Aix*Marseille université
Socialement engagée

Année 2022-23

Domaine Sciences et Technologies

Licence informatique : parcours MI

Approfondissements informatique :
Git et tests

Code UE : SIN6U28T

Gestion de versions et tests

Gestion de versions et tests

Arnaud Labourel

1 Introduction

1.1 Présentation de partie conception

Un tiers de l'enseignement d'approfondissements en informatiques est consacrée à ce que l'on nomme en programmation la conception. Cette partie est indépendante de la partie consacrée aux systèmes et réseaux. La conception consiste à donner à un projet de programmation une organisation souple mais structurante. Une telle organisation permettra de garder un code clair au cours des différentes évolutions que le code subira. Ces évolutions devront pouvoir être faites facilement, et le code pourra à tout moment être testé facilement. La modularité intelligente est une clé de voûte de la conception. Les principes de cette modularité sont énoncés dans les principes SOLID. Le savoir-faire de cette modularité est résumé dans les patrons de conceptions qui permettent d'éviter les pièges habituels de conception en suivant des modèles sous la forme de diagrammes UML.

Les différentes points abordés dans cette partie du cours sont :

— Documentation, spécification, automatisation des tests
— Diagrammes de classes UML
— Principes SOLID : SRP, OCP, LSP, ISP, DIP
— Présentation et utilisation de patrons de conceptions sur des exemples concrets

1.2 Bibliographie

Cette partie du cours sur la conception s'appuie en autre sur les ouvrages suivants :

— Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (trad. Jean-Marie Lasvergères), *Design Patterns - Catalogue de modèles de conceptions réutilisables*, France, Vuibert, 1999.
— Robert C. Martin, *Agile Software Development : Principles, Patterns, and Practices*, Upper Saddle River, NJ, Pearson Education, 2002.
— Robert C. Martin, *Coder proprement*, Pearson France, 2019.
— J B Rainsberger & Scott Stirling, *JUnit Recipes - Practical Methods for Programmer Testing*, Manning publishing

1.3 Déroulement des enseignements

Les cours, exercices et devoirs seront publiés tout au long de l'année scolaire. La note finale (NF) de cette partie de l'UE sera calculée comme suit avec la formule suivante :

$$NF = \max(E, 0,2 \times CC + 0,8 \times E)$$

avec E la note d'examen et CC la note de contrôle continu. Il y aura 3 devoirs dont le dernier, un peu plus long que les autres, comptera double. La note de contrôle continu est donc calculée avec la formule suivante :

$$CC = \frac{D1 + D2 + 2 \times D3}{4}$$

avec Di la note du devoir numéro i.

1.4 Langage de programmation

Le langage utilisé durant le cours sera le Java. Il est tout à fait possible d'appliquer les bonnes pratiques de programmation dont principes SOLID ainsi que les patrons de conceptions avec d'autres langages de programmation permettant le paradigme de programmation orientée objet comme Python, C#, C++, PHP, Swift, Kotlin, ... Normalement, vous devriez avoir suivi deux unités d'enseignement sur la programmation orientée objet nommées Programmation 1 et Programmation 2. Si vous avez besoin de réviser le fonctionnement de Java et les concepts de base de la programmation orientée objet, vous pouvez consulter les supports de cours de ces deux unités d'enseignements aux liens suivants :

— Programmation 1 : lien supports
— Programmation 2 : lien supports

1.5 Logiciels à installer

Contrairement à la partie Système et Réseaux, il n'y a pas d'obligation d'utiliser un système d'exploitation UNIX/Linux pour pouvoir travailler sur la partie conception car Java a été conçu pour être exécutable facilement sur n'importe quel système d'exploitation. Néanmoins, l'utilisation d'un tel système est bien sûr possible et peut faciliter l'utilisation de certains outils (comme la configuration de l'authentification via une clé SSH). Vous avez la possibilité d'utiliser la machine virtuelle *Lunix* disponible au lien suivant : système de virtualisation. Si jamais vous avez des problèmes pour accéder à ces machines virtuelles vous pouvez posez des questions dans le forum général d'échanges du cours ou bien formuler une demande d'aide personnalisée à la DIRNUM (DIRection du NUMérique) via le service de télé-demandes accessible depuis votre ENT.

1.5.1 Kit de développement Java

Afin de pouvoir compiler et exécuter du code Java, il vous faudra installer un Java Development Kit (JDK). Nous vous conseillons la version 17 de Java qui peut être téléchargée sur le site d'Oracle au lien suivant.

Sous Ubuntu ou tout système linux utilisant le gestionnaire de paquet apt (c'est-à-dire généralement une distribution debian), vous pouvez installer le JDK 17 avec la ligne de commande suivante : `apt install openjdk-17-jdk openjdk-17-jre`.

1.5.2 Environnement de développement

Pour coder proprement, il est souvent utile de disposer d'un environnement de développement (IDE pour *Integrated development environment*). Un IDE est un ensemble d'outils comportant un éditeur de texte dédiée à la programmation (avec une autocomplétion intelligente), des fonctions et menus qui permettent, de compiler et exécuter du code, d'accéder à des outils pour déboguer, de gérer la gestion de version, fournir des outils de refactoring efficaces, ...

Il existe de nombreux IDE adaptés au langage de programmation Java. Les plus utilisés sont les suivants :

— IntelliJ IDEA de JetBrains qui est l'IDE qui va être mis en avant dans les supports et qui est disponible sur les machines virtuelles fournies par AMU dans sa version ultimate;
— Eclipse qui est très similaire à IntelliJ IDEA;
— Visual Studio Code qui est plus léger que les deux IDE précédents et pour lequel il faut installer les extensions java pour avoir le même type de fonctionnalités.

Il est recommandé d'installer un de ces logiciels ou équivalent pour cette partie car utilisé un bon environnement de développement est une première étape pour construire proprement un projet logiciel d'une taille un peu conséquente.

1.5.3 Moteur de production

Un moteur de production est un logiciel dont la fonction principale consiste à automatiser le processus de création d'un logiciel à partir d'un code source. Cela comprend la compilation du code source, le *packaging* de l'executable (création d'installateur ou d'exécutable), la gestion des dépendances (liens avec des bibliothèques externes, téléchargements, ...), l'exécution automatisés de tests, ...

Pour les projets de ce cours, nous utiliserons gradle comme moteur de production. Il permet en autre de rendre facile la mise en place de tests unitaires via les bibliothèques JUnit 5 et AssertJ.

Pour installer Gradle il suffit de suivre les instructions d'installation en téléchargeant la version 7.5.1 de Gradle.

1.5.4 Gestion de version

Lorsqu'on travaille sur le code un peu conséquent, il est utile d'utiliser un outil de gestion de version qui permet en autre de :

— conserver les différentes versions du code: conservation de l'historique des changements sous la forme d'un dépôt qui permet de revenir à n'importe quelle version;
— stocker le code à des endroits différents (machines personnelles des développeurs, serveur permettant gérer le code, ...) avec des outils de synchronisation entre les différents dépôts;
— travailler en équipe en conservant l'origine de toutes les modifications (on sait donc qui blamer lorsqu'il y a des erreurs).

Dans ce cours, vous utiliserez git que vous pouvez télécharger au lien suivant : https://git-scm.com/download. Afin d'héberger votre code, vous avez accès à une instance gitlab à l'adresse suivante : https://etulab.univ-amu.fr/.

2 Bonnes pratiques de programmation

Avant de présenter les principes SOLID qui sont cinq principes de conception visant à produire des architecture logicielles qui sont plus compréhensibles et maintenables, il est important de faire un rappel sur les bonnes pratiques de programmation qui permettent de rendre le code plus lisible. Une des choses les plus importantes lorsqu'on écrit du code est de bien nommer les éléments du code. Nous allons donc commencer par expliquer les principes à respecter pour avoir un nommage efficace.

2.1 Une méthodologie pour bien nommer

Trouver des noms appropriés peut prendre beaucoup de temps sur le moment mais va vous permettre de gagner du temps par la suite. Les conventions de nommage sont extrêmement importante pour la maintenance et la lisibilité d'un programme. Il est donc important de :

- utiliser des noms cohérents pour tous les symboles:
- choisir un nom qui correspond au but/rôle du symbole (Le nom doit révéler le rôle de l'élément);
- utiliser l'anglais pour nommer vos éléments;
- choisir un nom en regardant le code source des *librairies* sérieuses comme celles fournies par la JDK.

Les raisons de l'importance du nommage sont les suivantes.

- Un nom bien choisi rend plus facile la lecture du programme.
- Lorsqu'on n'arrive pas à choisir un nom adapté, c'est souvent parce que son rôle n'est pas bien défini.
- Un programmeur passe 80% de son temps à lire le programme : il faut lui faciliter ce travail surtout qu'on peut passer beaucoup de temps à relire son propre code.
- Des incohérences logiques évidentes peuvent sauter aux yeux avec un bon nommage.
- Toutes les *librairies* sont codées en anglais.

2.2 Nommage des variables/arguments/attributs

Il y a certains écueils à éviter lorsqu'on cherche à trouver des noms pour des variables. Vous ne devez pas avoir des variables avec des :

- noms en une lettre (même pour les indices) : `i`, `j`, ...
- noms numérotés : `a1`, `a2`, `a3`, ...
- abréviations ayant plusieurs interprétations : `rec`, `res`, ...
- noms ne donnant pas le sens précis : `temporary`, `result`, ...
- noms trompeurs : par exemple un `accountList` doit être une `List` (et pas un `array` ou un autre type)
- types de l'objet au singulier pour une collection d'objet : une liste de personnes doit s'appeler `persons` et non `person`
- noms imprononçables : `genymdhms`, ...

Comme toute les bonnes pratiques, ce ne sont pas des règles absolues et on peut évidemment y déroger. Par exemple, il est généralement accepté d'utiliser `x` et `y` pour les coordonnées d'un Point comme cela est fait pour la classe `Point2D` de JavaFX. Néanmoins, comme pour toutes les règles qui ne sont pas absolues, il est nécessaire de se poser la question de savoir si on fait bien de ne pas les respecter.

Le **Java** et la quasi-totalité des langages de programmation utilise l'anglais (dans les mot-clés et les librairies standards). Par conséquent, on se doit de programmer en anglais pour avoir la cohérence du code. Utiliser l'anglais permet aussi d'augmenter le nombre de personnes pouvant lire le code et donc d'avoir de nombreux exemples existants pour s'en inspirer.

Une autre règle importante pour le nommage est de respecter le Code Style de Java pour le nommage des variables/arguments/attributs. Pour les noms composés d'un seul mot, on écrit tout simplement le mot en

minuscule. Pour un nom composé de plusieurs mots, on n'utilise ni espace ni ponctuation, et on sépare les mots en mettant en capitale la première lettre de chaque mot. Par exemple, cela donne : `flaggedCells`, `gameBoard`, ... Si le nom comportait des lettres en dehors des 26 caractères non-accentué classique de l'anglais (de `a` à `z`), on les remplace par des caractères inclus dans les 26 caractères de base. Par exemple, "root computed by Müller's method" devient `rootComputedByMuellersMethod`.

2.3 Nommage des méthodes

Comme pour les variables, il est important de bien nommer les méthodes. Le nom d'une méthode doit décrire le service rendu à celui qui l'appelle, et non pas comment elle le fait. La convention de nommage est la même que pour les variables (capitale pour la première lettre de chaque mot sauf le premier). Dans une très grande majorité des cas, la méthode se trouve dans une des catégories suivantes :

— **Ordre** : méthodes exécutant une action avec comme sujet l'objet avec lequel la méthode est appelée. Dans ce cas, on utilise le groupe verbal à l'infinitif. Cela donne par exemple : `connection.open()`, `list.sort(comparator)`, `comparator.compare(object1, object2)`.
— **Requête booléenne** : méthodes testant un prédicat sur l'objet. Dans ce cas, on utilise un groupe verbal au présent. Cela donne par exemple : `connection.isClosed()`, `list.isEmpty()`, `list.contains(object)`, `node.hasNext()`, `frame.canClose()`.
— **Requête non-booléenne** : méthodes renvoyant une partie de l'état de l'objet. Dans ce cas, on utilise un groupe nominal ou bien un *getter*. Cela donne par exemple : `list.size()`, `connection.getMetaData()`.
— **Conversion** : méthodes convertissant l'objet en un objet d'un autre type. Dans ce cas, on utilise `to` suivi du type ciblé. Cela donne par exemple : `list.toArray(...)`, `object.toString()`.

Comme pour le nommage des variables, ces règles ne sont pas absolues mais juste des conventions qui peuvent avoir des exceptions. En général le plus simple est de s'inspirer de l'existant : par exemple la **JDK** et de bien réfléchir lorsqu'on souhaite déroger aux règles.

Si vous avez des difficultés à nommer vos méthodes, c'est sans doute qu'elles font trop d'actions. Une méthode devrait avoir au maximum une dizaine de lignes de code. Il est toujours possible de satisfaire à cette contrainte en extrayant le plus possible les parties du code d'une méthode à d'autres méthodes. Il est donc important de :

— réfléchir avant de coder au rôle de la méthode ;
— se demander ce qui peut être confié à d'autres méthodes.

Une fonction ne doit donc faire qu'**une seule chose**. Pour cela, elle ne doit réaliser que des étapes de même niveau d'abstraction. Nous allons illustrer cela sur l'action de cuisiner. Pour faire la cuisine on doit (premier niveau d'abstraction) :

— choisir une recette ;
— réunir les ingrédients ;
— suivre la recette.

Pour choisir une recette, on doit (deuxième niveau d'abstraction):

— réfléchir à ce que j'ai envie de manger ;

— chercher sur marmiton.

Considérons le code suivant pour une méthode `cook` :

```
void cook(){
    // On choisit la recette
    Food wantToEat = thinkAboutFood();
    Recipe recipe = lookOnMarmiton(wantToEat);
    // On réunit les ingrédients
    openFridge();
    for(Ingredient ingredient :
        recipe.getFreshIngredients()){
        takeInFrige(ingredient);
    }
    closeFridge();
    openCupboard();
    ...
    // On suit la recette
    ...
}
```

Dans le code de la méthode `cook`, on utilise un niveau d'abstraction trop bas. Cette approche n'est pas la bonne et on a été obligé de rajouter des commentaires pour indiquer les étapes du premier niveau d'abstraction. La bonne approche consiste à définir des méthodes correspondant aux étapes du premier niveau d'abstraction et de les appeler dans la méthode `cook`. Cela nous donne le code suivant :

```
    void cook(){
        Recipe recipe = chooseRecipe();
        gatherIngredients(recipe);
        followRecipe(recipe);
    }

    Recipe chooseRecipe(){
        Food wantToEat = thinkAboutFood();
        Recipe recipe = lookOnMarmiton(wantToEat);
        return recipe;
    }

    ...
```

Un autre écueil à éviter est de mentir dans le nom d'une méthode. Par exemple, si on considère la méthode `checkPassword` dans la classe ci-dessous :

```
class User {
    private boolean authenticated;
    private String password;

    public boolean checkPassword(String password) {
        if (password.equals(this.password)) {
            authenticated = true;
            return true;
        }
        return false;
    }
```

```
12 }
```

Cette méthode authentifie l'utilisateur alors qu'elle ne devrait que vérifier la validité du mot de passe d'après son nom. Il y a donc un mensonge (la méthode fait plus que ce qu'elle dit) ce qui complique fortement la compréhension du code. C'est donc un comportement à éviter.

2.4 Nommage des classes/interfaces/records/enums

En Java, pour tous les éléments du code qui correspondent à des types (c'est-à-dire des classes, interfaces, records ou enums), on utilise une majuscule pour la première lettre de chaque mot composant le nom du type (y compris le premier mot contrairement aux variables et méthodes). Cela donne par exemple : `Shape`, `Rectangle`, `ArrayList`, `MountainBike`. Un type définissant généralement une catégorie d'objet, le nom d'une classe/interface/record/enum correspond dans la plupart des cas à un groupe nominal au singulier. Cela vaut en particulier pour les `enum` de java qui doivent être au singulier. L'`enum` `DayOfWeek` qui définit les sept jours de la semaine est au singulier car un objet de type `DayOfWeek` correspond à un jour de la semaine.

Parfois, on a besoin de regrouper un certain nombre de fonctions `static` ensemble. C'est par exemple ce qui se passe pour `Math` qui contient des fonctions mathématiques de base ou Collections qui contient des méthodes s'appliquant sur des collections. Dans ce cas très particuliers, la classe ne permet pas de créer des objets (par exemple le constructeur de `Math` est privé et n'est pas appelé dans la classe) et donc la convention de nommage ne s'applique pas vraiment. Le nom de la classe doit juste donner les liens entre les différentes méthodes statiques qu'on a regroupées dans celle-ci.

3 Gestion de versions

3.1 Introduction

Comme on l'a déjà dit, il est essentiel lorsqu'on travaille sur un projet logiciel un peu conséquent d'utiliser un système de gestion de version. Les raisons sont les suivantes :

- Cela facilite le travail collaboratif sur un projet et c'est donc quasiment indispensable pour le travail en équipe sur un projet.
- Cela permet de documenter toutes les modifications effectuées.
- Cela permet de sauvegarder un travail sur un serveur distant, et ainsi de prévenir sa perte en cas de problème avec un ordinateur (vol ou panne).
- Cela permet de revenir en arrière, et donc donne un filet de sauvetage au programmeur peu confiant ; il ne prend pas de risque, car il pourra toujours revenir en arrière.

Les principes de base de la gestion de version sont les suivants :

- Le code d'un projet est stocké dans un serveur.
- Les développeurs soumettent des modifications avec des commentaires à chaque fois.
- Le serveur conserve l'historique des mises à jour

Dans cet enseignement, nous allons utiliser le gestionnaire de version git. Les raisons sont les suivantes :

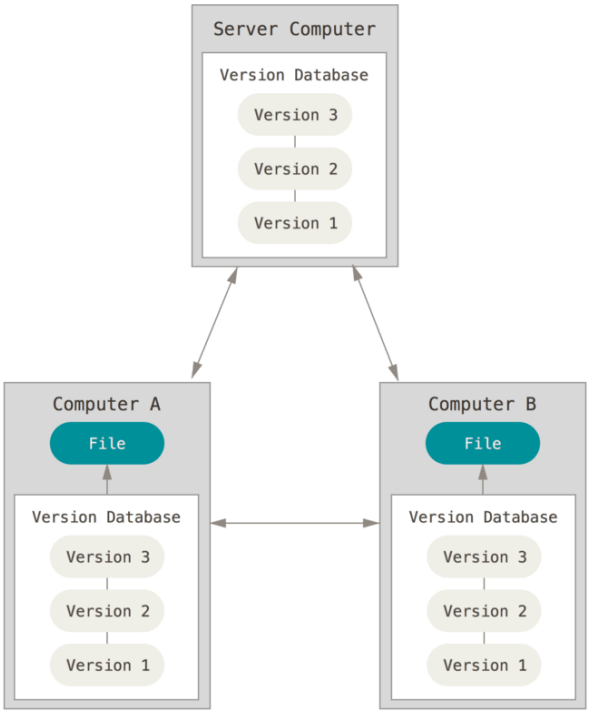
— c'est actuellement le logiciel de gestion de version le plus populaire et le plus utilisé ;
— il existe des serveurs gratuits : github ou gitlab
— il existe une version libre de logiciel serveur : gitlab auto-géré
— git permet la gestion de version décentralisée : la gestion de version se fait aussi en local ce qui permet de faire la gestion de versions sans accès au serveur.

On peut exécuter git via l'IDE (IntelliJ IDEA intègre la gestion de version dans ces menus) ou bien directement en ligne de commande.

De manière simple, l'utilisation que vous allez faire de git va suivre le déroulement suivant :

— À la première utilisation, on crée une copie locale du dépôt git (`clone` ou `init`).
— À chaque commit :
  — on récupère la version courante du dépôt sur le serveur (`pull`) ;
  — on ajoute les fichiers à modifier (`add`) ;
  — on finalise le commit en donnant un message résumant les modifications (`commit`).
— Après un ou plusieurs commits, on met à jour la version distante avec nos modifications (`push`).

Git est un système de gestion de version distribué. Dans un tel système et contrairement aux systèmes de gestion de version non-distribués, les clients n'extraient pas seulement la dernière version d'un fichier, mais ils dupliquent complètement le dépôt. Ainsi, si le serveur disparaît et si les systèmes collaboraient via ce serveur, n'importe quel dépôt d'un des clients peut être copié sur le serveur pour le restaurer. Chaque extraction devient une sauvegarde complète de toutes les données.



Les explications qui suivent sont tirées du Pro Git book.

3.2 Utilisation basique de Git

3.2.1 Première utilisation de Git

La première chose à faire est d'installer Git ce que vous pouvez faire en suivant les instructions au lien suivant : https://git-scm.com/book/fr/v2/D%C3%A9marrage-rapide-Installation-de-Git.

La première chose à faire après l'installation de Git est de renseigner votre nom et votre adresse de courriel. C'est une information importante car toutes les validations dans Git utilisent cette information.

```
$ git config --global user.name "Prénom Nom"
$ git config --global user.email votre.adresse@etu.uni-amu.fr
```

Vous pouvez obtenir de l'aide sur les commandes git à l'aide de la commande *help* :

```
$ git help nom_de_la_commande
```

3.2.2 Démarrer un dépôt Git

Vous pouvez démarrer un dépôt Git de deux manières.

— Vous pouvez prendre un répertoire existant et le transformer en dépôt Git.
— Vous pouvez cloner un dépôt Git existant sur un autre serveur.

Pour créer un dépôt local, il suffit d'appeler la commande `git init` dans le répertoire dans lequel vous voulez démarrer votre dépôt. Cela crée un nouveau sous-répertoire nommé `.git` qui contient tous les fichiers nécessaires au dépôt. Pour l'instant, aucun fichier n'est encore versionné.

Si vous souhaitez démarrer le contrôle de version sur des fichiers existants (par opposition à un répertoire vide), vous devrez probablement suivre ces fichiers et faire un commit initial. Vous pouvez le réaliser avec quelques commandes `add` qui spécifient les fichiers que vous souhaitez suivre, suivies par un `git commit` :

```
$ git add *.java
$ git commit -m 'initial project version'
```

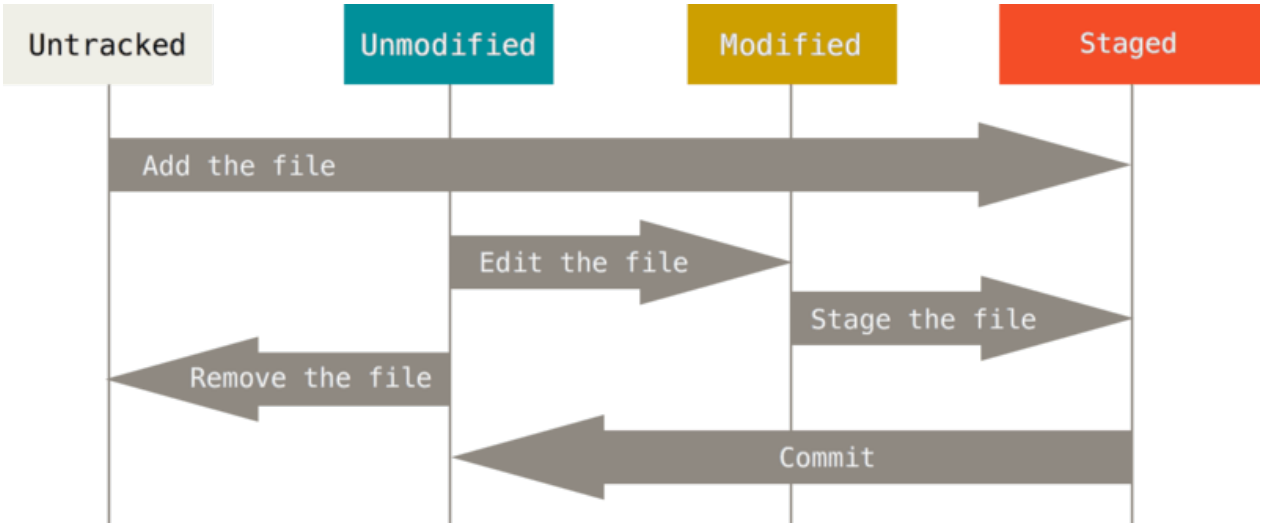
Pour obtenir une copie d'un dépôt Git existant, il faut utiliser la commande `git clone`. Vous clonez un dépôt avec `git clone url` avec url l'adresse serveur du dépôt.

3.2.3 Enregistrer des modifications dans le dépôt

Une fois que vous avez un dépôt Git valide et une extraction ou copie de travail du projet. Vous devez faire quelques modifications et valider des instantanés de ces modifications dans votre dépôt à chaque fois que votre projet atteint un état que vous souhaitez enregistrer.

Chaque fichier de votre copie de travail peut avoir deux états : sous suivi de version ou non suivi (*untracked*). Les fichiers suivis sont les fichiers qui appartenaient déjà au dernier instantané; ils peuvent être inchangés (*unmodified*), modifiés (*modified*) ou indexés (*staged*). En résumé, les fichiers suivis sont ceux que Git connaît. Tous les autres fichiers sont non suivis. Quand vous clonez un dépôt pour la première fois, tous les fichiers seront sous suivi de version et inchangés, car Git vient tout juste de les extraire et vous ne les avez pas encore édités.

Au fur et à mesure que vous éditez des fichiers, Git les considère comme modifiés, car vous les avez modifiés depuis le dernier instantané. Vous indexez ces fichiers modifiés et vous enregistrez toutes les modifications indexées, puis ce cycle se répète.



L'outil principal pour déterminer quels fichiers sont dans quel état est la commande `git status`. Supposons que vous souhaitez ajouter un nouveau fichier `README.md` que vous venez de créer. Ce fichier n'est pas en suivi de version. Pour commencer à suivre ce nouveau fichier, il faut utiliser la commande `git add` suivi du nom de fichier. Vous pouvez entrer ceci :

```
1 $ git add README.md
```

Si vous lancez à nouveau la commande git status, vous pouvez constater que votre fichier `README.md` est maintenant suivi et indexé. La commande `git add` accepte en paramètre un chemin qui correspond à un fichier ou un répertoire ; dans le cas d'un répertoire, la commande ajoute récursivement tous les fichiers de ce répertoire.

Il est aussi possible d'indexer (*stage*) des fichiers déjà suivis afin d'enregistrer par la suite dans le dépôt la modification du fichier. La commande `git add` est multi-usage : elle peut être utilisée pour placer un fichier sous suivi de version, pour indexer un fichier ou pour d'autres actions telles que marquer comme résolus des conflits de fusion de fichiers. Sa signification s'approche plus d'ajouter ce contenu pour la prochaine validation que d'ajouter ce contenu au projet.

3.2.4 Valider vos modifications

Maintenant que vous avez choisis les fichiers indexés, c'est-à-dire les fichiers dont les ajouts ou modifications seront stocké dans le dépôt, vous pouvez valider votre mise-à-jour. Souvenez-vous que tout ce qui est encore non indexé — tous les fichiers qui ont été créés ou modifiés, mais n'ont pas subi de `git add` depuis que vous les avez modifiés — ne feront pas partie de la prochaine validation. Ils resteront en tant que fichiers modifiés sur votre disque. Dans notre cas, la dernière fois que vous avez lancé `git status`, vous avez vérifié que tout était indexé, et vous êtes donc prêt à valider vos modifications. La manière la plus simple de valider est de taper `git commit`.

Vous constatez que le message de validation par défaut contient une ligne vide suivie en commentaire par le résultat de la commande `git status`. Vous pouvez effacer ces lignes de commentaire et saisir votre propre message de validation, ou vous pouvez les laisser en place pour vous aider à vous rappeler ce que vous êtes en train de valider.

Autrement, vous pouvez spécifier votre message de validation en ligne avec la commande git commit en le saisissant après l'option -m, comme ceci :

```
1 $ git commit -m "Story 182: Fix benchmarks for speed"
```

Souvenez-vous que la validation enregistre l'instantané que vous avez préparé dans la zone d'index. À chaque validation, vous enregistrez un instantané du projet en forme de jalon auquel vous pourrez revenir ou avec lequel comparer votre travail ultérieur.

3.2.5 Travailler avec des dépôts distants

Il y a plusieurs manières de travailler avec un dépôt distant. Si vous avez cloné un dépôt avec la commande `git clone`, votre dépôt est automatiquement lié au dépôt distant que vous avez cloné. Si vous avez besoin d'ajouter un nouveau dépôt distant Git, il faut exécuter la commande `git remote add url` avec `url` l'adresse du dépôt.

Lorsque votre dépôt vous semble prêt à être partagé, il faut le pousser en amont, c'est-à-dire envoyer vos commits dans le dépôt distant. La commande pour le faire est simple : `git push`.

Cette commande ne fonctionne que si vous avez cloné depuis un serveur sur lequel vous avez des droits d'accès en écriture et si personne n'a poussé dans l'intervalle. Si vous et quelqu'un d'autre clonez un dépôt au même moment et que cette autre personne pousse ses modifications et qu'après vous tentez de pousser les vôtres, votre poussée sera rejetée. Vous devrez tout d'abord tirer (commande `git pull`) les modifications de l'autre personne et les fusionner avec les vôtres avant de pouvoir pousser.

4 Tests

4.1 Qu'est-ce qu'un test en programmation ?

Il est important de tester le code que l'on écrit afin d'avoir des garanties sur son bon fonctionnement. Un code non testé n'a aucune valeur et par conséquent tout code doit être testé. En effet, même le développeur le plus aguerri peut faire des fautes d'inattention. Même une petite erreur peut faire perdre énormément de temps en débogage. Il est donc important de tester son code au fur et à mesure de sa production afin de trouver au plus tôt de telles erreurs. Il est difficile de donner une définition d'un test, car ceux-ci peuvent prendre tout un tas de forme. Rien qu'exécuter le code qu'on a écrit et vérifier à la main son comportement constitue en soi un test. C'est d'ailleurs généralement la dernière étape du processus de test d'un logiciel. En effet, il est souvent très complexe de tester automatiquement le bon comportement d'une interface utilisateur d'une application. Un autre type de test est le test automatique qui consiste en du code utilisant le code à tester et vérifiant sans contrôle d'une personne le bon fonctionnement du programme. C'est sur ce type de test que nous allons nous concentrer dans cet enseignement. Les tests constituent donc une étape importante du développement logiciel

et il existe une nomenclature donnant des noms aux différents types de tests. Les principaux types de test sont les suivants :

— **Tests unitaires** : Tester les différentes parties (méthodes, classes) d'un programme indépendamment les unes des autres.
— **Tests d'intégration** : Tester le bon comportant de partie de programmes formant un tout cohérent appelée module. Cela permet de tester le fonctionnement d'instances issues de classes différentes interagissant ensemble ce qui n'est pas possible avec les tests unitaires.
— **Tests systèmes (anciennement test fonctionnel)** : Tester que le fonctionnement constaté est identique à celui attendu dans des situations réelles d'utilisation et donc vérifier des scénarios de tests ou des schémas d'utilisation de bout en bout.

Par la suite, nous allons nous concentrer sur les tests unitaires.

4.2 Tests unitaires

Le point important à comprendre sur les tests unitaires est qu'ils testent de petites parties du code (généralement pas plus qu'un méthode appliquée sur un objet). Contrairement aux autres types de tests, les tests unitaires :

— ne communiquent pas avec une base de données ou par le réseau (pas d'action complexe) ;
— ne génèrent ni ne modifient de fichiers (mais ils peuvent en lire) ;
— peuvent être lancés en même temps que d'autres tests unitaires.

L'objectif d'un test unitaire est de permettre au développeur de s'assurer qu'une unité de code ne comporte pas d'erreurs. Dans un test de ce type, une petite partie de code est exécutée. Dans ce cours, on considérera que l'unité de code est la méthode. La plupart des tests consisteront donc à tester que le comportement d'une méthode est bien celui qu'on a défini dans la spécification de la méthode. Tester revient à vérifier que sur certains cas choisis (c'est généralement impossible de tester tous les cas, car ils sont trop nombreux), la méthode produit bien le résultat attendu. Considérons une méthode `void sort()` s'appelant sur une liste d'entiers et qui trie les entiers de la liste par ordre croissant. Tester cette méthode revient à vérifier que pour certaines listes, la liste après l'appel à la méthode `sort` contient bien les mêmes entiers que la liste de départ dans l'ordre croissant. Il est impossible de tester toutes les listes possibles, car il y en a une infinité (si on ne considère pas les limites de mémoire). Même dans les cas où on aurait un nombre fini de possibilités, il y en a souvent trop pour pouvoir tout tester dans un temps raisonnable.

Vous avez sans doute déjà testé votre code à la main (donc de manière non-automatique) en lançant le code et vérifiant votre code soit via des outils de débogage (avec des points de contrôles pour exécuter le code pas à pas) ou bien directement en vérifiant le bon comportement de votre programme via une utilisation normale de celui-ci. Ici, on peut se concentrer sur des tests automatisés. Un test sera donc un programme qui va appeler une méthode et vérifier que son comportement respecte les spécifications (le contrat de la méthode) via des assertions. Concrètement, un test, c'est du code qui vérifie que des assertions sont vraies. Généralement, un test unitaire consistera an trois étapes :

— créer un objet;
— appeler une méthode sur cet objet;

— vérifier que le résultat est bien le résultat attendu.

Bill Wake, auteur de *Refactoring Workbook*, a inventé le terme les trois “A” pour décrire ces étapes : *Arrange, Act, Assert* que l'on pourrait traduire par organiser, agir, vérifier. Se souvenir des trois “A” vous permet de rester concentré sur l'écriture d'un test unitaire efficace. Les tests produits via cette méthode sont reproductibles (pouvant être répétés avec le même résultat) puisqu'ils vérifient un comportement prévisible. De plus, puisque chaque test crée son objet afin de le tester, il est facile de tous les exécuter simultanément, car ils n'y a pas de dépendances entre les tests. Généralement, ce qu'on va faire, c'est regrouper des tests (par exemple les tests de toutes les méthodes d'une classe) en seule classe de test ce qui va nous permettre de les exécuter ensemble facilement. En effet, si l'objet est dans cet état et que je fais cela, alors cela se produira. Une partie du défi de vérifier du code par le biais de tests unitaires consiste à réduire tous les comportements du système à ces cas ciblés et prévisibles. Toute la difficulté de l'utilisation des tests unitaires est de trouver des moyens d'extraire des tests simples et prévisibles à partir de logiciels complexes.

Les règles de base pour l'écriture de test sont les suivantes.

— Écrire au moins un classe de test par classe à tester. Le nom de la classe de test est généralement le nom de la classe testée suivie de `Test`. Par exemple la classe de test d'une classe `Vector` est appelée `VectorTest`.
— Écrire une méthode de test par cas à tester. Le nom de la méthode de test doit indiquer le comportement de l'objet qui est testé. Une manière répandue d'écrire le nom d'une méthode de test est de construire le nom de la méthode de test en mettant `test`, suivi du nom de la méthode, suivi d'un `_` et du comportement testé. Par exemple, un test vérifiant le comportement d'une méthode `withdraw` (retrait) sur un compte en banque à découvert (*overdrawn*) pourra être nommée `testWithdraw_Overdrawn`.
— Pour une méthode à tester, il faut tester :
  — **les cas normaux**, utilisation naturelle de la méthode sur une donnée naturelle, par exemple un retrait d'argent d'un montant strictement positif ;
  — **les cas limites**, utilisation de la méthode sur une donnée “étrange”, par exemple par exemple un retrait d'argent d'un montant égal à 0 ;
  — **les cas anormaux**, vérification que les erreurs d'utilisation, c'est-à-dire que les cas d'erreurs sont bien pris en compte et gérés, par exemple un retrait d'argent d'un montant strictement négatif.

4.3 JUnit et assertJ

Afin d'automatiser les tests et de faciliter leur écriture, vous allez utiliser JUnit 5 avec assertJ. Junit est le framework de test unitaire pour Java le plus utilisé alors qu'AssertJ est une bibliothèque permettant de faciliter l'écriture d'assertions. Une bonne pratique à respecter est de séparer le code de test du code principal (code du logiciel principal aussi appelé code de production). Comme nous l'avons déjà indiqué précédemment, nous allons travailler avec le moteur de production `Gradle`. Dans un projet `Gradle`, le code de production en Java est dans le répertoire `src/main/java` et le code de test est dans le répertoire `src/test/java`. Il est nécessaire de séparer les tests du code de production car :

— on ne donne pas l'accès au code de test au client par exemple
— les tests ont un rôle spécifique différent du code de production

La commande `gradle test` (ou `./gradlew test` si vous utilisez le *wrapper* de Gradle) lancé à la racine de votre projet permet de lancer tous vos tests.

En JUnit 5, une méthode de test de base :

- a un modificateur d'accès `default` (pas de modificateur d'accès);
- est annotée avec `@Test` (à mettre avant la déclaration de la méthode);
- ne prend aucun paramètre;
- ne renvoie rien (retour de type `void`);
- contient des assertions `assertThat` qui lèvent une `assertionError` si elles sont fausses (échec du test).

Il existe des méthodes de test plus complexes (comme des méthodes de tests paramétrés) mais on en n'expliquera pas leur syntaxe durant cet enseignement. Le code d'une classe de test avec `JUnit 5` et `AssertJ` aura donc le format suivant :

```
1 import org.junit.jupiter.api.Test;
2 import static org.assertj.core.api.Assertions.*;
3
4 public class NameTestedClassTest {
5   @Test
6   void testNameTestedMethod_BehaviorTested(){
7     // code containing assertions to test nameTestedMethod
8   }
9 }
```

L'import de la première ligne permet d'utiliser l'annotation `@Test` alors que l'import de la deuxième ligne permet d'avoir accès aux assertions définies par AssertJ. La documentation de ces assertions est disponible à ce [lien][https://www.javadoc.io/doc/org.assertj/assertj-core/latest/org/assertj/core/api/Assertions.html).

Les assertions les plus utilisées d'AssertJ s'appellent de la manière suivante.

- `assertThat(condition).isTrue()` : vérifie que `condition` est vraie.
- `assertThat(condition).isFalse()` : vérifie que `condition` est faux.
- `assertThat(actual).isEqualTo(expected)` : vérifie que `expected` est égal à `actual` égal : `equals` pour les objets et `==` pour les types primitifs.
- `assertThat(actual).isCloseTo(expected, within(delta))` : vérifie que $|expected - actual| \leq delta$
- `assertThat(object).isNull()` : vérifie que la référence est `null`
- `assertThat(object).isNotNull()` : vérifie que la référence **n'**est **pas** `null`
- `assertThat(actual).isSameAs(expected)` : vérifie que les deux objets sont les mêmes (même référence).
- `assertThat(list).containsExactly(e1, e2, e3)` : vérifie que la liste `list` contient uniquement les éléments `e1`, `e2` et `e3` dans cet ordre.
- `assertThat(list1).containsExactlyElementsOf(list2)` : vérifie que les deux listes `list1` et `list2` contiennent les mêmes éléments dans le même ordre.
- `fail(message)` : échoue toujours en affichant message.

Il est possible de capturer une exception avec le code `Throwable thrown = catchThrowable(()-> { /*code that can throw an exception */})` puis de tester des propriétés sur l'exception. Par exemple, on peut tester

le contenu du message avec `assertThat(thrown).hasMessageContaining(text)`.

Il est possible de provoquer l'affichage d'un `message` lors d'un test faux en appelant `as(message)` sur le retour d'un `assertThat`. Par exemple, l'assertion suivante `assertThat(1+1).as("One plus one should be two").isEqualTo(2)` afficherait `"One plus one should be two"` en cas d'échec du test.

4.4 Exemples de tests unitaires

Afin d'illustrer le fonctionnement des tests unitaires et la manière de tester son code via ceux-ci, le plus simple est de prendre des exemples de classes dont on va tester le comportement.

4.4.1 Exemple de classe à tester : `RationalNumber`

On va commencer par une classe `RationalNumber` qui permet de représenter des nombres rationnels sous la forme de fraction de deux entiers et d'effectuer des opérations d'addition et de multiplications sur ceux-ci. Le code de la classe est le suivant :

```java
public class RationalNumber {
  public final int numerator;
  public final int denominator;

  public RationalNumber(int numerator, int denominator) {
    int gcd = gcd(numerator, denominator);
    this.numerator = numerator / gcd;
    this.denominator = denominator / gcd;
  }

  public RationalNumber add(RationalNumber val) {
    int numerator = (this.numerator * val.denominator)
      + (this.denominator * val.numerator);
    int denominator = this.denominator * val.denominator;
    return new RationalNumber(numerator, denominator);
  }

  public RationalNumber multiply(RationalNumber val) {
    int numerator = this.numerator * val.numerator
    int denominator = this.denominator * val.denominator;
    return new RationalNumber(numerator, denominator);
  }

  private static int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
  }
}
```

Pour ce premier cas assez simple, nous allons nous contenter de tester le bon comportement des deux méthodes `add` et `multiply` qui permettent respectivement d'ajouter et de multiplier deux nombres rationnels. Cela nous donne le code de test suivant dans une classe de test `RationalNumberTest` :

```java
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

```

```java
public class RationalNumberTest {
  @Test
  void testAdd(){
    RationalNumber one = new RationalNumber(1, 1);
    RationalNumber onePlusOne = one.add(one);
    assertThat(onePlusOne.numerator)
            .as("Numerator of one plus one is two.")
            .isEqualTo(2);
    assertThat(onePlusOne.denominator)
            .as("Denominator of one plus one is one.")
            .isEqualTo(1);
  }

  @Test
  void testMultiply(){
    RationalNumber twoThirds = new RationalNumber(2, 3);
    RationalNumber twoThirdsTimesTwoThirds = twoThirds.multiply(twoThirds);
    assertThat(twoThirdsTimesTwoThirds.numerator)
            .as("Numerator of twos thirds times two thirds should be four.")
            .isEqualTo(4);
    assertThat(twoThirdsTimesTwoThirds.denominator)
            .as("Denominator of twos thirds times two thirds should be nine.")
            .isEqualTo(9);
  }
}
```

On peut remarquer qu'on utilise `as` afin de donner des messages explicites en cas d'échec des tests. Ici, nous n'avons testé les méthodes qu'une fois et donc pour une seule paire de valeurs pour chaque opération. Généralement ce n'est pas suffisant, car on peut avoir la malchance que le test passe pour une valeur possible des objets. Une analogie est de considérer une horloge bloquée qui donne la bonne heure deux fois par jour. Même un code complètement faux peut donner la bonne réponse pour une valeur précise des arguments et de l'objet avec lequel elle est appelée. Il est donc important de tester chaque méthode avec au moins deux valeurs possibles des paramètres.

4.4.2 Exemple de classes à tester : `Box` et `Robot`

On va maintenant considérer une classe `Box` permettant de créer une boite avec un certain poids. Le code de la classe avec la documentation est le suivant :

```java
public class Box {
  /**
   * Create a box with the specified weight
   * @param weight the weight of the created box
   */
  public Box(int weight) {
    this.weight = weight;
  }

  /**
   * weight of the box
   */
  private int weight;

```

```
15    /**
16     * @return this box's weight
17     */
18    public int getWeight() {
19      return this.weight;
20    }
21  }
```

Il est assez facile de tester le code de la classe `Box` avec le code suivant :

```
1  import static org.assertj.core.api.Assertions.*;
2  import org.junit.jupiter.api.Test;
3
4  public class BoxTest {
5    @Test
6    public void testGetWeight() {
7      Box someBox = new Box(10);
8      assertThat(someBox.getWeight()).isEqualTo(10);
9      Box otherBox = new Box(100);
10     assertThat(otherBox.getWeight()).isEqualTo(100);
11   }
12 }
```

Maintenant, on souhaite créer une classe `Robot` permettant d'instancier des robots portant des caisses. La spécification informelle de cette classe est la suivante :

> Un robot peut porter une caisse d'un poids maximal défini à la construction du robot. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse, il ne peut en prendre une autre.

La classe `Robot` définira donc les éléments suivants :

- `Robot(int maxWeight)` : un constructeur qui permet d'instancier un robot pouvant transporter une boite dont le poids est inférieur ou égal au poids spécifié en argument ;
- `boolean isCarryingABox()` une méthode qui renvoie `true` si le robot porte une caisse et `false` sinon ;
- `boolean takeBox(Box box)` une méthode qui fait transporter par le robot la boite spécifiée en argument à deux conditions : le robot ne doit pas déjà transporter une boite et la boite doit peser moins que le poids que peut transporter le robot ;
- `Box getCarriedBox()` une méthode qui renvoie la boite transportée par le robot.

On va inverser le processus classique de développement en écrivant d'abord les tests. C'est une méthode appelée *Test-First Design*. On va donc commencer par appliquer les trois "A" (*Arrange, Act, Assert*) pour concevoir des tests unitaires :

1. **Arrange** : créer la situation initiale et vérifier les « préconditions ». Il est important de vérifier les préconditions, car cela permet de contrôler que l'état de l'objet est correct avant l'appel à la méthode testée. En effet, un test sur une méthode pourrait échouer à cause d'une erreur de code dans le constructeur de l'objet. Il est donc essentiel dans la mesure du possible de vérifier que l'état de l'objet avant l'appel à la

méthode testée est bien celui voulu.

2. **Act** : appeler la méthode testée. Si cette méthode renvoie une valeur, il est important de la stocker dans une variable afin de vérifier que sa valeur est correcte.
3. **Assert** : à l'aide d'assertions, vérifier les postconditions, c'est-à-dire la situation attendue après l'exécution de la méthode. Cela peut nécessiter plusieurs assertions.

On va commencer par tester qu'un robot créé ne porte pas de caisse (deuxième phrase de la spécification).

```
1  public class RobotTest {
2    @Test
3    public void notCarryingABoxWhenCreated() {
4      Robot robot = new Robot(15);
5      // no carried box ?
6      assertThat(robot.isCarryingABox()).isFalse();
7    }
8    }
```

On va continuer par un test de la méthode `takeBox` sur une boite d'un poids inférieur à la capacité de transport du robot (première phrase de la spécification) :

```
1  public class RobotTest {
2    @Test
3    public void robotTakeBox_LightEnoughBox() {
4      // initial configuration : a robot and a box
5      Robot robot = new Robot(15);
6      Box box = new Box(10);
7      // precondition : the robot does not carry a box
8      assertThat(robot.isCarryingABox()).isFalse();
9      // execution of the tested method
10     boolean boxTaken = robot.takeBox(box);
11     // postcondition : the carried box is the box taken
12     assertThat(boxTaken).isTrue();
13     assertThat(robot.isCarryingABox()).isTrue();
14     assertThat(robot.getCarriedBox()).isSameAs(box);
15   }
16 }
```

On continue avec un test de la méthode `takeBox` pour une boite d'un poids strictement supérieur à la capacité de transport du robot (toujours la deuxième phrase de la spécification) :

```
1  public class RobotTest {
2    @Test
3    public void robotTakeBox_TooHeavyBox() {
4      Robot robot = new Robot(15);
5      Box b = new Box(20);
6      // precondition : robot does not carry a box
7      assertThat(robot.isCarryingABox()).isFalse();
8      // execution of the tested method
9      boolean boxTaken = robot.takeBox(b);
10     // postcondition : no box is carried
11     assertThat(boxTaken).isFalse();
12     assertThat(robot.isCarryingABox()).isFalse();
13   }
```

```
14 }
```

Finalement, on teste l'appel à `takeBox` sur un robot transportant déjà une boite (troisième phrase de la spécification).

```
1    @Test
2    public void robotCanTakeOnlyOneBox() {
3      Robot robot = new Robot(15);
4      Box box1 = new Box(10);
5      Box box2 = new Box(4);
6      robot.takeBox(box1);
7      // precondition : the carried box is box1
8      assertThat(robot.getCarriedBox()).isSameAs(box1);
9      // execution of the tested method
10     boolean boxTaken = robot.takeBox(box2);
11     // postcondition: the carried box is not box2 and is box1
12     assertThat(boxTaken).isTrue();
13     assertThat(robot.getCarriedBox()).isNotSameAs(box2)
14             .isSameAs(box1);
15   }
```

Maintenant que l'on a écrit les tests pour la méthode `takeBox`, on peut écrire le code de production de la classe. Le fait d'avoir défini clairement le comportement voulu de la méthode (sa spécification) et qu'on a testé celui-ci nous permet de coder cette méthode sans crainte.

```
1  /**
2   * A class to instantiate robots that can carry boxes.
3   */
4  public class Robot {
5    /**
6     * The maximal weight of a box that can be carried by the robot.
7     */
8    private final int maxWeight;
9    /**
10    * The box carried by robot.
11    */private Box carriedBox;
12
13   /**
14    * Creates a robot that carry a box with a weight up to the specified
15    * weight.
16    *
17    * @param maxWeight The maximal weight of a box that can be
18    *                  carried by the robot.
19    */
20   public Robot(int maxWeight){
21     this.maxWeight = maxWeight;
22   }
23
24   /**
25    * Returns {@code true} if this robot carries a box.
26    *
27    * @return {@code true} if this robot carries a box
28    */
29   public boolean isCarryingABox(){
30     return carriedBox != null;
```

```
31    }
32
33    /**
34     * Makes the robot carry the specified box if it does not
35     * already carry a box and the box has a weight less or
36     * equal than the maximal weight that can be carried by
37     * the robot.
38     *
39     * @param box the box to be taken by the robot
40     * @return {@code true} if the box was taken by the robot
41     */
42    public boolean takeBox(Box box){
43      if(isCarryingABox() || box.getWeight() > maxWeight)
44        return false;
45      carriedBox = box;
46      return true;
47    }
48
49    /**
50     * Returns the box carried by the robot.
51     * @return the box carried by the robot.
52     */
53    public Box getCarriedBox() {
54      return carriedBox;
55    }
56  }
```

4.4.3 Exemple de classe à tester : `Emails`

Pour donner un dernier exemple, on va donner le code d'une classe `Emails` qui permet d'obtenir les noms d'utilisateurs à partir d'un texte contenant des adresses mails. Le nom d'utilisateur d'une adresse mail est la partie du texte contenu avant l'arobase (par exemple `arnaud.labourel` pour l'adresse arnaud.labourel@univ-amu.fr) et on considère que tout caractère différent d'un lettre, d'un chiffre ou d'un point sépare les adresses.

La classe `Emails` définira donc les méthodes suivantes :

— `Emails(String text)` : un constructeur qui permet d'instancier des emails à partir d'un texte
— `List<String> userNames()` : une méthode qui renvoie la liste des noms d'utilisateurs des emails.

On va commencer par tester la méthode `userNames` sur un cas assez simple qui correspond à une utilisation sur des données normales.

```
1  import org.junit.jupiter.api.Test;
2  import static org.assertj.core.api.Assertions.*;
3
4  public class EmailsTest{
5    @Test
6    public void testUserNames_NormalNames() {
7      Emails emails =
8        new Emails("foo bart@cs.edu xyz marge@ms.com baz");
9      assertThat(emails.getUserNames())
10       .containsExactly("bart", "marge");
11   }
```

Une fois ce premier test écrit, on peut s'attaquer au code de la classe `Emails` :

```
1  public class Emails {
2    private String text;
3
4    public Emails(String text) {
5      this.text = text;
6    }
7
8    public List<String> userNames() {
9      int pos = 0;
10     List<String> users = new ArrayList<String>();
11     for(;;) {
12       int atIndex = text.indexOf('@', pos);
13       if (atIndex == -1) break;
14       String userName = userName(atIndex);
15       if (userName.length() > 0) users.add(userName);
16       pos = atIndex + 1;
17     }
18     return users;
19   }
20
21   private String userName(int atIndex) {
22     int back = atIndex - 1;
23     while (back >= 0 &&
24             (Character.isLetterOrDigit(text.charAt(back))
25             || text.charAt(back) == '.')) {
26       back--;
27     }
28     return text.substring(back + 1, atIndex);
29   }
30 }
```

Afin de vérifier que notre code est correct, on peut maintenant le tester sur des adresses dont les noms d'utilisateurs sont étranges (composé que d'une lettre). Ici, c'est assez différent de ce que l'on a fait précédemment avec les tests sur les robots dans le sens où on teste la même méthode avec des cas potentiellement de plus en plus complexes à gérer (et non pas un comportement différent pour une autre situation).

```
1  public class EmailsTest{
2    @Test
3    public void testUserNames_NamesWithOneCharacter() {
4      Emails emails = new Emails("x y@cs 3@ @z@");
5      assertThat(emails.getUserNames())
6        .isNotEmpty();
7        .containsExactly("y", "3", "z");
8    }
9  }
```

Finalement, on peut aussi tester notre méthode sur des cas limites, c'est-à-dire des adresses emails créées à partir de texte vide ou ne contenant aucune adresse.

```
1  public class EmailsTest{
2    @Test
3    public void testUserNames_NullNames() {
```

```
4       Emails emails = new Emails("no emails here!");
5       assertThat(emails.getUserNames()).isEmpty();
6       emails = new Emails("@@@");
7       assertThat(emails.getUserNames()).isEmpty();
8       emails = new Emails("");
9       assertThat(emails.getUserNames()).isEmpty();
10    }
11  }
```

4.5 Test unitaires (version courte)

En résumé, ce que vous devez retenir sur les tests sont les points suivants :

- Il est essentiel de tester son code : cela fait gagner énormément de temps en cas d'erreur ;
- principe des trois "A" (*Arrange, Act, Assert* pour concevoir un test :
  1. ***Arrange** : créer la situation initiale et vérifier les « préconditions » ;
  2. ***Act** : appeler la méthode testée ;
  3. ***Assert** :à l'aide d'assertions, vérifier les « postconditions » = situation attendue après l'exécution de la méthode
- plusieurs méthodes de tests peuvent être nécessaires pour tester la correction d'une méthode (une méthode par comportement possible de la méthode en fonction de l'état de l'objet) ;
- Écrire au moins une méthode de test pour chaque méthode du code de production.
- Il est important de tester tous les types de cas :
  - cas normaux (utilisation naturelle de la méthode sur une donnée naturelle)
  - cas limites (utilisation de la méthode sur une donnée "étrange")
  - cas anormaux (vérification que les erreurs d'utilisation, c'est-à-dire que les cas d'erreurs sont bien pris en compte et gérés)

4.6 La suite : le TDD (Test Driven Development)

4.6.1 Définition du TDD

Le *Test-Driven Development (TDD)*, ou développement piloté par les tests, est une méthode de développement de logiciel qui consiste à concevoir un logiciel par des itérations successives très courtes, telles que chaque itération est accomplie en formulant un sous-problème à résoudre sous forme d'un test avant d'écrire le code source correspondant, et où le code est continuellement remanié dans une volonté de simplification. D'une certaine manière, le TDD est la suite logique *Test-First Design*. La différence est qu'au lieu d'écrire tous les tests pour ensuite écrire tout le code de production, le développeur écrit d'abord un test simple qui échoue (car le code de production correspondant n'existe pas) pour ensuite écrire le code de production qui permet au test de passer et répète ce processus en rajoutant du code de test et de production tant que le code ne satisfait pas à la spécification du logiciel.

Les trois lois à respecter pour suivre la méthodologie du TDD telle que définies par Robert C. Martin dans « Professionalism and Test-Driven Development » sont les suivantes.

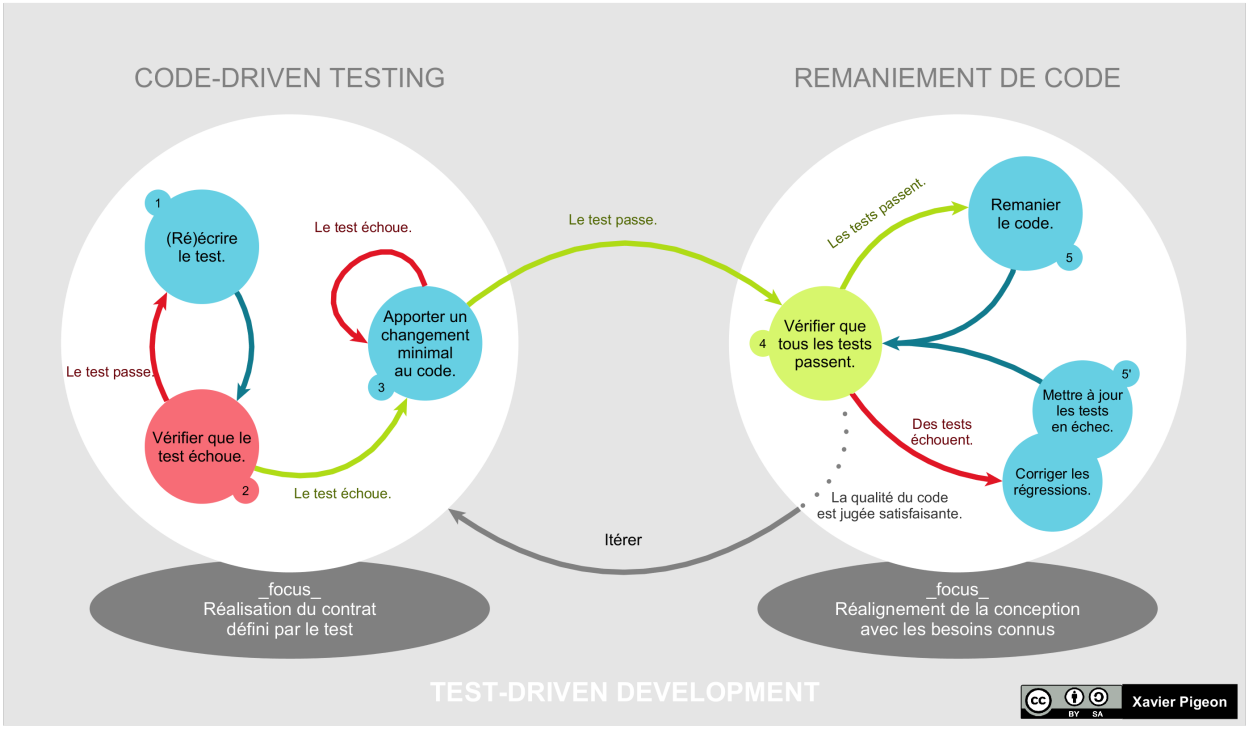
1. Écrivez un test qui échoue avant d'écrire le code de production correspondant.

2. Écrivez une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
3. Écrivez le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.

Le processus préconisé par la méthodologie TDD comporte cinq étapes.

1. Écrire un seul test qui décrit une partie du problème à résoudre. Il s'agit ici d'écrire un test très simple. La quantité de code de test ajouté doit être minime et le test ne doit vérifier qu'un petit ajout de fonctionnalité dans le code existant. De manière idéale, l'ajout ne devrait consister qu'en une seule ligne de code.
2. Vérifier que le test échoue, autrement dit qu'il est valide, c'est-à-dire que le code se rapportant à ce test n'existe pas encore.
3. Écrire juste assez de code pour que le test réussisse. Là aussi, le but est de rajouter la plus petite quantité de code possible afin de passer le test.
4. Vérifier que le test passe, ainsi que les autres tests existants. Il est important de s'assurer que le code ajouté n'a pas compromis les fonctionnalités existantes.
5. Remanier le code, c'est-à-dire l'améliorer sans en altérer le comportement, qu'il s'agisse du code de production ou du code de test. L'objectif est de simplifier le code le plus possible afin de le rendre le plus lisible possible. Cela passe en autre par la suppression éventuelle de code dupliqué.

Ce processus est répété en plusieurs cycles, jusqu'à résoudre le problème d'origine dans son intégralité. Ces cycles itératifs de développement sont appelés les micro-cycles de TDD. Ce processus est détaillé dans la figure ci-dessous qui a été produite par Xavier Pigeon.



4.6.2 Avantages du TDD

Dans le TDD, contrairement aux autres approches les tests font partie intégrante du processus d'écriture du code du logiciel. Avec le modèle d'organisation appelé cycle en V, les tests sont produits après l'étape d'implémentation et donc après l'écriture du code principal. Le TDD implique non seulement de commencer l'écriture du code par un test, mais impose aussi un aller-retour constant entre code de production et code de test. Les tests dans le processus TDD permettent d'explorer et de préciser le besoin, puis de spécifier le comportement souhaité du logiciel en fonction de son utilisation, avant chaque étape de codage. Le logiciel ainsi produit est tout à la fois pensé pour répondre avec justesse au besoin et conçu pour le faire avec une complexité minimale. On obtient donc un logiciel mieux conçu, mieux testé et plus fiable, autrement dit de meilleure qualité.

Quand les tests sont écrits après l'écriture du code, les choix d'implémentation contraignent leur écriture. Dans certains cas cela peut rendre le code difficile, voire impossible à tester (ou en tout cas trop coûteux en termes de temps de développement). Le processus de TDD qui impose d'écrire les tests d'abord force le développeur à faire des choix d'implémentation facilitant les tests. Cette propriété de testabilité du code favorise une meilleure conception ce qui permet d'éviter des erreurs de conception courantes.

Une autre propriété importante obtenue par le respect du processus de TDD est que chaque petite partie du code est associé à un test. Il est donc normalement facile d'identifier le problème dans le cas d'une régression. Dans le domaine du logiciel, une régression correspond au fait de perdre le bon fonctionnement d'un logiciel après une mise à jour. Ici, ce terme désigne le fait de perdre le comportement attendu du code suite à une réécriture de celui-ci. Dans le cadre du TDD (en supposant que les tests couvrent bien le code), une régression doit entraîner un échec à au moins un test. Si les tests sont bien nommés, il est facile de retrouver la partie du code principale responsable de la régression surtout que si la méthodologie TDD a été respectée, les ajouts dans le code principal depuis la dernière fois où les tests passaient sont minimaux. C'est en cela que les tests déjà écrits constituent un filet de sécurité contre des accidents de parcours où l'on perdrait le lien entre changement et régression. Ce filet de sécurité permet d'envisager avec sérénité n'importe quelle modification du code, qu'il s'agisse d'une transformation (modification qui affecte le comportement du logiciel) ou d'un remaniement (modification qui n'altère pas le comportement, mais par exemple sa lisibilité).

Pour résumer, le TDD fait gagner en productivité de plusieurs façons.

- Le TDD permet d'éviter des modifications de code sans lien avec le but recherché, car on se focalise à chaque cycle sur la satisfaction d'un besoin précis, en conservant le cap du problème d'ensemble à résoudre. Cela permet d'éviter un écueil classique du développeur qui code des fonctionnalités dont il n'aura jamais besoin : voir principe YAGNI *You Ain't Gonna Need It*.
- Le TDD permet d'éviter les accidents de parcours, où des tests échouent sans qu'on puisse identifier le changement responsable, ce qui aurait pour effet d'allonger la durée d'un cycle de développement.
- Le TDD permet de s'approprier plus facilement n'importe quelle partie du code en vue de le faire évoluer, car chaque test ajouté dans la construction du logiciel explique et documente le comportement du logiciel.

Pour finir, la méthodologie TDD est très puissante, mais peut être compliquée à appliquer si vous n'avez jamais écrit de tests automatiques au préalable. Elle pose néanmoins de bonnes bases sur comment un développeur professionnel devraient produire du code. Le temps qui peut sembler être perdu à écrire une quantité pouvant

paraître comme énorme de tests est en fait très souvent du temps gagné par la suite lorsqu'un problème est découvert dans le code.