POLYTECH MONTPELLIER

Projet transverse

Outillage

Auteur: John Tranier

Plan

1. Introduction aux tests unitaires
2. **JUnit** (tests unitaires)
3. **Mockito** (mocks)
4. **GitHub** (hébergement & gestion projet logiciel)
5. **Git** (gestion de version)

Tests unitaires

Les types de tests

- Test unitaire
- Test d’intégration
- Test fonctionnel
- Test de charge
- Recette
- Et bien d’autres (accessibilité, conformité W3C, …)

Test unitaire : Principe

- Isoler chaque « unité » d'un programme
  et vérifier qu'elle fonctionne comme prévu.

- Définir le contrat que l'unité de code doit satisfaire.
  - Ce contrat est « exécutable » ==> **Automatisation**.

- En POO : unité = la classe

A quoi servent les tests ?

- Assurer que le composant fonctionne comme prévu
- Mais pas que ça …
  - Assurer la **non régression**
  - Permettre le **refactoring**
  - Permettre de **comprendre** le code
  - Améliorer l'**architecture**
- Le code devient **explicité**, **organisé**, **assuré**

Limites

- Ne garanti pas le bon fonctionnement de l'application
- Evaluation partielle
- Un test peut contenir des bugs

Un test unitaire doit être …

- **Isolé** (indépendant de tous les autres tests)
- **Déterministe** (i.e. reproductible)
- **Simple** et **rapide** à développer et à exécuter
- **Exécutable automatiquement**
- Limité à une unité de code **isolée**

Physionomie d’un test

1. **Given …**

   Construction de « l’état du monde »

2. **When …**

   Exécution du code testé

3. **Then …**

   Vérification du résultat obtenu (assertions, métaphore de l’oracle)

Que faut-il tester ?

- **Cas en succès** : fonctionnement normal
- **Cas d'erreur** : test sur la gestion d'erreur
- **Cas aux limites** : test de la robustesse

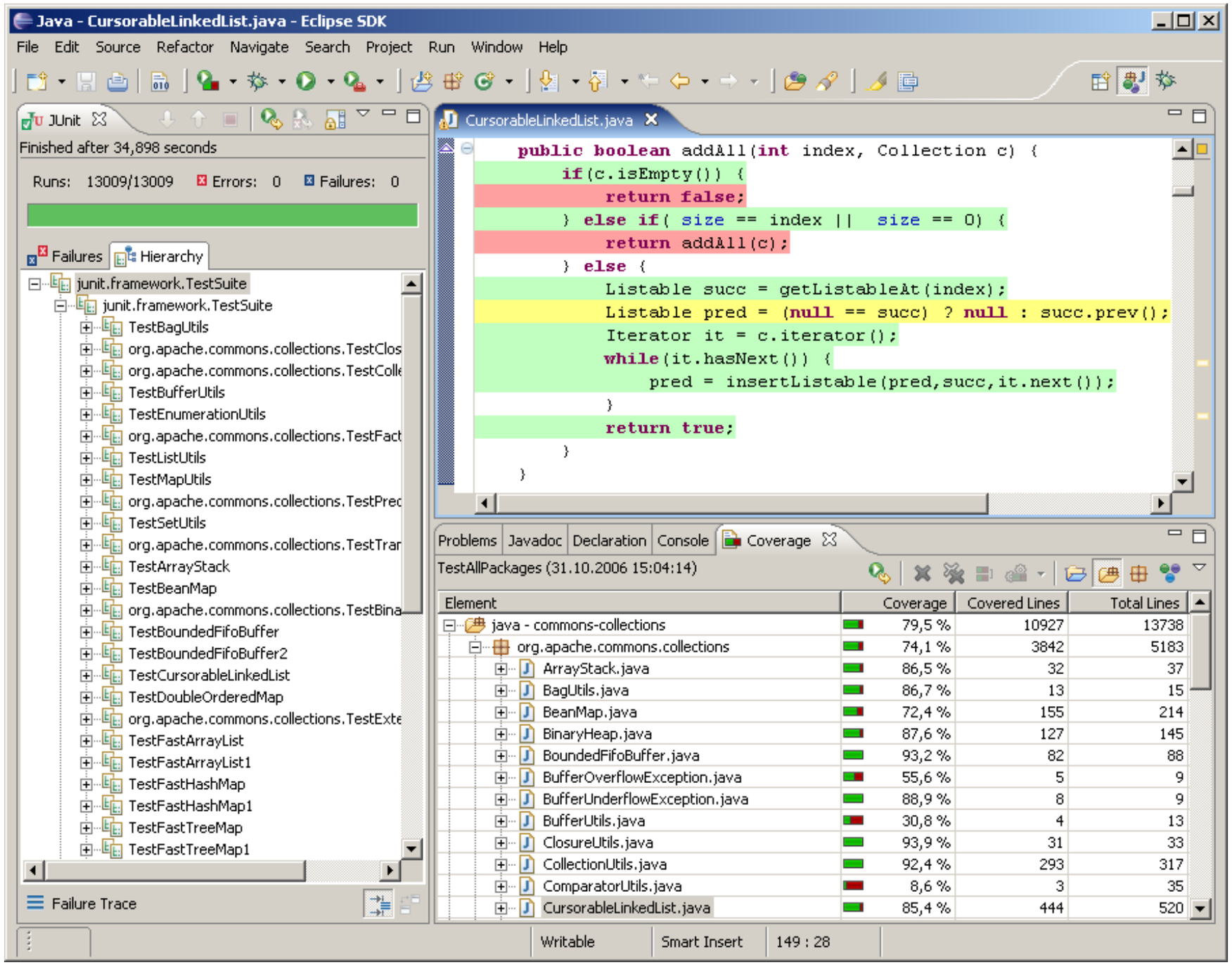
Le résultat d'un test

- **Success** : test réussi
- **Failure** : au moins une assertion est violée
- **Error** : erreur inattendue à l'exécution

Aspects méthodologiques

- Pour pouvoir tester unitairement :
  - Les composants doivent être séparables
  - Les composants doivent être simples
- Si on se retrouve à debugger à coup de « println » il vaut mieux écrire un test à la place
- Quand on trouve un bug, écrire un test qui le caractérise

Couverture de code



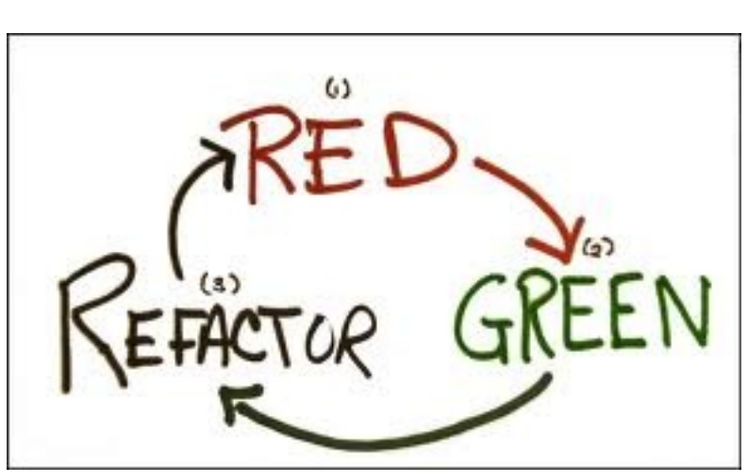
Intégration continue

- Automatisation de la construction d'un projet et répétition en continue
  (comprend l'exécution des tests)

- Objectif : détecter au plus tôt les anomalies

- Outils : Jenkins

Test-Driven Development (**TDD**)



TDD

- Initialement conceptualisé par *Erich Gamma* et *Kent Beck*

- Utilisation des tests unitaires comme spécification du code

- Plus généralement intégré aux approches de développement **agile** (eXtreme Programming, Scrum)

Cycle de TDD

1. Ecrire un test
2. Vérifier qu'il échoue (aucune implémentation)
3. Ecrire juste le code suffisant pour faire passer le test
4. Vérifier que le test passe
5. Réviser le code (*refactoring*), i.e. l'améliorer en conservant les mêmes fonctionnalités

Avantages du TDD

- On utilise le programme avant même qu'il existe (utilisation de l'interface avant même son implémentation)
- Diminue les erreurs de conception
- Construction conjointe du programme et de la suite de tests de non-régression
- Estimer l'état d'avancement du projet


JUnit.org

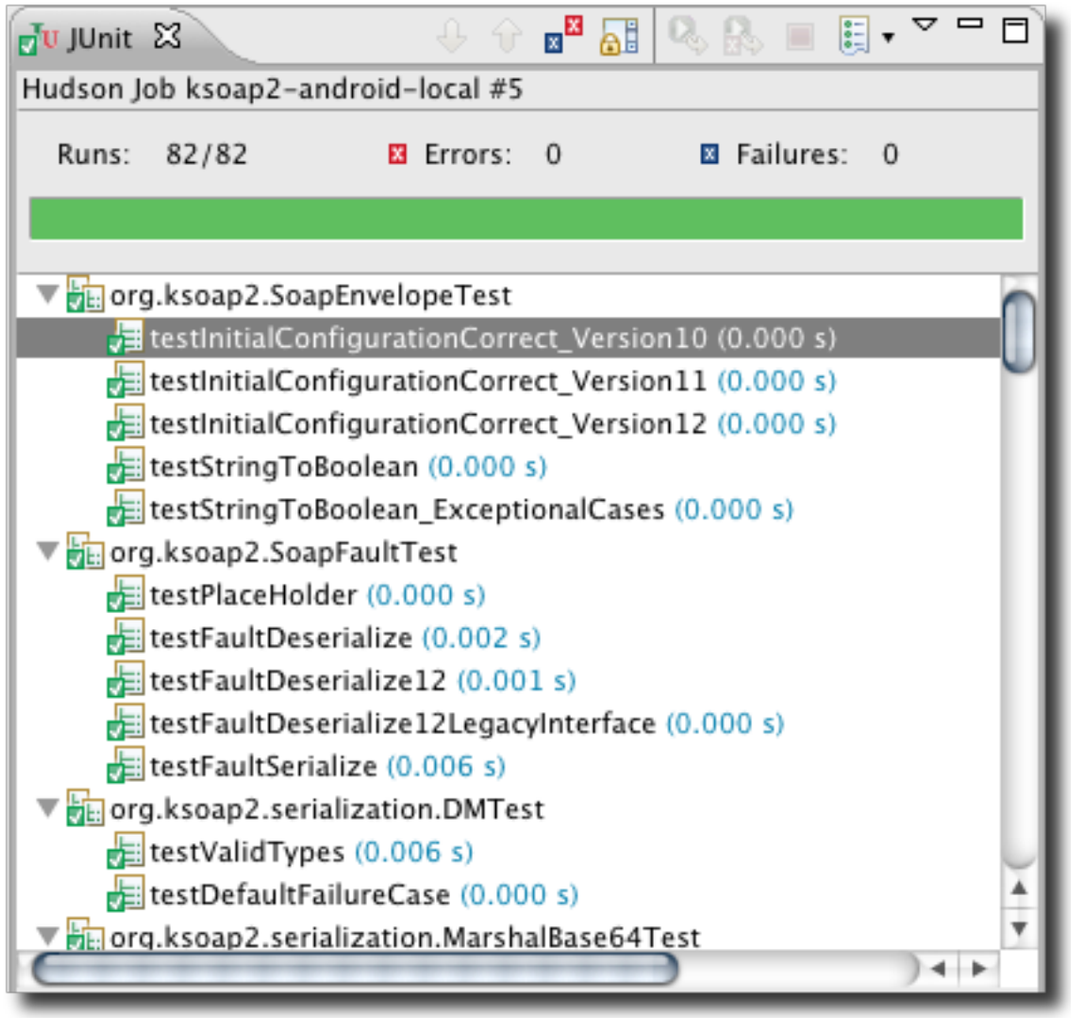
JUnit 4

- Framework pour les tests unitaires en Java
- Open Source (https://github.com/junit-team/junit)
- Intégration dans Eclipse & Maven
- Massivement utilisé dans l'industrie

Ce que JUnit offre

- Facilités pour la conception de tests unitaires
- Lancement automatique de suites de tests
- Génération de rapports d'exécution

Rapport


JUnit
Hudson Job ksoap2-android-local #5
Runs: 82/82
Errors: 0
Failures: 0
org.ksoap2.SoapEnvelopeTest
testInitialConfigurationCorrect_Version10 (0.000 s)
testInitialConfigurationCorrect_Version11 (0.000 s)
testInitialConfigurationCorrect_Version12 (0.000 s)
testStringToBoolean (0.000 s)
testStringToBoolean_ExceptionalCases (0.000 s)
org.ksoap2.SoapFaultTest
testPlaceHolder (0.000 s)
testFaultDeserialize (0.002 s)
testFaultDeserialize12 (0.001 s)
testFaultDeserialize12LegacyInterface (0.000 s)
testFaultSerialize (0.006 s)
org.ksoap2.serialization.DMTest
testValidTypes (0.006 s)
testDefaultFailureCase (0.000 s)
org.ksoap2.serialization.MarshalBase64Test

JUnit par l'example

- Une interface *Stack*
- Une 1ère implémentation vide *SimpleStack*
- Un test unitaire *SimpleStackTest*

```java
public interface Stack {

  /**
   * Tests if this stack is empty
   */
  public boolean isEmpty();

  /**
   * Returns the number of items in this stack.
   */
  public int getSize();

  /**
   * Pushes an item onto the top of this stack.
   * null item is allowed.
   */
  public void push(Item item);

  /**
   * Looks at the object at the top of this stack without removing it from the stack.
   */
  public Item peek() throws EmptyStackException;

  /**
   * Removes the object at the top of this stack and returns that object as the value of this
function.
   * @throws EmptyStackException if this stack is empty.
   */
  public Item pop() throws EmptyStackException;

}
```

```java
public class SimpleStack implements Stack {

    @Override
    public boolean isEmpty() {
        // TODO Auto-generated method stub
        return false;
    }

    @Override
    public int getSize() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public void push(Item item) {
        // TODO Auto-generated method stub

    }

    @Override
    public Item peek() throws EmptyStackException {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public Item pop() throws EmptyStackException {
        // TODO Auto-generated method stub
        return null;
    }
}
```

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

public class SimpleStackTest { // Test class

    @Test
    public void testCreateEmptyStack() { // Test case

        // Code under test
        Stack stack = new SimpleStack();

        // Assertions (oracle)
        assertTrue("A new stack should be empty", stack.isEmpty());
        assertEquals("A new stack has no element", 0, stack.getSize());
    }

    @Test
    public void testPush() throws EmptyStackException {

        // Setup the "state of the world"
        Stack stack = new SimpleStack();
        Item item = new SimpleItem();

        // Code under test
        stack.push(item);

        // assertions (oracle)
        assertFalse("The stack must be not empty", stack.isEmpty());
        assertEquals("The stack constains 1 item", 1, stack.getSize());
        assertSame("The pushed item is on top of the stack", item, stack.peek());
    }
}
```

```java
@Test(expected = EmptyStackException.class)
public void testPopOnEmptyStack()  throws EmptyStackException {
    // Setup the "state of the world"
    Stack stack = new SimpleStack();

    // Code under test
    stack.pop(); // should throws an EmptyStackException.
}
```

JUnit: terminologie

- **Test class** : contient plusieurs cas de tests (portant sur une même classe)
- **Test method** : un cas de test (portant sur une méthode)
- **Assertion** : expression dont on veut vérifier la véracité
- **Test fixture** : construction d'un état commun à plusieurs cas de test
- **Test suite** : regroupement de classes de test que l'on peut exécuter ensemble

Les assertions

Instruction	Description
`fail(message)`	fait échouer la méthode de test
`assertTrue([message], condition)`	teste si la condition est vraie
`assertEquals([message], expected, actual)`	teste si les valeurs sont les mêmes
`assertSame(([message], expected, actual)`	teste si les variables référencent le même objet
`assertNotSame([message], expected, actual)`	teste si les variables ne référencent pas le même objet
`assertNull([message], object)`	Teste si l'objet est null
`assertNotNull([message], object)`	Teste si l'objet est non null

Une assertion non vérifiée lève une exception

Les annotations

Instruction	Description
`@Test`	Méthode de test
`@Before`	Méthode exécutée avant chaque test de la classe
`@After`	Méthode exécutée après chaque test de la classe
`@BeforeAll`	Méthode exécutée avant le premier test de la classe
`@AfterAll`	Méthode exécutée avant le dernier test de la classe

Aspects pratiques

- Une classe de test par classe de programme

`<nomClasse>Test`

- Une ou plusieurs méthodes de test par méthode de la classe

`test<nomMethode>`
`[<cas>][<attendu>]`

- Fichiers sources des classes de tests séparées des classes du programme (/src/* et test/*)

`/src/…`
`/test/…`


mockito

Comment tester une classe de manière isolée ?

Qu’est-ce qu’un Mock ?

- Mock = objet factice
- Les mocks sont des objets simulés qui reproduisent le comportement d’objets réels de manière contrôlée
- Remarque :
  - Le terme « Mock » est souvent utilisé de manière générique ; on peut distinguer les mocks, les spies, les stubs, les fakes, les dummies

Principe

- Un mock a la même interface que l'objet qu'il simule
- L'objet client ignore qu'il interagit avec un objet réel ou un objet simulé
- La plupart des frameworks de mock permettent
  - De spécifier quelles méthodes vont être appelées, avec quels paramètres, et dans quel ordre
  - De spécifier les valeurs retournées par le mock

Mockito

- Framework de mock pour Java
- http://code.google.com/p/mockito/
- Focalisé sur
  - la définition du comportement
  - la vérification après l'exécution

Aspects pratiques

- import static org.mockito.Mockito.*
- @RunWith(MockitoJUnitRunner.class)

Création de Mock

- A partir d'une classe ou d'une interface

```
public class MyTest {

  MyInterface mock1 = mock(MyInterface.class);
  MyClass mock2 = mock(MyClass.class);

  @Mock
  MyClass mock3;
}
```

- Comportement par défaut

```
  @Test
  public void testMyMock() {
    assertEquals(0, mock.getInt());
    assertEquals(false, mock.getBoolean());
    assertEquals(0, mock.getList().getSize());
  }
```

Description du comportement

- Méthode avec un type de retour

```
public class MyTest {

  @Test
  public void testDefineBehaviour() {
    when(mock.getInt()).thenReturn(3);
    assertEquals(3, mock.getInt());
    assertEquals(3, mock.getInt()); // Can be repeated

    when(mock.getInt()).thenReturn(3, 4);
    assertEquals(3, mock.getInt());
    assertEquals(4, mock.getInt());
    assertEquals(4, mock.getInt()); // Can be repeated

    when(mock.getInt()).thenReturn(3).thenReturn(4); // Equivalent to previous notation

    when(mock.getInt()).thenThrow(new Exception());
    try {
      mock.getInt();
      fail();
    }
    catch(Exception e) {
      // Exception is expected
    }
  }

}
```

Description du comportement

- Méthode sans un type de retour

```
@Test
public void testDefineBehaviourForVoid() {
  doThrow(new Exception()).when(mock).voidMethod();

  try {
    mock.voidMethod();
    fail();
  }
  catch(Exception e) {
    // Exception is expected
  }
}
```

Description du comportement

- Méthode avec paramètre(s)

```
when(mock.add(1,2)).thenReturn(3);

assertEquals(3, mock.add(1,2));
assertEquals(0, mock.add(3,4)); // add(3,4) was not stubbed
```

```java
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;
import org.junit.runner.RunWith;

@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {

  UserService userService;
  EncryptionService encryptionService;

  @Before
  setup() {
    encryptionService = mock(EncryptionService.class);
    userService = new UserService();
    userService.encryptionservice = encryptionService; // Dependency injection
  }

  @Test
  testRegisterUser() {
    String login = "login";
    String password = "password";

    // Stub password encryption
    String encryptedPassword = "encrypted password";
    given(encryptionService.encrypt(password)).thenReturn(encryptedPassword);

    // When ...
    User user = userServive.registerUser(login, password);

    // Then ...
    assertNotNull(user);
    assertEquals(login, user.login);
    assertEquals(encryptedPassword, user.password);
  }
}
```

Vérification des interactions

- Méthode appelée 1 seule fois

```
verify(mock).getInt();
verify(mock, times(1)).getInt(); // equivalent notation
```

- Méthode appelée au plus ou au moins N fois

```
verify(mock, atLeastOnce()).getInt();
verify(mock, atMost(2)).getInt();
```

- Méthode jamais appelée

```
verify(mock, never()).getInt();
```

Vérification des interactions

- Une exception sera levée si une vérification échoue

- Il est possible de contrôler l'ordre des interactions (voir la documentation)

Matchers

- Mockito vérifie les arguments en utilisant *equals*
- Assez souvent, on veut spécifier un appel (*when* ou *verify*) sans définir la valeur précise des arguments
- On utilise pour cela des matchers

```
when(mock.get(anyInt())).thenReturn(3);
```

- Remarque : Si on utilise des matchers, tous les arguments doivent être des matchers

Les principaux Matchers

- any()
- anyInt(), anyDouble(), anyString(), …
- anyList(), anyMap(), anyCollection(), …
- Et bien d’autres (cf. documentation)

Ce que l'on ne peut pas mocker

- Constructeur
  - Ne pas instantier d'object avec new à l'extérieur d'une factory
  - Remplacer par l'injection de dépendance
- Eléments statiques (blocs, méthodes)
  - Ne pas utiliser « static » pour des comportements que l'on veut pouvoir débrancher dans les tests (contre exemple : une méthode statique qui écrit dans un fichier)


github
SOCIAL CODING

GitHub

- Hébergement de projet sous GIT
- Fonctionnalités de type « réseaux sociaux »
  - Les flux
  - Suivi de personnes ou de projets
  - Les graphes de réseaux pour les dépôts
- Pour chaque dépôt
  - Un wiki
  - Un logiciel de suivi d'anomalies & d'évolutions
  - Une page web de présentation du projet
- Un *pastebin* nommé Gist

Gist

SimpleStack.java — Java

```java
public class SimpleStack implements Stack {

	@Override
	public boolean isEmpty() {
		// TODO Auto-generated method stub
		return false;
	}

	@Override
	public int getSize() {
		// TODO Auto-generated method stub
		return 0;
	}

	@Override
	public void push(Item item) {
		// TODO Auto-generated method stub

	}

	@Override
	public Item peek() throws EmptyStackException {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public Item pop() throws EmptyStackException {
		// TODO Auto-generated method stub
		return null;
	}

}
```

Intégration avec Eclipse

- Le plugin EGit
- Tutoriel installation :
  - http://benjsicam.me/blog/how-to-setup-eclipse-git-plugin-egit-for-github-part-1-tutorial/

Aide GitHub

https://help.github.com


git

Découvrir GIT

http://try.github.com/

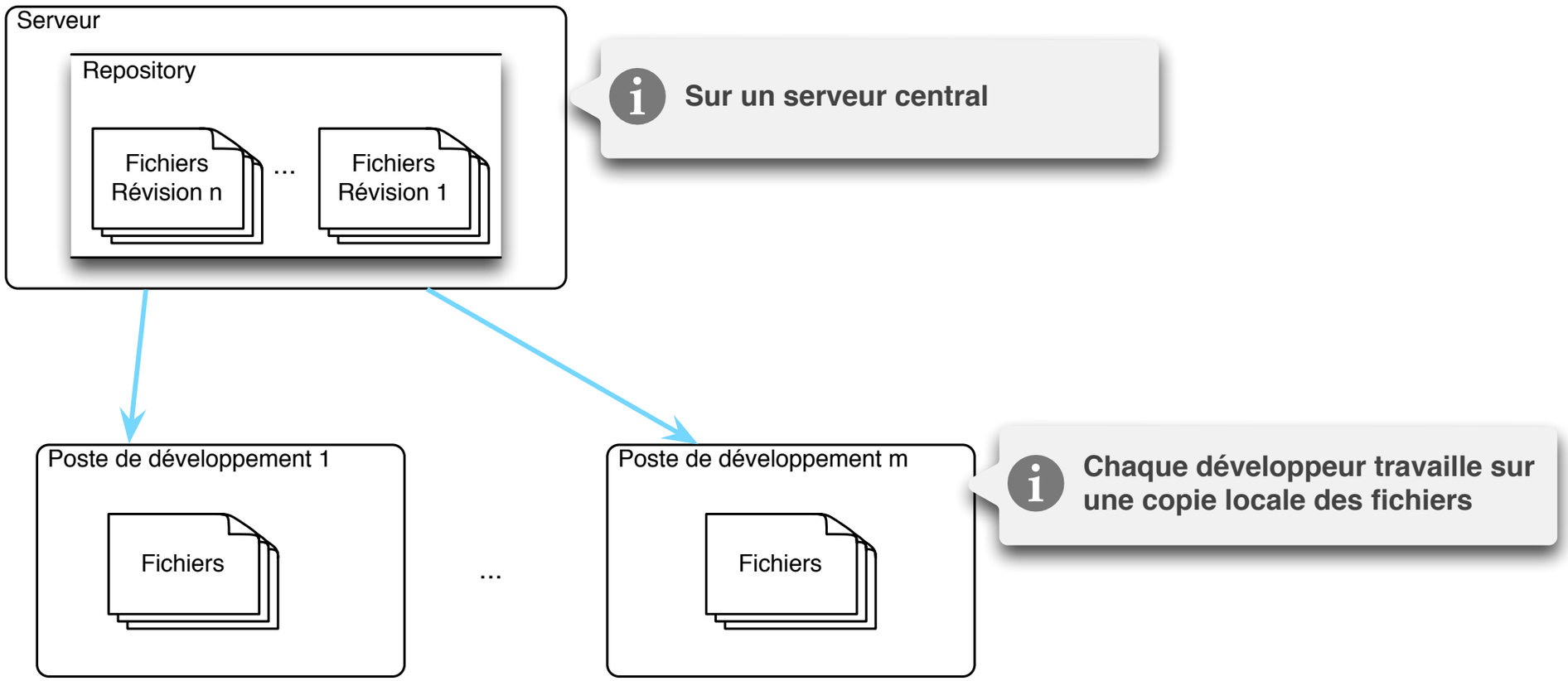
Version Control System (VCS)

- Gestion de version de code de source
  - Permet de gérer les changements sur le code source (révision)
  - Prise en compte des modifications provenant de plusieurs contributeurs
  - Possibilité de retrouver à tout moment le code source correspondant à une révision donnée

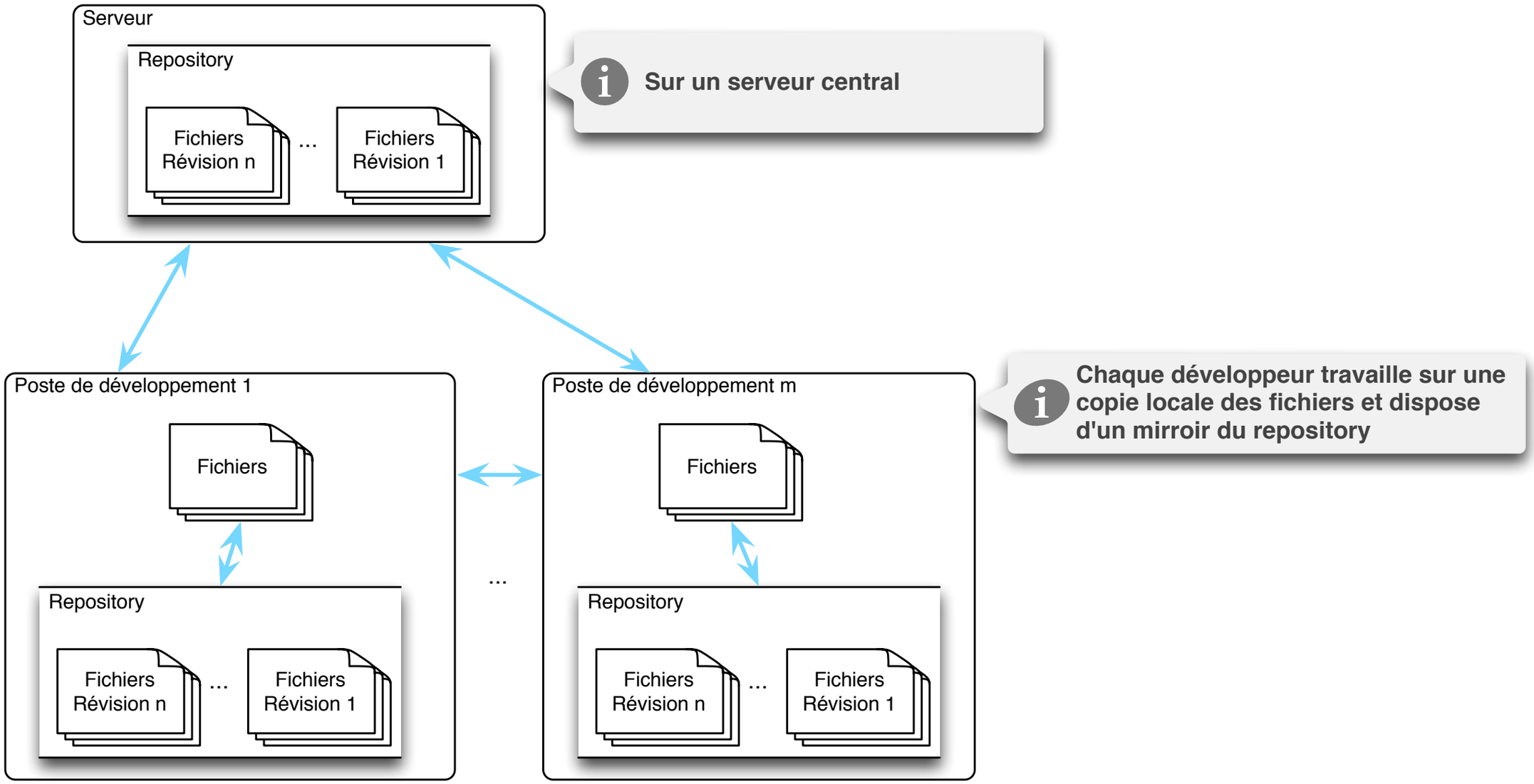
Historique

- Développement collaboratif du noyau Linux
  - changement de politique commerciale de BitKeeper
  - création d'un nouvel outil s'impose
- Avril 2005 : 1er commit de Linus Thorvald
- Aujourd'hui, utilisé par …
  - Git, Linux, Eclipse, Perl, Gnome, KDE, Ruby on Rails, Android, PostgreSQL, Debian, X.org, Grails, …

VCS: centralisé (ex SVN)



VCS Distribué (ex GIT)



Opérations principales

- Cloner un projet existant (*clone*)
- Initialiser un nouveau projet (*init*)
- Modifier le code source (*add*, *rm*, *commit*, *revert*)
- Brancher (branch, checkout)
  Fusionner (merge, mergetool)
- Synchroniser des dépôts (push, fetch, pull)
- Visualiser les changements (log, diff)

Cloner un dépôt existant

```
$ git clone git@github.com:FranckSilvestre/MonPremierProjetGithub.git
Cloning into 'MonPremierProjetGithub'...
remote: Counting objects: 31, done.
remote: Compressing objects: 100% (11/11), done.
remote: Total 31 (delta 7), reused 31 (delta 7)
Receiving objects: 100% (31/31), done.
Resolving deltas: 100% (7/7), done.
```

- Permet d'obtenir une copie intégrale d'un dépôt existant

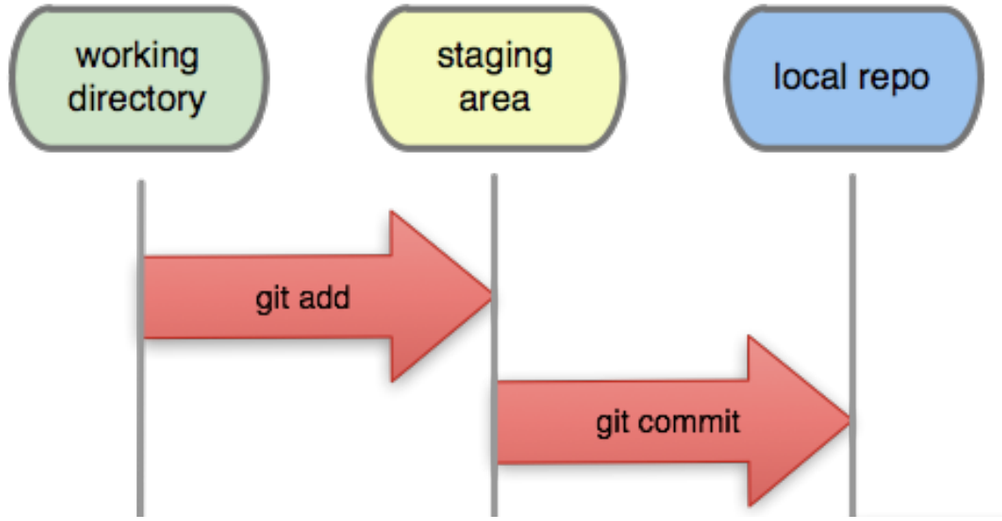
Initialiser un dépôt

```
mkdir MonProjet2
cd MonProjet2/
git init
Initialized empty Git repository in /Users/fsil/10_Dev/Other/Bacasable/MonProjet2/.git/
```

- Initialiser un dépôt GIT
  - Création dossier caché .git
- On peut initialiser un dépôt sur un projet existant

Modifier le code source

1. Modifier le code source
    - Ajouter / Supprimer / Modifier des fichiers
2. Sélectionner des fichiers dans l'aire d'embarquement
3. « Commiter »



Commiter

- Valider une révision du code source
- Un commit, c'est :
  - Un identifiant unique
  - Une date
  - Un auteur
  - Un message de commit (description)
  - Un nouvel état du code source
  - Un commit parent (ou 2)

Illustration: création de fichiers dans le workspace

```
MonProjet fsil$ groovy ../createFiles
MonProjet fsil$ ls
Fichier_0.txt    Fichier_2.txt    Fichier_4.txt    Fichier_6.txt    Fichier_8.txt
Fichier_1.txt    Fichier_3.txt    Fichier_5.txt    Fichier_7.txt    Fichier_9.txt
```

Illustration: visualiser « l'état des fichiers »

```
MonProjet fsil$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Fichier_0.txt
#       Fichier_1.txt
#       Fichier_2.txt
#       Fichier_3.txt
#       Fichier_4.txt
#       Fichier_5.txt
#       Fichier_6.txt
#       Fichier_7.txt
#       Fichier_8.txt
#       Fichier_9.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Illustration:
ajouter des fichiers

```
MonProjet fsil$ git add Fichier_0.txt
MonProjet fsil$ git add Fichier_1.txt
MonProjet fsil$ git status
# On branch master
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   Fichier_0.txt
#       new file:   Fichier_1.txt
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Fichier_2.txt
#       Fichier_3.txt
#       Fichier_4.txt
#       Fichier_5.txt
#       Fichier_6.txt
#       Fichier_7.txt
#       Fichier_8.txt
#       Fichier_9.txt
```

Illustration: commit

```
MonProjet fsil$ git commit -m'Premier commit !'
[master (root-commit) f331ccf] Premier commit !
 2 files changed, 2 insertions(+), 0 deletions(-)
 create mode 100644 Fichier_0.txt
 create mode 100644 Fichier_1.txt
```

```
MonProjet fsil$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       Fichier_2.txt
#       Fichier_3.txt
#       Fichier_4.txt
#       Fichier_5.txt
#       Fichier_6.txt
#       Fichier_7.txt
#       Fichier_8.txt
#       Fichier_9.txt
nothing added to commit but untracked files present (use "git add" to track)
```

Le statut des fichiers

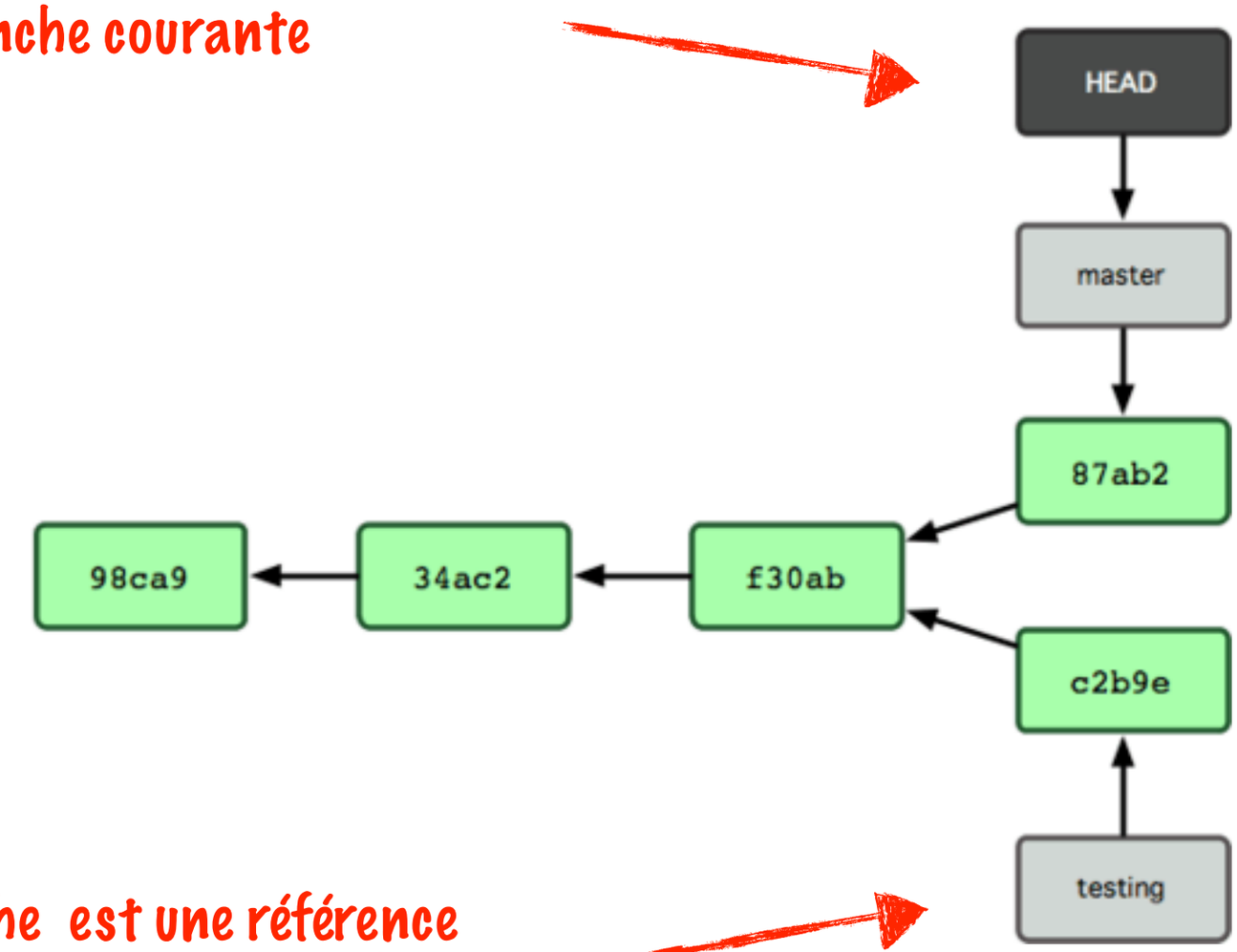
- *tracked* : fichiers « suivis » par GIT
  - Dans l'aire d'embarquement ou dans le *repository*
- *untracked* : fichiers inconnus pour GIT
  - Uniquement dans le *workspace*
- *ignored* : fichiers ignorés par GIT
  - Typiquement les fichiers de build et les fichiers propres à l'environnement local (ex: config IDE)
  - Définis par le(s) fichier(s) .gitignored

Branche

- Les branches permettent d'isoler des modifications (histoires parallèles)
- Usages :
  - Nouvelles fonctionnalités
  - Préparation d'une release (correction de bugs)
  - Maintenir différentes versions

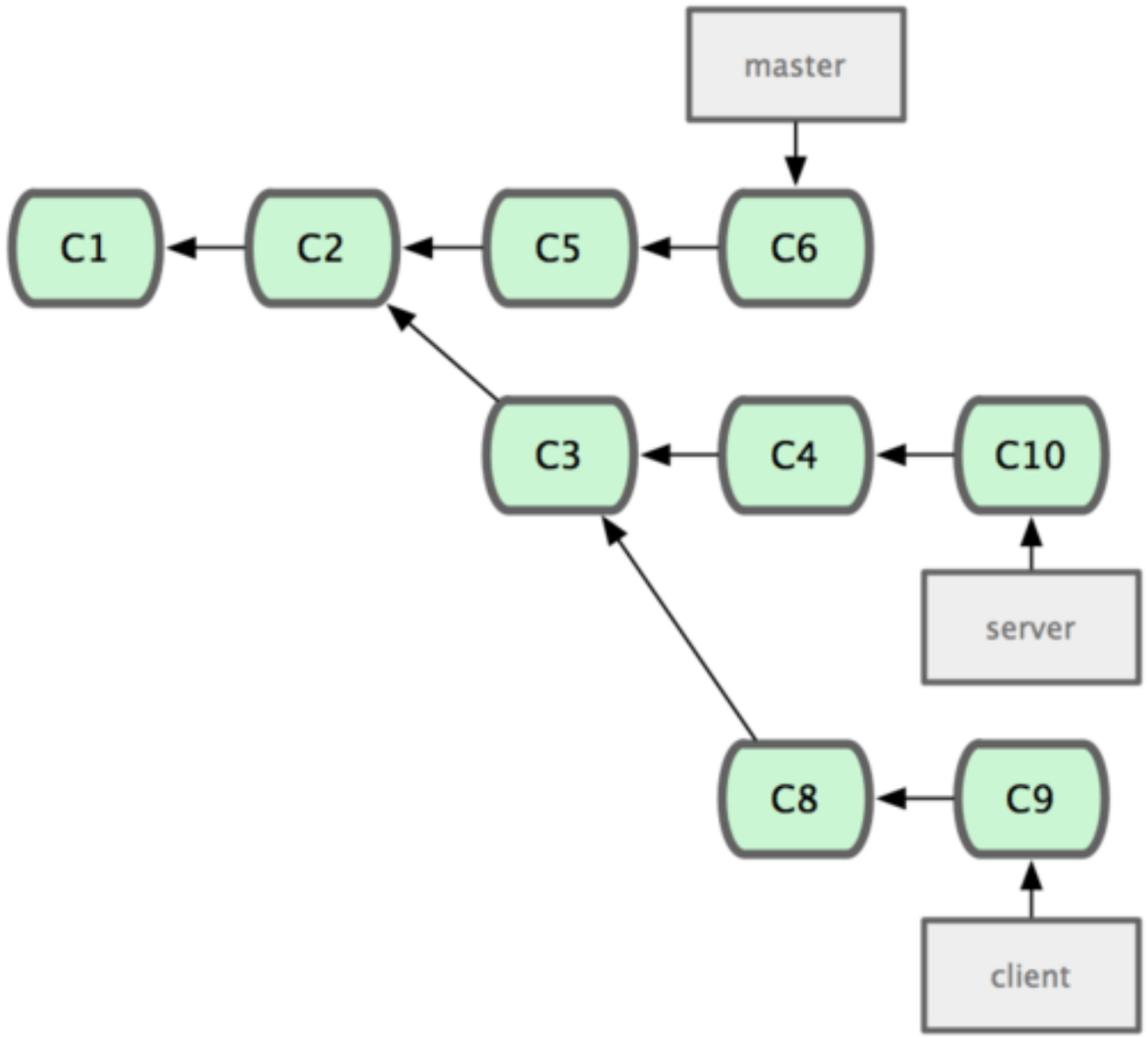
Le graphe des commits

HEAD est une référence sur la branche courante



Une branche est une référence sur un commit stockée dans un fichier du dossier refs/head

Plusieurs branches …



Créer une branche

- La création de branche avec GIT
  - Uniquement une référence sur un commit
  - Facile et rapide => à utiliser massivement

```
MonProjet fsil$ git branch dev/198
```

  - Création d'une branche nommée « dev/198 » à partir de « HEAD »

```
MonProjet fsil$ git checkout -b dev/198
```

  - Création d'une branche et changement de HEAD

Lister les branches

```
MonProjet fsil$ git branch
  dev/198
* master
```

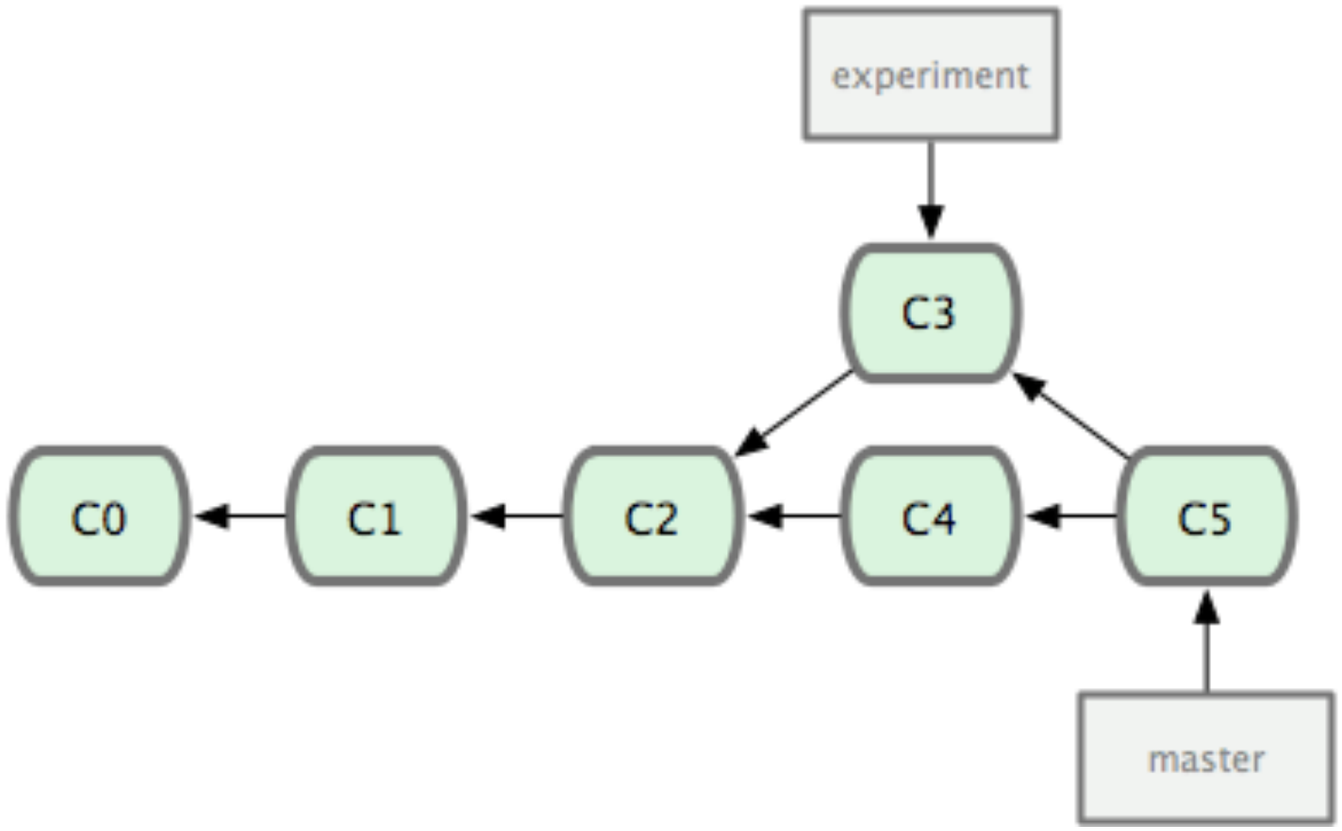
Changer de branche

```
MonProjet fsil$ git checkout dev/198
```

- Change la branche courante (HEAD)

- Charge le code source correspondant à la branche dans le workspace

Fusionner des branches

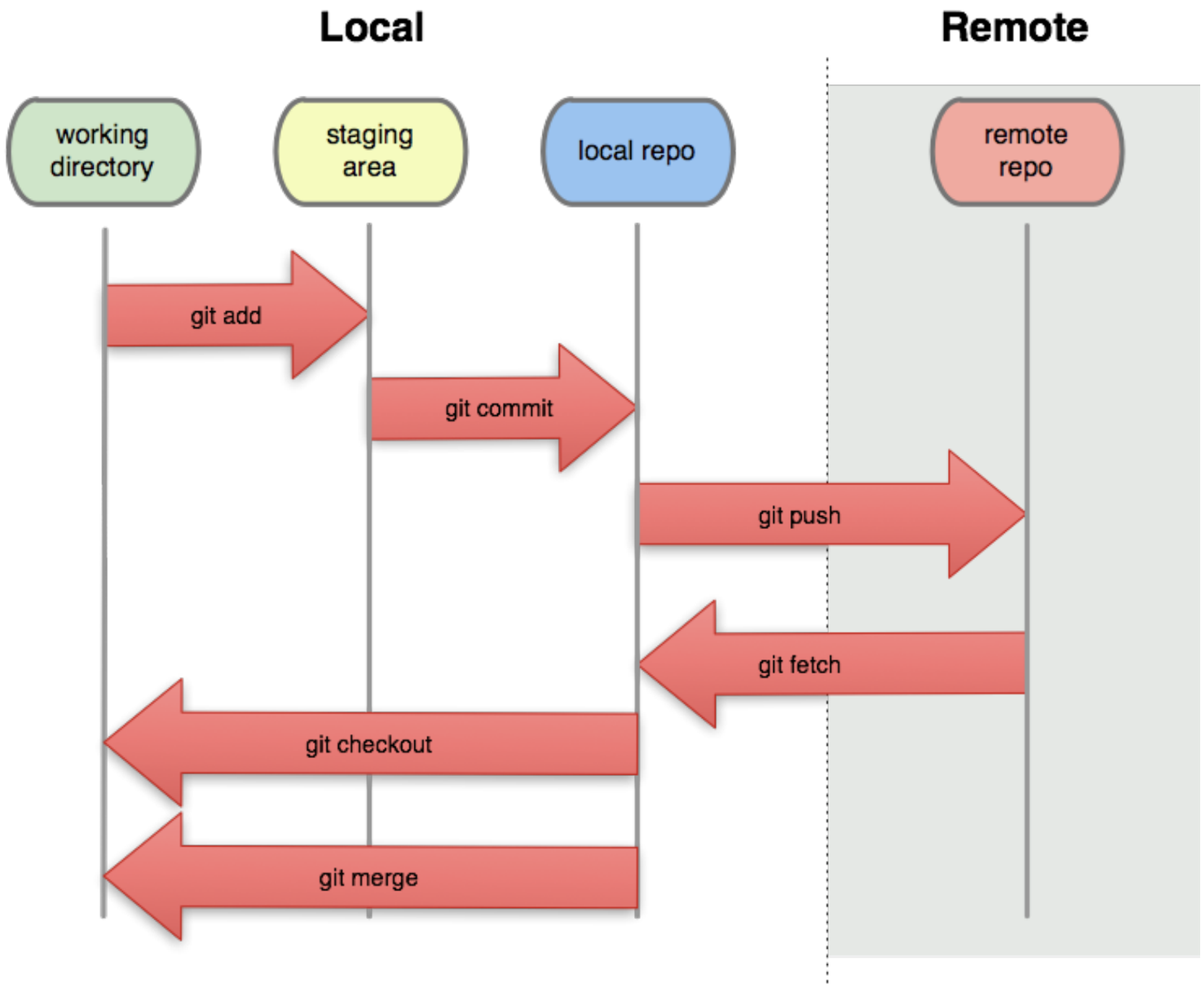


Merge

```
MonProjet fsil$ git merge dev/198
```

- L’opération merge
  - La branche courante fusionne avec la branche référencée
  - Si la branche courante est un « ancêtre » de la branche à fusionner => *fast-forward* (simple mise à jour de référence)
  - Si les branches ont divergées => *commit de merge*
  - En cas de conflit => résolution manuelle avec *git mergetool*

Synchroniser des dépôts



Les dépôts distants

- Un « remote » est un alias sur un repository distant
- Un repository local peut déclarer autant de « remotes » qu'il le souhaite

```
$ git remote add origin git@github.com:FranckSilvestre/MonPremierProjetGithub.git
$ git remote -v
origin  git@github.com:FranckSilvestre/MonPremierProjetGithub.git (fetch)
origin  git@github.com:FranckSilvestre/MonPremierProjetGithub.git (push)
```

Synchroniser un dépôt distant

```
$ git fetch origin
remote: Counting objects: 216, done.
remote: Compressing objects: 100% (53/53), done.
remote: Total 176 (delta 99), reused 142 (delta 84)
Receiving objects: 100% (176/176), 17.61 KiB | 0 bytes/s, done.
Resolving deltas: 100% (99/99), completed with 22 local objects.
From ssh://git.cloudbees.com/ticetime/elaastic
 * [new branch]      vter/AsynchronousEmails/EL-25/1 -> origin/vter/
AsynchronousEmails/EL-25/1
 * [new branch]      vter/ResendActivationEmail/EL-23/2 -> origin/vter/
ResendActivationEmail/EL-23/2
```

- Met à jour l'image des branches distantes du repository local
  - Les branches sur le repository distant (tracking branches) sont dissociées des branches sur le repository local.

Les branches distantes

```
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/dev/198
  remotes/origin/master
```

Synchroniser localement une branche

1. Synchroniser le dépôt distant
   - git fetch origin

2. Merger la branche distante avec la branche locale
   - git checkout master
   - git merge origin/master

- Alternative au merge : « rebase »
  - Voir http://git-scm.com/book/en/Git-Branching-Rebasing

Push

- Publier l'histoire d'une branche locale sur une branche distante

- L'histoire de la branche distant doit être « compatible » (sinon il faut synchroniser au préalable)

```
$ git push origin dev/198
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 314 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:FranckSilvestre/MonPremierProjetGithub.git
   5d1cf0b..f0cb3e4  dev/198 -> dev/198
```

Visualiser les changements

- Historique :
  - git log
  - git log --oneline --graph --decorate
- Visualiser les changements d'un commit
  - git show
  - git whatchanged
- Comparer des révisions
  - git diff (différences entre le workspace et la staging area)
  - git diff master dev/123 (différences entre deux branches)
  - git diff HEAD HEAD~3 (différences entre 2 commits)

Et bien d’autres choses …

```
git
usage: git [--version] [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p|--paginate|--no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           [-c name=value] [--help]
           <command> [<args>]

The most commonly used git commands are:
   add        Add file contents to the index
   bisect     Find by binary search the change that introduced a bug
   branch     List, create, or delete branches
   checkout   Checkout a branch or paths to the working tree
   clone      Clone a repository into a new directory
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   fetch      Download objects and refs from another repository
   grep       Print lines matching a pattern
   init       Create an empty git repository or reinitialize an existing one
   log        Show commit logs
   merge      Join two or more development histories together
   mv         Move or rename a file, a directory, or a symlink
   pull       Fetch from and merge with another repository or a local branch
   push       Update remote refs along with associated objects
   rebase     Forward-port local commits to the updated upstream head
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
   show       Show various types of objects
   status     Show the working tree status
   tag        Create, list, delete or verify a tag object signed with GPG

See 'git help <command>' for more information on a specific command.
```

Liens utiles pour la pratique

- http://git-scm.com/book/
- http://gitref.org/

Références

- Pour concevoir ce support, je me suis inspiré des sources suivantes dont j'ai copié et / ou adapté certains éléments :
  - https://github.com/FranckSilvestre/SupportsCours (F. Silvestre)
  - http://fr.slideshare.net/IsenDev/test-unitaire-28714183
  - http://deptinfo.unice.fr/twiki/pub/Minfo/GenieLog1213/cours3-GL-minfo-1213.pdf (P. Collet)
  - http://www.emse.fr/~picard/cours/2A/junit/junit.pdf (G. Picard)
  - http://dpt-info.u-strasbg.fr/~blansche/files/ogl_cours_1.pdf
  - http://www.lifl.fr/~nebut/ens/vl/fichiers/coursTest/coursJUnit.pdf (M. Nebut)
  - http://fr.slideshare.net/CedricGatay/tests-unitaires (C. Gatay)
  - http://fr.slideshare.net/fwendt/using-mockito (F. Wendt)

- Bien évidemment, le contenu du présent support n'engage que moi !