Techniques et outils de développement logiciel

Zoubida Kedad, Stéphane Lopes
zoubida.kedad@uvsq.fr, stephane.lopes@uvsq.fr

2017–2018

Université de Versailles St-Quentin-en-Yvelines
UFR des Sciences
université Paris-Saclay

Table des matières

1 Préambule **1**
- 1.1 Objectifs et prérequis . . . 1
- 1.2 Plan . . . 1
- 1.3 Bibliographie . . . 2

2 Introduction **3**
- 2.1 Qu'est-ce que le développement logiciel ? . . . 3
- 2.2 Situation intiale . . . 4
- 2.3 Création d'un projet . . . 5
- 2.4 Compléments sur les outils utilisés . . . 5
  - 2.4.1 Hébergement d'un projet . . . 5
  - 2.4.2 Gestion des versions . . . 6
  - 2.4.3 Édition du code source . . . 13
- 2.5 Bilan . . . 14
- 2.6 Exercices . . . 15

3 Débogage et tests **18**
- 3.1 Introduction . . . 18
- 3.2 Débogage . . . 18
- 3.3 Tests . . . 20
  - 3.3.1 Généralités . . . 20
  - 3.3.2 Tests unitaires . . . 21
  - 3.3.3 JUnit . . . 22
  - 3.3.4 Couverture de code . . . 26
  - 3.3.5 Développement piloté par les tests . . . 26
- 3.4 Bilan . . . 27
- 3.5 Exercices . . . 27

4 Qualité du code et documentation **29**
- 4.1 Introduction . . . 29
- 4.2 Style de codage . . . 30
  - 4.2.1 Conventions de codage . . . 30
  - 4.2.2 Outils . . . 31
- 4.3 Documentation . . . 32
  - 4.3.1 Documenter un code source . . . 32
  - 4.3.2 JavaDoc . . . 33
  - 4.3.3 Doxygen . . . 35
- 4.4 Analyse statique du code . . . 38
  - 4.4.1 Audit de code source . . . 38
  - 4.4.2 FindBugs . . . 40
  - 4.4.3 PMD . . . 44
- 4.5 Bilan . . . 44

4.6 Exercices . . . 45

5 Construction d'un projet 47
5.1 Introduction . . . 47
5.2 GNU Make . . . 48
5.3 Apache Maven . . . 51
5.3.1 Généralités . . . 51
5.3.2 POM . . . 52
5.3.3 Cycle de vie . . . 53
5.3.4 Plugin et but . . . 54
5.3.5 Référentiel . . . 54
5.4 Gradle . . . 54
5.5 Apache Ant . . . 56
5.6 Bilan . . . 60
5.7 Exercices . . . 60

6 Vers la livraison continue 62
6.1 Introduction . . . 62
6.2 Collaborer avec un VCS . . . 62
6.2.1 Branches . . . 62
6.2.2 Workflows . . . 64
6.3 Intégration continue . . . 65
6.3.1 Introduction . . . 65
6.3.2 Pratiques recommandées . . . 65
6.3.3 Workflow et outils . . . 67
6.4 Gestion et distribution des binaires . . . 67
6.5 Livraison continue . . . 67
6.6 Déploiement continu . . . 68
6.7 Outils pour le déploiement continu . . . 68
6.8 Bilan . . . 69
6.9 Exercices . . . 69

7 Bilan et conclusion 71
7.1 Bilan . . . 71
7.1.1 Situation initiale . . . 71
7.1.2 Hébergement et gestion des versions . . . 71
7.1.3 Débogage et tests . . . 73
7.1.4 Conventions et audit . . . 74
7.1.5 Automatisation du build . . . 75
7.1.6 Intégration continue . . . 76
7.1.7 Distribution et gestion des binaires . . . 77
7.1.8 Livraison et déploiement continus . . . 78
7.1.9 Standardisation des postes de développement . . . 79
7.1.10 Organisation et gestion de l'équipe . . . 80
7.2 Les indispensables . . . 80
7.3 Pour aller plus loin . . . 80
7.4 Exercices . . . 81

Table des figures

2.1 Iterative and incremental development (Wikipedia). 4
2.2 Les différents états d'un document. 8

4.1 Principe de Javadoc. 34
4.2 Principe de Doxygen. 36
4.3 Vue d'ensemble du rapport. 41
4.4 Vue par classes. 42
4.5 Vue par type de bogues. 43
4.6 Exemple de résultat pour PMD. 44

6.1 Des branches avec git (source Git scm book). 63

Listings

5.1 Un `Makefile` simple 49
5.2 Un `Makefile` amélioré 50
5.3 Un `Makefile` pour Java (1ère partie) 50
5.4 Un `Makefile` pour Java (2ème partie) 50
5.5 Un `Makefile` pour Java (3ème partie) 51
5.6 Un POM simple 52

Liste des exercices

2.1 Exercice (Introduction à Git) 15
2.2 Exercice (Introduction à Subversion) 15
2.3 Exercice (Découverte d'Eclipse) 16

3.1 Exercice (Tests unitaires/TDD) 27
3.2 Exercice (Débogage et tests unitaires) 28

4.1 Exercice (Documentation avec Javadoc) 45
4.2 Exercice (Audit des conventions de codage avec Checkstyle) 45
4.3 Exercice (Recherche de bogues avec Findbugs) 46

5.1 Exercice (Construction d'un projet avec Maven) 60

6.1 Exercice (Workflow git feature branch) 69
6.2 Exercice (Workflow git forking) 70

7.1 Exercice (Une application web avec Spring Boot) 81

Chapitre 1

Préambule

Sommaire

1.1 Objectifs et prérequis 1
1.2 Plan 1
1.3 Bibliographie 2

1.1 Objectifs et prérequis

Objectifs du cours

— Donner un aperçu de ce qu'est une approche professionnelle du développement logiciel
— Présenter quelques techniques de développement
— Présenter les principales familles d'outils de développement
— Illustrer ces techniques et outils dans un environnement Java
— Présenter une notation pour la modélisation (UML)

1

Prérequis

— Petite expérience de la programmation
— Connaissance des concepts objets
— Connaissance du langage Java

2

1.2 Plan

Plan général

— Introduction
— Débogage et tests
— Qualité du code et documentation
— Construction d'un projet
— Vers la livraison continue
— Bilan et conclusion
— La notation UML

3

1.3 Bibliographie

Barnes, David et Michael Kölling (2016). *Objects First with Java. A Practical Introduction using BlueJ*. English. 6th edition. Pearson. URL : http://www.bluej.org/objects-first/.

Hunt, Andrew et David Thomas (2001). *The Pragmatic Programmer*. the Pragmatic Bookshelf. URL : http://www.pragprog.com/titles/tpp/the-pragmatic-programmer.

Martin, Robert C. (2008). *Clean Code : A Handbook of Agile Software Craftsmanship*. Prentice Hall. URL : http://www.pearsonhighered.com/educator/academic/product/1,3110,0132350882,00.html.

Smart, John Ferguson (2008). *Java Power Tools*. O'Reilly. URL : http://www.wakaleo.com/java-power-tools.

Chapitre 2

Introduction

Sommaire

2.1 Qu'est-ce que le développement logiciel ? 3
2.2 Situation intiale 4
2.3 Création d'un projet 5
2.4 Compléments sur les outils utilisés 5
2.4.1 Hébergement d'un projet 5
2.4.2 Gestion des versions 6
2.4.3 Édition du code source 13
2.5 Bilan 14
2.6 Exercices 15

2.1 Qu'est-ce que le développement logiciel ?

Objectifs du développement logiciel

1. **Satisfaire l'utilisateur final**
   - le logiciel est réalisé dans le but de répondre à des *exigences*
   - nécessite de respecter certains *critères de qualité externes*
2. Faciliter l'évolution et la maintenance
   - permet de diminuer les coûts de développement
   - nécessite de respecter certains *critères de qualité internes*

4

Moyens pour y parvenir

- Évoluer du bricolage vers une approche professionnelle du développement logiciel
- Nécessite de gérer le cycle de vie d'une application (*Application lifecycle management*)
  - processus continu de gestion de la vie d'une application du lancement du projet jusqu'à la maintenance
  - prend en compte la gestion de l'entreprise et le génie logiciel
- Passe par l'adoption d'un processus de développement (*Software development process*)
  - organise un ensemble de tâches ou d'activités
  - définit des règles de gestion de l'équipe
  - différents modèles (chute d'eau, en V, en spirale, itératif et incrémental, agile, . . .)

5

Principales activités du processus de développement

- Gestion de projet
- Analyse des besoins/exigences
- Spécifications fonctionnelles et non fonctionnelles
- Architecture logicielle
- Conception
- Implémentation
- Tests
- Déploiement
- Évolution et maintenance

6

Exemple
Activités du processus unifié (RUP)
(voir figure 2.1).

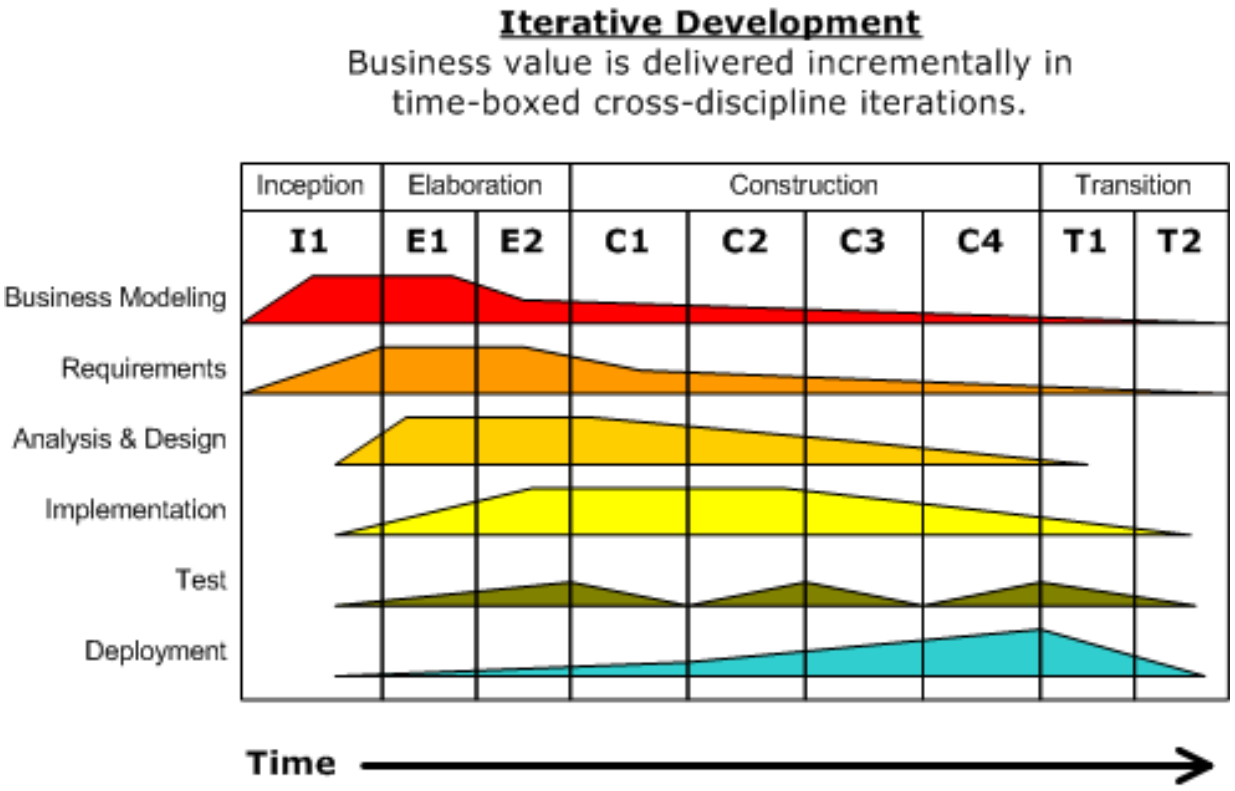
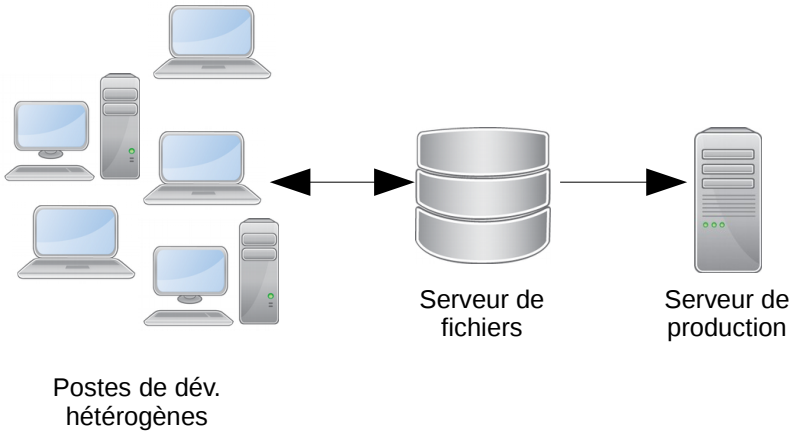


FIGURE 2.1 – Iterative and incremental development (Wikipedia).

7

2.2 Situation intiale

Collaboration par serveur de fichiers



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

8

2.3 Création d'un projet

Contexte

- Projet de petite taille en Java (peu de code)
- Un unique développeur

9

Démonstration

1. Création du projet sur une forge
2. Création du projet local avec git et un IDE
3. Ajout d'un fichier dans le projet
4. Synchronisation du projet local avec la forge
5. Quelques améliorations simples

10

2.4 Compléments sur les outils utilisés

2.4.1 Hébergement d'un projet

Forge logicielle

- Une *forge logicielle* est un système de gestion de développement collaboratif
- Une forge intègre un ensemble d'outils dans une interface web
  - un système de gestion de version
  - des outils de collaboration (fils de discussion, ...)
  - du suivi de tickets (bogues, ...)
  - un gestionnaire de documentation (wiki par exemple)

Quelques forges

GitHub, Bitbucket, SourceForge.

11

2.4.2 Gestion des versions

Généralités

Introduction

— La *gestion de versions* (*version control*, *revision control*, *source control* ou *source code management (SCM)*) consiste à maintenir l'ensemble des versions d'un document
— La gestion de versions concerne principalement le développement logiciel mais peut être utilisée pour tout document informatique
— C'est une activité fastidieuse et complexe qui bénéficie d'une façon importante d'un support logiciel (*système de gestion de versions*, *version control system (VCS)*)
— De nombreuses applications proposent ce genre de fonction (*Wiki*, *LibreOffice*, *Microsoft Word*, ...)

12

Les 3 opérations de base

— Obtenir une copie de travail (*checkout*)

```
cp unFichier unFichier.new
```

— Modifier la copie de travail

```
vi unFichier.new
```

— Transformer la copie de travail en version courante (*commit* ou *checkin*)

```
cp unFichier unFichier.old-<date>
cp unFichier.new unFichier
```

13

Difficultés de la gestion de versions

— Le nombre d'anciennes versions augmentent rapidement
— Occupation disque importante
  — un changement mineur provoque la copie totale du fichier
— *Sensible aux erreurs et aux oublis*

14

Une solution : les outils de gestion de versions

— Permet la 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
  — permet de revenir aisément à une version antérieure
— Permet le travail collaboratif
  — chaque utilisateur travaille sur une copie locale
  — le système signale les conflits

15

Périmètre d'utilisation des VCS

— Gestion d'un ensemble de fichiers textes
  — code source
  — pages html, xml, ...
  — documents LaTeX
— Supportent l'organisation en hiérarchie de répertoires et les métadonnées (permissions, ...)
— Supportent également les fichiers binaires mais l'intérêt diminue

16

Modèles pour la gestion de version

Problème de l'accès concurrent

La modification d'un même fichier par deux personnes au même moment risque de générer des incohérences.

Approche par verrouillage (*locking*)

- un fichier doit être verrouillé avant une modification
  - ⇒ une seule personne peut le modifier à un instant donné (approche *pessimiste*)
- simple mais réduit considérablement la concurrence

Approche par fusion (*merging*)

- plusieurs personnes peuvent modifier un fichier en parallèle (approche *optimiste*)
- le système s'occupe de fusionner les différentes modifications
- certains cas ne peuvent pas être traités automatiquement (*conflits*)

17

Architecture des VCS

- Les premiers VCS ne supportaient qu'un mode local
- La seconde génération fonctionnait selon un mode client/serveur (mode centralisé)
  - le référentiel est centralisé
  - tout doit être reporté sur ce référentiel
  - nécessite donc un accès au référentiel pour la plupart des opérations
- Les nouveaux VCS (*Distributed VCS* ou *DVCS*) supportent un mode pair à pair (mode réparti)
  - chaque développeur possède son propre référentiel
  - un utilisateur peut récupérer une partie d'un référentiel accessible (*pull*)
  - un utilisateur peut publier une partie de son référentiel dans un autre (*push*)

18

Quelques outils

- Mode local
  - SCCS puis RCS : anciens outils sous Unix
- Mode client/serveur
  - Subversion
  - CVS
- Mode réparti
  - git
  - Mercurial
  - Bazaar
  - darcs

19

Git

Introduction

- *Git* est un DVCS libre
- Initialement créé par Linus Torvalds pour le développement du noyau Linux
- Beaucoup de projets utilisent Git (noyau Linux, Perl, Gnome, Samba, X.org, ...)

20

Caractéristiques

— Supporte un mode de développement fortement distribué et non linéaire
— Chaque copie de travail est un dépôt qui peut être publié
— L'authenticité de l'historique est assurée grâce à des algorithmes de crytographie (SHA-1)
— Gère des instantanés d'arborescence de répertoires (pas de fichier individuel)

21

Les trois états d'un document

Validé (*committed*) conservé dans le dépôt
Modifié (*modified*) modifié dans le répertoire de travail
indexé (*staged* ou *indexed*) prêt pour la validation
(voir figure 2.2).

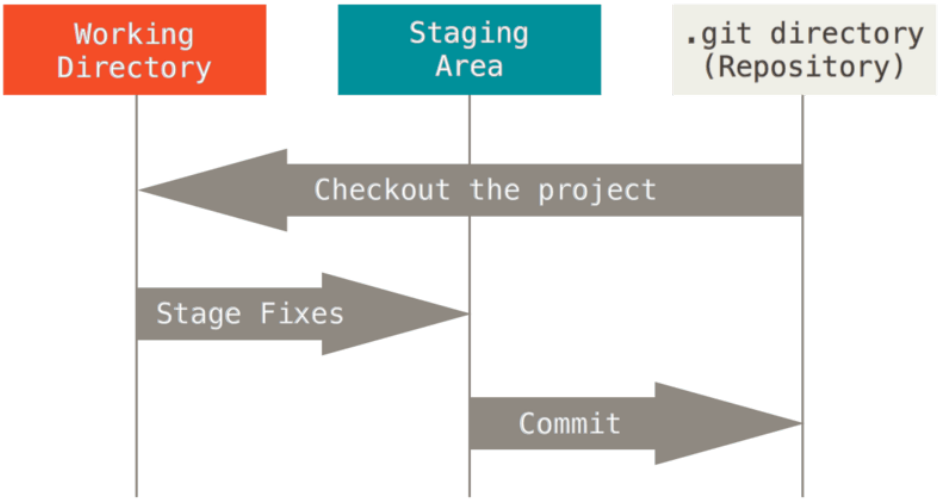


FIGURE 2.2 – Les différents états d'un document.

22

Workflow de base

1. Modifier la copie de travail
2. Ajouter le ou les documents à l'index
3. Valider l'ensemble des changements

23

Configuration initiale de git

— Le comportement de git peut être adapté grâce à la commande `git config`
— La configuration peut être : globale au système (`--system`), globale pour l'utilisateur (`--global`) ou spécifique au projet
— `git config --list` affiche la configuration courante
— Initialement, on définit au minimum son identité

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

24

Obtenir un dépôt git

— Initialiser un dépôt dans le répertoire courant (`git init`)
— Copier un dépôt existant (`git clone`)
— Le clonage peut utiliser les protocoles `https` ou `git` (avec `ssh`)

```
git clone https://github.com/libgit2/libgit2
```

25

Consulter l'état des fichiers

— La commande `git status` affiche l'état des documents
  — non suivi (nouveau), suivi et modifié, indexé
— L'option `--short` (ou `-s`) donne l'information de façon concise

26

Ajouter des fichiers dans l'index

— `git add` permet de placer des fichiers dans l'index
— *Cette commande doit être exécutée après chaque modification*
— `git rm` et `git mv` permettent de supprimer, déplacer/renommer les fichiers et d'indexer ces modifications

27

Examiner les changements

— `git diff` affiche le détail des changements sur les fichiers
— Sans option, les différences sont entre la copie de travail et l'index
— `--cached` compare l'index et le dernier commit
— Il est également possible de comparer une révision particulière avec la copie de travail, deux révisions, ...

28

Valider les modifications

— `git commit` valide le contenu de l'index
— Chaque *commit* est associé à un message (option `-m` de git commit)
— L'option `-a` permet de valider tous les changements des fichiers déjà suivis sans `git add` préalable

29

Ignorer des fichiers

— Certains fichiers ne doivent pas être suivis
  — résultat de la compilation
  — fichiers temporaires d'un éditeur
— Un fichier `.gitignore` placé dans le projet (et dans le dépôt) permet de lister les fichiers et répertoires à ne pas suivre
— Des exemples pour de nombreux types de projets sont disponibles

30

Consulter l'historique

- `git log` liste l'ensemble des révisions enregistrées
- L'option `-2` (ou `-n` avec n entier) limite aux n dernières
- `-p` affiche également les différences
- Le format de la sortie peut être adapté (`--pretty=oneline`, `-pretty=format:"..."`)
- `--graph` montre le graphe des branches et des fusions

31

Échanger avec un autre dépôt

- `git remote` permet de gérer les références à un dépôt distant

```
# ajoute une référence origin vers un dépôt
git remote add origin https://github.com/libgit2/libgit2
# liste les références
git remote -v
```

- `git fetch` récupère les révisions des dépôts référencés
- `git pull` récupère les révisions et les fusionne
- `git push` envoie les révisions locales vers une référence

```
git push origin master
```

32

Ressources

- Try git tutorial (interactif dans le navigateur)
- Documentation officielle
- Carte de référence GitHub
- Carte de référence (interactive)

33

Subversion

Introduction à Subversion

- Subversion est un VCS centralisé
- Distribué sous une licence libre
- Conçu comme un remplaçant de CVS ⇒ propose les mêmes concepts en les améliorant

34

Principaux apports de Subversion

- Les *commits* sont atomiques
- Le déplacement de fichiers et de répertoire est possible sans perdre l'historique
- Les métadonnées sont versionnées (permissions, ...)
- Supporte les fichiers binaires (diff binaire)

35

Accès au référentiel

- Un référentiel Subversion est identifié par une URL
- Cette URL précise le mode d'accès au référentiel (3 modes)
- Localement
  - le référentiel se trouve sur la machine locale
  - authentification et droits par le système
  - par exemple `file:///chemin/vers/repo`
- Protocole HTTP
  - support des protocoles `http` et `https`
  - authentification par le serveur web, droits de l'utilisateur web
  - par exemple `http://unserveur.domaine.fr/svnrepo`
- Protocole adhoc (`svnserve`)
  - ne nécessite pas de serveur web
  - authentification et droits par le système
  - peut utiliser `ssh`
  - par exemple `svn+ssh://unserveur.domain.fr/chemin/vers/repo`

Révisions

- Les numéros de révision (version) sont globaux au référentiel (pas de version par fichier)
- Une révision reflète l'état du référentiel à un instant donné
- Lors d'une validation, une nouvelle révision est créée
- La révision k représente l'état du référentiel après k validations
- L'option `-r` (`--revision`) de la plupart des commandes permet de faire référence à une version (ou une plage de versions) particulière

```
svn co -r 12 # checkout de la version 12 svn log -r 12:34 # messages entre les versions
12 # et 34
```

- Une révision peut aussi être référencée par mot clé ...

  HEAD version la plus récente du référentiel

  BASE version d'un item de la copie locale

  COMMITTED plus récente version ($\leq BASE$) où l'item a changé

  PREV la version précédant immédiatement `COMMITTED`

```
svn log -r HEAD # messages de la dernière version svn diff -r PREV:COMMITTED foo.c
# derniers changements # sur foo.c
```

- ...ou par date

```
svn co -r 2006-02-17 svn co -r 2006-02-17T15:30
```

Principales opérations

— Récupérer une copie de travail (checkout)

`svn co URL`

— Mettre à jour la copie de travail

`svn up`

— Manipuler des fichiers ou des répertoires

`svn add fichier.txt repertoire, svn rm fichier, svn mv fichier.txt autrefichier.txt, svn cp fichier.txt autrefichier.txt,`

— Examiner les changements

`svn status, svn status -show-updates -verbose, svn diff, svn log`

— Annuler les changements (local)

`svn revert`

— Valider les modifications

`svn ci -m"Message d'explication."`

38

Mise à jour de la copie de travail

— Les changements présents dans le référentiel sont reportés sur la copie local

— L'affichage précise le changement apporté par une lettre devant le nom du fichier

A ajout

B verrou "cassé" (uniquement en 3ème colonne)

C conflit détecté

D suppression

E existant

G fusion

U mis à jour

— La première colonne concerne le fichier, la deuxième les propriétés du fichier, la troisième le verrouillage

39

Résolution des conflits

— Un conflit est signalé lors d'un *update* par la lettre `C` en début de ligne

— Effets d'un conflit

  — des marqueurs sont placés dans le fichier à l'endroit des conflits

  — trois fichiers sont créés : la copie de travail, la version `BASE` et la dernière version `HEAD`

  — le *commit* est interdit

— Actions possibles

  — résoudre le conflit à la main (réaliser manuellement la fusion)

  — remplacer le fichier en conflit par l'une des 3 anciennes versions

  — exécuter `svn revert` pour anuuler les changements

— Signaler à Subversion que le conflit a été traité

  — `svn resolve` supprime les fichiers temporaires et autorise le *commit*

— il faut préciser l'alternative choisie

```
svn resolve --accept working foo.c # copie actuelle
svn resolve --accept base foo.c # BASE
svn resolve --accept mine-full foo.c # copie avant update
svn resolve --accept theirs-full foo.c # HEAD
```

40

Branches et tags

— Subversion considère les branches et les tags comme des copies d'une révision du projet

```
svn copy http://svn.example.com/repos/calc/trunk \
        http://svn.example.com/repos/calc/branches/my-calc-branch \
        -m "Creating a private branch of /calc/trunk."
```

— Après avoir réalisé un *checkout* de la branche et l'avoir fait évoluer, il est possible de fusioner une autre branche

```
svn merge http://svn.example.com/repos/calc/trunk
```

— Un tag est simplement une copie qui n'est pas destinée à être modifiée

```
svn copy http://svn.example.com/repos/calc/trunk \
        http://svn.example.com/repos/calc/tags/release-1.0 \
        -m "Tagging the 1.0 release of the 'calc' project."
```

41

Organisation des répertoires d'un projet

monprojet la racine du projet

- **trunk** la ligne de développement principale
- **branches** les lignes alternatives
  - **bug1234** correction du bogue 1234
  - ...
- **tags** les révisions marquées
  - **version-1.0** la version 1.0 du logiciel
  - ...

42

2.4.3 Édition du code source

Fonctionnalités de base pour un éditeur de code source

— Support de projets (ensemble de fichiers avec une configuration)
— Édition de fichiers en texte brut (*éditeur de texte* vs. traitement de texte)
— Support des jeux de caractères étendus (UTF-8, Latin1, ...)
— Défaire/refaire, couper/copier/coller, rechercher/remplacer, ...
— Aides à la saisie (coloration syntaxique, indentation automatique, reformatage, ...)
— Pliage (*folding*)/dépliage de texte
— Modèle (insertion de texte type)
— Index des fonctions (*ctags*)
— Appel de programmes externes (compilateur, ldots)

43

Environnement de développement intégré

— Un *environnement de développement intégré* (*Integrated Development Environment* ou *IDE*) regroupe et intègre un ensemble d'outils de développement
  — éditeur de code source,
  — compilateur/interpréteur,
  — débogueur,
  — un navigateur de classes,
  — un ou plusieurs systèmes de gestion de versions

44

Principales fonctionnalités d'un IDE

— Gestion de projets (fichiers, dépendances, configuration, ldots)
— Auto-complétion de code
— Navigation dans les classes
— *Refactoring*
— Débogage
— Profiling
— Gestion de versions

45

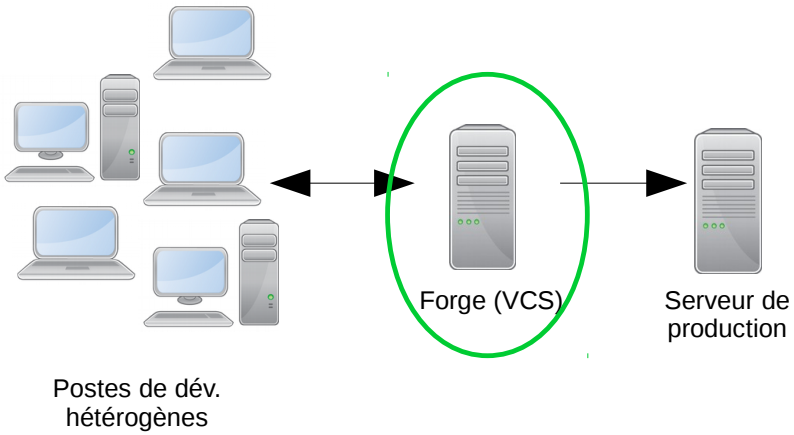
Quelques outils

IDE
Eclipse, IntellijIDEA, NetBeans, Microsoft Visual Studio, Code::Blocks, Geany.

Éditeurs
Vi, vim, Emacs, GEdit, Notepad++ Sublime Text.

46

2.5 Bilan



— Communication/tâches
— Fusion/versions
— Build
— Distribution des binaires
— Tests/intégration
— Ass. Qualité
— Livraison
— Documentation
— Postes de dév. standards

47

Bilan

Apports

— Sauvegarde du projet (réparti sur l'ensemble des machines et sur la forge)

— Maintien de l'historique des évolutions du projet

Limites

— Quel confiance peut-on avoir dans le code développé ?

— Comment faire collaborer plusieurs personnes sur le projet ?

2.6 Exercices

CONTRAINTES

— Pour l'ensemble des TDs, vous créerez un compte individuel sur Bitbucket. Vous nommerez ce compte de la façon suivante : `uvsq<MonNuméroÉtudiant>`. Par exemple, pour un étudiant de numéro 21601234, le compte sera `uvsq21601234`.

— Pour chaque commande VCS réalisée dans l'IDE, vous donnerez la ligne de commande correspondante.

— Vous pouvez utiliser l'IDE de votre choix. Sur le cartable numérique, ECLIPSE et INTELLIJ IDEA sont installés.

Exercice 2.1 (Introduction à GIT)
Dans cet exercice, vous créerez une classe `Fraction` représentant un nombre rationnel et une classe `Main` qui testera les méthodes de la classe `Fraction`. À chaque étape, consultez le statut des fichiers du projet ainsi que l'historique.

1. Sur la forge, créez le projet Java `SimpleFraction` ;
2. Localement, configurez GIT avec votre nom (*user.name*) et votre email (*user.email*) ;
3. Initialisez le dépôt GIT local pour le projet ;
4. Créez la classe `Fraction` (vide pour le moment) et la classe `Main` (avec un simple affichage) dans le projet ; Vérifiez que le projet compile et s'exécute dans l'IDE ; Validez les changements ;
5. Ajoutez un constructeur et la méthode `toString` à la classe `Fraction` et modifiez la classe `Main` en conséquence ; Validez les changements ;
6. Publiez vos modifications sur le dépôt distant ; Vous utiliserez le protocole `https` pour cela ; Vérifiez avec le navigateur ;
7. Sur la forge, ajoutez un fichier de documentation `README.md`. Quelle syntaxe est utilisée pour ce fichier ?
8. Récupérez localement les modifications effectuées sur la forge.
9. Ajoutez les répertoires et fichiers issus de la compilation aux fichiers ignorés par GIT (fichier `.gitignore`) ;
10. Retirez les fichiers de configuration de l'IDE du projet ; Ajoutez-les aux fichiers ignorés par GIT.
11. Configurez l'accès par clé publique/clé privée à la forge (cf. Use the SSH protocol with Bitbucket Cloud).

Exercice 2.2 (Introduction à SUBVERSION)
Dans cet exercice, vous travaillerez sur le même projet que dans l'exercice précédent.

1. Créez un dépôt SUBVERSION local dans le répertoire de votre choix ; Créez une copie de travail de ce dépôt ;
2. Créez la classe `Fraction` (vide pour le moment) et la classe `Main` (avec un simple affichage) dans le projet ; Vérifiez que le projet compile et s'exécute dans l'IDE ; Validez les changements ; Les fichiers devront se trouver dans le sous-répertoire `trunk` du dépôt ;
3. Ajoutez un constructeur et la méthode `toString` à la classe `Fraction` et modifiez la classe `Main` en conséquence ; Validez les changements ;
4. Ajoutez les répertoires et fichiers issus de la compilation aux fichiers ignorés par SUBVERSION (propriété `svn:ignore`) ;
5. Retirez les fichiers de configuration de l'IDE du projet ; Ajoutez-les aux fichiers ignorés par SUBVERSION.

Exercice 2.3 (Découverte d'Eclipse)
Cet exercice peut être adapté pour un IDE similaire.

1. Procédez au paramétrage initial d'Eclipse suivant (*Window* > *Preferences*) :
   — ajoutez un dictionnaire français (*General* > *Editors* > *Text Editors* > *Spelling* en ISO-8859-1),
   — modifiez l'encodage des caractères en *UTF-8* et les fins de ligne en *Unix* (*General* > *Worspace*),
   — vérifiez que l'option *Build automatically* est cochée (*General* > *Workspace*),
   — vérifiez les versions du JDK installées sur la machine et sélectionnez la plus récente (*Java*),
   — attachez les sources du JDK à la bibliothèque `rt.jar` (*Source Attachment...* dans les propriétés du JRE),
   — définissez les chemins des sources (`src/main/java`) et du bytecode (`target/classes`) pour les projets,
   — vérifiez la version du compilateur utilisé et sélectionnez la plus récente,
   — activez la mise en forme du code lors des sauvegardes (*Java* > *Editor* > *Save Actions*).
2. Dans la forge, effectuez un *fork* du projet SimpleShapes dans votre espace ;
3. Créez un projet Eclipse à partir de ce dépôt ;
4. Explorez le projet :
   — en utilisant la vue *Package Explorer*, naviguez dans les sources pour ouvrir le fichier `Main.java` dans un éditeur,
   — dans la vue *Outline*, utilisez la barre d'icones pour masquer/voir les différents éléments (membres statiques, ...),
   — quel impact la vue *Outline* a-t-elle sur l'éditeur ?
   — dans l'éditeur, faites apparaître la vue *Quick outline* (*Ctrl+o*) puis les membres hérités (*Ctrl+o*) et sélectionnez la méthode `java.lang.Object.toString()`,
   — dans l'éditeur, sur le fichier `Main.java`, placez le curseur sur le type `Shape` et faites apparaître la *hiérarchie des types* (*Ctrl+T* pour *Quick Type Hierarchy* ou *F4* pour la vue *Type Hierarchy*),
   — quelles classes implémentent `Shape` ?
5. Visualisez les erreurs du projet :
   — comment les erreurs dans le projet sont-elles signalées (vue et moyen) ?
   — placez le pointeur de la souris sur les différents symboles d'erreur dans l'éditeur,
   — placez le curseur sur l'erreur concernant le type `LogFactory`, faites apparaître la vue *Quick Fix* (*Ctrl+1*) et sélectionnez *Fix project setup...*.
6. Ajoutez la librairie de journalisation dans le projet :

— créez un répertoire `lib` dans le projet,

— récupérez et décompressez l'archive de la librairie *Apache Commons logging* (`http://commons.apache.org/logging/`) dans ce répertoire,

— ajoutez la librairie au projet (*Project* > *Properties*) en y associant les sources et la javadoc.

7. Exécutez le projet :
   — lancez le programme comme une application Java,
   — éditez la configuration d'exécution pour définir les propriétés systèmes suivantes (*VM arguments*) :

   ```
   -Dorg.apache.commons.logging.Log=org.apache.commons.logging.impl.SimpleLog
   -Dorg.apache.commons.logging.simplelog.defaultlog=trace
   ```

   — lancez à nouveau le programme. Que remarquez-vous ?
8. Éditez le code :
   — dans la classe `Main`, ajoutez la méthode `public void trace()`,
   — utilisez *Content Assist* pour saisir le corps de la méthode suivant (après `log.`, tapez *Ctrl+espace*, puis `tr`, ...) :

   ```
   log.trace(shapes);
   ```

   — ajoutez la méthode `public void translate()`,
   — saisissez le corps suivant en utilisant les *templates de code* (après `for`, tapez *Ctrl+espace*, puis sélectionner *foreach*, puis *Tab* entre les champs) :

   ```
   for (Shape s : shapes) {
       s.translate(2.0, 3.0);
   }
   ```

   — Organisez les importations de module :
   — dans la vue *Outline*, rendez visible les déclarations d'*import* (*View Menu* > *Filters...*),
   — dans la même vue, supprimez les déclarations d'*import* du fichier Java,
   — enfin, dans l'éditeur, ajoutez automatiquement les déclarations d'*import* (*Source* > *Organize Imports* ou *Shift+Ctrl+o*).
9. Apportez les modifications suivantes au projet (*Refactoring*) :
   — changez le nom du package `shapes` en `forms`,
10. Générez un Jar exécutable du projet contenant les dépendances :
    — exportez le projet comme un Jar exécutable,
    — vérifier en ligne de commande que l'archive fonctionne de façon autonome.

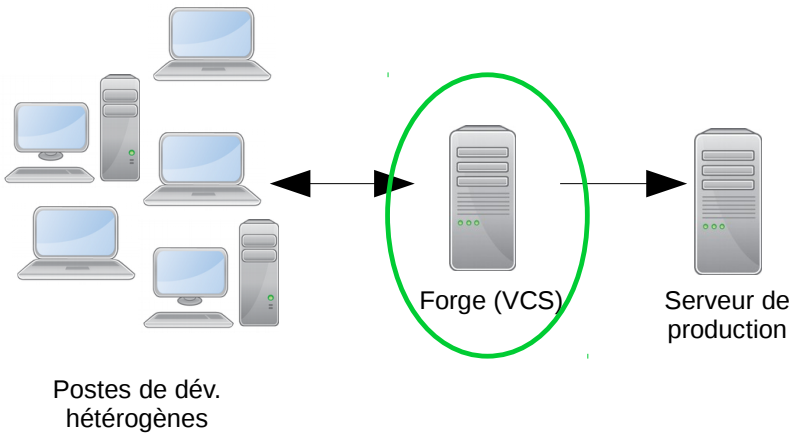
Chapitre 3

Débogage et tests

Sommaire

3.1 Introduction **18**
3.2 Débogage **18**
3.3 Tests **20**
3.3.1 Généralités 20
3.3.2 Tests unitaires 21
3.3.3 JUnit 22
3.3.4 Couverture de code 26
3.3.5 Développement piloté par les tests 26
3.4 Bilan **27**
3.5 Exercices **27**

3.1 Introduction



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- →**Tests/intégration**←
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

49

3.2 Débogage

Bogue, débogage et débogueur

- Un *bogue* (*bug*) est un défaut dans un programme qui l'empêche de fonctionner correctement
- Le *débogage* (*debugging*) est une activité ayant pour objectif de localiser les bogues dans un programme
- Le débogage est basé sur la *confirmation*

— Le débogage est un processus destiné à confirmer les choses que l'on croit vrai jusqu'à en trouver une qui ne l'est pas

— Un *débogueur* (*debugger*) est un outil fournissant une aide pour le débogage

Quelques outils

GDB/DDD, VALGRIND, IDE.

50

Pourquoi utiliser un débogueur ?

Pour gagner du temps !

Alternatives

— Utiliser des affichages (`printf`, ...)
  — perte de temps
  — beaucoup d'éditions/compilations pour ajouter/enlever les affichages
  — moins informatif
— Utiliser une bibliothèque de journalisation (*logging*)
  — plus pratique que les affichages
  — moins informatif que le débogueur

51

Fonctionnalités

— Exécution contrôlée du programme
  — pas à pas sommaire (sans entrer dans les fonctions)
  — pas à pas détaillé
  — retour en arrière (plus rare)
— Pose de *points d'arrêt* (*breakpoints*)
  — repère sur une instruction signalant au débogueur qu'il doit faire un pause dans l'exécution lorsqu'il arrive à cette instruction
  — peut être également associé à une condition
  — un *point d'observation* (*watchpoint*) stoppe le programme lorsque l'état d'une expression change
  — un *catchpoint* fait de même quand un événement se déclenche
— Visualisation de l'état du programme
  — variables, pile d'appel, ...
  — certains débogueurs permettent l'affichage de structure de données complexes
— Modification de l'état du programme
— Débogage à distance

52

Processus de débogage

1. Tenter de reproduire le bogue
2. Simplifier les entrées du programme
3. Exécuter le programme sous le contrôle du débogueur
4. Se positionner à l'endroit de l'erreur signalée ou au milieu du programme (pose d'un breakpoint) si aucune erreur n'est signalée
5. Examiner le contexte : confirmer que les variables possèdent les valeurs attendues
6. Déterminer la position suivante à étudier et retourner en 5

Remarque

— Nécessite de compiler le programme avec les informations de débogage (option `-g` de `javac`)

53

3.3 Tests

3.3.1 Généralités

Définition et intérêt

— Les *tests* visent à mettre en évidence des défauts de l'élément testé
— L'objectif final est bien sûr de réduire le nombre de bogues présents dans un programme
— Un test est un ensemble de *cas à tester* (conditions initiales, entrées, observations attendues)
— *Un test ne permet pas de prouver l'absence de bogue* ($\neq$ méthodes formelles)
— Il est impossible d'exécuter des tests exhaustifs dans la plupart des cas
— Les tests sont toutefois une aide précieuse pour améliorer la qualité du logiciel

54

Types de tests

— Un test *boite blanche* (*white box*) s'appuie sur une connaissance de la structure interne de l'élément à tester
— Un test *boite noire* (*black box*) s'appuie sur les spécifications de l'élément
— Un test de *non régression* vérifie que le système ne se dégrade pas lors de ses évolutions
— Un test *fonctionnel* s'assure que les résultats attendus sont bien obtenus
— Un test *non fonctionnel* analyse les propriétés non fonctionnelles d'un système
  — test *des performances* pour vérifier l'efficacité du système
  — test *de sécurité* pour s'assurer du respect des règles de confidentialité

55

Niveaux de tests

Unitaire Les tests unitaires vérifient la conformité des éléments de base d'un programme (fonctions, classes, ...) et sont en général réalisés par le développeur.

Intégration Les tests d'intégration vérifient la cohérence des différents modules et la bonne communication entre eux.

Système Les tests systèmes concernent l'ensemble du projet et son intégration dans son environnement.

Recette Les tests de recette (ou d'acceptation) confirment la conformité du système avec les besoins.

56

Intégration au processus de développement

— Généralement (cycle de développement en V par exemple), les tests sont réalisés par un groupe de testeurs après la réalisation des fonctionnalités

— Une pratique encouragée par les méthodes Agiles et XP consiste à débuter le processus par les tests (*Développement dirigé par les tests*)

Quelques outils pour les tests

— unitaires et d'intégration : JUnit, TestNG.

— de recette : Fit, FitNesse.

— non fonctionnels : Apache JMeter, JUnitPerf.

57

3.3.2 Tests unitaires

Définition et objectifs

— Un *test unitaire* (*unit test*) vise à augmenter la confiance du programmeur dans des portions du code source

— Une *unité* fait référence à la plus petite partie testable de l'application (fonction, méthode)

— Peuvent être réalisés avec un débogueur ou avec un framework spécialisé de type `xUnit`

— Le but des tests unitaires est d'isoler chaque partie du programme pour la tester indépendamment

— Isoler les différents éléments nécessite souvent d'avoir recours à du code de substitution (*stub*, *fake* ou *mock object*)

58

Principe

— Pour chaque unité, on écrit une ou plusieurs méthodes de test

  — un outil de gestion est nécessaire vu le nombre de tests

— Une possibilité intéressante est d'écrire le test avant la méthode

  — précise d'abord ce que doit faire la méthode

— L'ensemble des tests peut ensuite être répété autant que nécessaire

  — l'exécution des tests après chaque modification permet de vérifier la non régression

59

Caractéristiques des tests unitaires

— Petits (analyse d'un point précis) et rapides (exécutés souvent)

— Totalement automatisés

— Toujours au niveau de l'unité

— Indépendants les uns des autres (pas de contraintes d'ordre)

— N'utilisent pas de ressources externes (SGBD, . . .)

60

Doublure de tests

- Un test unitaire se focalise sur un élément particulier
- Ce dernier peut être dépendant d'autres éléments
- Une *doublure de test* permet de remplacer ces dépendances
- Plusieurs types de doublure

fantôme un objet *fantôme* (*dummy*) sert juste à remplir des listes de paramètres

substitut un objet *substitut* (*fake*) fournit une implémentation sumplifiée

bouchon un objet *bouchon* (*stub*) retourne des réponses prédéfinies spécifiques aux tests

simulacre un objet *simulacre* (*mock*) sont préprogrammés par des attentes, i.e. une spécification du comportement attendu

61

3.3.3 JUnit

Introduction à xUnit

- `xUnit` est le nom d'un ensemble de frameworks de test unitaire
- Ces frameworks fournissent un cadre pour développer les tests et automatisent leurs exécutions
- Ils sont basés sur SUnit, le framework de Kent Beck pour Smalltalk
- Existent pour de nombreux langages (Java, C++, PHP, ...)
- JUnit a été développé par Kent Beck et Erich Gamma

62

Concepts

Test case un test proprement dit

Test fixtures le contexte du test

Test suite un ensemble de tests partageant le même contexte

Assertion un prédicat vérifiant l'état (ou le comportement) de l'élément à tester

Test runner un moyen pour exécuter des tests

63

Principe de JUnit

- Un cas de test est une méthode d'une classe Java
- En général, une classe regroupe plusieurs cas de test
- Les tests peuvent être regroupés en suite de tests
- Marche à suivre
  1. créer l'instance de l'élément à tester ainsi que ses dépendances
  2. appeler la méthode à tester avec les paramètres adéquat
  3. comparer les résultats obtenus avec les résultats attendus avec des assertions
- *Les cas de tests doivent être indépendant les uns des autres* (pas de contraintes d'ordre, ...)

64

JUnit 4

- Repose sur l'utilisation des annotations de Java 1.5
- Contenu dans le package `org.junit`

65

Création d'une classe de test

— Ne nécessite aucun traitement particulier
— Il faut juste importer les packages de JUnit
— En général, les assertions sont importées en `static`

```
import org.junit.*;
import static org.junit.Assert.*;

public class UneClasseDeTest {
  //...
}
```

66

Écrire un cas de test

— La méthode représentant le cas de test doit être annotée avec `org.junit.Test`
— Elle contient en général
  — une initialisation,
  — l'appel de la méthode à tester
  — des assertions pour vérifier les résultats

```
@Test
public void testEmptyCollection() {
  Collection<Object>collection = new ArrayList<Object>();
  assertTrue(collection.isEmpty());
}
```

67

Qu'est-ce qu'une assertion ?

— Une *assertion* est une expression booléenne exprimant une propriété sémantique (*prédicat*)
— C'est un outil simple pour exprimer et valider la correction d'une partie de code
— L'assertion formalise ce que le développeur *croit* vrai à un emplacement donné du programme
— Une assertion n'impose pas de surcoût dans la version de production
  — destinées aux versions de débogage
  — éliminées lors de la compilation pour les versions de productions

68

Intérêt des assertions

— Une assertion permet de vérifier que ce que l'on croit « trivialement vrai » reste actuellement vrai
— Apporte une aide pour :
  — la construction de programme correct (*spécification*)
  — la documentation (*programmation par contrat*)
  — le débogage
  — le raisonnement

69

Utilisations des assertions

Pré-condition exprime les contraintes pour l'appel d'un sous-programme, i.e. décrit les conditions dans lesquelles le composant logiciel fonctionnera

Post-condition exprime les garanties lors de la sortie d'un sous-programme, i.e. décrit les conditions établies en sortie du composant

Invariant de classe propriété qui caractérise les instances d'une classe

Invariant de boucle propriété toujours vraie lors de l'exécution d'une boucle

70

Assertion ou gestion d'erreurs

— La **gestion d'erreur** est utilisée pour vérifier les événements qui *peuvent* se produire même de façon très improbable

— Les **assertions** sont utilisées pour les événements que l'on pense ne devoir se produire en *aucune circonstance*

⇒ *Une assertion qui échoue signale une erreur de conception ou du programmeur, jamais une erreur de l'utilisateur*

71

Assertion et langage de programmation

— Avec le langage Eiffel, les assertions font partie du processus de conception

— En C/C++, la macro `assert` (`assert.h` en C, `cassert` en C++) permet d'insérer des assertions dans le programme

— En Java, le mot-clé `assert` fait de même

72

Assertion et JUnit

— Les assertions sont des méthodes de classe de la classe `org.junit.Assert`

— Chaque assertions existent en deux versions : avec ou sans message (1er paramètre)

assertArrayEquals égalité de deux tableaux

assertEquals égalité de deux éléments

assertFalse condition fausse

assertNotNull référence non `null`

assertNotSame identité de deux références

assertNull référence `null`

assertSame identité de deux références

assertThat vérifie une condition précisée par une instance de `Matcher`

assertTrue condition vraie

fail échec du test

73

Exécution des tests

— L'exécution peut se faire à partir de la console

```
java org.junit.runner.JUnitCore UneClasseDeTest
```

— JUnit s'intègre dans la plupart des IDE

74

Initialisation des tests

— Les initialisations communes à des cas de tests se font dans le *test fixture*

— Les méthodes annotées avec `org.junit.Before` sont exécutées avant chaque test

  — permet d'effectuer les créations d'instances

— Les méthodes annotées avec `org.junit.After` sont exécutées après chaque test

  — permet de libérer les ressources

```
@Before
public void setUp() {
  collection = new ArrayList<Object>();
}
```

75

Tests et exception

— Une exception attendue peut être spécifiée avec l'attribut `expected` de l'annotation `Test` (dans ce cas, le test passe)

```
@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
  ArrayList<Object> emptyList = new ArrayList<Object>();
  Object o = emptyList.get(0);
}
```

— Si une exception inattendue se produit, le test échoue

76

Quelques bonnes pratiques

— Placer la classe de test dans le même package que la classe testée

  — permet d'accéder aux membres accessibles du package

— Placer les fichiers de tests dans une arborescence séparée mais parallèle

  — simplifie la distribution du projet

— Tester ce qui peut raisonnablement échouer

  — *test until fear turns to boredom*

— Un élément difficile à tester peut nécessiter une nouvelle conception

— Exécuter les tests aussi souvent que possible

— Quand un bogue est identifié, écrire un test qui le mette en évidence

77

JUnit 3

— Le package à inclure est `junit.framework`

— Une classe de test hérite de `TestCase`

— Chaque cas de test est représenté par une méthode dont le nom débute par `test`

— Les méthodes d'initialisation/nettoyage des tests sont nommées `setUp/tearDown`

— L'exécution des tests est réalisée par la classe `junit.textui.TestRunner` (console), `junit.swingui.TestR` (GUI Swing) ou `junit.awtui.TestRunner` (GUI AWT)

— Un objet `TestSuite` est utilisé pour regrouper les tests

78

3.3.4 Couverture de code

Définition

- L'objectif est de vérifier que les tests unitaires couvrent bien l'ensemble du code écrit
- La *couverture de code* (*code coverage*) est un outil de mesure de la qualité des tests effectués
- Le degré de couverture est mesuré par des indices statistiques (rapport de la quantité de code testé sur la quantité de code écrit)
- Les portions de codes non testées sont mises en évidence

Un score de 100% ne garantit pas la correction du programme.Ce n'est même pas un objectif !

Quelques outils

Cobertura, EMMA, Clover.

79

Quelques métriques

- Le *Statement Coverage* (ou *Line Coverage*) mesure le degré d'exécution de chaque ligne
  - simple mais ignore un certain nombre d'erreurs simples (ne prend pas en compte la logique du programme)
- Le *Condition Coverage* indique si toutes les conditions ont été évaluées
  - les conditions doivent être évaluées à vrai et à faux pour obtenir un taux de 100%
  - aide à résoudre les problèmes de la mesure précédente
- Le *Path coverage* examine si chaque chemin a été parcouru
- Le *Function Coverage* vérifie si chaque fonction a été appelée

80

3.3.5 Développement piloté par les tests

Introduction

- Le *développement piloté par les tests* (*Test Driven Development* ou *TDD*) est une méthode de développement mettant l'accent sur les tests unitaires
- Cette méthode préconise d'écrire les tests avant le code
  - *Only ever write code to fix a failing test*
- Cette approche permet de spécifier ce que l'on attend du système avant de le réaliser
- Elle est basée sur les tests et le *refactoring*
- Le refactoring consiste à améliorer la conception du programme sans en changer le comportement (les fonctionnalités)
- Le TDD n'est pas limité aux tests unitaires mais s'applique aussi aux tests de recette (*Acceptance TDD*)

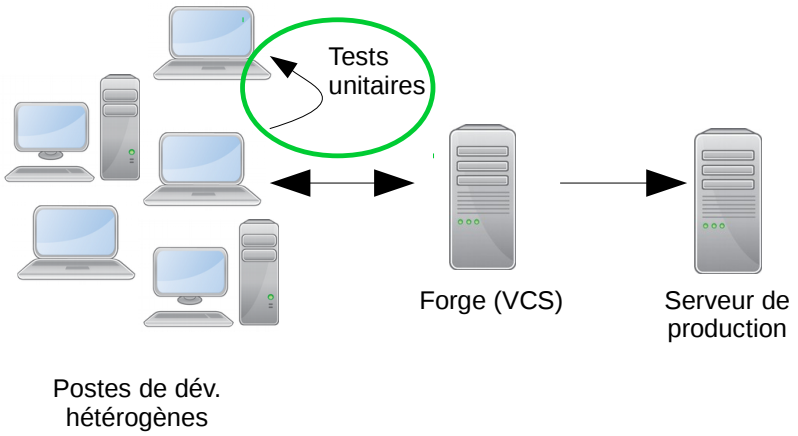
81

Cycle de développement

- Le TDD s'appuie sur de courtes itérations
- Chaque itération possède cinq étapes
  1. Ecrire un test
  2. Exécuter les tests et vérifier que le nouveau échoue
  3. Ecrire juste le code nécessaire pour faire passer le test
  4. Réexécuter les tests et vérifier que tous les tests passent
  5. Corriger la conception du système (*refactoring*)
- La phase de refactoring s'applique aussi bien au code de l'application qu'au code des tests

82

3.4 Bilan



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

83

Bilan

Apports

- Les techniques de débogage permettent une recherche de bogues plus efficace
- La présence des tests améliore la confiance dans le logiciel
- Les modifications sont plus aisées car sous la garde des tests

Limites

- Comment faire collaborer plusieurs personnes sur le projet ?
  - « règles » et bonnes pratiques de rédaction du code
  - automatisation des tâches

84

3.5 Exercices

CONTRAINTES

- Pour ces exercices, vous utiliserez votre dépôt BitBucket pour stocker vos réalisations.
- Vous créerez une nouvelle version à la fin de chaque cas de test. De cette façon, il sera possible de naviguer dans l'historique (`git checkout`) pour consulter l'état du projet avant le premier test, après le premier, après le second test, ...
- Chaque version que vous créerez devra compiler et faire passer l'ensemble des tests déjà implémentés.

Exercice 3.1 (Tests unitaires/TDD)
On veut implémenter une classe représentant un compte bancaire. Les opérations que devra supporter le compte sont :

- l'initialisation avec un solde initial,
- la consultation du solde,
- le crédit,
- le débit et,
- le virement vers un autre compte.

La réalisation devra tenir compte des contraintes suivantes :

— un compte n'est jamais à découvert,

— seules des sommes positives peuvent être passées en paramètre des opérations,

— l'invocation du débit ou du virement ne peut se faire qu'avec une somme inférieure au solde du compte concerné.

Vous utiliserez une approche pilotée par les tests pour la réalisation, i.e. écrivez toujours un test avant le code et pensez au refactoring. La prise en compte des contraintes se fera par de la gestion d'erreurs.

1. Énumérez une liste de cas de tests à réaliser.
2. Pour chaque cas de test,
   (a) écrivez le test JUnit correspondant dans la classe de test,
   (b) vérifiez qu'il échoue,
   (c) implémentez la fonctionnalité dans la classe `Compte`,
   (d) vérifiez que le test passe,
   (e) appliquez un étape de refactoring sur les tests et la classe `Compte` si nécessaire.

Exercice 3.2 (Débogage et tests unitaires)
L'exemple utilisé est issu du tutoriel de Cay Horstmann. La classe `WordAnalyser` permet d'analyser un mot en invoquant diverses méthodes. Cependant, cette classe contient plusieurs bogues.

Récupérez le fichier WordAnalyzer.java ainsi que les programmes de test (1, 2, 3). Chaque programme de test vérifie le fonctionnement d'une méthode.

1. Créez un projet dans l'IDE pour la classe `WordAnalyser`.
2. Créez la classe de test JUnit `TestWordAnalyser` qui contiendra les tests de `WordAnalyser`.
3. Transformez chaque test des programmes de test en test JUnit.
4. Si un test échoue, déboguez le programme pour identifier et corriger le problème.

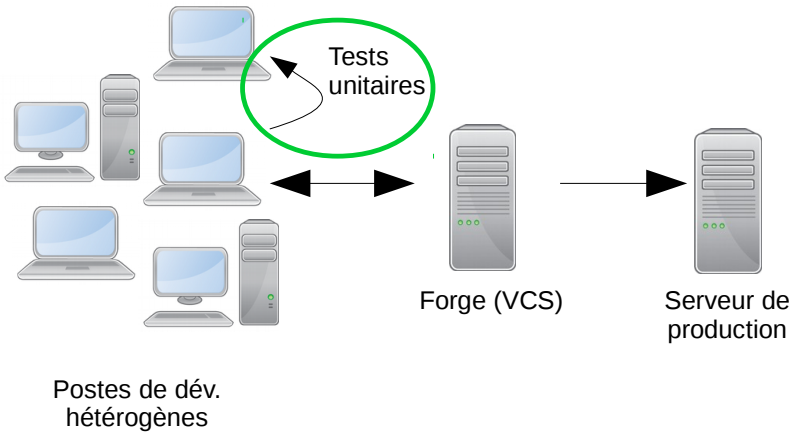
Chapitre 4

Qualité du code et documentation

Sommaire

4.1 Introduction 29
4.2 Style de codage 30
4.2.1 Conventions de codage 30
4.2.2 Outils 31
4.3 Documentation 32
4.3.1 Documenter un code source 32
4.3.2 JavaDoc 33
4.3.3 Doxygen 35
4.4 Analyse statique du code 38
4.4.1 Audit de code source 38
4.4.2 FindBugs 40
4.4.3 PMD 44
4.5 Bilan 44
4.6 Exercices 45

4.1 Introduction



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- →**Ass. Qualité**←
- Livraison
- Documentation
- Postes de dév. standards

85

4.2 Style de codage

4.2.1 Conventions de codage

Style de codage

— 80% du coût d'un logiciel concerne la maintenance

— Un logiciel n'est pas forcément maintenu par son auteur

— Améliore la lisibilité du code source ⇒ le code est plus facile à comprendre ⇒ le code est plus facile à maintenir

— On s'appuie généralement sur l'éditeur de texte et/ou un outil de vérification

L'important est de choisir un style et de s'y tenir !

86

Exemple

Une portion de code ne respectant pas de convention

```
public static <T extends Comparable<? super T>> void f(List<T> l) {
int s=l.size();
int i,j;
for(i=0;i<s-1;++i)
{
for(j=0;j<s-1-i;++j){
if(l.get(j+1).compareTo(l.get(j))<0){
T t=l.get(j);
l.set(j,l.get(j+1));l.set(j+1,t);
}}}
}
```

exemples/chap3/BubbleSortExample.java

87

Exemple

Une portion de code respectant une convention

```
    /**
     * Tri la liste passee en parametre en ordre ascendant
     * en respectant l'ordre naturel de ses elements.
     * Les elements de la liste doivent implementer
     * l'interface <code>Comparable</code>.
     * Cet methode utilise un algorithme de tri a bulle.
     *
     * @param aList la liste a trier
     */
  public static <T extends Comparable<? super T>> void bubbleSort(List<T> aList) {
  int size = aList.size();
  for (int i = 0; i < size - 1; ++i) {
    for (int j = 0; j < size - 1 - i; ++j) {
    if (aList.get(j+1).compareTo(aList.get(j)) < 0) { // compare les deux voisins
      // echange les deux voisins
      T tmp = aList.get(j);
      aList.set(j, aList.get(j + 1));
      aList.set(j + 1, tmp);
    }}
  }}
```

exemples/chap3/BubbleSortExample.java

88

Quelques conventions

— Google Style Guides, Google, 2014.

  — pour de multiples langages

— Java code conventions, SUN, 1999.

  — simples et très utilisées

— Writing robust Java code, Scott W. Ambler, AmbySoft, 2007.

  — un article, une carte de référence et un livre

  — discussion sur différentes normes et leurs motivations plus de nombreux liens sur le sujet

  — très complet

— Java Programming Style Guidelines, GeoSoft, 2008.
  — chaque règle est commentée
  — un fichier de configuration pour `Checkstyle` est disponible
— Java Language Coding Guidelines, Cay Horstmann, 2004.
— Standards de programmation C++, Herb Sutter, Andrei Alexandrescu, Pearson Education, 2005 (UVSQ : 005.13C++ SUT).
  — aborde le style de codage et plus généralement les bonnes pratiques de programmation C++
— Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, Prentice Hall, 2008.

Thèmes abordés par les conventions

— Structure et contenu d'un fichier source
  — ex : *Files longer than 2000 lines are cumbersome and should be avoided.*
— Conventions de nommage
  — ex : *Class names should be nouns, in mixed case with the first letter of each internal word capitalized.*
— Format et emplacement des commentaires (cf. châpitre suivant)
  — ex : *Short comments can appear on a single line indented to the level of the code that follows.*
— Indentation des lignes
  — ex : *Four spaces should be used as the unit of indentation.*
— Format des déclarations, des instructions et emplacement des espaces
  — ex : *Each line should contain at most one statement.*
— Quelques bonnes pratiques
  — ex : *Don't make any instance or class variable public without good reason.*

89

4.2.2 Outils

Rôle des outils de vérification

— Réaliser un audit des fichiers sources
— Générer un rapport des violations des conventions de codage
— Sont généralement configurables pour différentes conventions

90

Quelques outils de vérification

— CheckStyle
  — fourni un rapport sur le projet analysé (commentaire javadoc mal formé, ...)
  — supporte par défaut la norme SUN
  — configurable (fichier XML)
  — intégration possible dans de nombreux outils (IDE, `ant`, ...)
— JCSC
  — fourni un rapport sur le projet analysé (commentaire javadoc mal formé, ...) ainsi que des statistiques
  — supporte par défaut la norme SUN
  — configurable (fichier XML + GUI)
  — intégration possible dans quelques outils

91

Exemple

Extraits des résultats avec `Checkstyle`

```
BubbleSortExample.java:10: L'utilisation des import.* est
                           prohibé - java.util.*.
BubbleSortExample.java:12: Commentaire javadoc manquant.
BubbleSortExample.java:12:1: Les classes utilitaires ne
                             doivent pas avoir de constructeur
                             par défaut ou public.
BubbleSortExample.java:13: La première ligne doit se terminer
                           avec un point.
BubbleSortExample.java:14:23: La variable 'MAX' devrait être
                              privée et avoir des accesseurs.
BubbleSortExample.java:14:23: Le nom 'MAX' n'est pas conforme
                              à l'expression '^[a-z][a-zA-Z0-9]*$'.
BubbleSortExample.java:14:29: '10' devrait être défini
                              comme une constante.
BubbleSortExample.java:25: La ligne excède 80 caractères.
BubbleSortExample.java:25:20: Balise javadoc @param manquante pour '<T>'.
BubbleSortExample.java:25:69: Le paramètre aList devrait être final.
BubbleSortExample.java:29:32: Il manque une espace avant '+'.
BubbleSortExample.java:29:33: Il manque une espace après '+'.
BubbleSortExample.java:42:4: Il manque une espace après 'for'.
BubbleSortExample.java:43:1: '{' devrait être sur la ligne précédente.
BubbleSortExample.java:48:1: '}' devrait être seul sur sa ligne.
BubbleSortExample.java:51:40: Les crochets du tableau ne sont pas
                              placés au bon endroit.
```

92

Exemple

Extrait des résultats avec `JCSC`

```
JavaEx.java:12:21:class Declaration JavaDoc does not provide the required
                  '@author' tag
JavaEx.java:22:7:class 'myWindow' name must match the following regexp
                 '[A-Z][\w\d]*'
JavaEx.java:25:12:public ctor declaration does not provide any JavaDoc
JavaEx.java:26:8:is a tab '\t' character which is not allowed
JavaEx.java:28:82:line is too long 0..80 characters are allowed
JavaEx.java:54:52:public method declaration JavaDoc does not provide
                  the required '@param' tag for all parameters

Metrics:
25:12:myWindow.CTOR():NCSS-16:CCN-1
54:52:myWindow.MyWindowAdapter.windowClosing():NCSS-2:CCN-1
77:42:JavaEx.main():NCSS-3:CCN-1
Total NCSS count      : 27
Total Methods count   : 3
Unit Test Class count : 0
Unit Tests count      : 0
```

— Vérification des règles pour JavaDoc, les conventions de nommage (classe dans l'exemple)

— Métriques

  — NCSS (Non Commenting Source Statements) : nombre de lignes de code réel

  — CCN (Cyclomatic Complexity Number) : métrique de McCabe, nombre de chemins dans le code pour chaque méthode (quand le flot de contrôle se partage, le compte est incrémenté)

93

Outils de reformatage

— Mettre en conformité le code source avec des conventions

— Concerne uniquement la mise en forme du code (indentation, espaces, ...)

— La plupart des IDE assurent également ce service

94

4.3 Documentation

4.3.1 Documenter un code source

Intérêt

— Générer automatiquement la documentation (dans diverses formats) du code source
— Permet de garder plus facilement la documentation en phase avec le code

Quelques outils
JavaDoc, Doxygen.

95

Qu'est-ce qu'une bonne documentation ?

— Un commentaire doit clarifier le code
  — la documentation du code doit permettre à une autre personne de mieux comprendre le code
— Évitez les commentaires décoratifs (bannières, ...)
  — ajoute peu de valeurs à la documentation
  — est une perte de temps
— Rédigez des commentaires simples et concis
— Écrivez la documentation avant d'écrire le code
  — permet de définir l'objectif en premier
— Documentez pourquoi les choses sont faites et pas simplement ce qui est fait
  — ne paraphrasez pas le code

Quelques outils
Idéalement, un code bien écrit doit se suffire à lui-même, i.e. doit se lire (et se comprendre) facilement sans commentaire.

96

Types de commentaires en Java

Documentation `/** ... */`
  — documente chaque élément du code (classes, attributs, méthodes, ...)
  — commentaires traités par un outil automatique (JavaDoc par exemple)

Style C `/* ... */`
  — pour mettre en commentaire une portion du code obsolète mais que l'on veut tout de même conserver

Mono-ligne `// ...`
  — pour les commentaires internes aux méthodes par exemple

97

4.3.2 JavaDoc

JavaDoc

— Outil standard fourni avec le JDK
— Génération automatique de la documentation au format HTML
— Documente les classes, les interfaces, les membres (données et fonctions), les modules et les fichiers sources
— Un exemple du résultat est donné par la documentation des API du JDK
— Le contenu de la sortie peut être paramétrée (membres privés ou non, ...)
— Un *Doclet* permet de modifier la sortie (XML, ...)

98

Commentaires pour la documentation

— `Javadoc` utilise un type de commentaire particulier (`/** ...*/`)

```
/**
 * This is the typical format of a simple documentation comment
 * that spans two lines.
 */
// ou
/** This comment takes up only one line. */
```

— Le *commentaire de documentation* doit être placé immédiatement avant l'entité qu'il documente.

— Un seul commentaire de documentation est reconnu par entité

— Un commentaire est composé d'une *description principale* suivie d'une *section de tags*

— Le texte du commentaire est écrit en HTML

— La première phrase de chaque commentaire est utilisée comme résumé

99

Tags `JavaDoc`

— Un *tag* est un mot reconnu et interprété par `Javadoc` dans un commentaire

— Un tag débute par le caractère `@`

— On différencie les *tags de bloc* (`@tag`) des *tags en ligne* (`{@tag}`)

— Tous les tags ne sont pas valides dans tous les contextes

Quelques tags (voir la liste complète) :
`@author` (noms des auteurs de l'entité), `{@code}` (formatage de code), `@deprecated` (précise que l'entité ne devrait plus être utilisée), `@exception` ou `@throws` (exceptions pouvant être lancées), `@param` (paramètre de méthode), `@return` (valeur de retour d'une méthode).

100

Principe

(voir figure 4.1).

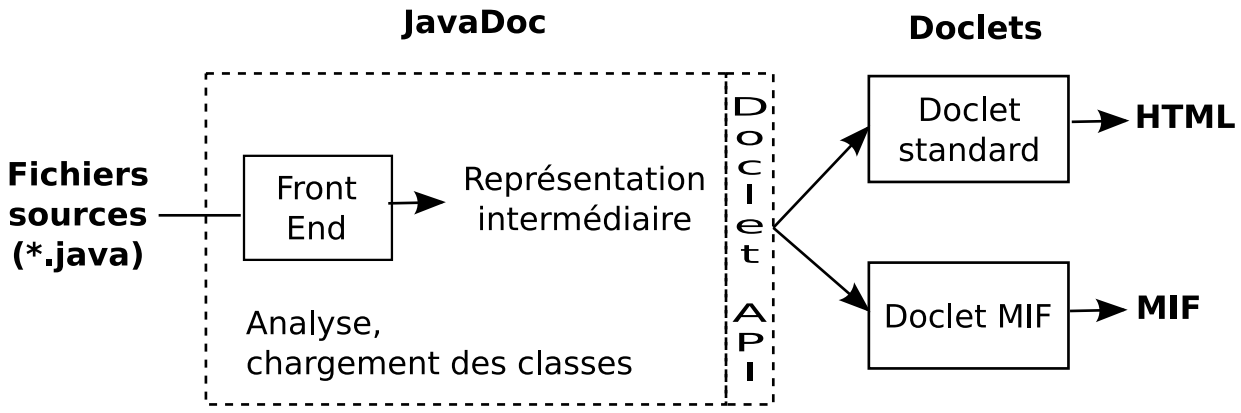


FIGURE 4.1 – Principe de Javadoc.

— MIF : format d'échange d'Adobe FrameMaker's

— Le doclet MIF permet la génération de PDF

101

Syntaxe

```
javadoc [ options ] [ packagenames ] [ sourcefilenames ]\
        [ -subpackages pkg1:pkg2:... ] [ @argfiles ]
```

Quelques options

`-public` traite uniquement les classes et les membres publiques

`-protected` traite les entités publiques et protégées (`-package` et `-private` existent aussi)

`-source` version des sources

`-sourcepath` chemin pour trouver les fichiers sources

`-d` répertoire de destination de la documentation

Exemple

Commentaire `JavaDoc` pour une méthode

```
/**
  * Returns an Image object that can then be painted on the screen.
  * The url argument must specify an absolute {@link URL}. The name
  * argument is a specifier that is relative to the url argument.
  * <p>
  * This method always returns immediately, whether or not the
  * image exists.
  *
  * @param  url  an absolute URL giving the base location of the image
  *         name the location of the image, relative to the url argument
  * @return      the image at the specified URL
  * @see         Image
  */
  public Image getImage(URL url, String name) {
```

— Le commentaire est en HTML (tag `<p>`)

— On utilise des tags particuliers (@param, @return, ...)

4.3.3 Doxygen

Doxygen

— Logiciel libre plus complet que JavaDoc

— Supporte C, C++, Java, C#, ...

— Génère du HTML, TEX, RTF, PDF, ...

— Permet aussi d'extraire la structure d'un code non documenté

Principe

(voir figure 4.2).

Mise en œuvre

1. Créer un fichier de configuration

```
doxygen -g <config-file>
```

2. Éditer le fichier de configuration
3. Éxécuter doxygen

```
doxygen <config-file>
```

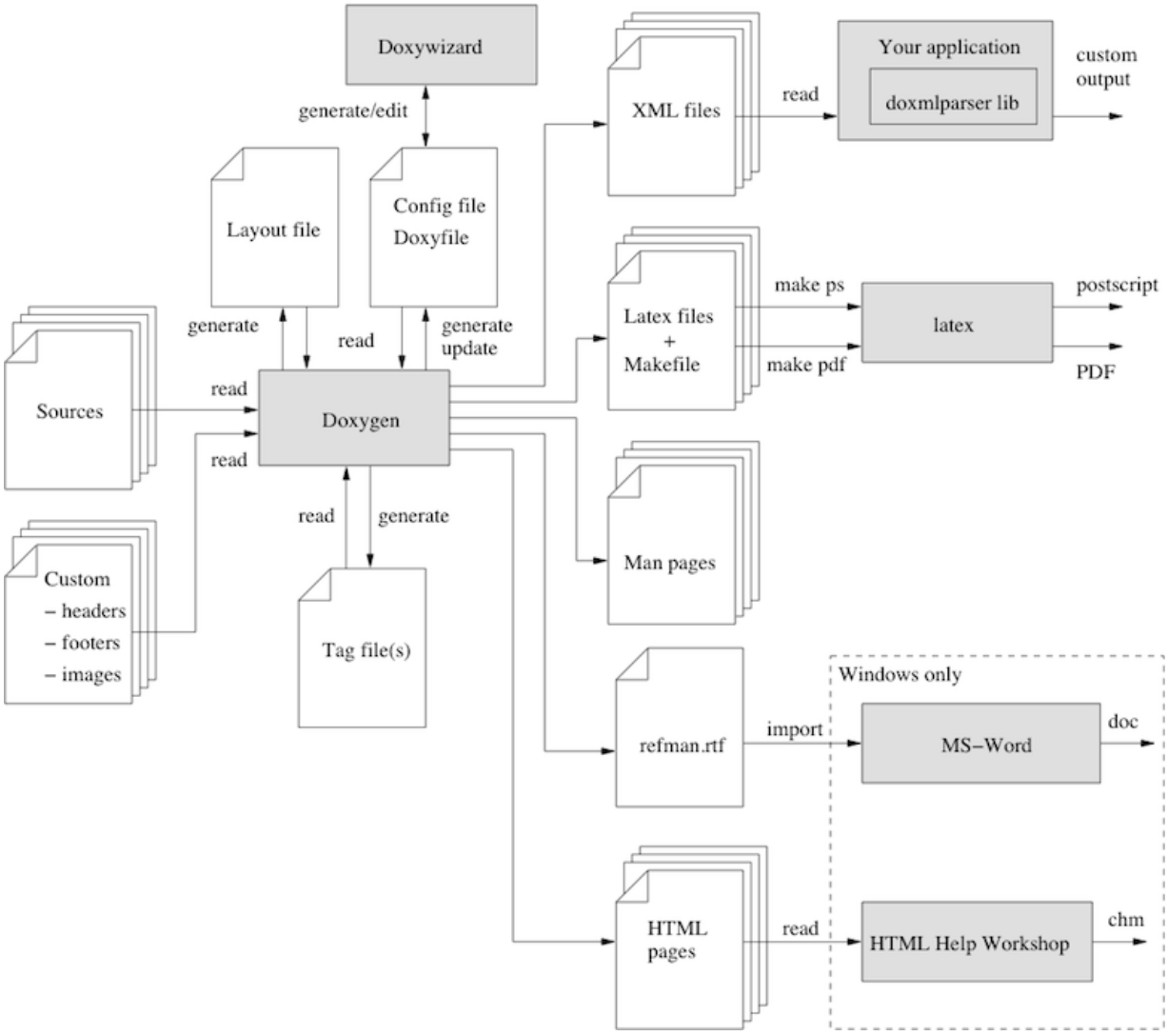


FIGURE 4.2 – Principe de Doxygen.

Exemple

Main Page | Related Pages | Modules | Data Structures | Files | Search for

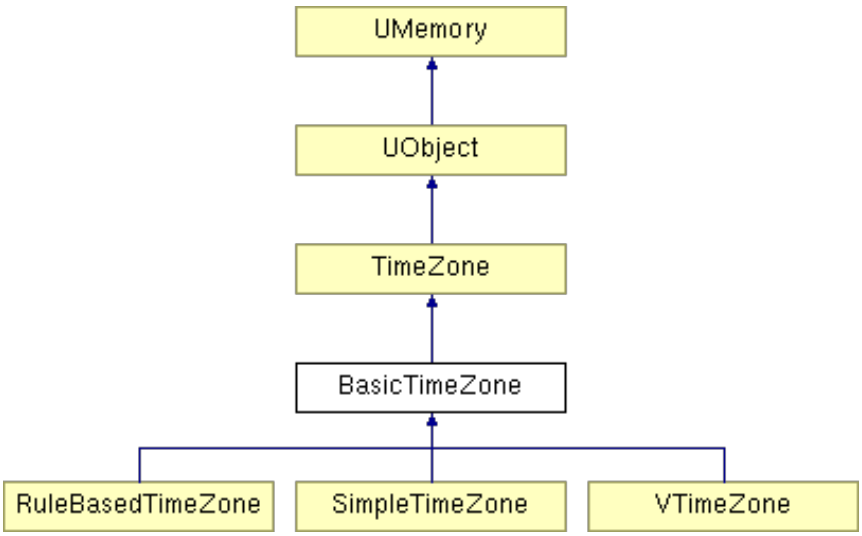
Alphabetical List | Data Structures | Class Hierarchy | Data Fields

BasicTimeZone Class Reference

BasicTimeZone is an abstract class extending **TimeZone**. More...

`#include <basictz.h>`

Inheritance diagram for BasicTimeZone:



Public Types

enum	{ **kStandard** = 0x01, **kDaylight** = 0x03, **kFormer** = 0x04, **kLatter** = 0x0C } *The time type option bit flags used by getOffsetFromLocal. More...*

Public Member Functions

virtual	**~BasicTimeZone** () *Destructor.*
virtual **UBool**	**getNextTransition** (**UDate** base, **UBool** inclusive, **TimeZoneTransition** &result)=0 *Gets the first time zone transition after the base time.*
virtual **UBool**	**getPreviousTransition** (**UDate** base, **UBool** inclusive, **TimeZoneTransition** &result)=0 *Gets the most recent time zone transition before the base time.*
virtual **UBool**	**hasEquivalentTransitions** (**BasicTimeZone** &tz, **UDate** start, **UDate** end, **UBool** ignoreDstAmount, **UErrorCode** &ec) *Checks if the time zone has equivalent transitions in the time range.*
virtual int32_t	**countTransitionRules** (**UErrorCode** &status)=0 *Returns the number of **TimeZoneRule**s which represents time transitions, for this time zone, that is, all **TimeZoneRule**s for this time zone except **InitialTimeZoneRule**.*
virtual void	**getTimeZoneRules** (const **InitialTimeZoneRule** *&initial, const **TimeZoneRule** *trsrules[], int32_t &trscount, **UErrorCode** &status)=0 *Gets the **InitialTimeZoneRule** and the set of **TimeZoneRule** which represent time transitions for this time zone.*
virtual void	**getSimpleRulesNear** (**UDate** date, **InitialTimeZoneRule** *&initial, **AnnualTimeZoneRule** *&std, **AnnualTimeZoneRule** *&dst, **UErrorCode** &status) *Gets the set of time zone rules valid at the specified time.*
virtual void	**getOffsetFromLocal** (**UDate** date, int32_t nonExistingTimeOpt, int32_t duplicatedTimeOpt, int32_t &rawOffset, int32_t &dstOffset, **UErrorCode** &status) *Get time zone offsets from local wall time.*

Protected Types

enum	{ **kStdDstMask** = kDaylight, **kFormerLatterMask** = kLatter } *The time type option bit masks used by getOffsetFromLocal. More...*

107

4.4 Analyse statique du code

4.4.1 Audit de code source

Audit de code source

— L'*audit* ou *revue* de code consiste à étudier attentivement un code source afin de détecter et de corriger des erreurs

— L'objectif est d'améliorer la qualité du logiciel et l'expérience des développeurs

— Peut prendre différentes formes

Inspection/**Fagan inspection** est un processus formel pour l'audit de code

« par dessus l'épaule » un développeur suit en temps réel ce qu'un autre écrit

par email un email est envoyé au relecteur lors des validations de version

programmation par binôme deux développeurs travaillent de concert et échange leur rôle régulièrement (vient de eXtreme Programming (XP))

assisté par un outil s'appuie sur des outils pour une analyse systématique

108

Analyse statique du code

— L'*analyse statique* permet d'obtenir des informations sur un programme sans l'exécuter
— Elle est un bon complément aux tests
— En général, elle n'a pas connaissance de ce que le programme doit faire (recherche de motifs généraux)
— Certaines erreurs « d'inattention » se reproduisent fréquemment dans un fichier source (`;` après un `for`, ...)
— La plupart de ces erreurs peuvent être recherchées de façon systématique
— Des outils proposent un moteur ainsi qu'un ensemble de règles permettant de trouver ce type d'erreurs dans un fichier source
— L'ensemble de règles peut éventuellement être modifiable

Quelques bogues courants

— Boucle récursive infinie

```
public MaClasse() {
  MaClasse m = new MaClasse();
}
```

— Déréférencement d'une référence null

```
if (c == null && c.uneMethode()) //...
```

— Auto affectation d'attribut

```
public MaClasse(String uneChaine) {
  this.chaine = chaine;
}
```

— Valeur de retour ignorée

```
String nom = //...
nom.replace('/', '.');
```


Catégories de bogues

Correction le code ne fait clairement pas ce qui est attendu
— *déréférencement d'une référence null*

Mauvaise pratique le code ne respecte pas les bonnes pratiques
— *redéfinition d'`equals` sans `hashCode`, comparaison de chaîne avec `==`*

Problème de sécurité le code est vulnérable à un usage malveillant
— *injection SQL*

Code suspect le code utilise des pratiques non usuelles

Performance le code est inefficace

Correction multithread il y a un problème de correction en environnement multithread

Mise en œuvre de l'analyse statique

- Intégration au processus de développement
  - *intégration à l'IDE, exécution comme les tests unitaires, ...*
- Réglage de l'outil utilisé
  - *éviter les faux positifs, paramétrer le niveau de détail, ...*
- Réfléchir à la prise de décision
  - *consultation des rapports, processus pour la correction du bogue, ne pas corriger le bogue, ...*

112

Quelques outils

- FindBugs
  - recherche les bogues dans les programmes Java
  - basé sur des *modèles de bogues* (*bug patterns*) (idiome de code considéré en général comme une erreur)
  - analyse le *bytecode*
  - supporte un système de *plugins*
- Pmd
  - analyse le code source
  - recherche les bogues potentiels, les expressions trop complexes, le code dupliqué, ...
  - sortie au format texte, XML, HTML
  - permet de préciser l'ensemble de règles à utiliser voire de définir des règles personnalisées

113

4.4.2 FindBugs

Exécution

- avec l'interface graphique

```
java -jar $FINDBUGS_HOME/lib/findbugs.jar
```

- en ligne de commande

```
java -jar $FINDBUGS_HOME/lib/findbugs.jar -textui
```

- intégré au build
- comme extension pour l'IDE

114

Résultats de FindBugs sur les sources du JDK

(voir figure 4.5).
(voir figure 4.5).
(voir figure 4.5).

115

FindBugs (1.2.1-dev-20070506) Analysis for jdk1.7.0-b12

Bug Summary | Analysis Information | List bugs by bug category | List bugs by package

FindBugs Analysis generated at: Sun, 6 May 2007 03:12:12 -0400

Package	Code Size	Bugs	Bugs p1	Bugs p2	Bugs p3	Bugs Exp.	Ratio
Overall (736 packages), (16445 classes)	963957	3901	259	3642			
java.awt	22029	90	10	80			
java.awt.color	942	1		1			
java.awt.event	1525	4	1	3			
java.io	8669	23	3	20			
java.lang	9085	46	3	43			
java.math	3151	7		7			
java.net	7955	53	2	51			
java.nio	6188	13	11	2			
java.util	18749	31		31			
javax.swing	35662	205	16	189			
javax.swing.colorchooser	1275	13		13			
javax.swing.event	617	6		6			
javax.swing.filechooser	342	2		2			
javax.swing.plaf	340	2		2			
javax.swing.plaf.basic	24666	104	4	100			
javax.swing.plaf.metal	8472	59	9	50			
javax.swing.plaf.synth	9593	32	2	30			
javax.swing.table	1363	9	1	8			
javax.swing.text	15353	90	7	83			
javax.swing.text.html	12170	68	8	60			
javax.swing.text.html.parser	1873	35		35			
javax.swing.text.rtf	1906	22		22			
javax.swing.tree	3390	16	1	15			
javax.swing.undo	407	1		1			

FIGURE 4.3 – Vue d'ensemble du rapport.

FindBugs (1.2.1-dev-20070506) Analysis for jdk1.7.0-b12

Bug Summary | Analysis Information | List bugs by bug category | List bugs by package

FindBugs Analysis generated at: Sun, 6 May 2007 03:12:12 -0400

Package	Code Size	Bugs	Bugs p1	Bugs p2	Bugs p3	Bugs Exp.	Ratio
Overall (736 packages), (16445 classes)	963957	3901	259	3642			
java.awt	22029	90	10	80			
java.awt.color	942	1		1			
java.awt.event	1525	4	1	3			
java.io	8669	23	3	20			
java.lang	9085	46	3	43			
java.math	3151	7		7			
java.net	7955	53	2	51			
java.nio	6188	13	11	2			
java.util	18749	31		31			
javax.swing	35662	205	16	189			
javax.swing.colorchooser	1275	13		13			
javax.swing.event	617	6		6			
javax.swing.filechooser	342	2		2			
javax.swing.plaf	340	2		2			
javax.swing.plaf.basic	24666	104	4	100			
javax.swing.plaf.metal	8472	59	9	50			
javax.swing.plaf.synth	9593	32	2	30			
javax.swing.table	1363	9	1	8			
javax.swing.text	15353	90	7	83			
javax.swing.text.html	12170	68	8	60			
javax.swing.text.html.parser	1873	35		35			
javax.swing.text.rtf	1906	22		22			
javax.swing.tree	3390	16	1	15			
javax.swing.undo	407	1		1			

FIGURE 4.4 – Vue par classes.

FindBugs (1.2.1-dev-20070506) Analysis for jdk1.7.0-b12

Bug Summary | Analysis Information | List bugs by bug category | List bugs by package

FindBugs Analysis generated at: Sun, 6 May 2007 03:12:12 -0400

Package	Code Size	Bugs	Bugs p1	Bugs p2	Bugs p3	Bugs Exp.	Ratio
Overall (736 packages), (16445 classes)	963957	3901	259	3642			
java.awt	22029	90	10	80			
java.awt.color	942	1		1			
java.awt.event	1525	4	1	3			
java.io	8669	23	3	20			
java.lang	9085	46	3	43			
java.math	3151	7		7			
java.net	7955	53	2	51			
java.nio	6188	13	11	2			
java.util	18749	31		31			
javax.swing	35662	205	16	189			
javax.swing.colorchooser	1275	13		13			
javax.swing.event	617	6		6			
javax.swing.filechooser	342	2		2			
javax.swing.plaf	340	2		2			
javax.swing.plaf.basic	24666	104	4	100			
javax.swing.plaf.metal	8472	59	9	50			
javax.swing.plaf.synth	9593	32	2	30			
javax.swing.table	1363	9	1	8			
javax.swing.text	15353	90	7	83			
javax.swing.text.html	12170	68	8	60			
javax.swing.text.html.parser	1873	35		35			
javax.swing.text.rtf	1906	22		22			
javax.swing.tree	3390	16	1	15			
javax.swing.undo	407	1		1			

FIGURE 4.5 – Vue par type de bogues.

4.4.3 PMD

Exécution

— en ligne de commande

```
$PMD_HOME/bin/pmd.sh BubbleSortExample.java\
text\
basic,imports,unusedcode
```

— intégré au build

— comme extension pour l'IDE

116

Exemple

Extrait du résultat de PMD sur un projet SourceForge
(voir figure 4.6).

PMD report

Problems found

#	File	Line	Problem
1	de/hunsicker/io/FileBackup.java	353	Avoid unused private methods such as 'getVersionName(File,int)'
2	de/hunsicker/jalopy/Jalopy.java	1511	Avoid unused private methods such as 'hasOutput()'
3	de/hunsicker/jalopy/debug/ASTFrame.java	62	Avoid unused local variables such as 'listener'
4	de/hunsicker/jalopy/language/CodeInspector.java	93	Avoid unused private fields such as 'STR_ADHERE_TO_NAMING_CONVENTION'
5	de/hunsicker/jalopy/language/CodeInspector.java	191	Avoid unused private fields such as '_file'
6	de/hunsicker/jalopy/language/CodeInspector.java	816	Avoid unused local variables such as 'hashCodeNode'
7	de/hunsicker/jalopy/language/CodeInspector.java	1009	Avoid unused formal parameters such as 'method'
8	de/hunsicker/jalopy/language/DeclarationType.java	143	Avoid unused private fields such as '_key'
9	de/hunsicker/jalopy/language/ImportTransformation.java	75	Avoid unused private fields such as 'EMPTY_STRING_ARRAY'
10	de/hunsicker/jalopy/language/ImportTransformation.java	560	Avoid unused private methods such as 'getPossibleTypes(List)'
11	de/hunsicker/jalopy/language/ImportTransformation.java	639	Avoid unused formal parameters such as 'source'
12	de/hunsicker/jalopy/language/ImportTransformation.java	639	Avoid unused formal parameters such as 'target'
13	de/hunsicker/jalopy/language/ImportTransformation.java	686	Avoid unused private methods such as 'collapse()'
14	de/hunsicker/jalopy/language/ImportTransformation.java	963	Avoid unused local variables such as 'settings'
15	de/hunsicker/jalopy/language/ImportTransformation.java	1081	Avoid unused private methods such as 'expand()'
16	de/hunsicker/jalopy/language/JavaLexer.java	495	Avoid unused local variables such as '_saveIndex'
327	de/hunsicker/jalopy/swing/syntax/SyntaxUtilities.java	43	Avoid unused private fields such as 'COLORS'
328	de/hunsicker/jalopy/swing/syntax/SyntaxView.java	396	Avoid unused private methods such as 'drawLineNumber(Graphics,int,int,int)'

FIGURE 4.6 – Exemple de résultat pour PMD.

117

4.5 Bilan

Agrégation des métriques

— La difficulté est de visualiser l'ensemble des rapports produits

— Il peut être utile d'agréger les différentes mesures pour générer des indicateurs synthétiques

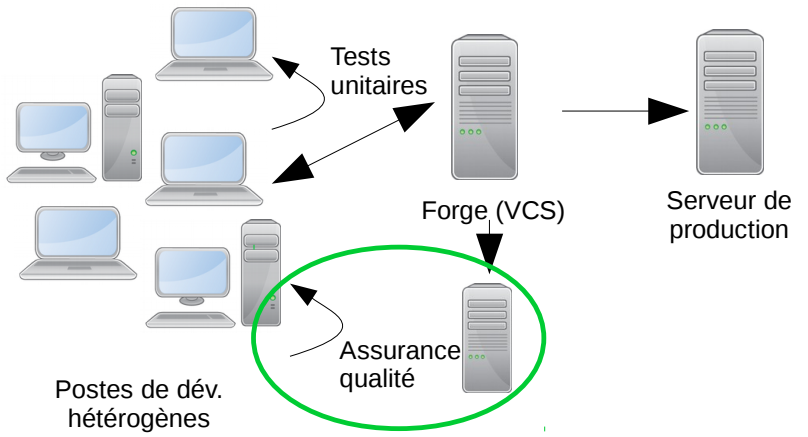
— Ces indicateurs doivent ensuite être disponibles pour l'ensemble des membres du projet

— Les réactions face à ces rapports doivent aussi être anticipées

Outils

SONARQUBE

118



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

119

Bilan

Apports

- Le code est plus lisible et donc plus facile à maintenir
- Tous les participants d'un projet partagent ces règles

Limites

- De nombreux outils doivent être appliqués sur le code source
- L'ensemble des développeurs doit utiliser les différents outils
  - automatisation des tâches pour standardiser le *build*

120

4.6 Exercices

CONTRAINTES

- Vous pouvez travailler dans l'IDE ou en ligne de commande.
- Dans l'IDE, vous vous assurerez que les plugins pour les outils utilisés sont correctement installés.

Exercice 4.1 (Documentation avec JAVADOC)
Pour cet exercice, vous travaillerez sur les sources du projet `Compte` réalisé dans le TD précédent.

1. Documentez la classe `Compte` à l'aide de commentaires Javadoc respectant les conventions de codage SUN ;
2. Générez la Javadoc en plaçant les fichiers résultants dans `docs/apidocs` ;
3. Vérifiez le résultat avec un navigateur.

Exercice 4.2 (Audit des conventions de codage avec CHECKSTYLE)
Pour cet exercice, vous travaillerez sur les sources du projet SimpleShapes.

1. Activez Checkstyle avec les conventions par défaut pour le projet ;
2. Consultez la liste des avertissements dans l'éditeur puis dans la vue Checkstyle ;
3. Remplacez le fichier de configuration par les conventions Google puis activez-le sur le projet ;
4. Créez une configuration personnalisée :
   - copiez le fichier de configuration Google,
   - désactivez la vérification des tabulations,

— sélectionnez cette configuration pour le projet.

Exercice 4.3 (Recherche de bogues avec FINDBUGS)
Dans cet exercice, vous travaillerez sur le projet SimpleShapes.

1. Activez Findbugs pour qu'il s'exécute automatiquement ;
2. Affectez `null` à la variable `shapes` au début de la méthode `void run(String[] args)` et réexécutez Finbugs ;
3. Quel problème est détecté ?

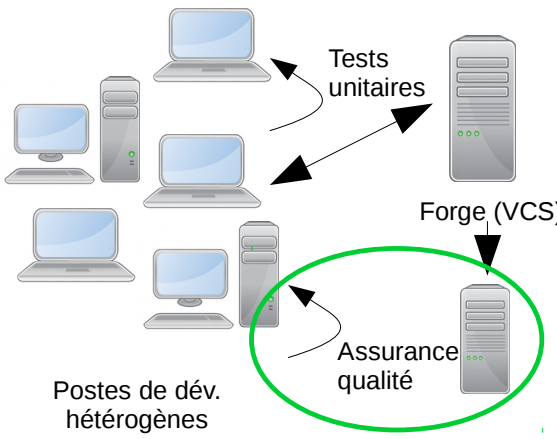
Chapitre 5

Construction d'un projet

Sommaire

5.1 Introduction 47
5.2 GNU Make 48
5.3 Apache Maven 51
5.3.1 Généralités 51
5.3.2 POM 52
5.3.3 Cycle de vie 53
5.3.4 Plugin et but 54
5.3.5 Référentiel 54
5.4 Gradle 54
5.5 Apache Ant 56
5.6 Bilan 60
5.7 Exercices 60

5.1 Introduction



- Communication/tâches
- Fusion/versions
- →**Build**←
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

121

Intérêt

- La gestion de la compilation (*Build automation*) consiste à automatiser les tâches répétitives des développeurs
  - compilation (mode normal, mode débogage, ...)
  - génération de la version de distribution

  - génération de la documentation et des notes de version
  - lancement des tests
  - déploiement
- Ces tâches sont alors réalisées plus efficacement et moins sujettes aux erreurs
  - évite les fastidieuses lignes de commande
  - permet une compilation « intelligente »
  - rend le processus reproductible
- Peut être déclenché

  à la demande l'utilisateur exécute un script

  par un ordonnanceur exécuté à un instant donné

  par un événement provoqué par un événement particulier

122

Quelques outils

Dans un environnement Java
Apache Maven, Gradle, Apache Ant, sbt

Dans un environnement C/C++
CMake, Make

Dans d'autres contextes
Waf, SCons, Rake

123

5.2 GNU Make

Introduction

- *Make* permet d'automatiser les tâches répétitives de compilation, ...
- Il détermine automatiquement les fichiers à mettre à jour et l'ordre des mises à jour ⇒ n'est mis à jour que ce qui est nécessaire
- Il exécute des commandes *shell* ⇒ n'est pas spécifique à un langage de programmation mais au système d'exploitation
- La configuration est fournie dans un *Makefile*

124

Exécution de `make`

```
make [-B] [-d] [-f makefile] [-n] [VAR=val] [but1 but2 ...]
```

- `-B` reconstruit tout
- `-d` affiche des informations de débogage
- `-f` précise le *makefile* à considérer (`Makefile` par défaut)
- `-n` affiche les commandes à exécuter mais sans les exécuter
- Des variables peuvent être définies en ligne de commande
- Si le but n'est pas précisé, la première cible du `Makefile` est choisie

125

Makefile

— Un *makefile* est un fichier texte regroupant un ensemble de *règles* et de *déclarations*
— Une règle comporte une *cible*, des *dépendances* et une liste de *commandes*
— La cible est en général un nom de fichier à construire ou une action (*phony target*)
— Les dépendances indiquent de quoi dépend la cible (fichier ou autre cible)
— Les commandes précisent à `make` comment obtenir la cible à partir des dépendances
— *Chaque commande doit être précédée du caractère* ***tabulation***
— Un commentaire débute par `#`

126

Exemple

Un `Makefile` *simple*

```
objects = main.o kbd.o command.o display.o \
          insert.o

edit : $(objects)
  cc -o edit $(objects)

main.o : main.c defs.h
  cc -c main.c

kbd.o : kbd.c defs.h command.h
  cc -c kbd.c

command.o : command.c defs.h command.h
  cc -c command.c

display.o : display.c defs.h buffer.h
  cc -c display.c

insert.o : insert.c defs.h buffer.h
  cc -c insert.c

clean :
  rm edit $(objects)
```

Listing 5.1 – Un `Makefile` simple

127

Compléments sur les règles

— Certaines règles sont déjà connues de `make` (*règles implicites*)
— Les règles implicites peuvent être adaptées à l'aide de variables (`CFLAGS` par exemple)
— Il est possible de définir des règles implicites (*règle motif*)

```
%.o : %.cc
    $(CXX) $(CPPFLAGS) $(CXXFLAGS) -c $<
```

— Certaines variables ont une signification particulière (*variables automatiques*)
  $@ nom de la cible
  $< nom de la première dépendance
  $? noms des dépendances plus récentes que la cible
  $^ noms de toutes les dépendances
— Les cibles qui ne représentent pas des fichiers sont déclarées avec la cible `.PHONY`

128

Exemple

Le `Makefile` «corrigé»

```
objects = main.o kbd.o command.o display.o \
          insert.o

edit : $(objects)
  cc -o edit $(objects)

main.o : main.c defs.h
  cc -c main.c

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h

.PHONY : clean
clean :
  -rm edit $(objects)
```

Listing 5.2 – Un `Makefile` amélioré

— Pas de commandes de compilation (règle implicite)
— Pas de dépendances avec le fichier `.c` (règle implicite)
— `.PHONY` précise les cibles qui ne sont pas des fichiers
— Le moins « - » devant rm permet d'ignorer les erreurs si certains fichiers n'existent pas

129

Cibles standards (par convention)

`all` Fabrique les cibles de premier niveau

`clean` Supprime les fichiers temporaires

`distclean` Supprime tous les fichiers produits par `make`

`install` Installe le programme sur le système

130

Exemple I

Un `Makefile` plus sophistiqué pour Java

```
# Les constantes pour les repertoires
CLASS_DIR = ./class
DOC_DIR = ./doc

# Les constantes pour les fichiers
SOURCE_FILES = $(wildcard *.java)
CLASS_FILES = $(SOURCE_FILES:%.java=$(CLASS_DIR)/%.class)

# Les outils
RM = rm
MKDIR = mkdir

# Le compilateur Java (doit etre accessible dans le PATH)
JAVAC = javac
JAVAC_OPTIONS = -d $(CLASS_DIR) -deprecation # Ajouter -g pour le deboguage

# JavaDoc
JAVADOC = javadoc
JAVADOC_OPTIONS = -d $(DOC_DIR) -author -package -use -version
```

Listing 5.3 – Un `Makefile` pour Java (1ère partie)

131

Exemple II

Un `Makefile` plus sophistiqué pour Java

```
# Les regles
.PHONY : all build javadoc clean
all : build

$(CLASS_DIR) :
  -$(MKDIR) $@

$(DOC_DIR) :
  -$(MKDIR) $@

# .java -> .class
$(CLASS_DIR)/%.class : %.java
  $(JAVAC) $(JAVAC_OPTIONS) $<
```

Listing 5.4 – Un `Makefile` pour Java (2ème partie)

132

Exemple III
Un `Makefile` plus sophistiqué pour Java

```
# Invocation avec make ou make build
build : $(CLASS_DIR) $(CLASS_FILES)

javadoc : $(DOC_DIR)
  $(JAVADOC) $(JAVADOC_OPTIONS) $(SOURCE_FILES)

clean :
  -$(RM) $(CLASS_FILES)
```

Listing 5.5 – Un `Makefile` pour Java (3ème partie)

133

Bilan

Concepts importants

description langage spécialisé (*DSL*) pour décrire le build

cible/but le résultat à obtenir

règle comment construire une cible ?

dépendance que faut-il pour réaliser le but ?

Limites

— Le langage de description est limité (expressivité, syntaxe)

— Les commandes sont liées au *shell*

134

5.3 Apache Maven

5.3.1 Généralités

Introduction

— Maven est un projet open source de la fondation Apache

— C'est un outil de *gestion et de compréhension de projet logiciel*

— Il fournit une aide pour la gestion de la compilation, de la documentation, des dépendances, ...

— Il est basé sur un ensemble de conventions et de bonnes pratiques pour simplifier la gestion d'un projet

— Maven utilise une approche déclarative décrivant le projet plutôt qu'une approche par tâche comme `ant` et `make`

135

Principaux concepts

POM Le *modèle objet du projet* (*project object model* ou *POM*) est une description du projet

Cycle de vie Le *cycle de vie* (*build lifecycle*) définit précisément les processus de compilation

Hiérarchie standard L'arborescence de répertoires d'un projet respecte un ensemble de conventions

Plugin Un *plugin* est un programme exécuté par maven pour réaliser une tâche

But Un *but* (*goal*) est l'une des tâches proposées par un plugin

Référentiel Un *référentiel* (*repository*) contient les objets générés et les dépendances

136

5.3.2 POM

Modèle objet du projet

— Le *modèle objet du projet* est une description détaillée du projet

— Il contient les versions et les configurations, les dépendances, ...

— Il est contenu dans un fichier XML nommé `pom.xml` placé dans le répertoire de base du projet

137

Exemple

Un exemple de POM (fichier `pom.xml`)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.8.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Listing 5.6 – Un POM simple

138

Eléments de base du POM

project est la racine du document `pom.xml`

modelVersion indique la version du POM

groupId identifie de façon unique le groupe qui a créé le projet

artifactId est le nom du principal objet généré du projet

packaging est le type de distribution

version précise la version de l'objet généré

name est le nom du projet (pour l'affichage)

url donne l'URL du site web du projet

description est une description du projet

139

Coordonnées d'un projet

— Les éléments `groupId`, `artifactId`, `packaging` et `version` représentent les *coordonnées* d'un projet

— C'est un moyen de faire référence à un projet parmi l'ensemble des projets

— Les éléments `groupId`, `artifactId` et `version` forment un identifiant unique pour un projet (aucun autre projet ne peut avoir les trois mêmes éléments)

— Cet identifiant permet de faire référence à un autre projet (dépendance par exemple)

```
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>4.12</version>
```

140

Archétype

— Un *archétype* (*archetype*) permet de créer un squelette de projet Maven
— Il permet de créer simplement une structure de projet conforme aux conventions d'une organisation
— Le plugin `archetype` fournit les fonctionnalités pour les archétypes
— Différents squelettes sont disponibles

141

Création du POM à partir d'un modèle

```
# Version de maven installee
mvn -version
# Generation du projet (but generate du plugin archetype)
mvn archetype:generate \
    -DarchetypeGroupId=org.apache.maven.archetypes \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DinteractiveMode=false \
    -DgroupId=fr.uvsq.tod.monprojet \
    -DartifactId=MonProjet \
    -Dversion=1.0-SNAPSHOT
```

142

Principaux répertoires d'un projet

pom.xml le POM du projet
src les sources du projet
    main les fichiers de l'application
        java fichiers java
        resources les ressources
        config fichiers de configuration
    test les tests
        java les sources java des tests
        resources les ressources
    site le site du projet
target ce qui est généré
    classes le résultat de la compilation de l'application
    test-classes le résultat de la compilation des tests

143

5.3.3 Cycle de vie

Cycle de vie

— Dans Maven, le processus de production d'un objet est clairement défini
— Le cycle de vie est formé de *phases* : *validate*, *compile*, *test*, *package*, *install*
— Une phase représente une étape du processus
— Les différentes phases sont exécutées séquentiellement pour réaliser le cycle de vie
— Une phase est associée à un ou plusieurs *buts* (tâches spécifiques)

144

Invocation de maven

```
# Generer la distribution du projet (phase package du cycle default)
mvn package
# Supprimer les fichiers de compilation (phase clean du cycle clean)
mvn clean
# Generer un site web (phase site du cycle site)
mvn site
```

145

5.3.4 Plugin et but

Plugin et but

— Toute tâche sous Maven est réalisée par un plugin
  — exemple : `clean`, `compiler`, `jar`, `javadoc`, ...
— Un plugin fournit un ensemble de buts à Maven
  — exemple : `compiler:compile`, `compiler:testCompile`, `jar:jar`, `javadoc:javadoc`, ...

146

5.3.5 Référentiel

Référentiel

— Maven s'appuie sur un référentiel pour récupérer les plugins et les objets nécessaires
— Par défaut, Maven utilise un référentiel distant (`http://repo1.maven.org/maven2`)
— La structure d'un référentiel reflète le système de coordonnées afin de faciliter la localisation des objets
— Les objets téléchargés sont stockés dans un référentiel local (`~/.m2/repository`)
— Le résultat de l'installation d'un projet (`mvn install`) copie les objets résultants dans le référentiel local

147

Rechercher dans un référentiel

— Le nombre de bibliothèques dans les référentiels publiques est très important ⇒ des outils de recherche sont nécessaires
— Des moteurs de recherche spécialisés existent :
  — search.maven le moteur officiel de *maven central*
  — MVNRepository le plus utilisé (recherche dans plusieurs référentiels)
  — JavaLibs recherche dans plusieurs référentiels

148

5.4 Gradle

Gradle

— Gradle est un système de build basé sur groovy
— La description du projet se trouve dans le fichier `build.gradle`
— Le fichier `build.gradle` utilise un DSL basé sur groovy
— Gradle gère les dépendances et supporte les dépôts maven et Ivy
— Un build comporte un ou plusieurs *projets*
— Chaque projet se compose de *tâches*

Un ***`build.gradle`*** *simple*

`build.gradle`

```
task hello << {println 'Hello Gradle'}
```

Bash

```
$ gradle hello
Hello Gradle
```

149

Dépendances entre tâches

— Le paramètre `dependsOn` d'une tâche précise de quelle autre tâche elle dépend

`other` dépend de `compile`
`build.gradle`

```
task compile << {
  println 'Executing the compile task'
}

task other(dependsOn: 'compile') << {
  println "I'm not a default task!"
}
```

150

Dépendances avec des bibliothèques

— Le bloc `dependencies` précise les bibliothèques nécessaires au projet

Expression des dépendances
`build.gradle`

```
dependencies {
  compile group: 'commons-collections', name: 'commons-collections', version: '3.2'
  testCompile group: 'junit', name: 'junit', version: '4.+'
  runtime files('libs/library1.jar', 'libs/library2.jar')
  runtime fileTree(dir: 'libs', include: '*.jar')
}
```

151

Dépôt pour les binaires

— Le bloc `repositories` précise les dépôts où chercher les bibliothèques

Définition des dépôts
`build.gradle`

```
repositories {
  mavenCentral()
  mavenLocal()
  maven { url "http://repo.uvsq.fr/maven2" }
  ivy { url "http://repo.uvsq.fr/repo" }
}
```

152

Plugins

— Gradle fonctionne grâce à un ensemble de plugins

— Un plugin est intégré au build avec `apply plugin:`

Intégration de plugins
`build.gradle` pour Java

```
apply plugin: 'java'
```

`build.gradle` pour Groovy

```
apply plugin: 'groovy'

repositories {
  mavenCentral()
}

dependencies {
  compile 'org.codehaus.groovy:groovy-all:2.4.4'
  testCompile 'junit:junit:4.+'
}
```

153

5.5 Apache Ant

Introduction

— `ant` est un outil de gestion de la compilation en environnement Java
— Implémenté en Java
— Logiciel libre maintenu par *Apache*
— Permet d'automatiser un grand nombre de tâches
  — plus de 80 tâches principales
  — plus de 60 tâches optionnelles
  — plus de 100 extensions
  — possibilité d'écrire ses propres extensions
— Exécute des programmes Java (fonctionnalités étendues par des classes) ⇒ non spécifique à un SE, plus contraint
— Fichier de configuration en XML ⇒ plus facile à écrire
— Indépendant de la plate-forme (peut appeler des commandes systèmes mais on perd l'indépendance)

154

Exécution de Ant

```
ant [-p] [-Dnom=valeur] [cible1 cible2 ...]
```

— sans argument, lance la cible par défaut du fichier `build.xml`
— avec des cibles, exécute les tâches associées aux cibles
— `-p` affiche les cibles possibles et leurs descriptions
— `-D` définit des propriétés

155

Fichier de configuration pour Ant

— Le fichier de configuration est en XML (`build.xml` par défaut)
— Un fichier contient un *projet*
— Chaque projet est composé de cibles (*targets*)
— Chaque cible est un ensemble de *tâches* (*tasks*)

156

Projet

— Le projet (`<project>`) est l'élément de premier niveau du script Ant
— Il possède trois attributs optionnels :
  name nom du projet
  default la cible par défaut
  basedir le répertoire de base du projet

157

Cible

— Un élément cible (`<target>`) est un ensemble de *tâches* (*tasks*)
— Une cible possède un nom (attribut `name`)
— Une cible peut dépendre d'autres cibles (attribut `depends`)
— Une cible peut être exécutée de façon conditionnelle (attributs `if` ou `unless`)
— L'attribut `description` fournit une aide pour la cible

```
<target name="A"/>
<target name="B" depends="A"/>
<target name="C" depends="A"/>
<target name="D" depends="B,C"/>
```

158

Tâche

— Une tâche est une morceau de code à exécuter (programme Java)
— Une tâche peut être paramétrée par des attributs ou des sous-éléments
— La valeur des paramètres peut utiliser des *propriétés*

159

Propriété

— Une propriété possède un nom et une valeur
— Peut être utilisée comme valeur d'un attribut de tâche (encadré par `${` et `}`)
— Les propriétés sont immuables
  — une fois fixée, la valeur d'une propriété ne change pas
— Définition d'une propriété
  1. sur la ligne de commande `-Dnom=valeur`
  2. sous-élément `<property>` du projet
  3. sous-élément `<property>` d'une tâche
— Permet d'accéder aux propriétés systèmes

160

Exemple

Définir des propriétés

— Définir `source.dir` à la valeur `src`

```
<property name="source.dir" value="src"/>
<!--
L'attribut "location" définit la propriété comme étant
le chemin absolu vers un fichier.
-->
<property name="source.dir" location="src"/>
```

— Lire un ensemble de propriétés à partir d'un fichier

```
<property file="unfichier.properties"/>
```

— Lire un ensemble de propriétés à partir d'une ressource du *classpath*

```
<property resource="chemin.properties"/>
```

— Lire l'ensemble des variables d'environnement et les rendre accessibles comme propriétés (préfixées par `env`)

```
<property environment="env"/>
```

161

Répertoires

- Les éléments `<path>` et `classpath` permettent de définir des chemins pour les cibles
- Le sous-élément `<pathelement` représente une partie d'un chemin
- L'attribut `location` spécifie un unique fichier ou répertoire relativement au répertoire du projet ou comme chemin absolu
- L'attribut `path` peut contenir un ensemble de répertoires séparés par « :» ou « ;»
- Un chemin peut être nommé avec l'attribut `id` et réutilisé par la suite (attribut `refid`)

```
<classpath>
    <pathelement path="${classpath}"/>
    <pathelement location="lib/helper.jar"/>
</classpath>
```

162

Collections de ressources

- Une collection de ressources permet de regrouper ensemble des ressources (fichiers, propriétés, ...)
- Un élément `<fileset>` représente un ensemble de fichiers
- Un élément `<dirset>` regroupe des répertoires
- Les deux éléments précédents utilisent des filtres pour sélectionner les ressources
- Un élément `<filelist>` est un ensemble de fichiers explicitement nommés (attribut `files`)

163

Exemple

Utilisation des ressources

```
<classpath>
    <pathelement path="${classpath}"/>
    <fileset dir="lib">
        <include name="**/*.jar"/>
    </fileset>
    <pathelement location="classes"/>
    <dirset dir="${build.dir}">
        <include name="apps/**/classes"/>
        <exclude name="apps/**/*Test*"/>
    </dirset>
    <filelist refid="third-party_jars"/>
</classpath>
```

164

Principales tâches

- Compilation
  - `<javac>`, `<apt>` (processeur d'annotations), `<rmic>` (compilateur RMI)
- Manipulation d'archives
  - `<zip>`, `<unzip>`, `<jar>`, `<unjar>`, `<war>`, `<unwar>`, `<ear>`
- Manipulation de fichiers
  - `<copy>`, `<concat>`, `<delete>`, `<filter>`, `<fixcrlf>`, `<get>`, `<mkdir>`, `<move>`, `<replace>`, `<sync>`, `<tempfile>`, `<touch>`
- Tests
  - `<junit>`, `<junitreport>`
- Divers
  - `<java>`, `<echo>`, `<javadoc>`, `<sql>`

165

Exemple

Le projet

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="AProject" basedir="." default="compile" >
    <description>
        Le projet AProject ...
        Executer "ant -Drelease.build="" " pour la version finale
        (pas d'infos de debogage).
    </description>

    <property name="src.dir"     value="src"/>
    <property name="classes.dir" value="classes"/>
    <property name="jar.dir"     value="jar"/>
    <property name="doc.dir"     value="doc"/>

    <property name="main-class"  value="monpackage.AProject"/>

    ...

</project>
```

166

Exemple

La compilation

```
    <target name="init">
        <mkdir dir="${classes.dir}"/>
    </target>

    <target name="init-release" if="release.build">
        <echo message="Version release (pas d'infos de debogage)"/>
        <property name="debug" value="off"/>
    </target>

    <target name="compile" depends="init, init-release" description="Compilation" >
        <property name="debug" value="true"/>
        <depend srcdir="${src.dir}" destdir="${classes.dir}" closure="yes"/>
        <javac srcdir="${src.dir}" destdir="${classes.dir}" debug="${debug}"/>
    </target>
```

167

Exemple

Génération d'un jar

```
    <target name="jar" depends="compile" description="Generation d'un jar">
        <mkdir dir="${jar.dir}"/>
        <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}">
            <manifest>
                <attribute name="Main-Class" value="${main-class}"/>
            </manifest>
        </jar>
    </target>
```

168

Exemple

Génération de la documentation

```
    <target name="javadoc" description="Generation de la documentation" >
        <mkdir dir="${doc.dir}"/>
        <javadoc sourcepath="${src.dir}" destdir="${doc.dir}">
            <fileset dir="${src.dir}"/>
        </javadoc>
    </target>
```

169

Exemple

Suppression des fichiers de compilation

```
    <target name="clean" description="Suppression des fichiers de compilation" >
        <delete dir="${classes.dir}"/>
        <delete dir="${jar.dir}"/>
    </target>
```

170

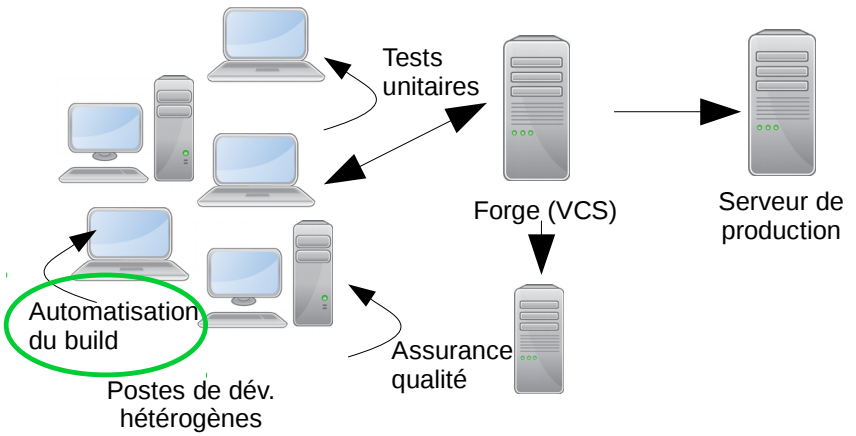
Exemple

Exécution du programme

```
<target name="run" depends="jar" description="Execution du programme">
    <java jar="${jar.dir}/${ant.project.name}.jar" fork="true"/>
</target>
```

171

5.6 Bilan



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

172

Bilan

Apports

- La construction du projet est automatique
- Les scripts de construction sont versionnés et standardisés

Limites

- Comment déployer l'application ?

173

5.7 Exercices

CONTRAINTES

- Pour ces exercices, vous utiliserez votre compte Bitbucket pour conserver votre travail.
- Vous créerez une nouvelle version après chaque question.

Exercice 5.1 (Construction d'un projet avec Maven)
Pour cet exercice, vous travaillerez sur les sources du projet `Compte` réalisé dans un TD précédent. Il pourra également être nécessaire de configurer le proxy (`http://wwwcache.uvsq.fr:3128`) pour accéder au dépôt Maven. À chaque étape, exécutez le cycle maven adapté pour vérifier vos manipulations.

1. Créez un nouveau projet Maven en sélectionnant l'archétype `maven-archetype-quickstart` et en définissant le groupe à `fr.uvsq.<login>` (login est votre login Bitbucket) et l'artéfact à `compte` ;
2. Placez les sources de l'application et des tests dans les répertoires adéquats ; Quelles conventions utilise Maven pour l'arborescence des répertoires ?
3. Modifiez la configuration du projet de la façon suivante :

   - si nécessaire, ajoutez une propriété précisant l'encodage des fichiers du projet (cf. FAQ)
   - modifiez la version des sources et des `.class` pour utiliser la version 1.8 de Java
   - utilisez la version 4.12 de JUnit
4. Modifiez le POM pour intégrer les éléments suivants :
   (a) la génération de la documentation `Javadoc` ;
   (b) la vérification des conventions de codage avec `checkstyle` ;
   (c) la recherche de bogues avec `findbugs` ;
5. Modifiez le POM pour que le jar généré soit exécutable ;
6. En utilisant le plugin `assembly` (ou le plugin shade), générez une archive du projet contenant ses dépendances (*uber-jar*) ;
7. En partant de ce projet, créez un archetype et utilisez-le comme modèle pour un autre projet.

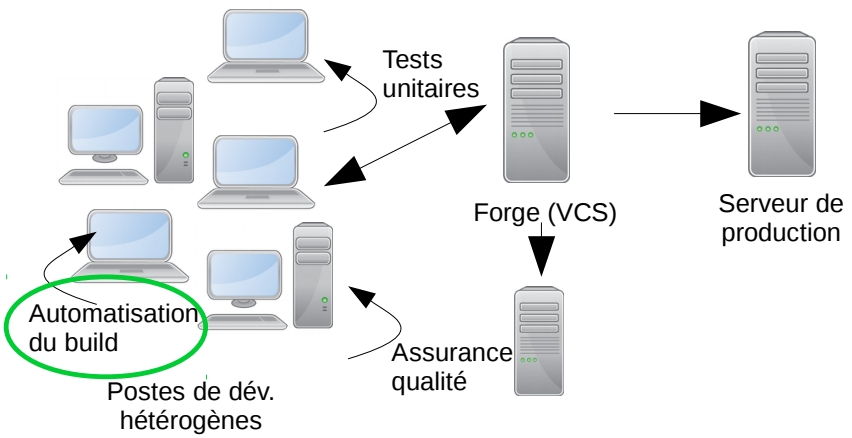
Chapitre 6

Vers la livraison continue

Sommaire

6.1 Introduction 62
6.2 Collaborer avec un VCS 62
6.2.1 Branches 62
6.2.2 Workflows 64
6.3 Intégration continue 65
6.3.1 Introduction 65
6.3.2 Pratiques recommandées 65
6.3.3 Workflow et outils 67
6.4 Gestion et distribution des binaires 67
6.5 Livraison continue 67
6.6 Déploiement continu 68
6.7 Outils pour le déploiement continu 68
6.8 Bilan 69
6.9 Exercices 69

6.1 Introduction



- Communication/tâches
- Fusion/versions
- Build
- →Distribution des binaires←
- Tests/intégration
- Ass. Qualité
- →Livraison←
- Documentation
- Postes de dév. standards

174

6.2 Collaborer avec un VCS

6.2.1 Branches

Branches

Le problème

— Comment gérer les évolutions de développements légèrement différents (nouvelle fonctionnalité, correction de bogue, ...) ?

— Comment reporter les modifications de l'une des évolutions sur les autres ?

— Une *branche* est une ligne de développement indépendante de la ligne principale mais qui partage le même historique (voir figure 6.1).

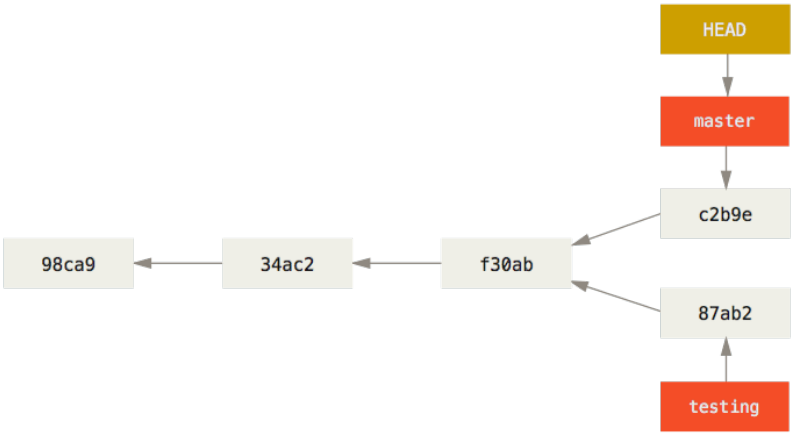


FIGURE 6.1 – Des branches avec git (source Git scm book).

— Une branche peut ensuite être fusionnée avec une autre afin d'y reporter les modifications

175

Manipulation de branches

— L'initialisation d'un dépôt crée une branche nommée *master*

```
git init
```

— Création de la branche *testing*

```
git branch testing
```

— Basculer sur la branche *testing*

```
git checkout testing
```

— Création et bascule sur la branche *testing*

```
git checkout -b testing
```

— Suppression de la branche *testing*

```
git branch -d testing
```

176

Fusion de branches

— La *fusion* permet de « reporter » les changements d'une branche sur une autre.

```
git checkout master
git merge testing
```

— La fusion peut provoquer des conflits

177

Branches distantes

— Une branche distante fait référence a une branche dans un dépôt distant
— *git clone* crée automatiquement le dépôt *origin*
— La commande *git remote* permet de gérer les dépôts distants
— Récupérer les modifications d'une branche distante

```
git fetch origin
```

— Envoyer les modifications sur une branche distante

```
git push origin master
```

— Récupérer et fusionner les modifications d'une branche distante

```
git pull origin master
```

178

Tags

— Un *tag* est un marqueur qui fait référence à une révision particulière
— Lister les tags

```
git tag
```

— Placer un tag sur la révision courante (*HEAD*)

```
git tag -a v1.0 -m"Version 1.0"
```

— Envoyer le tag *v1.0* sur le dépôt *origin*

```
git push origin v1.0
```

— Envoyer tous les tags sur le dépôt *origin*

```
git push origin --tags
```

— Se positionner sur le tag *v1.0*

```
git checkout -b version1 v1.0
```

179

6.2.2 Workflows

Workflows avec git

— Un *workflow* décrit un ensemble d'activités ainsi que la manière dont elles s'enchaînent
— Un workflow git décrit :
  — la façon d'utiliser les branches,
  — quand et comment les fusionner.

180

Workflow centralisé

— C'est le workflow le plus simple et le plus proche de celui des VCS centralisés
— Un dépôt fait référence
— Tout se passe sur la branche principale (*master*)
— Quand l'historique local est satisfaisant, il est publié sur le dépôt de référence
— En cas de conflit, les modifications distantes sont reportées localement (*rebase*)
— L'historique est toujours linéaire

181

Workflow *une branche par fonctionnalité*

— Chaque développement se déroule sur une branche spécifique (*feature branch*)
— La branche principale (*master*) ne contient que du code « fiable »
— Un dépôt fait référence
— Les *feature branches* sont poussées sur le dépôt central
— Quand les modifications sont satisfaisantes, la feature branch est fusionnée avec *master*
— Il est possible d'utiliser les *pull requests* pour discuter d'une branche spécifique

182

6.3 Intégration continue

6.3.1 Introduction

Introduction

— L'*intégration continue* (*continuous Integration*) est une pratique de développement où les membres d'une équipe intègre fréquemment (au moins une fois par jour) leur travail
— Un outil automatique est chargé de vérifier et de détecter les problèmes d'intégration au plus tôt
— Est issu d'*Extreme Programming*
— S'appuie généralement sur un *serveur d'intégration continue*

Quelques outils

Jenkins, drone.io, Travis CI, CruiseControl

183

6.3.2 Pratiques recommandées

Maintenir un référentiel unique des sources

— en général avec un outil de gestion de versions
— tous les développeurs doivent utiliser ce référentiel
— tout ce qui est nécessaire à la compilation doit se trouver dans le référentiel (tests, fichier de propriétés, scripts SQL, ...) (à partir d'une machine vierge, un simple checkout doit permettre d'obtenir un système compilable)
— limiter l'utilisation des branches (favoriser la branche principale)
— ne placer aucun produit de la compilation dans le référentiel

184

Automatiser les compilations

— chaque tâche répétitive doit être automatisée (création de la BD, ...) (à partir d'une machine vierge, un checkout et une unique commande doit permettre d'obtenir un système utilisable)

— l'outil doit permettre de ne recompiler que ce qui est nécessaire

— l'outil doit permettre de définir différentes cibles

— le système de compilation de l'IDE ne suffit pas

185

Rendre les compilations auto-testantes

— un ensemble de tests automatisés doit être diaponible

— la compilation doit inclure l'exécution des tests

— l'échec d'un test doit être reporté comme un échec de la compilation

186

Tout le monde valide chaque jour

— des validations fréquentes favorisent une détection rapide des problèmes d'intégration

— validation fréquente ⇒ moins d'endroits où les conflits peuvent se rpoduire ⇒ détection plus rapide des problèmes

— des validations fréquentes encouragent les développeur à découper leur travail en tâches

187

Chaque validation doit compiler la branche principale sur une machine d'intégration

— permet d'avoir une compilation de référence

— la validation dépend de la réussite de cette compilation

— c'est en général le rôle du serveur d'intégration continue

— le serveur notifie le développeur de la réussite de la compilation

188

Maintenir une compilation courte

— pour obtenir un feedback rapide

— XP recommande un maximum de 10mn

189

Tester dans un environnement de production cloné

— chaque différence avec l'environnement de production peut conduire à des résultats de tests différents

— parfois difficile mais il faut s'en approcher au maximum

190

Rendre disponible facilement le dernier exécutable

— chacun doit pouvoir utiliser la dernière version du système

191

Tout le monde peut voir ce qui se passe

— le but de l'intégration continue est de faciliter la communication

— tout le monde doit voir l'état de la branche principale (compilation en cours, échec de la compilation, ...)

— et les changements apportés

192

Automatiser le déploiement

- la copie des exécutables dans les différents environnements doit être automatique
- il peut être necessaire de mettre aussi en place un mécanisme pour annuler un déploiement (en production par exemple)

193

6.3.3 Workflow et outils

Intégration continue vs. outils d'intégration continue

- L'intégration continue ne dépend pas d'un outil
- C'est une pratique qui doit être acceptée par l'équipe de développement
  - la dernière version du code dans le référentiel doit toujours compiler et passer tous les tests
  - le code doit être validé fréquement
- Processus
  1. avant la validation, s'assurer que la compilation et les tests réussissent
  2. prévenir l'équipe de ne pas mettre à jour le référentiel à cause de l'intégration en cours
  3. valider
  4. aller sur la machine d'intégration, récupérer la dernière version du référentiel et s'assurer que la compilation et les tests réussissent
  5. prévenir l'équipe que les mises à jour peuvent reprendre
- Un outil d'intégration continue permet d'automatiser l'étape 4

194

6.4 Gestion et distribution des binaires

Dépôt de binaires

- Le *dépôt de binaires* permet de centraliser l'entreposage et la distribution des résultats de la compilation
- Il doit s'intégrer avec :
  - le serveur d'intégration continue,
  - le système de build utilisé.
- Il gère également le contrôle d'accès aux différents paquets

Quelques outils

Apache Archiva, Sonatype Nexus, JFrog Artifactory

195

6.5 Livraison continue

Livraison continue

- La *livraison continue* (*continuous delivery*) est une approche visant à
  - produire un logiciel en cycles courts,
  - pouvoir livrer le logiciel à tout moment.
- La construction, les tests et la distribution doivent être plus fréquent
- Cette approche s'appuie sur un déploiement fiable et reproductible
- La décision de déployer reste « manuelle »

196

Pipeline de déploiement

— Un *pipeline de déploiement* couvre trois aspects

Visibilité Toutes les phases du processus de la livraison sont visibles par tous les membres de l'équipe

Feedback Les problèmes doivent être remontés au plus tôt à l'équipe

Automatisation Toute version du logiciel peut être déployée sur n'importe quelle plate-forme automatiquement

— La construction du projet est décomposée en étapes successives
  — chaque étape améliore la confiance dans le logiciel
  — les premières étapes fournissent un feedback rapidement

197

6.6 Déploiement continu

Déploiement continu

— Le *déploiement continu* (*continuous deployment*) doit permettre de déployer **automatiquement** en production tout changement

— Peut conduire à mettre à jour le système en production plusieurs fois par jour

198

6.7 Outils pour le déploiement continu

Infrastructure as code

— L'*Infrastructure as code* consiste à gérer et installer les infrastructures (serveur physique, machine virtuelle, ...) en utilisant des fichiers de description exécutables par la machine

— Ces descriptions sont ensuite ajoutées dans le VCS pour maintenir les différentes versions

— C'est une réponse à la question du passage à l'échelle pour l'installation de machines

Quelques outils

Ansible/Ansible Tower, Puppet/Foreman, Chef

199

Machine virtuelle

— Une *machine virtuelle* (VM) est une émulation d'un système informatique

— Elles permettent d'optimiser l'usage des serveurs physiques et apporte de la souplesse

Quelques outils de virtualisation

Virtualbox, VMWare Workstation/ Server, QEMU

Quelques outils de création/configuration de VM

Packer, Vagrant

200

Conteneur logiciel

— La virtualisation au niveau du système d'exploitation permet de mettre en place des espaces isolés nommés *conteneur*

— Dans un conteneur, les processus sont isolés et l'OS propose un mécanisme de gestion des ressources

— C'est une version évoluées du mécanisme de *chroot* disponible sous Unix

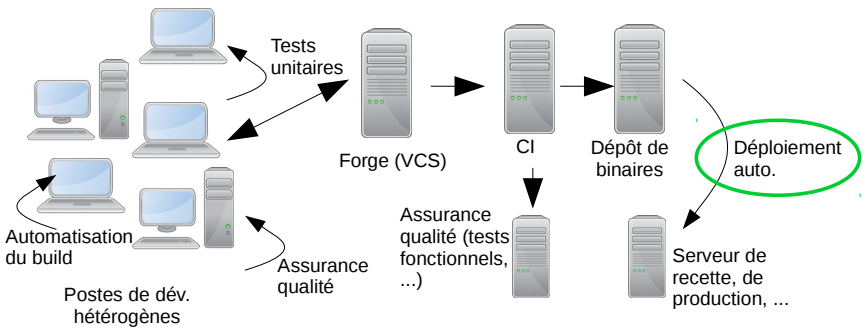
Quelques outils

Docker, Linux Containers, OpenVZ

201

6.8 Bilan

Livraison continue



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

202

Bilan

Apports

- Assure une installation rapide et fiable d'une application
- Met en évidence les problèmes au plus tôt
- Permet la réversibilité des déploiements

Limites

- Complexité de la mise en œuvre

203

6.9 Exercices

CONTRAINTES

- Pour ces exercices, vous utiliserez votre compte Bitbucket pour conserver votre travail.
- Vous créerez une nouvelle version après chaque question.
- Les projets doivent se compiler avec maven.

Exercice 6.1 (Workflow git feature branch)

CONTRAINTES

- Pour cet exercice, vous travaillerez avec un binôme sur un dépôt partagé.
- Vous développerez en suivant une approche TDD.
- Vous travaillerez en parallèle (chaque membre du binôme sur une fonctionnalité) en respectant le workflow feature branch.
- Intégrez fréquemment.

L'objectif de cet exercice est de réaliser une classe **immuable** `Fraction` qui représente un nombre rationnel. Une fraction comporte un numérateur et un dénominateur (nombres entiers). Un exemple d'interface pour une telle classe est donné par la classe `Fraction` de la bibliothèque Apache Commons Math.

Implémentez la classe en fournissant l'interface suivante :

1. initialisation avec (i) un numérateur et un dénominateur, (ii) juste avec le numérateur (dénominateur égal à 1) ou (iii) sans argument (numérateur égal 0 et dénominateur égal à 1),

2. les constantes ZERO (0, 1) et UN (1, 1),
3. consultation du numérateur et du dénominateur,
4. consultation de la valeur sous la forme d'un nombre en virgule flottante (`double`),
5. conversion en chaîne de caractères.

Exercice 6.2 (Workflow git forking)

CONTRAINTES

— Pour cet exercice, vous travaillerez avec un binôme sur deux dépôts indépendants.
— Vous développerez en suivant une approche TDD.
— Vous travaillerez en parallèle (chaque membre du binôme sur une fonctionnalité) en respectant le workflow forking.
— Intégrez fréquemment.

Pour cet exercice, vous continuerez la classe `Fraction` commencée ci-dessus en ajoutant :

1. test d'égalité entre fractions (deux fractions sont égales si elles représentent la même fraction réduite),
2. comparaison de fractions selon l'ordre naturel.

Chapitre 7

Bilan et conclusion

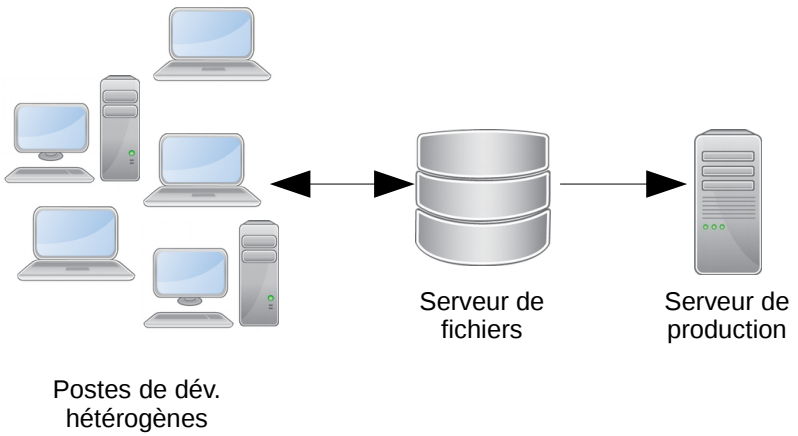
Sommaire

7.1 Bilan 71
7.1.1 Situation initiale 71
7.1.2 Hébergement et gestion des versions 71
7.1.3 Débogage et tests 73
7.1.4 Conventions et audit 74
7.1.5 Automatisation du build 75
7.1.6 Intégration continue 76
7.1.7 Distribution et gestion des binaires 77
7.1.8 Livraison et déploiement continus 78
7.1.9 Standardisation des postes de développement 79
7.1.10 Organisation et gestion de l'équipe 80
7.2 Les indispensables 80
7.3 Pour aller plus loin 80
7.4 Exercices 81

7.1 Bilan

7.1.1 Situation initiale

Collaboration par serveur de fichiers



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

7.1.2 Hébergement et gestion des versions

Gestion de versions

Problème
Comment suivre l'évolution du projet ?

Objectifs
Savoir qui a modifié quoi et en garder la trace.

Moyens

— Utilisation d'un système de gestion de versions (*Source Control Managment* ou *SCM*)
— Suivi des modifications, gestion des conflits, ...

205

Quelques outils

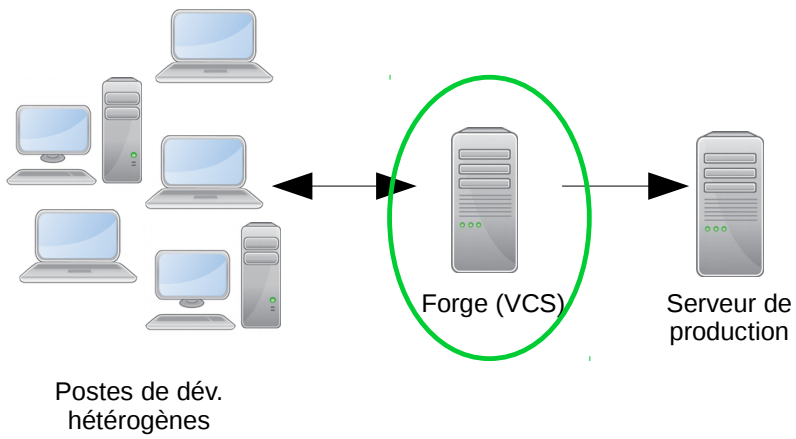
Mode local
SCCS, RCS.

Mode contralisé (client/serveur)
Subversion (*Apache*), CVS.

Mode distribué
Git, Mercurial.

Forges
GitHub, Bitbucket, SourceForge.

206



— Communication/tâches
— Fusion/versions
— Build
— Distribution des binaires
— Tests/intégration
— Ass. Qualité
— Livraison
— Documentation
— Postes de dév. standards

207

Apports et limites

Apports

— Sauvegarde du projet (réparti sur l'ensemble des machines et sur la forge)
— Maintien de l'historique des évolutions du projet

Limites

— Quel confiance peut-on avoir dans le code développé ?
— Comment faire collaborer plusieurs personnes sur le projet ?

208

7.1.3 Débogage et tests

Tests

Problème
Comment réduire les défaillances des logiciels ?

Objectifs
Améliorer la qualité du logiciel.

Moyens

- Mettre en place des tests (automatisés) à différents niveaux (unitaire, d'intégration, fonctionnel, non fonctionnel)
- Utiliser une approche orientée test (*Test Driven Development (TDD)*, Behavior driven development (BDD))

209

Quelques outils

Tests unitaires
xUnit (CUnit, JUnit), TestNG.

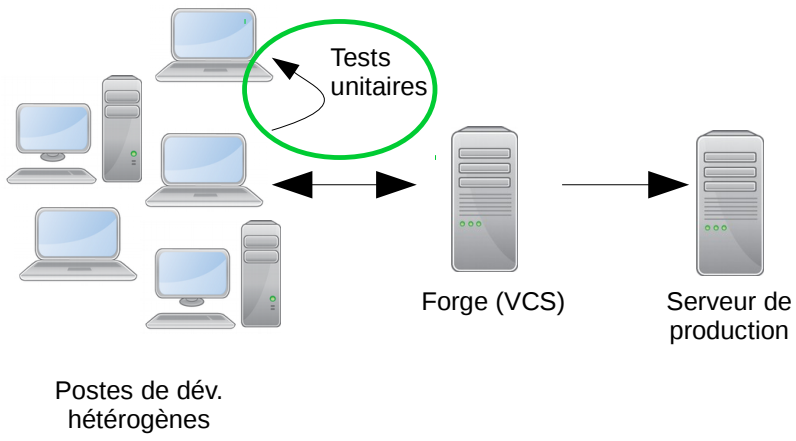
Mocking Framework
Mockito, JMockit, EasyMock, PowerMock.

Tests fonctionnels
Fitness, Selenium, Cargo, Arquilian.

Tests non fonctionnels
JMeter (*Apache*).

Behaviour-Driven Development
Cucumber, Spock, JBehave.

210



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

211

Apports et limites

Apports

— Les techniques de débogage permettent une recherche de bogues plus efficace
— La présence des tests améliore la confiance dans le logiciel
— Les modifications sont plus aisées car sous la garde des tests

Limites

— Comment faire collaborer plusieurs personnes sur le projet ?
  — « règles » et bonnes pratiques de rédaction du code
  — automatisation des tâches

212

7.1.4 Conventions et audit

Assurance qualité

Problème
Comment augmenter la confiance dans le logiciel ?

Objectifs
Avoir une mesure objective de la qualité du projet.

Moyens

— Introduire des techniques et des outils dans le processus de développement
  — style de codage, relecture de code, ...
— Surveiller l'évolution de certaines métriques

213

Gestion de la documentation

Problème
Comment maintenir la documentation technique d'un projet ?

Objectifs
Faire en sorte que la documentation soit en phase avec le code source.

Moyens

— Placer la documentation dans le SCM (texte, Word, OpenOffice)
— Utiliser des outils collaboratifs (Wiki, ...)
— Intégrer des outils spécifiques

214

Quelques outils

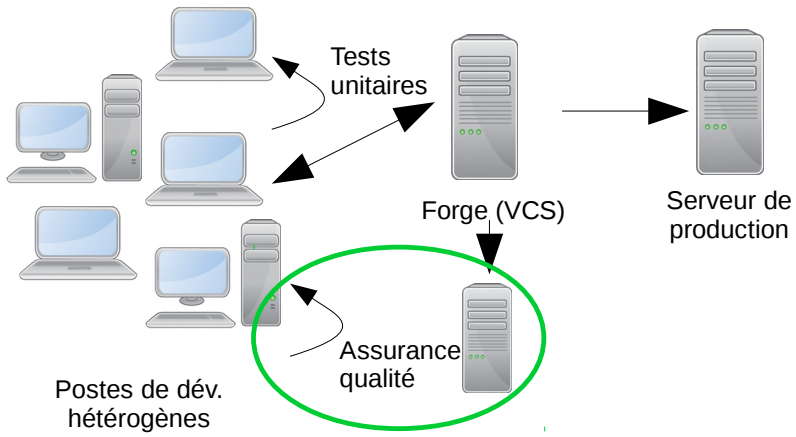
Documentation
Doxygen, JavaDoc (*Oracle*), XWiki.

Audit des conventions
Checkstyle.

Recherche de bogues (analyse statique)
PMD, Findbugs.

Agrégation de métriques
SonarQube.

215



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

216

Apports et limites

Apports

- Le code est plus lisible et donc plus facile à maintenir
- Tous les participants d'un projet partagent ces règles

Limites

- De nombreux outils doivent être appliqués sur le code source
- L'ensemble des développeurs doit utiliser les différents outils
  - automatisation des tâches pour standardiser le *build*

217

7.1.5 Automatisation du build

Automatisation du build

Problème
Comment améliorer le processus de construction du projet ?

Objectifs

Automatiser les tâches répétitives (compilation, ...).

Moyens

- Outils de *Build Automation*

218

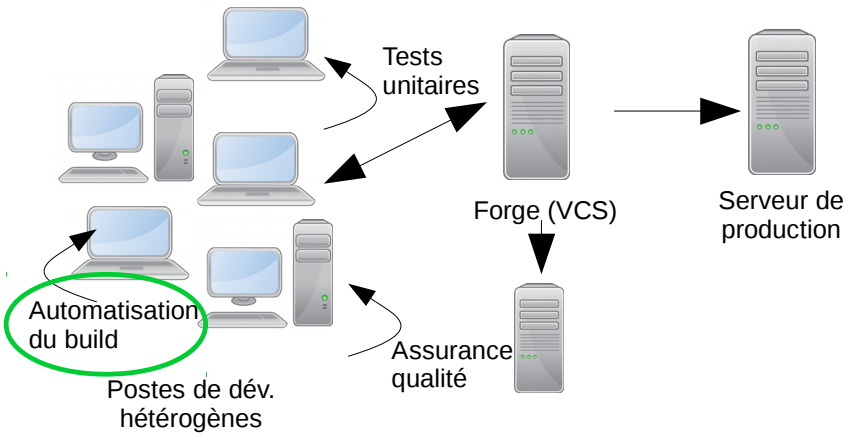
Quelques outils

Dans un environnement Java
Apache Maven, Gradle, Apache Ant, sbt

Dans un environnement C/C++
CMake, Make

Dans d'autres contextes
Waf, SCons, Rake

219



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

220

Apports et limites

Apports

- La construction du projet est automatique
- Les scripts de construction sont versionnés et standardisés

Limites

- Comment déployer l'application ?

221

7.1.6 Intégration continue

Intégration continue

Problème
Comment minimiser les problèmes d'intégration ?

Objectifs

Intégrer fréquemment le travail de chacun et valider cette intégration.

Moyens

- Mettre en place une approche d'*intégration continue*
- Gérer le « feedback » (mail, IM, dispositif physique)

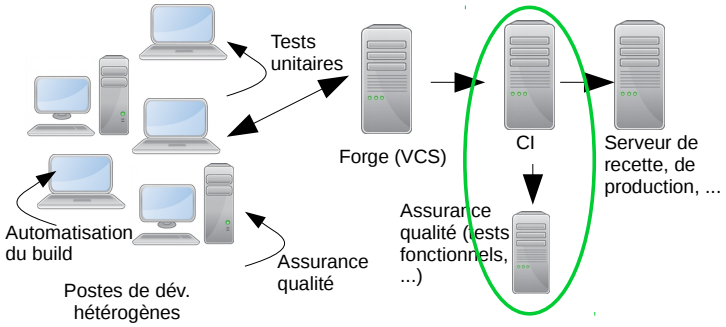
222

Quelques outils

CI
Jenkins, Cloudbees, drone.io, Travis CI, CruiseControl.

223

Intégration continue



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

224

Apports et limites

Apports

- Construction dans un environnement standard
- Mise en évidence des problèmes au plus tôt
- Transparence pour l'équipe de développement

Limites

- Comment mettre les binaires à disposition ?

225

7.1.7 Distribution et gestion des binaires

Gestion des binaires

Problème
Comment contrôler la publication des binaires sans utiliser le SCM ?

Objectifs

Distribuer la version adéquat des binaires aux bonnes personnes.

Moyens

- Utilisation d'un gestionnaire de dépôt binaire (*Repository manager*)
- Mettre en place un contrôle d'accès

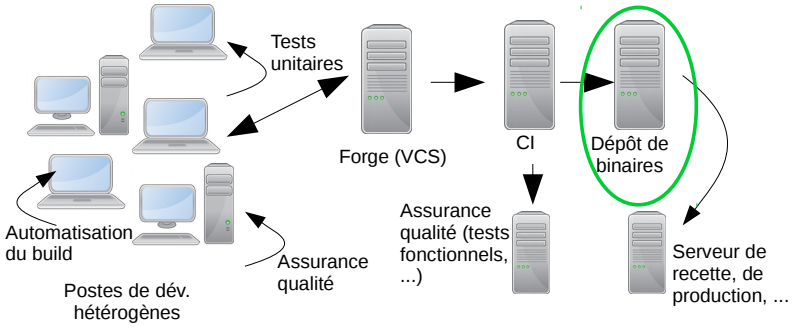
226

Quelques outils

Repository Manager
Apache Archiva, Nexus (*Sonatype*), Artifactory (*JFrog*).

227

Intégration continue



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

228

Apports et limites

Apports

- Fournit un espace centralisé de référence pour les binaires
- Gère le contrôle d'accès

Limites

- Comment déployer l'application ?

229

7.1.8 Livraison et déploiement continus

Livraison

Problème
Comment livrer et déployer l'application ?

Objectifs

Assurer une installation rapide et fiable de l'application

Moyens

- Produire automatiquement une version distribuable (*release*) des binaires
- Automatiser le déploiement (serveur d'application, SGBD, ...)

230

Quelques outils

Infrastructure as code
Ansible/Ansible Tower, Puppet/Foreman, Chef

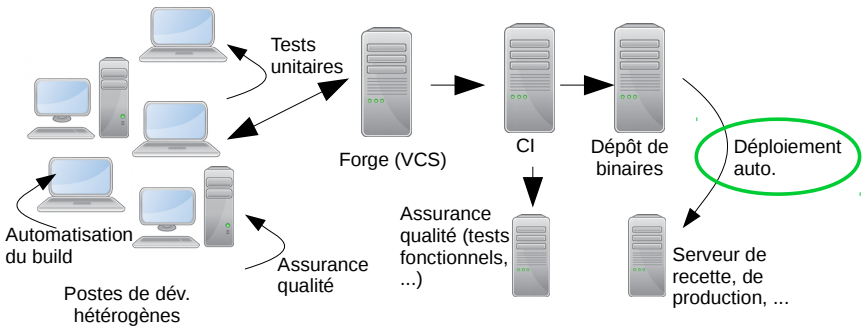
Virtualisation
Virtualbox, VMWare Workstation/ Server, QEMU

Création/configuration de VM
Packer, Vagrant

Conteneur logiciel
Docker, Linux Containers, OpenVZ

231

Livraison continue



- Communication/tâches
- Fusion/versions
- Build
- Distribution des binaires
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dév. standards

232

Bilan

Apports

- Assure une installation rapide et fiable d'une application
- Met en évidence les problèmes au plus tôt
- Permet la réversibilité des déploiements

Limites

- Complexité de la mise en œuvre

233

7.1.9 Standardisation des postes de développement

Poste de développement

Problème

Installer et configurer un poste de développement complet peut être complexe et long.

Objectifs

Automatiser la mise en place d'un poste de développement.

Moyens

- Fournir un environnement type (archive)
- Utiliser une machine virtuelle
- Développer dans le Cloud

234

Quelques outils

Virtualisation

VIRTUALBOX (*Oracle*), ...

Cloud IDE

ECLIPSE CHE, CODENVY, CLOUD9.

235

7.1.10 Organisation et gestion de l'équipe

Organisation et gestion de l'équipe

Problème
Comment organiser le travail de l'équipe de développement ?

Objectifs
Améliorer la communication et optimiser les interactions.

Moyens

- Méthode de gestion de projet (XP, Scrum, UP, ...)
- Intégration d'outils (suivi de tickets, ...)

236

Quelques outils

Gestion de projets
Jira (*Atlassian*), Trello, Mur de Post-It.

Techniques de planification
Pomodoro, Getting Things Done.

237

7.2 Les indispensables

Les indispensables

- Une machine de développement
  - correctement installée
  - maintenue à jour
  - **dont on maîtrise le *shell***
- Un éditeur de texte ou un IDE performant
  - **que l'on connaît bien**
- Un système de gestion de version (*git*)
  - **que l'on utilise systématiquement**
- Un outil de gestion de la compilation (*gradle*, *maven*)
  - que l'on utilise systématiquement

238

7.3 Pour aller plus loin

Software Craftmanship

- Manifeste *Software Craftsmanship* (artisan du logiciel)
- Met l'accent sur les compétences des développeurs et leur conscience professionnelle
  - *des logiciels bien conçus*
  - *l'ajout constant de la valeur*
  - *une communauté de professionnels*
  - *des partenariats productifs*
- s'inspire du *compagnonnage*

239

DevOps

1. Mouvement visant à réduire le gap entre développement (*DEVelopment*) et exploitation (*OPerationS*)
2. L'objectif est de *délivrer le meilleur logiciel aux clients de l'entreprise*
   — en faisant coopérer Devs et Ops sur l'ensemble du projet
   — en modernisant les outils (usine logicielle) et en mettant en place du *monitoring*
   — en livrant en continu

240

The Twelve-Factor App

1. The Twelve-Factor App
2. Décrit une méthodologie pour concevoir des applications *as a service*

241

7.4 Exercices

Exercice 7.1 (Une application web avec Spring Boot)
Dans cette exercice, vous réaliserez une application web pour une liste de tâches (*TODO List*).

> CONTRAINTES
> — Vous travaillerez avec un binôme sur un dépôt partagé.
> — À partir de la question 2, vous travaillerez en parallèle (chaque membre du binôme sur une fonctionnalité) en respectant le workflow feature branch.
> — Pour la question 3, vous devrez installer Docker.

1. Générez le squelette du projet avec Spring Initializr (group = fr.uvsq.tod, artifact=spring-todo). Vous intégrerez la dépendance *Web*.
2. En suivant le tutoriel Building an Application with Spring Boot, retournez un simple message pour la requête HTTP `GET /`
3. Créez une image docker pour l'application (cf. Spring Boot with Docker)
4. Modifiez l'application pour que la requête HTTP `GET /` renvoie la liste des tâches (cf. Building a RESTful Web Service)
5. Ajoutez la création d'une tâche avec une requête HTTP `POST /` ayant le titre de la tâche en paramètre.

** Rendez la liste persistante en utilisant JPA et Derby. (cf. Accessing Data with JPA).

** Développez une interface web en utilisant le moteur de templates thymeleaf (cf. Serving Web Content with Spring MVC).

Exemples à consulter

— Spring Boot Example
— Thymeleaf Course