

Techniques et outils d'ingénierie

Stéphane Lopes

stephane.lopes@prism.uvsq.fr



2013-2014

Chapitre I : Preamble

1 Objectifs et prérequis

2 Plan

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 de modélisation (UML)

Prérequis

- Petite expérience de la programmation
- Notions de langage Java

Plan général

- Introduction
- Gestion du code source
- Construction et gestion des binaires
- Mise au point de programmes
- La notation UML

Chapitre II : Introduction

- 3 Qu'est-ce que le développement logiciel ?
- 4 Professionnalisation du développement
- 5 Exemples de mise en œuvre

Chapitre II : Introduction

Section 3 : 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*

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 une ensemble de tâches ou d'activités
 - différents modèles (chute d'eau, en V, en spirale, itératif et incrémental, agile, ...)

Principales activités du processus de développement

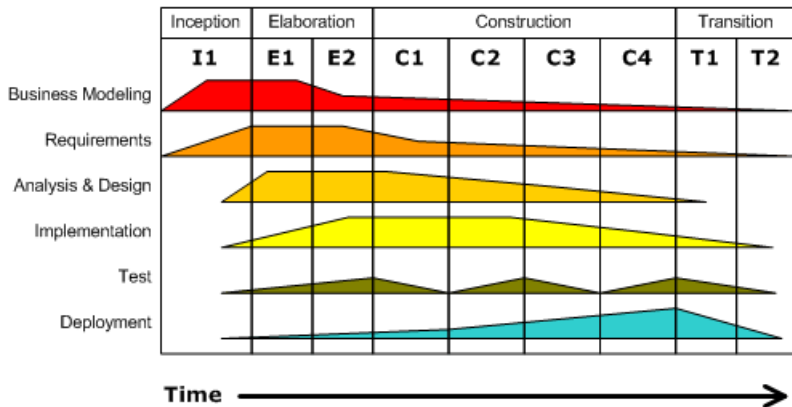
- Gestion de projet
- Analyse des besoins/exigences
- Spécifications fonctionnelles et non fonctionnelles
- Architecture logicielle
- Conception
- Implémentation
- Tests
- Déploiement
- Évolution et maintenance

Exemple

Activités du processus unifié (RUP)

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.



source : [Iterative and incremental development](#) (Wikipedia)

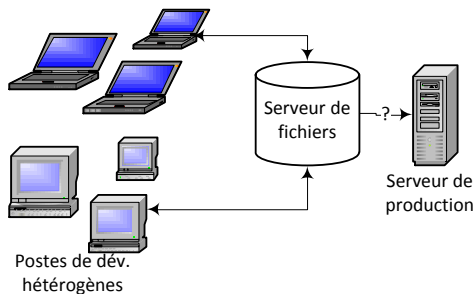
Chapitre II : Introduction

Section 4 : Professionnalisation du développement

Professionnalisation du développement

- Industrialisation du logiciel
 - le logiciel est vu comme un produit manufacturé
 - forge/usine logicielle
 - automatisation des traitements répétitifs
- Artisanat (manifeste *Software Craftsmanship*)
 - met l'accent sur les compétences des développeurs
 - s'inspire du *compagnonnage*
- DevOps
 - mouvement visant à réduire le gap entre développement (*DEvelopment*) et exploitation (*OPerationS*)
 - l'objectif est de permettre des livraisons fréquentes du logiciel
 - automatisation des déploiements

Situation initiale

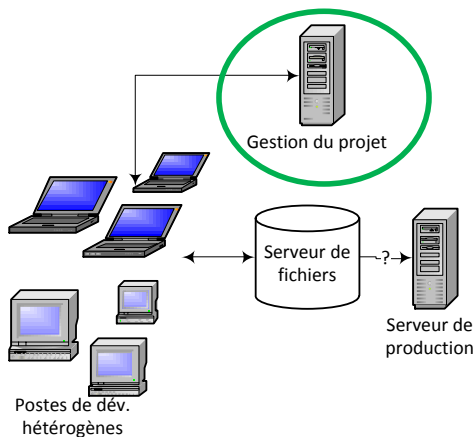


- Communication/tâches
- Fusion/versions
- Build
- Distribution des bin.
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dev. standards

Efficacité de l'équipe I

- Améliorer la communication
- Optimiser les interactions
- Méthode de gestion de projet (XP, Scrum, Agile, Lean, . . .)
- Intégration d'outils
 - Atlassian [Jira Studio](#), Atlassian [Greenhopper](#), mur de post-it

Efficacité de l'équipe II

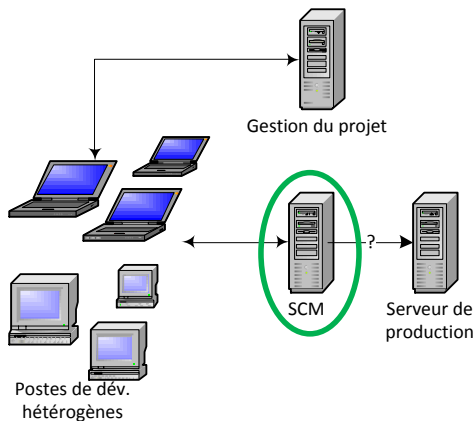


- Communication/tâches
- Fusion/versions
- Build
- Distribution des bin.
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dev. standards

Travail de groupe/gestion de versions I

- Utilisation indispensable d'un système de gestion de versions (*Source Control Managment* ou *SCM*)
- Suivi des modifications
- Gestion des conflits
- Apache/CollabNet Subversion, Git

Travail de groupe/gestion de versions II

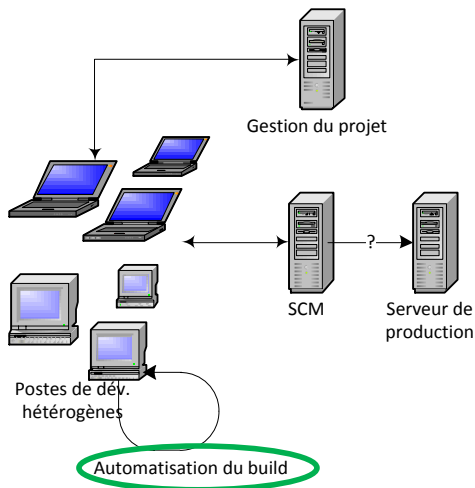


- Communication/tâches
- Fusion/versions
- Build
- Distribution des bin.
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dev. standards

Gestion du « build » I

- Améliorer le processus de construction du projet
- Automatisation des tâches répétitives (génération des binaires, exécution des tests, métriques, ...)
- Apache Ant, Apache Maven

Gestion du « build » II

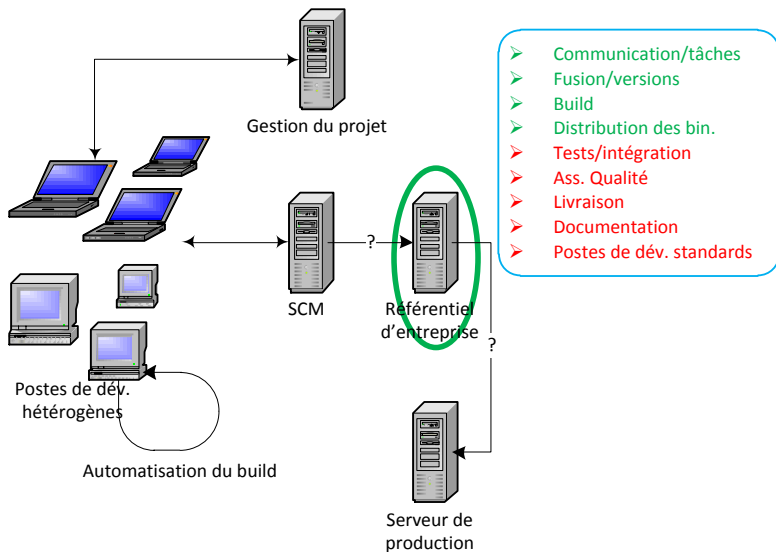


- Communication/tâches
- Fusion/versions
- Build
- Distribution des bin.
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dev. standards

Gestion des binaires I

- Règle : pas d'artefact généré dans le SCM
- Distribution des binaires
- Contrôle d'accès
- Utilisation d'un gestionnaire de dépôt binaire (*Repository manager*)
 - Sonatype Nexus

Gestion des binaires II



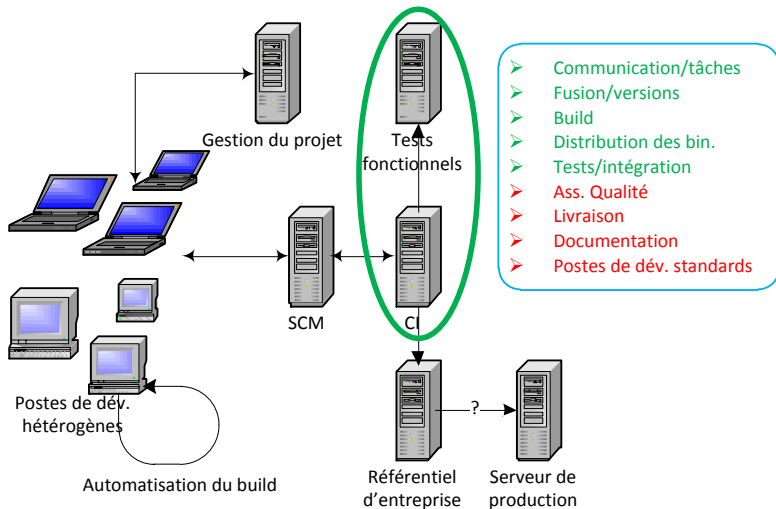
Tests

- Les tests sont indispensables pour améliorer la qualité du logiciel
- Approches
 - classique : les tests sont implémentés après la fonctionnalité
 - développement piloté par les tests (*Test Driven Development* ou *TDD*) : tests écrits avant la fonctionnalité
 - développement piloté par le comportement (*Behavior driven development* ou *BDD*) : tests écrits en langage naturel (*User stories*, *JBehave*, *Cucumber*)
- Types de tests
 - unitaires, d'intégration (JUnit, TestNG) : vérifie le fonctionnement des modules, fournit une documentation
 - fonctionnels (Fitness) : est une traduction exécutable des exigences, nécessite un déploiement automatique (*Cargo*, *Arquillian*)
 - IHM (*Selenium*) : automatise les tests utilisateurs
 - non fonctionnel (Apache *JMeter*, *Clif*) : performances, sécurité, ...
- Fréquence des tests : aussi souvent que possible \Rightarrow automatiser

Intégration continue I

- Intégrer fréquemment le travail de chacun afin de minimiser les efforts d'intégration
- Disposer en permanence d'une version opérationnelle du projet
- Comment gérer le « feedback » ? (mail, IM, dispositif physique)
- Serveur d'intégration continue ([Jenkins](#))

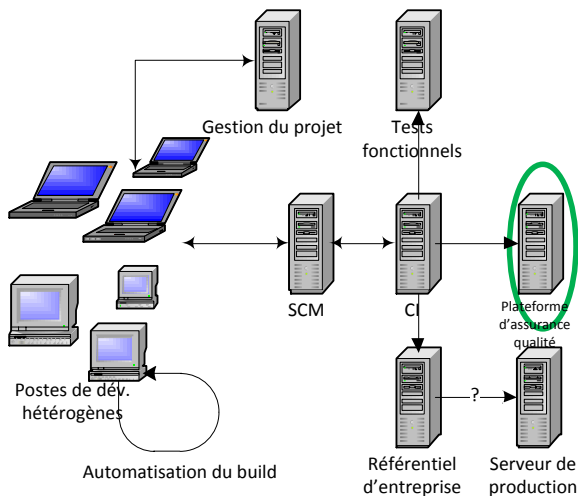
Intégration continue II



Assurance qualité I

- Objectif : augmenter la confiance dans le logiciel
- Nombreuses approches (relecture de code, ...), outils (Checkstyle, PMD, Findbugs, ...) et métriques
- Difficulté pour synthétiser et communiquer les résultats
- Intégration de plusieurs outils ([Sonar](#))

Assurance qualité II

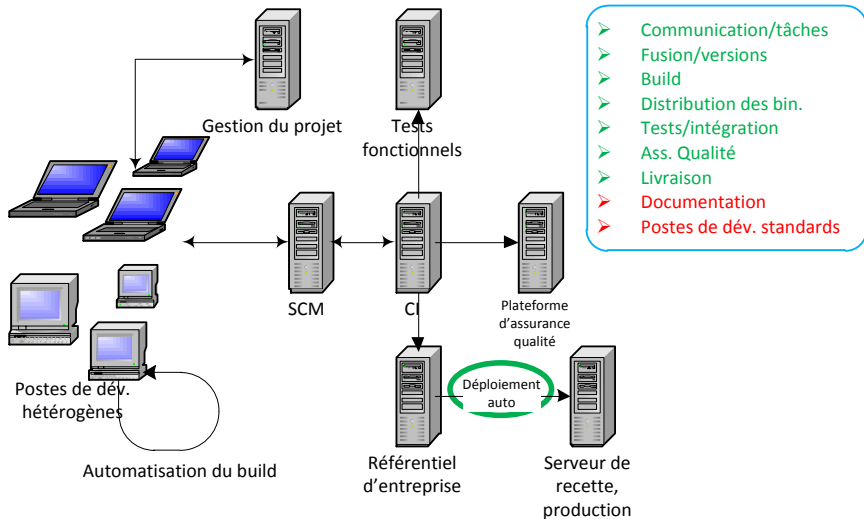


- Communication/tâches
- Fusion/versions
- Build
- Distribution des bin.
- Tests/intégration
- Ass. Qualité
- Livraison
- Documentation
- Postes de dev. standards

Livraison I

- Produire automatiquement une version de production (*release*) des binaires (plugin *release* de maven)
- Automatiser le déploiement : serveur d'application, SGBD ([DBMaintain](#))

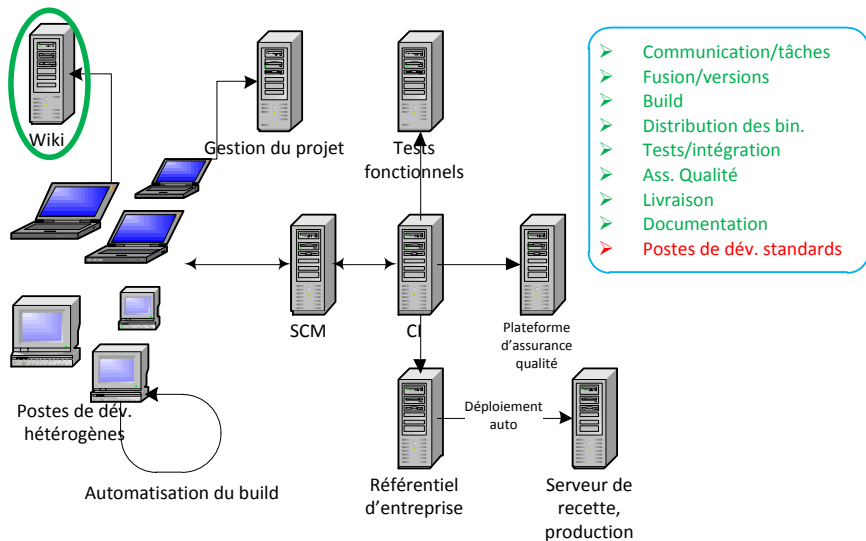
Livraison II



Documentation I

- Comment faire en sorte que la documentation soit en phase avec le code ?
 - fichiers de documentation dans le SCM (texte, Word, OpenOffice)
 - Wiki ([XWiki](#))

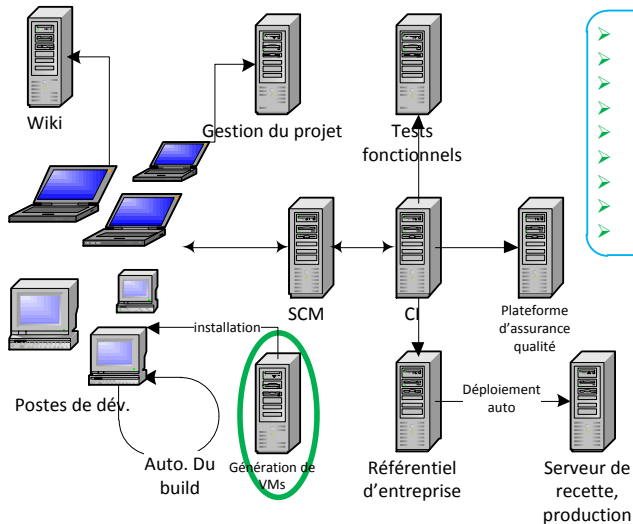
Documentation II



Poste de développement I

- Problème : installer et configurer un poste de développement complet peut être coûteux
- Comment créer un poste de développement prêt à coder ?
 - fournir un environnement type (archive)
 - utiliser une machine virtuelle
 - générateur de forge ([Blacksmith Project](#))

Poste de développement II



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

Chapitre II : Introduction

Section 5 : Exemples de mise en œuvre

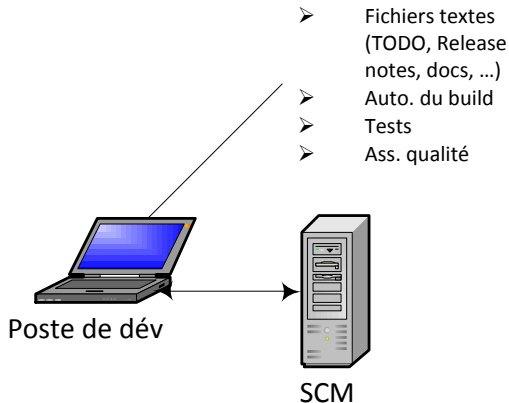
Petit projet I

TP, projet dans une UE, ...

Comm./tâches	Fichier texte
SCM	Oui
Automatisation du build	Oui
Gestion des binaires	Manuelle
Tests/intégration	Poste de dév.
Ass. qualité	Poste de dév.
Livraison	Manuelle
Documentation	Fichiers

Petit projet II

TP, projet dans une UE, ...



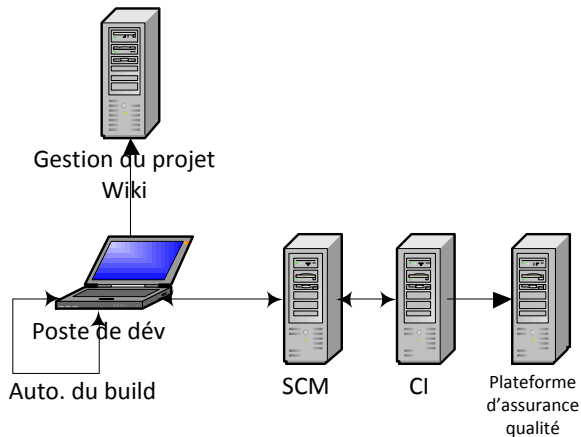
Projet important I

Projet annuel, ...

Comm./tâches	Appli. web
SCM	Oui
Automatisation du build	Oui
Gestion des binaires	Manuelle
Tests/intégration	CI
Ass. qualité	Serveur
Livraison	Manuelle
Documentation	Wiki

Projet important II

Projet annuel, ...



Chapitre III : Gestion du code source

- 6 Environnements de développement
- 7 Style de codage
- 8 Documentation
- 9 Analyse statique du code
- 10 Gestion des versions/Travail de groupe

Chapitre III : Gestion du code source

Section 6 : Environnements de développement

- Editeur de texte pour la programmation
- IDE
 - Introduction
 - Eclipse

Chapitre III : Gestion du code source

Section 6 : Environnements de développement

- Editeur de texte pour la programmation
- IDE
 - Introduction
 - Eclipse

Editeur de texte

- Un *éditeur de texte* est un programme permettant l'édition de fichiers en texte brut.
- Principales fonctionnalités
 - Rechercher/Remplacer
 - Défaire/Refaire
 - Couper/Copier/Coller
 - Support des jeux de caractères étendus
 - Langage de macros
 - Vérification de l'orthographe

Editeur de code source

- L'éditeur est spécialisé pour la rédaction de programme
 - facilite la saisie du code source (coloration syntaxique, ...)
 - respect des standards (indentation automatique, reformatage, ...)
- Principales fonctionnalités
 - Support de projet (ensemble de fichiers)
 - Coloration syntaxique
 - Mise en évidence des parenthèses
 - Indentation automatique
 - Pliage (*folding*)/dépliage de texte
 - Modèle (insertion de texte type)
 - Index des fonctions (*ctags*)
 - Appel de programmes externes (compilateur, ...)

Quelques outils

VI, VIM, EMACS, GEDIT, NOTEPAD++.

Chapitre III : Gestion du code source

Section 6 : Environnements de développement

- Editeur de texte pour la programmation
- IDE
 - Introduction
 - Eclipse

Environnement de développement intégré

- Un environnement de développement intégré (*Integrated Development Environment* ou *IDE*) regroupe un ensemble d'outils de développement
- Principaux outils intégrés
 - un éditeur de code source
 - un compilateur/interpréteur
 - un débogueur
- Plus rarement
 - un navigateur de classes
 - un assistant au développement d'interfaces graphiques
 - une interface pour la gestion de versions
- Un IDE est parfois complexe à prendre en main

Quelques outils

ECLIPSE, INTELLIJIDEA, NETBEANS, MICROSOFT VISUAL STUDIO.

Principales fonctionnalités

- Les mêmes fonctionnalités que les éditeurs de code source
- Gestion de projets (fichiers, dépendances, ...)
- Auto-complétion de code
- Navigation dans les classes
- *Refactoring*
- Débogage
- Profiling
- Intégration de la gestion de versions

Editeur ou IDE ?

- Avantages d'un éditeur
 - plus léger
 - contrôle total du processus de développement
 - choix d'outil de façon indépendante (éditeur, débogueur, ...)
- Avantage d'un IDE
 - ensemble d'outils intégrés
 - fonctionnalités puissantes

Eclipse

- *Eclipse* est un projet libre dont le but est de fournir une plate-forme de développement extensible.
- Les différents développements sont organisés en *projets*.
- La *plate-forme Eclipse* est écrite en Java et extensible par un système de *plugins*.
- La *fondation Eclipse* est soutenue par de nombreuses organisations (Borland, IBM, Intel, OMG, Oracle, ...).
- Le code initial ainsi qu'une partie des extensions principales a été donné par IBM (à partir de *Visual Age*).

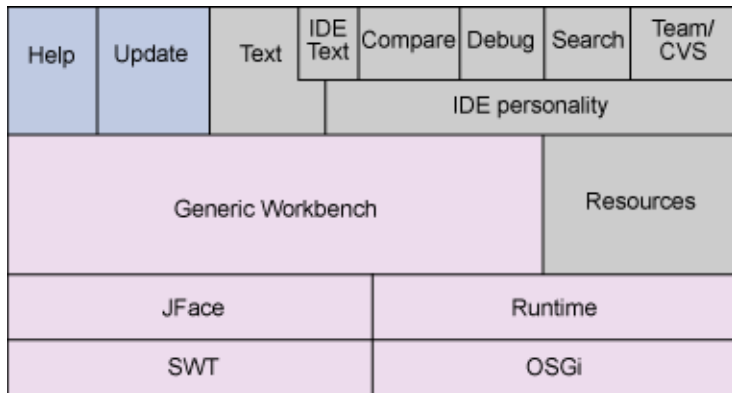
Utilisation

- IDE pour Java (plate-forme + JDT)
- *Framework* pour un IDE (CDT pour C/C++, PDT pour PHP, ...)
- Plate-forme pour intégrer des outils (BI, modélisation, ...)
- *Framework* pour des applications (application de type client riche, ...)
- Plate-forme d'exécution ([Equinox](#)/[OSGi](#))

Projets de premier niveau

- *Eclipse* (plate-forme, JDT, PDE)
- *Tools*
- *Web Tools Platform*
- *Test and Performance Tools Platform* (TPTP)
- *Business Intelligence and Reporting Tools* (BIRT)
- *Modeling*
- *Data Tools Platform*
- *Device Software Development Platform*
- *SOA Tools Platform*
- *Technology*
- *RT* (Equinox)

Architecture



source : [Get started with the Eclipse Platform](#)

Quelques éléments de l'architecture

Runtime/OSGi système d'exécution à la base de l'architecture des plugins

Rich Client Platform plus petit sous-ensemble de plugins nécessaires pour une application

Standard Widget Toolkit (SWT) ensemble de composants graphiques pour les applications

JFace composants graphiques de plus haut niveau

Workbench éléments de base de l'interface d'Eclipse

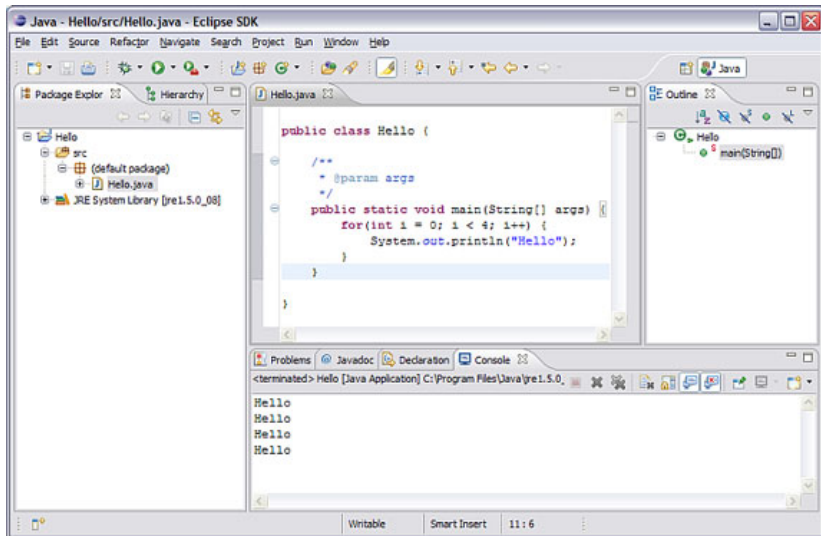
Interface graphique de l'IDE

- Le *workbench* regroupe un ensemble de fenêtres
- Un *éditeur* permet à l'utilisateur d'ouvrir, d'éditer et de sauver des objets
- Une *vue* fournit des informations sur les objets manipulés
- Une *perspective* correspond à une organisation des fenêtres à l'écran pour une tâche particulière

Manipulation de projets

- Le *Workspace* est un emplacement sur le système qui contiendra le travail de l'utilisateur
- L'espace de travail contient des *ressources* (projets, répertoires et fichiers)

Perspective Java



source : [Get started with the Eclipse Platform](#)

Chapitre III : Gestion du code source

Section 7 : Style de codage

- Conventions de codage
- Outils

Chapitre III : Gestion du code source

Section 7 : Style de codage

- Conventions de codage
- Outils

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 !

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);  
            }  
        }  
    }  
}
```

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 Comparable.
 * 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);
            }
        }
    }
}
```

Quelques conventions I

- *Java code conventions*, **SUN**, 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**, GeoSoft (2008)
 - chaque règle est commentée
 - un fichier de configuration pour Checkstyle est disponible

Quelques conventions II

- *Java Language Coding Guidelines*, **Cay Horstmann**, 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 SUN

- 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. chapitre 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.*

Chapitre III : Gestion du code source

Section 7 : Style de codage

- Conventions de codage
- 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

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

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.
```

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
```

Rôle des 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

Quelques outils de reformatage

- JACOB

- supporte le langage Java
- par défaut, respecte le style SUN et est configurable
- plugins pour ant et eclipse

- ARTISTIC STYLE

- supporte les langages C, C++, C# et Java
- configurable

Chapitre III : Gestion du code source

Section 8 : Documentation

- Documenter un code source
- JavaDoc
- Doxygen

Chapitre III : Gestion du code source

Section 8 : Documentation

- Documenter un code source
 - JavaDoc
 - Doxygen

Intérêt

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

Quelques outils

JAVADOC, DOXYGEN.

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

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

Chapitre III : Gestion du code source

Section 8 : Documentation

- Documenter un code source
- **JavaDoc**
- Doxygen

JavaDoc

- Outil standard fourni avec le JDK
- Génération automatique de la documentation au format HTML
- Documente les classes, les interfaces, les membres , les modules et les fichiers sources
- Le contenu de la sortie peut être paramétrée (membres privés ou non, ...)
- Un *Doclet* permet de modifier la sortie (XML, ...)

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

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

Liste des tags J

`@author` noms des auteurs de l'entité

`{@code}` affiche le texte en mode «code»

`{@docRoot}` chemin de la documentation

`@deprecated` précise que l'entité ne devrait plus être utilisée

`@exception` synonyme de `@throws`

`{@inheritDoc}` hérite la documentation

`{@link}` insère un lien interne vers la documentation

`{@linkplain}` idem avec une police en forme normal

`{@literal}` affiche le texte sans interprétation

`@param` documente un paramètre de méthode

`@return` décrit la valeur de retour d'une méthode

Liste des tags II

`@see` ajoute une section *See Also*

`@serial` lié à la sérialisation

`@serialData` idem

`@serialField` idem

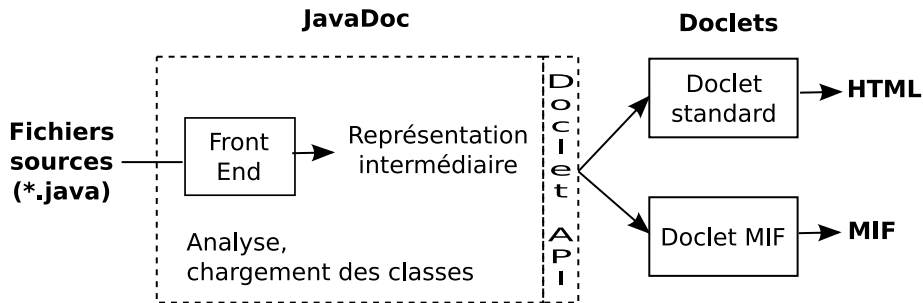
`@since` version où l'entité a été ajoutée

`@throws` exception lancée par une méthode

`{@value}` affiche la valeur d'une constante

`@version` version de l'entité

Principe



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) {
```

Chapitre III : Gestion du code source

Section 8 : Documentation

- Documenter un code source
- JavaDoc
- 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

Exemple I

Résultat avec Doxygen

[Main Page](#) [Related Pages](#) [Modules](#) [Data Structures](#) [Files](#)[Alphabetical List](#) [Data Structures](#) [Class Hierarchy](#) [Data Fields](#)

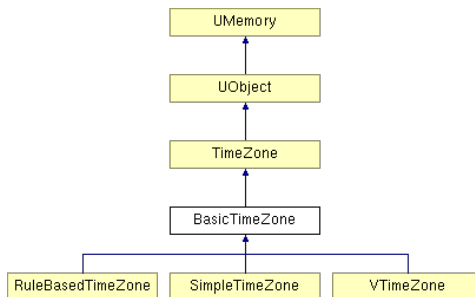
Search for

BasicTimeZone Class Reference

BasicTimeZone is an abstract class extending [TimeZone](#). [More...](#)

```
#include <basictz.h>
```

Inheritance diagram for BasicTimeZone:



Exemple II

Résultat avec Doxygen

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 TimeZoneRules which represents time transitions, for this time zone, that is, all TimeZoneRules 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...
```


Chapitre III : Gestion du code source

Section 9 : Analyse statique du code

- Audit de code source
- FindBugs
- PMD

Chapitre III : Gestion du code source

Section 9 : Analyse statique du code

- Audit de code source
 - FindBugs
 - PMD

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 échangent leur rôle régulièrement (vient de **eXtreme Programming** (XP))
 - assisté par un outil** s'appuie sur des outils pour une analyse systématique

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, ...*

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

• HAMMURAPI

- analyse le code source

Chapitre III : Gestion du code source

Section 9 : Analyse statique du code

- Audit de code source
- FindBugs
- PMD

Exécution

- avec l'interface graphique

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

- en ligne de commande

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

- comme tâche ant
- comme extension pour Eclipse

Exemple 1

Extrait du résultat de FindBugs sur les sources du JDK

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			

Exemple II

Extrait du résultat de FindBugs sur les sources du JDK

java.io (23: 3/20/0/0)

java.lang (46: 3/43/0/0)

java.lang.AssertionStatusDirectives (4: 0/4/0/0)

java.lang.Byte (1: 0/1/0/0)

java.lang.Byte\$ByteCache (1: 0/1/0/0)

java.lang.Character (1: 0/1/0/0)

java.lang.Character\$CharacterCache (1: 0/1/0/0)

java.lang.Class (2: 1/1/0/0)

Load of known null value (1)

Non-transient non-serializable instance field in serializable class (1)

java.lang.Class\$EnclosingMethodInfo (1: 0/1/0/0)

java.lang.Class\$MethodArray (1: 0/1/0/0)

java.lang.ClassLoader (7: 0/7/0/0)

java.lang.ClassLoader\$3 (1: 0/1/0/0)

java.lang.ConditionalSpecialCasing (2: 0/2/0/0)

java.lang.Integer (6: 0/6/0/0)

Method invokes inefficient Number constructor; use static valueOf instead (5)

Method invokes inefficient new String(String) constructor (1)

java.lang.Integer\$IntegerCache (1: 0/1/0/0)

java.lang.Long (6: 0/6/0/0)

java.lang.Long\$LongCache (1: 0/1/0/0)

java.lang.Object (1: 0/1/0/0)

java.lang.Package (1: 1/0/0/0)

java.lang.SecurityManager (2: 0/2/0/0)

java.lang.Short (2: 0/2/0/0)

java.lang.Short\$ShortCache (1: 0/1/0/0)

java.lang.String (1: 1/0/0/0)

java.lang.Thread (1: 0/1/0/0)


java.lang.Throwable (1: 0/1/0/0)

Dm / DM_STRING_CTOR
Using the java.lang.String(String) constructor wastes memory because the object so constructed will be functionally indistinguishable from the String passed as a parameter. Just use the argument String directly.

Exemple III

Extrait du résultat de FindBugs sur les sources du JDK

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
			
Bad practice (954: 118/836/0/0)			
Correctness (249: 81/168/0/0)			
BC: Bad casts of object references (4: 1/3/0/0)			
Bx: Questionable Boxing of primitive value (1: 0/1/0/0)			
DMI: Dubious method invocation (17: 0/17/0/0)			
EC: Suspicious equals() comparison (3: 2/1/0/0)			
FE: Test for floating point equality (6: 6/0/0/0)			
ICAST: Casting from integer values (5: 5/0/0/0)			
IP: Ignored parameter (2: 0/2/0/0)			
MF: Masked Field (16: 8/8/0/0)			
Class defines field that masks a superclass field (16: 8/8/0/0)			
Nm: Confusing method name (3: 2/1/0/0)			
NP: Null pointer dereference (69: 17/52/0/0)			
Null pointer dereference (4: 4/0/0/0)			
NP / NP_ALWAYS_NULL			
A known null pointer is dereferenced here. This will lead to a NullPointerException when the code is executed.			
Possible null pointer dereference in method on exception path (14: 0/14/0/0)			
Method call passes null for unconditionally dereferenced parameter (13: 4/9/0/0)			
Non-virtual method call passes null for unconditionally dereferenced parameter (1: 1/0/0/0)			
Read of unwritten field (7: 0/7/0/0)			
NS: Suspicious use of non-short-circuit boolean operator (4: 4/0/0/0)			
RC: Suspicious reference comparison (8: 0/8/0/0)			
RCN: Redundant comparison to null (35: 10/25/0/0)			
RV: Bad use of return value from method (7: 3/4/0/0)			
SA: Useless self-operation (14: 12/2/0/0)			
SF: Switch case falls through (3: 3/0/0/0)			
UCF: Useless control flow (1: 1/0/0/0)			
UMAC: Uncallable method of anonymous class (6: 6/0/0/0)			
UR: Uninitialized read of field in constructor (5: 0/5/0/0)			
UwF: Unwritten field (40: 1/39/0/0)			
Multithreaded correctness (272: 1/271/0/0)			
Performance (1772: 7/1765/0/0)			
Dodgy (654: 52/602/0/0)			

Chapitre III : Gestion du code source

Section 9 : Analyse statique du code

- Audit de code source
- FindBugs
- PMD

Exécution

- en ligne de commande

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

- comme tâche ant ou plugin maven
- comme extension pour un IDE (Eclipse, Netbeans, ...)

Exemple

Extrait du résultat de PMD sur un projet SourceForge

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)'

Chapitre III : Gestion du code source

Section 10 : Gestion des versions/Travail de groupe

- Généralités
- Subversion
- Git
- Forge logicielle

Chapitre III : Gestion du code source

Section 10 : Gestion des versions/Travail de groupe

- Généralités
- Subversion
- Git
- Forge logicielle

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*, *OpenOffice*, *Microsoft Word*, ...)

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
```

Problèmes

- Le nombre de fichiers d'ancienne version augmentent rapidement
- Occupation disque importante
 - un changement mineur provoque la copie totale du fichier
- Sensible aux erreurs et aux oublis

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

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
- Supportent également les fichiers binaires mais l'intérêt diminue

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 \Rightarrow 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*)

Concepts I

Branche un ensemble de fichier qui subissent une évolution alternative (*branch* ou *fork*)

Changement une modification apportée à un document (*diff* ou *delta*)

Change set un ensemble de changements atomiques (*changeset* ou *patch*)

Checkout créer une copie de travail à partir du référentiel

Commit créer une nouvelle version dans le référentiel à partir des modifications faites sur la copie de travail (*commit* ou *checkin*)

Conflit le système ne peut pas fusionner les changements (*conflict*)

Concepts II

Copie de travail une copie d'une version des fichiers du référentiel
(*working copy*)

Head la version la plus récente

Référentiel l'endroit où sont conservés toutes les versions (*repository*)

Resolve résoudre un conflit détecter par le système

Tag une version nommée

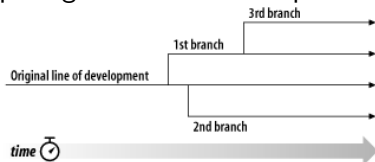
Tronc la ligne principal (*trunk*)

Update mettre à jour la copie locale à partir du référentiel

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



source : *Version Control with Subversion, 2nd ed.*

- Une branche peut ensuite être fusionnée avec une autre afin de reporter les modifications

Conflit

- Le modèle par fusion ne permet pas de traiter automatiquement tous les cas
- Certaines situations appelées *conflits* nécessitent une intervention humaine
 - ① Alice et Bob récupère une copie de travail d'un fichier
 - ② Alice modifie la première ligne et valide
 - ③ Bob supprime la première ligne et valide
 - ④ la validation de Bob échoue à cause d'un conflit
- Le VCS détecte, localise et signale le problème
- Le développeur doit alors résoudre le conflit avant de valider
- Nécessite de communiquer entre les membres de l'équipe pour minimiser ces situations

Marche à suivre recommandée

- ❶ Récupération d'une copie locale d'une version d'un projet (*checkout*)
- ❷ Modifications de la copie locale (indépendant du VCS)
- ❸ Récupération des mises à jour à partir du serveur (*update*)
- ❹ (Optionnel) Résolution des conflits éventuels au niveau de la copie locale
 - lors de l'*update* de la copie locale, les emplacements des conflits sont précisés
 - c'est au programmeur de les résoudre
- ❺ Envoi des modifications de la copie locale sur le serveur pour créer une nouvelle version (*commit*)

Architecture des VCS

- Les premiers VCS ne supportaient qu'un mode local
- Les VCS les plus répandus actuellement fonctionnent 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*)

VCS vs. DVCS

Limitations des VCS

- La fusion entre branches est difficile (maintenance de l'historique)
- Impossible d'envoyer des changements à un autre développeur sans passer par le dépôt central
- Validation hors-ligne impossible

Apports des DVCS

- Pas de dépôt centralisé (chaque copie de travail est un dépôt)
- Opérations déconnectées possibles
- Création/suppression de branches simplifiée
- Collaboration entre pair simplifiée

Désavantages des DVCS

- L'organisation des dépôts n'est pas imposé
- Paradigme nouveau et complexe

Ressources

- *Mercurial: The Definitive Guide*, **Bryan O'Sullivan**, O'Reilly Media, 2009
- *Essential CVS, Second Edition* , **Jennifer Vesperman**, O'Reilly, 2006

Quelques outils

- Mode local
 - **SCCS** puis **RCS** : anciens outils sous Unix
- Mode client/serveur
 - **CVS**
 - **SUBVERSION** : le remplaçant de CVS
- Mode réparti
 - **GIT** : utilisé pour le noyau Linux
 - **MERCURIAL**
 - **BAZAAR**
 - **DARCS**

Chapitre III : Gestion du code source

Section 10 : Gestion des versions/Travail de groupe

- Généralités
- Subversion
- Git
- Forge logicielle

Introduction

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

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)

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.domaine.fr/chemin/vers/repo`

Révisions I

- 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
```

Révisions II

- 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 I

- 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
```


Principales opérations II

- 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."
```

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

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 annuler 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
```

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 fusionner 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."
```

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

...

Ressources

- *Version Control with Subversion, 2nd ed.*, **C Pilato, Ben Collins-Sussman, Brian Fitzpatrick**, O'Reilly, 2008

Chapitre III : Gestion du code source

Section 10 : Gestion des versions/Travail de groupe

- Généralités
- Subversion
- **Git**
- Forge logicielle

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, ...)

Caractéristiques

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

Concepts

index (modifiable) conserve les modifications de la copie de travail avant la validation

object database (immuable) conserve les révisions

blob le contenu d'un fichier

tree l'équivalent d'un répertoire (décrit un instantané de l'arborescence de répertoires)

commit relie les objets *tree* en un historique

tag un conteneur qui référence un autre objet avec des méta-données

Quelques commandes

- Initialiser un dépôt/copie de travail
`git init`
- Prendre un instantané du répertoire courant et l'ajouter dans l'index
`git add .`
- Ajouter les changements au dépôt
`git commit -m "Description des changements."`
- Visualiser les modifications avant la validation
`git diff --cached`
`git status`
- Récupérer une copie d'un projet
`git clone /chemin/project projetlocal`

Ressources

- *Pragmatic Version Control Using Git*, **Travis Swicegood**, The Pragmatic Bookshelf, 2008
- *Version Control with Git*, **Jon Loeliger**, O'Reilly, 2009

Chapitre III : Gestion du code source

Section 10 : Gestion des versions/Travail de groupe

- Généralités
- Subversion
- Git
- Forge logicielle

Forge logicielle

- Une *forge logicielle* est un système de gestion de développement collaboratif
- Une forge intègre un ensemble d'outils
 - un système de gestion de version
 - un gestionnaire de liste de diffusion (ou de forum)
 - un outil de suivi de bogues
 - un gestionnaire de documentation (wiki par exemple)
 - ...

Quelques forges

GIT^{HUB}, BIT^{BUCKET}, GOO^{GLE} CO^{DE}, SOURCE^{FORGE}.

Chapitre IV : Construction et gestion des binaires

11 Gestion de la compilation

Chapitre IV : Construction et gestion des binaires

Section 11 : Gestion de la compilation

- Généralités
- GNU Make
- Ant
- Maven
 - Généralités
 - POM
 - Cycle de vie
 - Arborescence standard
 - Plugin et but
 - Archétype
 - Référentiel
 - Ressources
- Intégration continue

Chapitre IV : Construction et gestion des binaires

Section 11 : Gestion de la compilation

- Généralités
- GNU Make
- Ant
- Maven
 - Généralités
 - POM
 - Cycle de vie
 - Arborescence standard
 - Plugin et but
 - Archétype
 - Référentiel
 - Ressources
- Intégration continue

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 donc réalisées plus rapidement et sont moins sujettes aux erreurs
- Évite les fastidieuses lignes de commande
- Permet une compilation « intelligente » (n'exécute que ce qui est nécessaire)
- Peut être déclenché
 - à la demande l'utilisateur exécute un script
 - par un ordonnanceur déclenché à un instant donné
 - par un événement déclenché par un événement particulier

Quelques outils

• MAKE

- contrôle la génération d'exécutables à partir de fichiers sources
- un fichier *makefile* décrit les règles et les commandes pour générer les exécutables
- différentes versions (GNU notamment)

• APACHE ANT

- indépendant de la plate-forme (écrit en Java)
- basé sur un fichier XML
- peut être couplé à *Ivy* pour la gestion des dépendances entre module

• APACHE MAVEN

- outil de gestion et de compréhension de projet logiciel
- basé sur le concept de *modèle objet de projet* (*POM*)

Ressources

- La série « [Automation for the people](#) », developerWorks

Chapitre IV : Construction et gestion des binaires

Section 11 : Gestion de la compilation

- Généralités
- **GNU Make**
- Ant
- Maven
 - Généralités
 - POM
 - Cycle de vie
 - Arborescence standard
 - Plugin et but
 - Archétype
 - Référentiel
 - Ressources
- Intégration continue

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 \Rightarrow n'est mis à jour que ce qui est nécessaire
- Il exécute des commandes *shell* \Rightarrow n'est pas spécifique à un langage de programmation mais au système d'exploitation
- La configuration est fournie dans un *makefile*

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, c'est la première cible du Makefile qui est choisie

Makefile

- Un *makefile* est un fichier texte regroupant un ensemble de *règles* et de *déclaration*
- 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
- Chaque commande doit être précédée du caractère *tabulation*
- Un commentaire

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)
```

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 : %.c`

commande

- 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`

Exemple

Le Makefile «corrigé»

```
objects = main.o kbd.o command.o display.o \  
        insert.o  
  
edit : $(objects)  
    cc -o edit $(objects)  
  
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)
```

Cibles standards

`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

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 debogage

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

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) $<

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

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

Exemple III

Un Makefile plus sophistiqué pour Java

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

Ressources

- Site officiel GNU Make
- Manuel de GNU Make
- *Managing Projects with GNU make*, **Robert Mecklenburg**, O'Reilly, 2004
- *Compilez sous GNU/Linux !*, ?, Site du zéro (?)
- *Compilation séparée et Make*, **Benjamin Drieu**, April (1997)
- *Introduction à Makefile*, , Developpez.com (2005)
- Linux kernel Makefiles
 - une description de l'utilisation des *Makefiles* pour la compilation du noyau Linux

Chapitre IV : Construction et gestion des binaires

Section 11 : Gestion de la compilation

- Généralités
- GNU Make
- Ant
- Maven
 - Généralités
 - POM
 - Cycle de vie
 - Arborescence standard
 - Plugin et but
 - Archétype
 - Référentiel
 - Ressources
- Intégration continue

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)
- Fichier de configuration en XML
- Indépendant de la plate-forme

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

Fichier de configuration pour Ant

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

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

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"/>
```

Tâche

- Une tâche est un 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*

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**

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"/>
```

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>
```

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`)

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>
```

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>`

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>
```

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>
```

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>
```


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>
```

Exemple

Suppression des fichiers de compilation

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

Exemple

Exécution du programme

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

Ressources

- Le [site](#) officiel
- Le [wiki](#) officiel
- Le [manuel](#)
- Le [chapitre](#) du cours Java de Jean-Michel DOUDOUX
- La présentation *[Introduction to Ant](#)*
- Un [livre](#) sur WikiBooks
- Un [didacticiel](#)
- *[Top 15 Ant Best Practices](#)*, **Eric M. Burke**, [OnJava.com](#) (2003)
- [Site](#) officiel de Ivy
- *[Ant in Action](#)*, **Steve Loughran, Erik Hatcher**, Manning, 2007 (UVSQ : 005.13jav LOU)
- *[Ant: The Definitive Guide, 2ed](#)*, **Steve Holzner**, O'Reilly, 2005

Chapitre IV : Construction et gestion des binaires

Section 11 : Gestion de la compilation

- Généralités
- GNU Make
- Ant
- Maven
 - Généralités
 - POM
 - Cycle de vie
 - Arborescence standard
 - Plugin et but
 - Archétype
 - Référentiel
 - Ressources
- Intégration continue

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`

Concepts

POM Le *modèle objet 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 une des tâches proposées par un plugin

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

Modèle objet projet

- Le modèle objet projet est une description détaillée du projet
- Il contient
 - les versions et les configurations
 - les dépendances
 - les tests
 - ...
- Il est contenu dans un fichier XML nommé `pom.xml` placé dans le répertoire de base du projet

Exemple

Un exemple de POM (fichier pom.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<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>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Éléments 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

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>3.8.1</version>
```

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 *but*s (tâches spécifiques)

Exemple

Invocation de maven

- Vérifier l'installation de Maven

```
mvn --version
```

- Générer la distribution du projet (phase package du cycle default)

```
mvn package
```

- Supprimer les fichiers de compilation (phase clean du cycle clean)

```
mvn clean
```

- Générer un site web (phase site du cycle site)

```
mvn site
```

- Générer un squelette de projet (but create du plugin archetype)

```
mvn archetype:create \  
  -DarchetypeGroupId=org.apache.maven.archetypes \  
  -DgroupId=fr.uvsq.isty.isty1 \  
  -DartifactId=mon-projet
```

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

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`, ...

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

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

Ressources

- Le [site](#) officiel
- La [documentation](#)
- Un [site](#) pour rechercher dans le référentiel principal
- *Introduction to Apache Maven 2*, **Sing Li**, developerWorks (2006)
- [FAQ pour Maven 2 et Continuum \(en français\)](#), Eric Reboisson, 2007
- *Introduction à Maven 2*, **John Ferguson Smart**, **Denis Cabasson**, Developpez.com (2006)
- *Maven: The Definitive Guide*, **Sonatype Company**, O'Reilly, 2008
- *Better Builds with Maven*, **John Casey**, **Vincent Massol**, **Brett Porter**, **Carlos Sanchez**, Exist, 2008

Chapitre IV : Construction et gestion des binaires

Section 11 : Gestion de la compilation

- Généralités
- GNU Make
- Ant
- Maven
 - Généralités
 - POM
 - Cycle de vie
 - Arborescence standard
 - Plugin et but
 - Archétype
 - Référentiel
 - Ressources
- Intégration continue

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*

Pratiques recommandées I

- 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, ...)
 - limiter l'utilisation des branches (favoriser la branche principale)
 - ne placer aucun produit de la compilation dans le référentiel
- Automatiser les compilations
 - chaque tâche répétitive doit être automatisée (création de la BD, ...)
 - 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

Pratiques recommandées II

- Rendre les compilations auto-testantes
 - un ensemble de tests automatisés doit être disponible
 - la compilation doit inclure l'exécution des tests
 - l'échec d'un test doit être reporté comme un échec de la compilation
- Tout le monde valide chaque jour
 - des validations fréquentes favorisent une détection rapide des problèmes d'intégration
 - validation fréquente \Rightarrow moins d'endroits où les conflits peuvent se reproduire \Rightarrow détection plus rapide des problèmes
 - des validations fréquentes encouragent les développeur à découper leur travail en tâches

Pratiques recommandées III

- 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
- Maintenir une compilation courte
 - pour obtenir un feedback rapide
 - XP recommande un maximum de 10mn
- 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

Pratiques recommandées IV

- Rendre disponible facilement le dernier exécutable
 - chacun doit pouvoir utiliser la dernière version du système
- 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
- Automatiser le déploiement
 - la copie des exécutables dans les différents environnements doit être automatique
 - il peut être nécessaire de mettre aussi en place un mécanisme pour annuler un déploiement (en production par exemple)

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 fréquemment
- 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

Quelques outils

- **APACHE CONTINUUM**

- *Apache Continuum - A Detailed Look at the Open Source Continuous Integration System*, **Soumya Sinha**, Simple Thoughts (2008)

- **CABIE** (Continuous Automated Build and Integration Environment)

- **CRUISECONTROL**

- une [présentation](#) de CruiseControl
- *Cruise Control - Serveur d'intégration continue en JAVA*, **Loïc Mathieu**, Developpez.com (2008)

- **CRUISECONTROL.NET**

- *L'intégration continue avec CruiseControl.NET*, **Minosis**, Developpez.com (2005)

- **HUDSON**

- *L'intégration Continue avec Hudson*, **Romain Linsolas**, Developpez.com (2008)

- Un [tableau](#) comparatif des outils d'intégration continue

Ressources

- L'article fondateur de **Martin Fowler**
- *Continuous Integration anti-patterns*, **Paul Duvall**, developerWorks (2007) (1ère partie, 2ème partie)
- Le tutoriel *Spot defects early with Continuous Integration*, **Andrew Glover**, developerWorks (2007)
- *The Best Continuous Integration Tools*, **Vlad Kofman**, Gamelan (2009)
- *Continuous Integration. Improving Software Quality and Reducing Risk*, **Paul Duvall**, Addison-Wesley, 2007

Chapitre V : Mise au point de programmes

12 Assertions/programmation par contrat

13 Tests

14 Débogage

15 Optimisation et analyse dynamique

Chapitre V : Mise au point de programmes

Section 12 : Assertions/programmation par contrat

- Généralités
- Assertions en Java
- Programmation par contrat

Chapitre V : Mise au point de programmes

Section 12 : Assertions/programmation par contrat

- Généralités
- Assertions en Java
- Programmation par contrat

Intérêt des assertions

- Outil simple pour assister la réalisation de programmes corrects
- Normalement pas de surcoût dans la version de production
 - destinées aux versions de débogage
 - éliminées lors de la compilation pour les versions de productions

Définition et utilité I

- Une assertion est une expression booléenne exprimant une propriété sémantique (*prédicat*)
- L'assertion exprime que le développeur *croit* que ce prédicat est **toujours vrai à cet emplacement**
- C'est un outil pour exprimer et valider la correction d'une partie de code
- Une assertion permet de vérifier que ce que l'on croit « trivialement vrai » reste actuellement vrai

Définition et utilité II

- Plus une assertion est triviale, plus elle est utile !
 - lié à la théorie de l'information
 - la quantité d'information dans un événement décroît avec la probabilité qu'il survienne
 - si on débogue un programme sans assertion, on cherche d'abord le plus évident puis on va vers le plus improbable
- Apporte une aide pour différents aspects :
 - pour la construction de programme correct (*spécification*)
 - pour la documentation (*programmation par contrat*)
 - pour le débogage
 - pour le raisonnement

Utilisations

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

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

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

Ressources

- La [page](#) wikipedia
- *An axiomatic basis for computer programming*, **C.A.R. Hoare**, Communications of the ACM (1969)
- La présentation *Assert early, assert often*, **Hoare**, Microsoft Research (2002)
- *The benefits of programming with assertions*, **Philip Guo**, (2008)
- *Programming With Assertions*, **?**, SUN (2002)
- *Using assertions*, **,** JDC Tech Tips (2002)

Chapitre V : Mise au point de programmes

Section 12 : Assertions/programmation par contrat

- Généralités
- **Assertions en Java**
- Programmation par contrat

Syntaxe et fonctionnement

- Deux syntaxes

```
assert predicat;  
// ou
```

```
assert predicat : expression;
```

- Lors de l'exécution de l'assertion, si le prédicat est évalué comme faux, une exception `AssertionError` est lancée
- Dans la deuxième syntaxe, la valeur de l'expression est passée au constructeur de `AssertionError`

Activation/désactivation des assertions

- Les assertions peuvent être activées ou désactivées (défaut)
 - Désactivation par défaut pour des raisons de performance et de compatibilité
 - L'ajout du mot réservé `assert` en Java peut invalider certains programmes
 - Donc, par défaut, les assertions sont invalidées (à la compilation et à l'exécution)
- Quand elles sont désactivées, chaque assertion est équivalente à une instruction vide
- Comme elles peuvent être désactivées, **elles ne doivent pas être utilisées pour effectuer une tâche normale du programme**

Utilisation

- Les assertions sont disponibles depuis la version 1.4 du JDK (nouveau mot-clé `assert`)
- Nécessite l'ajout de l'option `-source 1.4` (ou ultérieure) lors de la compilation

```
javac -source 1.6 monpackage/MonProgramme.java
```

- Nécessite l'ajout de l'option `-enableassertions` (ou `-ea`) lors de l'exécution

```
java -ea monpackage.MonProgramme
```

Exemple

Attention aux nombres négatifs

```
if (unEntier % 3 == 0) {  
    // ...  
} else if (unEntier % 3 == 1) {  
    // ...  
} else {  
    assert unEntier % 3 == 2 : unEntier;  
    // ...  
}
```

Exemple

Seules les réponses 'O' et 'N' peuvent survenir cependant ...

```
switch (reponse) {  
  case 'O':  
    //...  
    break;  
  case 'N':  
    //...  
    break;  
  default:  
    assert false : reponse;  
}
```

Chapitre V : Mise au point de programmes

Section 12 : Assertions/programmation par contrat

- Généralités
- Assertions en Java
- Programmation par contrat

Définition

- La *programmation par contrat* (*design by contract* ou *Programming by contract*) est une approche pour la conception de logiciel
- Les interfaces des composants doivent être spécifiées de façon formelle et vérifiable
- Introduit par [Bertrand Meyer](#) pour le langage [Eiffel](#)
- Le terme *Design by contract* est déposé par [Interactive Software Engineering, Inc](#)
- Prend ses racines dans la vérification et la spécification formelle ainsi que dans la logique de Hoare

Qu'est-ce qu'un contrat ?

- Métaphore du contrat
 - le fournisseur doit fournir un certain produit (obligation) et sera payé pour cela (bénéfice)
 - le client doit payer pour le produit (obligation) et recevra le produit (bénéfice)
 - les deux parties doivent respecter certaines règles (lois, ...) comme dans tout contrat
- Une méthode fournissant une fonctionnalité peut
 - imposer certaines contraintes (pré-conditions)
 - garantir certaines propriétés en sortie (post-conditions)
 - maintenir certaines propriétés (invariant de classe)
- Le programme n'a pas besoin de vérifier les post-conditions durant l'exécution (différence par rapport à la *programmation défensive*)
- La programmation par contrat est lié à la notion de test

Exemple

Pré et post-condition pour la méthode put de la classe Dictionary (Eiffel)

```
put (x: ELEMENT; key: STRING) is
  -- Insert x so that it will be retrievable through key.
require
  count <= capacity
  not key.empty
do
  ... Some insertion algorithm ...
ensure
  has (x)
  item (key) = x
  count = old count + 1
end
```

Exemple

Invariant de la classe Dictionary (Eiffel)

```
class DICTIONARY [ELEMENT]
feature
    ... Interface specifications of other features ...
invariant
    0 <= count
    count <= capacity
end
```


Ressources

- *Conception et programmation orientées objet*, 2ème ed., **Bertrand Meyer**, Eyrolles, 2008 (UVSQ : 005.12 MEY)
- *Applying "Design by contract"*, **Bertrand Meyer**, IEEE Computer (1992)
- Building bug-free O-O software: An introduction to Design by Contract
- Deux interviews de B. Meyer :
 - *The Demand for Software Quality*, **Bill Venners**, Artima Developer (2003)
 - *Design by Contract*, **Bill Venners**, Artima Developer (2003)
- **CODE CONTRACTS** pour C# et .NET 4.0
- **CONTRACT4J** pour Java

Chapitre V : Mise au point de programmes

Section 13 : Tests

- Généralités
- Tests unitaires
- JUnit
- Couverture de code
- Développement piloté par les tests

Chapitre V : Mise au point de programmes

Section 13 : Tests

- Généralités
 - Tests unitaires
 - JUnit
 - Couverture de code
 - Développement piloté par les tests

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

Types de tests

- Un test *boîte blanche* (*white box*) s'appuie sur une connaissance de la structure interne de l'élément à tester
- Un test *boîte 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

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.

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 débiter le processus par les tests (*Développement dirigé par les tests*)

Quelques outils et frameworks

- Une **liste** d'outils
- Tests unitaires (voir section suivante)
- Tests de recette
 - **FIT** utilise des tableaux HTML pour vérifier automatiquement le comportement de programmes
 - **FITNESSE** utilise le même principe que Fit mais avec un wiki
- Tests non fonctionnels
 - **JUNITPERF** est une extension à JUnit pour la mesure de performance et le passage à l'échelle
 - **APACHE JMETER** permet de simuler une charge importante sur un élément à tester

Ressources

- SoftwareQATest.com Des ressources (FAQ, liens, ...) sur l'assurance qualité et sur les tests
- OpenSourceTesting.org Outils libres et informations pour les tests logiciels
- *Test Logiciel en pratique*, **John Watkins**, Vuibert, 2002 (UVSQ : 005.1 WAT)
- [Comité Français du Test Logiciel](#)
- [Testing Standards Working Party](#)
- *Seven Principles of Software Testing*, **Bertrand Meyer**, Eiffel Software (2008)

Chapitre V : Mise au point de programmes

Section 13 : Tests

- Généralités
- Tests unitaires
- JUnit
- Couverture de code
- Développement piloté par les tests

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*)

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

Caractéristiques des tests unitaires

- Petits (analyse d'un point précis) et rapides (exécutés souvent)
- Totalemt automatisés
- Toujours au niveau de l'unité
- Indépendants les uns des autres (pas de contraintes d'ordre)
- N'utilisent pas de ressources externes (SGBD, ...)

Doubleure 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 simplifiée
 - 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

Quelques outils et frameworks

- JUNIT
- TESTNG
- DBUNIT, ANYDBTEST et SQLUNIT pour les applications BD
- PHPUNIT pour PHP
- SUNIT est le framework original pour Smalltalk
- JMOCK
- EASYMOCK
- MOCKITO

Ressources

- *Simple Smalltalk Testing: With Patterns*, **Kent Beck**, First Class Software, Inc. (1994)
- *The Art Of Unit Testing*, **Roy Osherove**, Manning, 2007
- *XUnit Test Patterns*, **Gerard Meszaros**, Addison-Wesley, 2007
- *Mocks aren't stubs*, **Martin Fowler**, MartinFowler.com (2007)
- *Mock Roles, Not Objects*, **Steeve Freeman**, **Tim Mackinnon**, **Nat Pryce**, **Joe Walnes**, OOPSLA (2004)
- Le [site MockObjects.com](http://MockObjects.com)
- *Unit Testing Guidelines*, , GeoSoft (2007)
- *Advanced Unit Testing*, **Marc Clifton**, [The Code Project](http://TheCodeProject.com) (2003)
(1ère partie, 2ème partie, 3ème partie et 4ème partie)
- Un [site](#) sur les doublures de test

Chapitre V : Mise au point de programmes

Section 13 : Tests

- Généralités
- Tests unitaires
- JUnit
- Couverture de code
- Développement piloté par les tests

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
- Existents pour de nombreux langages (Java, C++, PHP, ...)
- JUnit a été développé par [Kent Beck](#) et [Erich Gamma](#)

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

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
 - ➊ créer l'instance de l'élément à tester ainsi que ses dépendances
 - ➋ appeler la méthode à tester avec les paramètres adéquats
 - ➌ comparer les résultats obtenus avec les résultats attendus avec des assertions
- **Les cas de tests doivent être indépendants les uns des autres** (pas de contraintes d'ordre, ...)

JUnit 4

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

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 {
    //...
}
```

Ecrire 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());
}
```

Assertions

- 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

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

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>();  
}
```

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

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

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.TestRunner` (GUI Swing) ou `junit.awtui.TestRunner` (GUI AWT)
- Un objet `TestSuite` est utilisé pour regrouper les tests

Ressources

- Le [site](#) officiel ([FAQ](#), [Cookbook](#))
- Une [carte](#) de référence JUnit/EasyMock
- *Tests unitaires automatisés avec JUnit4*, **Régis POUILLER**, Developpez.com (2009)
- Un [chapitre](#) du cours Java de Jean-Michel DOUDOUX
- [JUnit HOWTOs](#)
- *Easier testing with EasyMock*, **Elliott Rusty Harold**, developerWorks (2009)
- [JUnit A Cook's Tour](#) (version 3.8)
- [JUnit Test Infected: Programmers Love Writing Tests](#) (version 3.8)
- *Conception de tests unitaires avec JUnit*, **Romain Guy**, Developpez.com (2006) (version 3.8)

Chapitre V : Mise au point de programmes

Section 13 : Tests

- Généralités
- Tests unitaires
- JUnit
- Couverture de code
- Développement piloté par les tests

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
- 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 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

Quelques outils

- COBERTURA
- EMMA
- CLOVER
- GROBOUTILS

Ressources

- *Introduction to Code Coverage*, **Lasse Koskela**, JavaRanch (2004)
- *Code Coverage Analysis*, **Steve Cornett**, Bullseye Testing Technology (2008)

Chapitre V : Mise au point de programmes

Section 13 : Tests

- Généralités
- Tests unitaires
- JUnit
- Couverture de code
- 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*)

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

Ressources I

- *Test Driven Development: By Example*, **Kent Beck**, Addison-Wesley Professional, 2003
- *Test Driven: Practical TDD and Acceptance TDD for Java Developers*, **Lasse Koskela**, Manning, 2007 (UVSQ : 005.13 KOS)
- *Introduction to Test Driven Design*, **Scott W. Ambler**, AgileData.org ()
- *Introducing Behaviour-Driven Development (BDD)*, **Dan North**, DanNorth.net (2006)
- *Test-driven design*, **Neal Ford**, developerWorks (2009) (1^{re}partie, 2^epartie)
- Le site TestDriven.com
- *Growing Object-Oriented Software, Guided by Tests*, **Steve Freeman**, **Nat Pryce**, , 2009

Ressources II

- Tutoriel *Développement dirigé par les tests : mise en pratique*, **David Boissier**, Developpez.com ()
- Tutoriel *Développement Dirigé par les Tests*, **Bruno Orsier**, Developpez.com (2007)
- *Test-Driven Requirements*, **Gilles Mantel**, Developpez.com (2008)
- *Refactoring: Improving the Design of Existing Code*, **Martin Fowler, Kent Beck**, Addison-Wesley Professional, 1999
- *Refactoring To Patterns*, **Joshua Kerievsky**, Addison Wesley, 2004

Chapitre V : Mise au point de programmes

Section 14 : 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

Pourquoi utiliser un débogueur ?

Pour gagner du temps !

Alternatives

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

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

Processus de débogage

- ❶ Tenter de reproduire le bogue
- ❷ Simplifier les entrées du programme
- ❸ Exécuter le programme sous le contrôle du débogueur
- ❹ Se positionner à l'endroit de l'erreur signalée ou au milieu du programme (pose d'un breakpoint) si aucune erreur n'est signalée
- ❺ Examiner le contexte : confirmer que les variables possèdent les valeurs attendues
- ❻ 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`)

Quelques outils

- **GDB/DDD**
 - *GNU Debugger* et une GUI pour gdb (*Data Display Debugger*)
- **VALGRIND**
 - infrastructure pour le développement d'outils d'analyse dynamique
 - inclus plusieurs outils (détection d'erreurs mémoires, ...)
- **JDB**
 - le débogueur du JDK (en ligne de commande)
 - la plateforme Java fournit une architecture pour construire des débogueurs (*Java™ Platform Debugger Architecture* ou *JPDA*)
- **JSWAT**
 - débogueur graphique basé sur JPDA
- **JDEBUGTOOL** (commercial)
- BlueJ ou un autre IDE
 - la plupart des IDE possèdent un débogueur intégré

Ressources

- *Debug It!: Find, Repair, and Prevent Bugs in Your Code*, **Paul Butcher**, The Pragmatic Bookshelf, 2009
- *The Art of Debugging with GDB, DDD, and Eclipse*, **Norman Matloff, Peter Jay Salzman**, O'Reilly, 2008
- *The Developer's Guide to Debugging*, **T. Grötter, U. Holtmann, H. Keding, M. Wloka**, Springer, 2008 ([site compagnon](#))
- *Why Programs Fail: A Guide to Systematic Debugging*, **Andreas Zeller**, Morgan Kaufmann, 2006 ([site compagnon](#))

Chapitre V : Mise au point de programmes

Section 15 : Optimisation et analyse dynamique

- Introduction
- HProf

Chapitre V : Mise au point de programmes

Section 15 : Optimisation et analyse dynamique

- Introduction
- HProf

Généralités I

- L'*optimisation* est la pratique qui consiste à modifier un système pour qu'il fonctionne plus efficacement ou en consommant moins de ressources
- L'*analyse dynamique (profiling)* d'un programme a pour objectif de collecter des informations sur le comportement d'une application pendant son exécution
- Les éléments à surveiller sont l'utilisation des CPU, l'utilisation de la mémoire, les *threads*, ...
- Ce type d'analyse a un impact sur le comportement de l'application
- Un outil d'analyse dynamique permet donc de collecter et de présenter ces informations

Généralités II

- Utilisé pour l'analyse de performances, un tel outil permet de localiser les « points chauds » (*hot spots*) du programme
- L'optimisation est souvent un compromis entre différents facteurs
- La phase d'optimisation ne doit intervenir qu'une fois que le programme fonctionne et répond aux spécifications fonctionnelles

A propos de l'optimisation prématurée

- *More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity., W.A. Wulf*
- *We should forget about small efficiencies, say about 97% of the time : premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%., Donald Knuth*
- *Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you have proven that's where the bottleneck is., Rob Pike*
- *The First Rule of Program Optimization : Don't do it. The Second Rule of Program Optimization (for experts only!) : Don't do it yet., Michael A. Jackson*

Optimisation à différents niveaux

- Conception
 - algorithmes, architecture de l'application, ...
- Code source
 - utilisation d'idiomes adaptés au langage
 - **attention de ne pas perturber le compilateur**
- Compilateur
 - utiliser les optimisations fournies par le compilateur
- Assembleur
 - spécifique à une plateforme
- Exécution
 - compilateur *just in time*

Marche à suivre pour l'optimisation

- ① Choisir un paramètre à optimiser (temps CPU, occupation mémoire, ...)
- ② Localiser les portions de code les plus coûteuse vis à vis de ce paramètre
 - permet d'obtenir le meilleur rendement
 - règle des 80/20
- ③ Appliquer les optimisations puis mesurer le résultat

Quelques outils

- GNU Profiler ([GPROF](#))
- [HPROF/JHAT](#) (SUN/JDK)
- [ECLIPSE TEST AND PERFORMANCE TOOLS PLATFORM \(TPTP\) PROJECT](#)
- [NETBEANS PROFILER](#)
- [EXTENSIBLE JAVA PROFILER](#)
- [VTUNE](#) (Intel)
- [JPROFILER](#) (ej-technologies)
- [JPROBE](#) (Quest Software)
- Un [article](#) sur les outils d'analyse de performances

Chapitre V : Mise au point de programmes

Section 15 : Optimisation et analyse dynamique

- Introduction
- HProf

Introduction

- HProf est l'outil fourni avec Java SE pour le profiling
- Permet différentes analyses
 - des temps CPU
 - de l'occupation mémoire
- Les résultats peuvent être générés en format texte ou binaire
- Le format binaire peut être importé par d'autres outils pour analyse (JHat par exemple)
- HProf est basé sur l'interface *Java Virtual Machine Tool Interface (JVM TI)*

Utilisation

- HProf est invoqué en ligne de commande directement avec la machine virtuelle

```
java -agentlib:hprof[=options] ToBeProfiledClass
```

```
# ou
```

```
java -Xrunhprof[:options] ToBeProfiledClass
```

```
# pour l'aide
```

```
java -agentlib:hprof=help
```

- Les options permettent de préciser l'analyse à effectuer
- Par défaut, le résultat de l'analyse est stocké dans le fichier `java.hprof.txt`

Types d'analyse

- Mémoire dynamique (*heap*)
 - profils d'allocation (*heap=sites*)
 - instantané complet (*heap=dump*)
- CPU
 - utilisation du CPU (échantillonnée) (*cpu=samples*)
 - utilisation du CPU (mesure) (*cpu=times*)

JHat

- JHat (*Heap Analysis Tool*) est outil expérimental fourni avec le JDK
- Analyse les fichiers *dump* de la mémoire dynamique d'un programme
- JHat démarre un serveur web permettant
 - de naviguer dans l'instantané
 - d'exécuter des requêtes prédéfinies
 - d'exécuter des requêtes OQL libres
- OQL est un langage de type SQL permettant d'interroger le *dump*

Exemple

Profil d'allocation en mémoire dynamique (*Heap Allocation Profiles*)

...

TRACE 300008:

java.util.Arrays.copyOf(Arrays.java:2882)

java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:1

java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:390)

java.lang.StringBuilder.append(StringBuilder.java:119)

...

rank	percent		live		alloc'ed		stack	class
	self	accum	bytes	objs	bytes	objs		
1	74,90%	74,90%	277213672	7639	900662648	29999	300008	char[]
2	21,75%	96,65%	80519800	3507	100140000	5000	300204	char[]
3	3,09%	99,74%	11454880	303	100140000	5000	300198	char[]
4	0,05%	99,80%	196280	3505	280000	5000	300202	char[]
5	0,04%	99,84%	147648	11	147648	11	300207	char[]
6	0,04%	99,87%	140280	3507	200000	5000	300203	java.lang.String
7	0,03%	99,91%	112128	3504	160000	5000	300201	java.lang.StringB

Exemple

Echantillonnage de l'utilisation CPU

...

TRACE 300025:

java.util.Arrays.copyOf(Arrays.java:2882)

java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:1

java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:390)

java.lang.StringBuilder.append(StringBuilder.java:119)

...

rank	self	accum	count	trace	method
------	------	-------	-------	-------	--------

1	27,87%	27,87%	17	300025	java.util.Arrays.copyOf
---	--------	--------	----	--------	-------------------------

2	13,11%	40,98%	8	300034	TestHprof.makeStringInline
---	--------	--------	---	--------	----------------------------

3	9,84%	50,82%	6	300026	java.util.Arrays.copyOfRange
---	-------	--------	---	--------	------------------------------

4	8,20%	59,02%	5	300040	TestHprof.makeStringWithLocal
---	-------	--------	---	--------	-------------------------------

5	8,20%	67,21%	5	300041	TestHprof.makeStringWithLocal
---	-------	--------	---	--------	-------------------------------

6	6,56%	73,77%	4	300032	java.util.Arrays.copyOfRange
---	-------	--------	---	--------	------------------------------

Exemple

Mesure de l'utilisation CPU

...

TRACE 300038:

java.util.Arrays.copyOf(Arrays.java:Unknown line)

java.lang.AbstractStringBuilder.expandCapacity(AbstractStringBuilder.java:U

java.lang.AbstractStringBuilder.append(AbstractStringBuilder.java:Unknown l

java.lang.StringBuilder.append(StringBuilder.java:Unknown line)

...

rank	self	accum	count	trace	method
------	------	-------	-------	-------	--------

1	27,61%	27,61%	30001	300038	java.util.Arrays.copyOf
---	--------	--------	-------	--------	-------------------------

2	5,47%	33,08%	5000	301033	java.util.Arrays.copyOfRange
---	-------	--------	------	--------	------------------------------

3	4,65%	37,74%	5000	300993	java.util.Arrays.copyOfRange
---	-------	--------	------	--------	------------------------------

4	4,59%	42,33%	10000	301030	java.lang.AbstractStringBuilder.append
---	-------	--------	-------	--------	--

5	3,90%	46,23%	10000	300990	java.lang.AbstractStringBuilder.append
---	-------	--------	-------	--------	--

6	3,65%	49,87%	10000	301042	java.lang.AbstractStringBuilder.append
---	-------	--------	-------	--------	--

Ressources

- Un [article](#) sur HProf
- La documentation de HProf se trouve dans le livre [Advanced Programming for the Java 2 Platform](#)

Chapitre VI : Bibliothèques et frameworks

16 Bibliothèques

17 Frameworks

Chapitre VI : Bibliothèques et frameworks

Section 16 : Bibliothèques

Quelques modules de la bibliothèque standard

`java.lang.reflect` API de réflexion (manipulation des structures internes du langage) ([tutoriel](#))

`java.util.concurrent` Support de la programmation parallèle (multi-thread) ([tutoriel](#))

`java.beans` Support de l'architecture *JavaBeans* pour le développement de composants ([tutoriel](#))

`java.lang.annotation` pour les annotations

Apache Commons

- Le projet Apache *Commons* fournit divers composants utilitaires
- Quelques exemples de bibliothèque
 - CLI* analyse les arguments en ligne de commande
 - Codec* implémente divers encodeurs/décodeurs (Base64, ...)
 - Collections* étend la bibliothèque standard de collection (ne supporte pas les génériques)
 - Configuration* permet d'accéder à plusieurs formats de fichiers de configuration
 - Lang* étend les fonctionnalités de `java.lang`
 - Logging* fournit une API de journalisation
 - Math* fournit des utilitaires mathématiques/statistiques
 - Primitives* permet la manipulation efficace des types primitifs

GUI

[JFreeChart](#) pour réaliser des diagrammes, histogrammes, ...

[JGraph](#) pour la visualisation de graphes

[SWT/JFace](#) bibliothèques de composants UI utilisées par Eclipse

[Java Power Tools](#) bibliothèques pour le développement de GUI

Divers

- Journalisation
 - Apache Commons Logging
 - Log4J
 - `java.util.logging`
- Collections
 - Apache Commons collections
 - Google Collection Library
- Autres
 - Joda Time manipulation de dates et d'heures

Chapitre VI : Bibliothèques et frameworks

Section 17 : Frameworks

Frameworks

- [Swing Application Framework](#) un framework « simple » pour développer des applications Swing ([doc](#))
- [Jt](#) un framework orienté patterns