Gestion de Projet

Introduction et Outils

Introduction

Contenu du cours

- Pas du tout un cours complet de Gestion de projet
- Quelques Outils (Maven - Git - Trello ....)
- Quelques pratiques (Tests - CI ...)
- Les rendus attendus (Code - Rapport)

Support

Le projet !

Projet - Cahier des charges

Le projet

- Projet à réaliser par groupe de 3
- Cahier des charges fournis
- 2 rendus attendus :
  - 1er rendu, commun à tout le monde : l'application de base
  - 2ème rendu : spécifique à chaque groupe
- Deadlines
  - 1er rendu : semaine du 13 Mars
  - 2ème rendu : le vendredi 14 Avril à 14h

Projet - Cahier des charges

Le projet

- Projet à réaliser par groupe de 3
- Cahier des charges fournis
- 2 rendus attendus :
  - 1er rendu, commun à tout le monde : l'application de base
  - 2ème rendu : spécifique à chaque groupe
- Deadlines
  - 1er rendu : semaine du 13 Mars
  - 2ème rendu : le vendredi 14 Avril à 14h

Les rendus

Au prochain cours ! (Il faut commencer par les outils)

La base : un éditeur

IDE

- L'éditeur doit permettre d'écrire le code, mais pas que ⇒ Integrated Development Environnement
- Le but d'utiliser un IDE puissant est donc de ne pas faire "que" écrire le code avec
- Il faut prendre le temps d'apprendre à l'utiliser efficacement (Cheat Sheet)

Lequel ?

Dans ce cours nous proposons d'utiliser VScode, et notamment LiveShare pour le travail collaboratif, mais rien n'est obligatoire

Comment gérer un projet

Qu'est-ce qu'un projet ?

- Du code

Comment gérer un projet

Qu'est-ce qu'un projet ?

- Du code
- Des dépendances
- Des tests
- Des compilations
- De la documentation
- Des Releases
- Du déploiement
- etc

Comment gérer un projet

Qu'est-ce qu'un projet ?

- Du code
- Des dépendances
- Des tests
- Des compilations
- De la documentation
- Des Releases
- Du déploiement
- etc

Gardez en tête de rester fénéant !

Les phases du cycle de vie

Maven

- **validate** vérifie que le projet est correct et que tout ce qui est nécessaire est présent
- **compile** compile les sources du projet
- **test** teste les code source compilé en utilisant un framework de tests unitaires
- **package** prend le code compilé et le package dans un format distribuable

Les phases du cycle de vie

Maven

- **integration-test** effectue les tests d'intégration (après un éventuel déploiement dans un environnement où ils peuvent être effectué)
- **verify** effectue toute vérification pour s'assurer que le package est valide et satisfait les critères de qualité
- **install** installe le package locallement, pour une utilisation en tant que dépendances dans d'autres projets locaux
- **deploy** copie le package final dans la cible distante, le rendant accessible aux autres développeurs et projets.

Gestion des dépendances

à l'aide des coordonnées Maven

- groupId
- artifactId
- version (peut être vide mais c'est déconseillé)

`https://search.maven.org`
`https://mvnrepository.com`

```
<dependencies>
    <dependency>
        <groupId>org.boofcv</groupId>
        <artifactId>boofcv-core</artifactId>
        <version>0.39.1</version>
    </dependency>
</dependencies>
```

Project Object Model

```
<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.x
<modelVersion>4.0.0</modelVersion>

<groupId>pdl.l3</groupId>
<artifactId>part1</artifactId>

<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>Mon Super Projet</name>
<url>http://siteduprojet.org</url>
        .
        .
</project>
```

Project Object Model

Le POM

C'est un peu verbeux : `build`, `plugin`, `scope`, `parent`, ...

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Project Object Model

Multi-modules

- Il faut packager le module principal en tant que "pom"
- le repertoire racine contient le pom.xml
- on spécifie dedans les sous modules :

```
<modules>
    <module>artifactIdEnfant1</module>
    <module>artifactIdEnfant2</module>
</modules>
```

- chaque sous module est dans un sous repertoire

```
<parent>
    <artifactId>artifactIdParent</artifactId>
    <groupId>pdl</groupId>
    <version>1.0-SNAPSHOT</version>
</parent>
```

La solution : Les Archetypes

Il existe un certain nombre de projets type préconfiguré, et on peut créer ses propres types

```
mvn archetype:generate
    -DarchetypeGroupId=maven.archetype
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DgroupId=pdl
    -DartifactId=parent
```

Maven est integré dans tous les IDE courant (par défaut ou avec un plugin)

Un point très important

La communication

- On ne vous demande pas de mettre en place des méthodologies de gestion de projet...
- ... MAIS vous allez être obligé de mettre en place des choses vu les conditions
- En gestion de projet vous apprendrez que la communication est primordiale, et il va falloir mettre en place une organisation qui en inclus un maximum.

Gestion du code

Utilisation d'une forge logicielle

C'est un outil de travail collaboratif qui inclus un dépôt bien sur, mais aussi des outils de gestion des tâches et des métriques.

Gitlab

Nous allons utliser le gitlab du CREMI :
`https://gitlab.emi.u-bordeaux.fr/`

- Il est en version beta
- Il utilise un dépôt git
- Accessible aux étudiants et enseignants

Git

Gestionnaire de version

- Git permet la gestion des versions du code (comme d'autres)
- Il peut s'utiliser en ligne de commande
- On trouve quelques interfaces dédiées
- Il est intégré dans tous les IDE digne de ce nom... mais il est vivement conseillé d'apprendre et de comprendre la ligne de commande avant de l'utiliser par l'éditeur

Les commandes de bases

Encore et toujours le Cheat Sheet

- Création : `init`, `clone`
- Gestion des fichiers : `add`, `remove`, `mv`, `status`
- Gestions des versions : `commit`, `checkout`
- Gestion des branches : `branch`, `merge`, `rebase`
- Synchronisation des dépôts : `pull`, `push`, `fetch`

Utilisation classique (et rarement bonne)

1. On `clone` le dépôt distant sur lequel on veut travailler
2. On modifie un ou plusieurs fichiers
3. on `commit` nos modifications
4. on réitère 2 et 3 jusqu'à ce qu'on soit assez satisfait...
5. ...et on `push` sur le dépôt

Accrochez vous aux branches

- Git utilise un système de branches, qui regroupe des suites de commits
- Chaque dépôts est initialisé avec une branche "Master"
- La branche principale doit rester "propre"
- créez autant de branche que nécessaire ! (typiquement pour chaque feature et bugfix)

Utilisation des branches

1. on crée une branche
2. on travaille dessus : x commits jusqu'à satisfaction
3. on `merge` avec la branche parente
4. on efface la branche

Manipuler les branches

- Créer une branche : `git branch` *`nom-de-branche`*
- Changer de branche : `git checkout` *`nom-de-branche`*
- Lister les branches : `git branch [-a]`
- Effacer une branche : `git branch -d` *`nom-de-branche`*
- Fusionner *`branche2`* avec sa branche : `git merge` *`branche2`*
- Quand une branche est jolie on y met un tag, qui ne bougera plus : `git tag` *`nom-de-tag`*

Tailler les arbres

quelques commandes utiles pour gérer les merge :

`git pull`, `git stash`, `git rebase`, `git rebase -i`

retrouver une erreur : `git blame`

Tailler les arbres

quelques commandes utiles pour gérer les merge :
`git pull`, `git stash`, `git rebase`, `git rebase -i`
retrouver une erreur : `git blame`

Combien de dépôts ?

- Soit un dépôt commun dans lequel tout le monde travaille
- Soit un dépôt par développeur (on "fork" le dépôt), en plus du dépôt commun dans lequel le code est accepté après une revue de code : mécanisme de `pull-request`

Gitlab

D'autres fonctionnalités

- Gestion Des tâches : description, statut, personne assignée, commentaires, étiquette, etc
- Les issues peuvent servir à traquer les bug ou remplacer un KanBan (sinon vous pouvez utiliser Trello)
- Gestion des pull-request/merge-request (avec des fonctionnalités similaires aux Issues)
- Intégration continue

Intégration Continue

Pincipe

- Permet de savoir à tout instant si tout va bien (et ne pas découvrir au dernier moment qu'un test passe pas), et pouvoir garder une branche principale toujours fonctionnelle
- Automatisation des tâches courantes :
  - Compilation
  - Tests
  - Audit de code (qualité, couverture)
  - Fourni des rapports et des métriques

Tests

Qu'est ce qu'un test ?

- Permet de vérifier le comportement d'un programme
- Confronte une implémentation à sa spécification
- Il existe différents types de tests (tests unitaires, tests d'intégration, tests fonctionnels)
- Les tests peuvent avoir différents niveaux de connaissances du programme :
  - Tests boîtes blanches : avec connaissance
  - Tests boîtes noires : sans connaissance

Tests

Qu'est ce qu'un test unitaire ?

- Test une *unité* du programme : une classe, une méthode, etc
  - Vérifie les comportement attendus
  - Vérifie les comportement innatendus (mauvaise valeur de paramètres, ...)
- Approche incrémentale
- Doivent être rejouer afin de vérifier la non-regression

Exemple

Implémentation d'une Pile

Quelles fonctionnalités ?

- Création
- Push
- Pop

Quels tests on pourrait définir avant de coder ?

Exemple

Implémentation d'une Pile

- Quand je crée une pile elle doit être vide
- Quand je push un élément dans la pile sa taille doit être égale à un
- Quand je push 3 éléments dans la pile sa taille doit être 3
- Quand je pop un élement de la pile avec un élément, la pile doit être vide
- Quand je pop un élément de la pile avec 3 éléments, la taille doit être 2
- ....

Est-ce que cette suite de tests est suffisantes ?

Est-ce que l'on couvre tout le code ?

Test Driven Development (TDD)

TDD

- Concept qui est utilisé dans plusieurs méthodologie de gestion de projet
- Utilise les tests unitaires comme spécification
- On écrit le test avant le code
- Les tests sont automatisé et peuvent donc être executés régulièrement

Fonctionnement du TDD

1. On écrit letest en premier
2. On vérifie qu'il échoue (puisque le code n'existe pas encore)
3. On écrit juste assez de code pour passer le test
4. On vérifie que le test passe
5. On fait du refactoring pour améliorer le code

Bénéfices

Les avantages du TDD

- Réduit les erreurs de design
- Augmente la confiance du programmeur lors du refactoring
- Permet la construction conjointe du programme et des tests (évite les éceuils de certains cycle de développments séquentiels)
- Permet les tests de non-regression
- Permet d'estimer la progression du développement du projet

Le framework JUnit

Un test est une méthode Java avec :

- visibilité `public`
- type de retour `void`
- pas de paramètres
- elle peut renvoyer une exception
- annotée `@Test`
- Utilise les instructions de test du framework défini dans `org.junit.Assert`

Instructions de tests

Instruction	Signification
`fail(String)`	échoue toujours
`assertTrue(true)`	réussi toujours
`assertsEquals(expected, actual)`	vérifie que les valeurs sont les mêmes
`assertsEquals(expected, actual, tolerance)`	vérifie que les valeurs sont les mêmes avec une tolérance
`assertNull(object)`	vérifie qu'un objet est null
`assertNotNull(object)`	vérifie qu'un objet n'est pas null
`assertSame(expected, actual)`	vérifie que les variables référencent le même objet
`assertNotSame(expected, actual)`	vérifie que les variables ne référencent pas le même objet
`assertTrue(boolean condition)`	vérifie que la condition est vrai

Exemples

```
@Test
public void test() {
    Stack<Integer> s = new Stack<Integer>();
    assertTrue(s.isEmpty());
}

@Test
public void test2() {
    Stack<Integer> s = new Stack<Integer>();
    s.push(1);
    assertEquals(1, s.size());
}
```

Exemples

Exceptions

- Vérification des comportements inapproprés (mauvaise valeurs de paramètres, ...)
- Repose sur le mécanisme des Exceptions Java

```
@Test(expected = Exception.class)
public void test5() throws Exception {
    Stack<Integer> s = new Stack<Integer>();
    s.pop();
}
```

Annotations

Annotation	Description
`@Test`	Méthode de test
`@Before`	Appelé avant chaque méthode de test
`@After`	appelé après chaque méthode
`@BeforeClass`	appelé avant le premier test
`@AfterClass`	appelé après le dernier test

Ordre d'execution

- Les méthodes annotées avec `@BeforeClass`
- Pour chaque méthode annotée avec `@Test`
  - Les méthodes annotées avec `@Before`
  - La méthode avec `@Test`
  - Les méthode annotées avec `@After`
- Les méthodes annotées avec `@AfterClass`

Mocking

A quoi servent les Mock

- Isoler les tests unitaires
  - Confiance dans les autres unités (dépendances, ...)
  - Localisation précise de l'erreur
- Créer un environnement de test qui simule le comportement d'autres unités

Objets Mock

- Imitent le comportement des objets réels de manières contrôlée
- Ne contiennent pas de fonctionnalité, mais peuvent retourner des valeurs

Mocking

Quand utiliser des objets Mock ?

Quand les objets à émuler

- N'existe pas encore ou vont changer de comportement
- Ont des résultats non déterministes (par ex. La température)
- Ont des état difficile à reproduire (par ex. une erreur réseau)
- Sont coutêuse (par ex. un accès à une base de données)

Exemple

Test d'une classe A à l'aide d'un objet Mock B

```
public class A {
    private B b;
    public A(B b) {
        this.b = b;
    }
    public int inc2(int i) {
        return b.inc(b.inc(i));
    }
}

public interface B {
    public int inc(int i);
}
```

Exemple

```
import static org.junit.Assert.assertEquals;
import org.easymock.EasyMock;
import org.junit.Test;

public class ATest {

    @Test
    public void test_m() {
        B b = EasyMock.createMock(B.class);
        A a = new A(b);
        EasyMock.expect(b.inc(1)).andReturn(2);
        EasyMock.expect(b.inc(2)).andReturn(3);
        EasyMock.replay(b);
        assertEquals(a.inc2(1),3);
        EasyMock.verify(b);
    }
}
```