Cours 3 Outils de développement

Programmation objets, web et mobiles en Java
Licence 3 Professionnelle - Multimédia

Pierre Talbot (talbot@ircam.fr)

ATER UPMC/IRCAM

14 novembre 2017

Le menu

- IDE
- JUnit
- Maven : Automatisation de production
- Gestionnaire de version
- Documentation Java

Environnement de développement (IDE)

Un IDE est un logiciel regroupant plusieurs fonctionnalités afin de développer un logiciel, notamment :

- **Éditeur de texte** (e.g. raccourcis claviers, auto-complétion, navigation dans le code source)
- **Automation de la compilation** (e.g. sélection version Java, gestion des dépendances, mode de test/debug/production)
- **Debugger** : exécuter le code pas à pas pour comprendre un comportement ou un bug.

Configurer son IDE

IDEs Java

Eclipse, Netbeans, IntelliJ, ...

- Peu importe l'IDE, il faut le configurer et l'apprendre si on veut qu'il soit utile !
- En particulier maitriser les raccourcis claviers permet de réduire la frustration de devoir faire des changements car ça devient facile.

Rupture du "flow"

- Faire l'aller-retour entre le clavier et la souris casse votre "flow"... Ce n'est pas ergonomique ni agréable.
- Par exemple déplacer le curseur sans arrêt et cliquer sur les fichiers pour les ouvrir.

Génération de code

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package upmc.game;

/**
 *
 * @author ptalbot
 */
public class Game {
}
```

Template de code

- *Netbeans* : `tools -> templates`
- *Eclipse* : `Window -> Preferences -> Java -> Code Style -> Code Templates`

Configuration de l'éditeur

Instructions pour Eclipse :

- Utiliser des espaces pour indenter : `Window > Preferences > General > Editors > Text Editors` cochez la case insert *spaces for tabs*
- Fichier en UTF8 : `Window > Preferences > General > Workspace > Text File Encoding` : *utf8*

Source : Cours DLP Master 1 STL / UPMC.

Naviguer dans le code

Raccourcis pour Eclipse :

- `CTRL + SHIFT + R` : Ouvrir n'importe quel fichier.
- `CTRL + O` : Aller à une méthode du fichier ouvert.

Les raccourcis suivant sont utiles mais les flèches ne sont pas très ergonomiques...

- `CTRL + Flèche gauche / droite` : Aller au mot précédent/suivant.
- `END` : Aller à la fin de la phrase.
- `HOME` : Aller au début de la phrase.

`CTRL + SHIFT + L` permet de lister tous les raccourcis disponibles et ce raccourci deux fois permet de les modifier.

Manipulation du code

- `CTRL + SHIFT + 1` : Quick fix d'une typo / erreur de syntaxe / ...
- `CTRL + SHIFT + o` : Organiser / réparer les imports manquants.
- `CTRL + /` : Commenter ou dé-commenter une ligne.
- `CTRL + SHIFT + F` : Correction de l'indentation du code.
- `CTRL + D` : Effacer une ligne.
- `CTRL + Up / Down` : Dupliquer une ligne.

Ce n'est que quelques exemples, n'hésitez pas à essayer de faire tout au clavier et chercher de nouveaux raccourcis.

Le menu

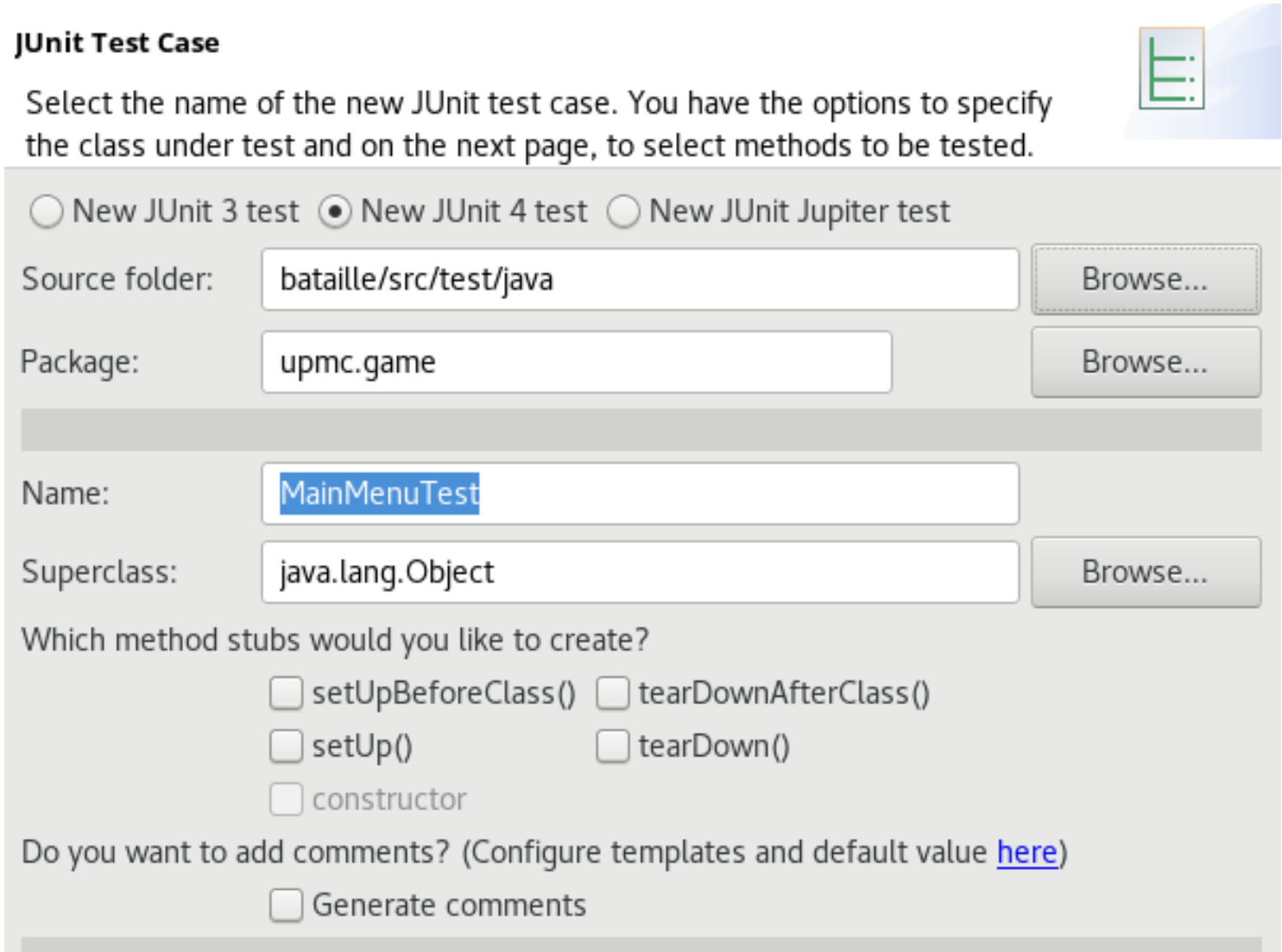
- IDE
- JUnit
- Maven : Automatisation de production
- Gestionnaire de version
- Documentation Java

Pourquoi tester son projet ?

- Tester une méthode, c'est l'utiliser en avant-première et vérifier que son interface est bonne.
- C'est avoir **confiance** en son code : les tests permettent de vérifier qu'une modification dans le code n'entrainent pas de bugs.
- C'est gagner du temps au final : débugger peut être très long et douloureux.
- C'est avoir un "flow" plus agréable de travail.

Créer un test JUnit

Eclipse : Clique droit sur le fichier à tester, puis `New > JUnit Test Case`.



Tester une méthode

```
@Test
public void testVerifyMenuChoice() {
  // Arrange
  int menu = 0;
  int maxEntries = 2;
  MenuTools tools = new MenuTools();
  // Act
  boolean result = tools.verifyMenuChoice(menu, maxEntries);
  // Assert
  assert(result == false);
}
```

- `@Test` : seules les méthodes annotées sont testées.
- `assert(expr)` vérifie que l'expression est bien égale à *true*.

Assertions JUnit

JUnit fournit de nombreux outils pour tester le code :

- `assertFalse(e)` / `assertTrue(e)`
- `fail(msg)` fait échouer le test.
- `assertNotNull(o)` vérifie que l'objet *o* est différent de `null`.
- Voir `http://junit.org/junit4/javadoc/latest/org/junit/Assert.html`

Before / After

On a vu qu'il faut créer l'objet et d'autres données ; on peut automatiser ce processus :

```
private MenuTools tools;

@Before
public void createMenuTools() {
  tools = new MenuTools();
}
```

La méthode `createMenuTools` sera appelée avant chaque test.

Tester les exceptions

```
@Test(expected=BadMenuChoiceException.class)
public void testVerifyMenuChoiceException() {
  // Arrange
  int menu = 0;
  int maxEntries = 2;
  // Act
  tools.verifyMenuChoiceException(menu, maxEntries);
}
```

Une technique plus récente existe avec les `@Rule` (voir livre).

Qu'est-ce qu'un bon test ? FIRST

- *[F]ast* : Il faut qu'ils soient assez rapides pour être lancés fréquemment.
- *[I]solated* : Pas d'interaction avec base de données, réseau, ...
- *[R]epeatable* : Les tests sont déterministes, pas d'aléatoires, si le test échoue il doit échouer tout le temps et vice-versa si il réussit.
- *[S]elf-validating* : Vérifier si un test est correct ou non ne doit pas nécessiter notre intervention (e.g. afficher sur la console et vérifier à l'il nu n'est pas un test !)
- *[T]imely* : N'écrivez plus de Java sans le tester, dès maintenant, une méthode $\equiv$ un test.

Source : Pragmatic unit testing in Java 8 with JUnit

Écrire un bon test : Right-BICEP

- *Right* Are the results right ?
- *B* Are all the boundary conditions correct ?
- *I* Can you check inverse relationships ?
- *C* Can you cross-check results using other means ?
- *E* Can you force error conditions to happen ?
- *P* Are performance characteristics within bounds ?

Source : Pragmatic unit testing in Java 8 with JUnit

En savoir plus...

Mock objects

Ils répondent à la question de comment tester, par exemple, une classe utilisant la console ? Une base de données ? Une requête du web ?

Test Driven Development (TDD)

Méthodologie de développement dans laquelle on met les tests au cur du projet, notamment :

- Au lieu d'écrire le code et puis d'écrire les tests, on fait le contraire !
- En écrivant les tests d'abord, on réfléchit sur l'interface du code que l'on va écrire.

Le menu

- IDE
- JUnit
- Maven : Automatisation de production
- Gestionnaire de version
- Documentation Java

Compilation Java

Compiler les sources d'un projet Java peut se faire en console :

```
> javac Test.java
> java Test
```

Laborieux

- Quand on a des packages et de nombreux fichiers c'est laborieux car il faut tous les lister...
- Il faut aussi gérer les dépendances à la main.

Automatisation de production

- Les IDEs possèdent leur propre système d'automatisation de production.
- Par exemple, on parlera de "projet Eclipse" ou "projet Netbeans".

Avantages

- Le projet est construit automatiquement.
- Les dépendances ajoutées au projet sont +- automatiquement prises en compte.

Désavantages

- Dépendant de l'IDE : il faut que tous le monde soit sur le même.
- Pas toujours intuitif de savoir comment gérer les options.

Maven

Maven est un outil non-dépendent d'un IDE particulier.

Caractéristiques

- "Convention plutôt que configuration" : impose une hiérarchie du projet.
- *Build* divisé en parties appelées *goal*, exécutées à tour de rôle.
- Le fichier `pom.xml` est le fichier de configuration.

Structure du projet

- `src/main/java` : Application/Library sources
- `src/main/resources` Application/Library resources
- `src/main/filters` Resource filter files
- `src/main/webapp` : Web application sources
- `src/test/java` : Test sources
- `src/test/resources` : Test resources
- `src/site` : Site
- `LICENSE.txt` : Project's license
- `README.txt` : Project's readme

Source : `https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html`

Cycle de vie

Le build est divisé en plusieurs parties. En voici quelques unes :

- `validate` : Vérifie que le projet est complet avec toutes les informations nécessaires (e.g. présence du `pom.xml`)
- `generate-sources` : Générer des `.java` à partir d'autres fichiers (e.g. grammaire ANTLR4)
- `compile` : Compiler le projet, `.java -> .class`
- `test-compile` : Compiler les tests JUnit and co.
- `test` : Exécuter les tests.
- `package` : Crée le `.jar` associé au projet
- `integration-test` : Exécuter les tests d'intégrations (e.g. le logiciel fonctionne toujours sur un serveur).
- `install` : Installer le package dans le dépôt local de Maven.
- `deploy` : Déployer le package pour partage avec d'autre personne (release).

Source : voir le livre "Maven : The complete reference"

Goals

- Ce cycle est lancé avec `mvn package` ou `mvn compile` suivant jusqu'où on veut aller.
- `mvn test` lance les tests du projet.
- En général `mvn` *goal* où vous sélectionnez le goal dans la liste précédente.
- `mvn exec:java -Dexec.mainClass="upmc.game.Bataille"` pour exécuter la méthode `main`.
- `mvn clean` pour nettoyer le projet (supprimer les fichiers générés).

Bien sûr, toutes ces fonctionnalités peuvent être lancées via votre IDE.

Dépendances

Utiliser une bibliothèque dans un projet

- C'est souvent comme installer un logiciel sur votre PC.
- Répliquer l'installation est parfois difficile sur les différentes plateformes.
- Quid de la compilation automatisée ?

Cas du C et C++

- C'est problématique car chaque bibliothèque s'installe d'une manière différente.
- Perte de temps et difficulté de répliquer et automatiser la compilation (essayez d'installer C++ Boost Libraries...)

Répertoire de bibliothèques centralisé

Solution

- Un répertoire central listant toutes les bibliothèques d'un langage.
- Problème : Cela ne dit pas comment construire une bibliothèque.
- Solution : Associer ce répertoire central à un builder particulier.

- Python : PIP est le *package manager* et PyPI est le répertoire.
- Javascript : NPM / `npmjs.com`
- Rust : Cargo / `crates.io`

Choisir un langage, c'est aussi adopter son écosystème.

Et en Java ?

- Répertoire de bibliothèques Java utilisables dans notre projet via *Maven* : `https://www.mvnrepository.com/`
- Il suffit de trouver sa bibliothèque sur ce site et de copier/coller la dépendances dans le `pom.xml`.
- En fait, ce répertoire marche aussi pour d'autres systèmes de build de Java (Gradle, SBT,...)

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchem
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.
  <modelVersion>4.0.0</modelVersion>

  <groupId>upmc.pcg</groupId>
  <artifactId>pokedeck</artifactId>
  <version>0.0.1</version>
  <packaging>jar</packaging>

  <name>pokedeck</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

pom.xml : Identification du projet

- `groupId` est l'identifiant d'un groupe de projets, souvent le domaine d'une entreprise à l'envers : `fr.upmc`
- `artifactId` est l'identifiant du projet courant.

Le `groupId` + `artifactId` + `version` forment ce qu'on appelle une "coordonnée" ; on l'utilise pour déclarer précisément ses dépendances.

- `name` : Le nom du projet tel qu'il est connu dans l'équipe ou commercial. Par exemple, "com.google-android-5.0.0" est la coordonnée mais le nom est "Lollipop".

Exercice : Ajouter la dépendance JSON au Pokédeck.

Numéro de version

Le numéro de version d'un logiciel, par exemple `2.0.3-beta`, est très important ; il permet de :

- Préciser exactement "à quel point" le logiciel a changé.
- Informer les utilisateurs des potentiels incompatibilités avec les anciennes versions.

En Maven le format de version est le suivant :

```
<major version>.<minor version>.<incremental version>-<qualifier>
```

Semantic versioning

Citation de `http://semver.org/` :

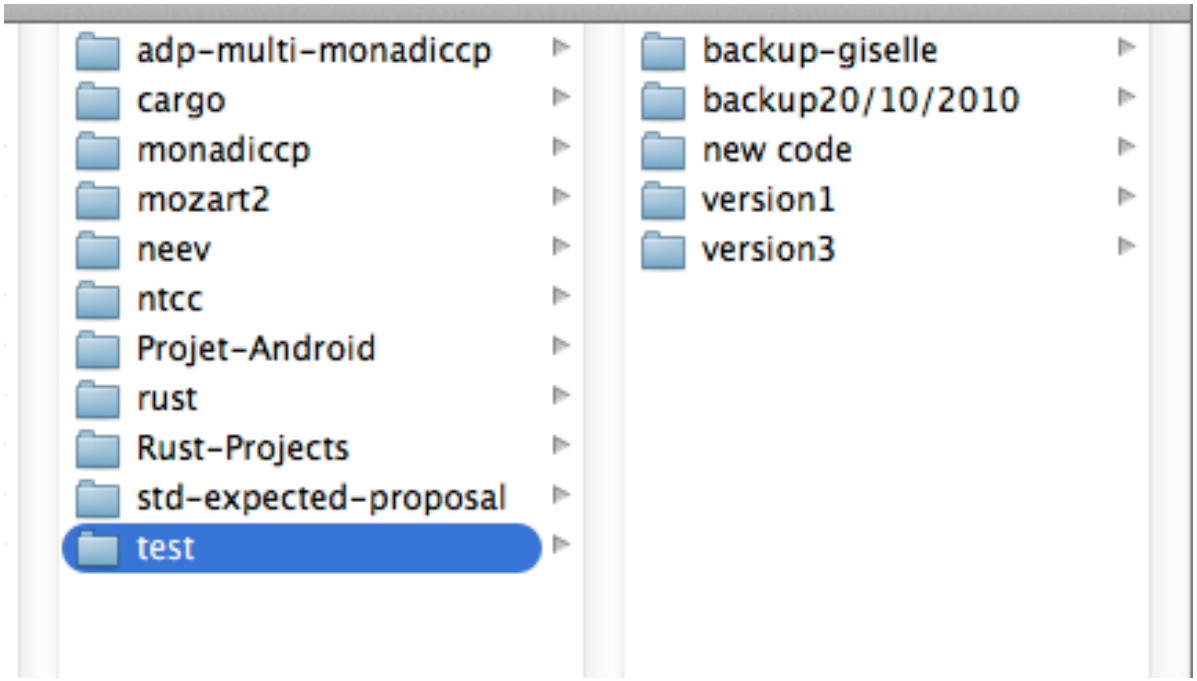
```
Given a version number MAJOR.MINOR.PATCH, increment the:

  MAJOR version when you make incompatible API changes,
  MINOR version when you add functionality in a
    backwards-compatible manner, and
  PATCH version when you make backwards-compatible
    bug fixes.

Additional labels for pre-release and build metadata are
available as extensions to the MAJOR.MINOR.PATCH format.
```

Le menu

- IDE
- JUnit
- Maven : Automatisation de production
- Gestionnaire de version
- Documentation Java

La gestion old-school

Gérer ses sources à la main. . .



Gestionnaire de version

Les gestionnaires de version permettent de :

- Travailler facilement à plusieurs ou via plusieurs machines sur un projet.
- Garder une trace de toutes les modifications.
- Revenir en arrière sur une partie ou tous le projet.
- Tester des fonctionnalités en gardant le projet compilable.
- Et bien plus. . .

Point culture

Deux grandes familles

1. Gestionnaire de version centralisé (Subversion (svn), CVS, . . .).
2. Gestionnaire de version décentralisé (Git, Mercurial, . . .).

La grande différence est que les gestionnaires de version décentralisés impliquent que tout le monde peut devenir un serveur, et donc que tous les développeurs possèdent une copie locale des versions.

Avantages

1. Vous pouvez travailler sans être connecté au serveur.
2. Si le serveur crash, l'historique n'est pas perdu et un développeur peut devenir le serveur.
3. La plupart des opérations sont plus rapides car effectuées en local.

Comment ça marche ?

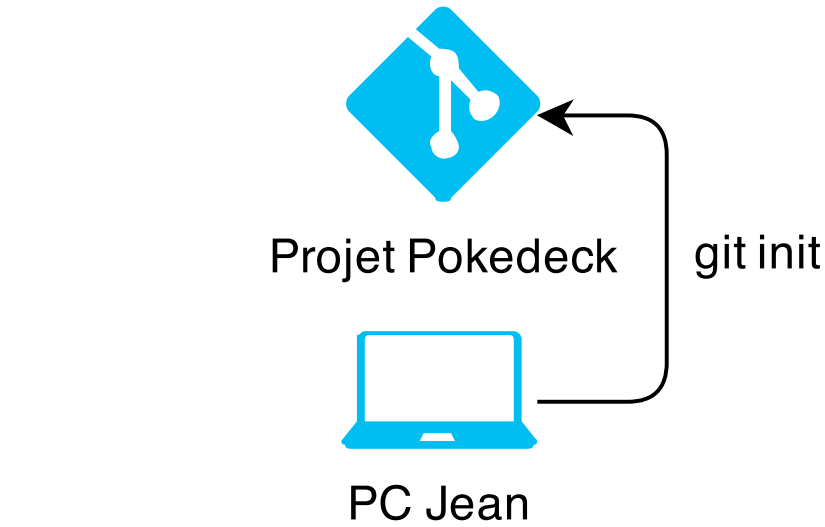
PC Jean

FIGURE – Situation de base

2 situations

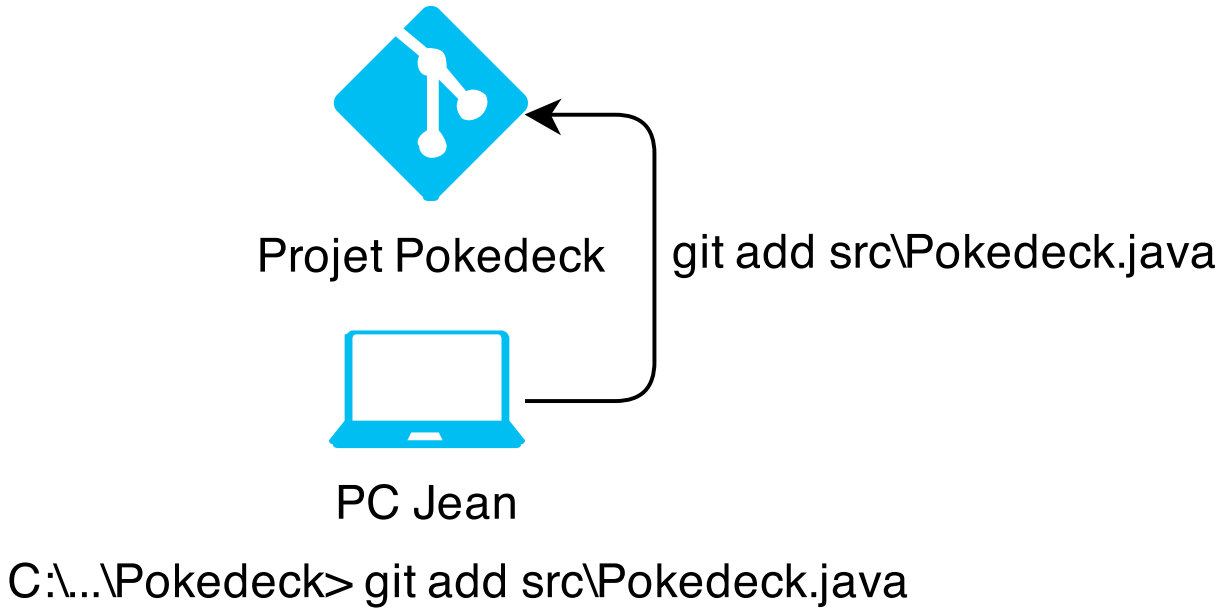
1. J'ai un projet que je veux versionner.
2. Le projet existe déjà et je veux juste le récupérer.

Création d'un répertoire



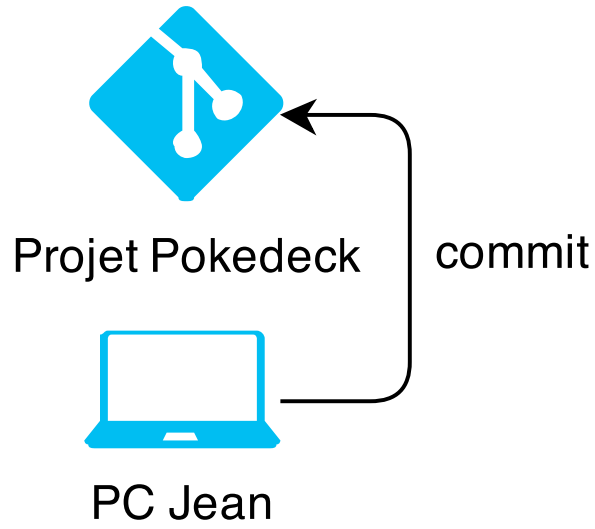
Cette commande est lancée dans le répertoire de votre projet, à la racine du projet (Pokedeck/).

Ajouter un fichier



Un fichier est ajouté au projet, mais il faudra quand même “valider” cet ajout avec `git commit`.

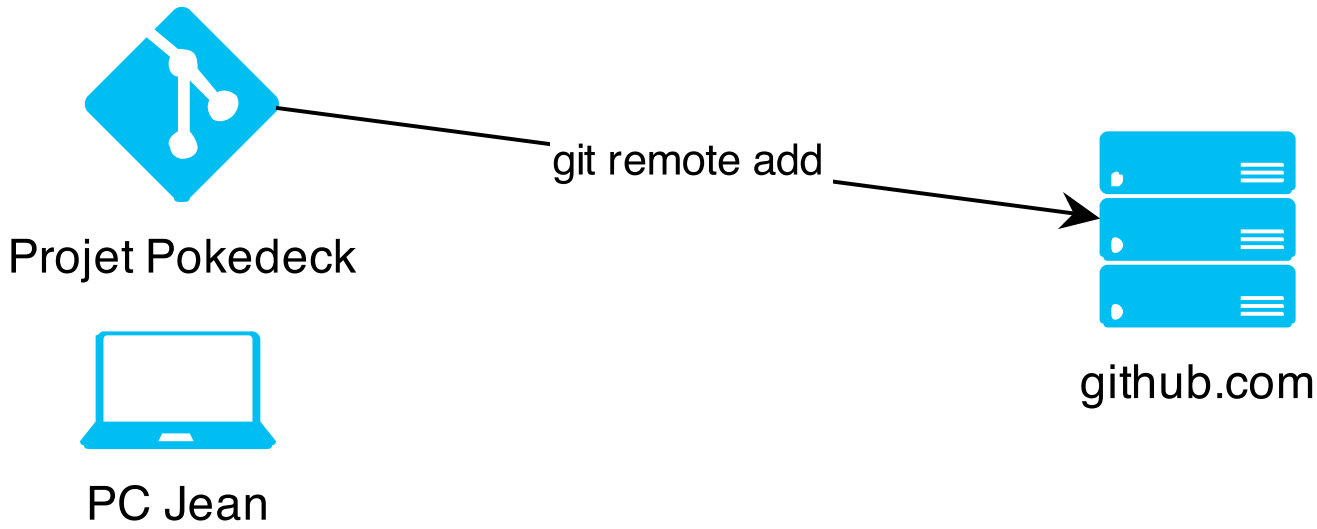
Valider les changements



C:\...\Pokedeck> git commit -a -m "Initialize main class"

Les changements sur les fichiers ajoutés aux projets sont validés **en local**.
On peut voir `commit` comme un `push` sur un serveur local.

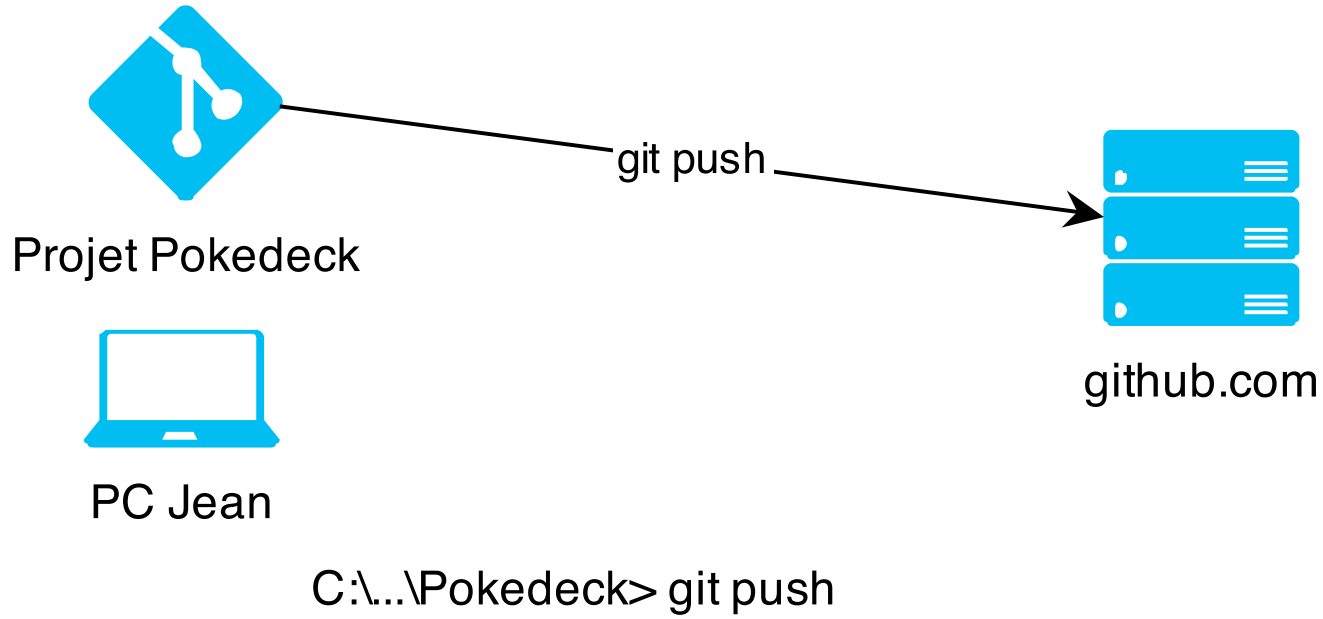
Ajouter son projet sur un serveur distant



C:\...\Pokedeck> git remote add origin https://github.com/ptal/Pokedeck.git

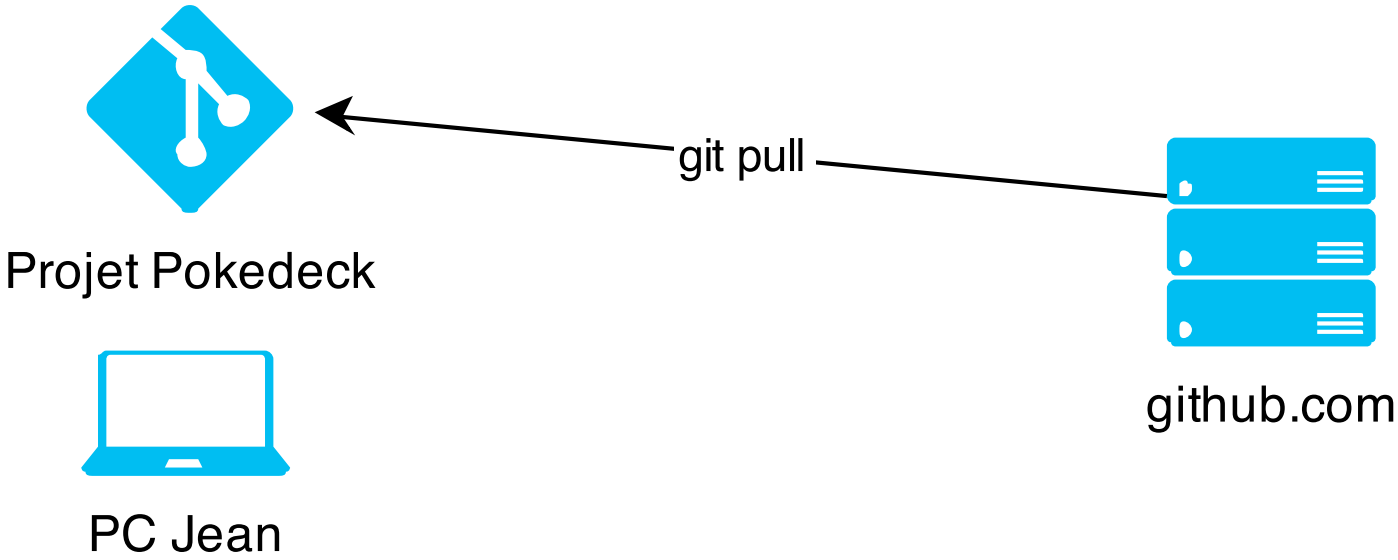
github devient le serveur référenciel, `origin` est juste un alias arbitraire vers l'adresse exacte.
Note : Le *commit* n'envoie pas les changements sur le serveur distant.

Envoyer les commits sur le serveur distant



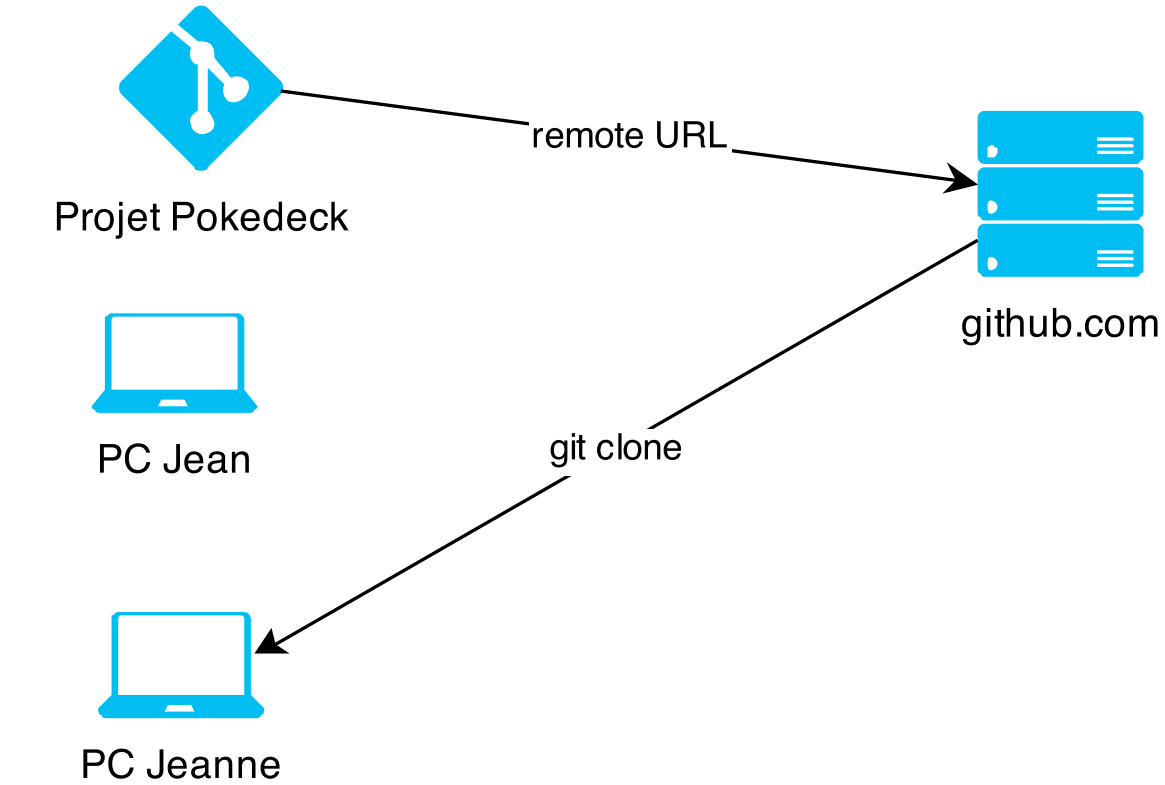
Avant de partir, ou si une autre personne à besoin de nos changements, faire un `push` pour envoyer les *commits* sur le serveur.
Note : Pour le premier push, vous devez dire explicitement sur quel serveur et quel branche vous envoyez les changements : `git push -u origin master`.

Récupérer les commits du serveur distant



Quand on arrive, ou pendant qu'on travaille si une personne a `push` des changements qu'on veut récupérer, faire un `pull` pour récupérer les changements du serveur.

Cloner un projet d'un serveur distant



On récupère simplement le dossier du projet avec `git clone`.

Concrètement

Vous pouvez taper ces commandes dans un *shell* (ou une console, ou un *bash*, . . .). Elles sont identiques pour *tous* les systèmes d'exploitations.

Le shell sous Windows

Existe en deux versions :

- Une version console (*git bash*) imitant celle de Linux/MacOSX : http ://msysgit.github.io/
- Une version console (*git shell*) avec les attributs de Windows (nom de fichier, ...) : https ://windows.github.com/

Ces outils contiennent plus que une console, mais également une interface graphique. Si vous voulez utiliser l'interface graphique, le deuxième outil est probablement le meilleur.

Un cours en ligne

Vous pouvez suivre ce tutoriel de 15 minutes : `https://try.github.io/levels/1/challenges/1`, il se concentre sur la ligne de commande mais le nom des boutons dans l'interface graphique sont similaires.
Lisez également la documentation de l'outil que vous avez choisi d'installer.

Directement dans l'IDE

Il existe des plugins pour git qui sont intégrables à l'IDE, plus qu'à faire clique-droit $>$ git $>$ ajouter, pour ajouter un nouveau fichier ou lancer n'importe quelle commande.

Github

`github.com` est probablement le service le plus populaire pour héberger des projets (principalement open-source) en 2017.

Gestion de projet

- But principal : Versionner ses sources.
- Mais contient aussi un gestionnaire de ticket (*issues*) que vous pouvez tagger.
- La possibilité de créer un Wiki.
- De récolter des statistiques en tout genre
- ...

Ticket Github

Un ticket représente une action à réaliser :

- Un bug à résoudre.
- Une fonctionnalité à implémenter.
- Du code à commenter.
- ...

Vous pouvez fermer automatiquement des tickets en ajoutant dans votre message de commit : “It closes #2” où 2 est le numéro du ticket.

Le menu

- IDE
- JUnit
- Maven : Automatisation de production
- Gestionnaire de version
- Documentation Java

Documentation Java

Voici le lien pour Java 8 : `http://docs.oracle.com/javase/8/docs/api/overview-summary.html`.

- Généralement on explore pas la doc pour le fun.
- On veut un renseignement précis sur les méthodes qu'une classe propose.
- Mais comment faire pour trouver une fonctionnalité si on ne connaît pas la classe ?

Trouver son bonheur

Il faut exprimer clairement ce qu'on recherche et faire le point sur :

- Est-ce que je sais comment faire (via expériences) et qu'il me faut les noms des classes ?
- Si oui, une recherche Google suffit généralement à trouver des exemples.
- Si non, s'il Google ne donne pas de résultat pertinent et qu'on n'arrive pas à mettre des mots sur ce qu'on recherche, il faut demander (au prof ou à vos collègues). Vous gagnerez du temps.
- Parfois quelque chose qu'on pense simple est en fait plus global et compliqué et c'est difficile de s'en rendre compte (c'est pour ça qu'il faut demander).

Documenter son code

Vous pouvez facilement faire une documentation comme celle de Java en documentant votre code avec des balises spéciales.

```
/**
 * 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.
 *
 * @param  url  an absolute URL giving the base location of the image
 * @param  name the location of the image, relative to the url argume
 * @return      the image at the specified URL
 * @see         Image
 */
public Image getImage(URL url, String name) {
  try {
    return getImage(new URL(url, name));
  } catch (MalformedURLException e) {
    return null;
  }
}
```

Documenter son code

Les commentaires sont interprétés en HTML et les balises (@param, @return, ...) décrivent le code et sont interprétées spécialement par `javadoc`.
Une liste est disponible : `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#javadoctags`.

En Maven

```
mvn javadoc:javadoc
```

- Le résultat est dans `target/site/apidocs`, ouvrez le `index.html` avec votre navigateur.
- Essayer aussi `mvn site` pour générer un site Maven.

Documentation “Top-down”

Bonne pratique

Privilégier la documentation de plus haut niveau avant de documenter les spécifiques.

En effet, à quoi sert d'avoir la documentation :

- d'une méthode get si on ne sait pas à quoi sert la classe, et
- d'une classe si on ne sait pas à quoi sert le projet ?

Privilégier d'abord la documentation en tête de classe et réfléchissez : “Qu'est-ce que quelqu'un qui arrive sur ce code a besoin de savoir”.