Module Génie Logiciel du Cours Turing EPFL

Ce module enseigne les bases du génie logiciel, afin de passer de *l'écriture de code* au *développement de logiciel*.

Pour obtenir une copie du matériel sur votre ordinateur, veuillez consulter les instructions de clonage

Ce module est une adaptation du cours "Software Engineering" de l'EPFL, qui était enseigné par le Prof. George Candea et Solal Pirelli.

Objectifs

À la fin du cours, les étudiants devraient être capables de :

- Reconnaître les besoins et problèmes courants dans le développement logiciel
- Expliquer pourquoi certaines techniques aident et pourquoi certaines techniques plus simples sont inadéquates
- Implémenter des logiciels corrects, efficaces, et fiables au niveau de fonctions, modules, et programmes
- Développer un logiciel en équipe, de l'organisation à la revue de code
- Critiquer les logiciels développés par d'autres d'une manière constructive et scientifique
- Produire des logiciels qui permettent aux utilisateurs de faire les tâches dont ils ont besoin de manière fiable et efficace

Prérequis

Ce module demande une familiarité basique avec la programmation, et spécifiquement le langage de programmation Python, pour mettre en pratique les concepts théoriques enseignés dans le cours.

Philosophie d'enseignement

Ce cours utilise des méthodes modernes et basées sur des preuves, particulièrement en ce qui concerne l'interactivité pendant les cours ainsi qu'une structure claire avec des objectifs et des critères d'évaluation.

Introduction

Bienvenue dans ce cours de génie logiciel ! Commençons par expliquer pourquoi vous devriez vous intéresser aux logiciels, pourquoi le *génie* logiciel est important, quel est l'objectif de ce cours, et une introduction rapide aux outils modernes.

Pourquoi s'intéresser aux logiciels ?

Le but des logiciels est d'aider les humains en automatisant des tâches. Il s'agit d'une question d'échelle : effectuer des tâches que les humains ne pourraient raisonnablement pas faire eux-mêmes parce que cela prendrait trop de temps ou d'efforts.

Prenons l'exemple du recensement aux États-Unis. Pour savoir combien de personnes vivent à quel endroit, le gouvernement américain procède à un recensement tous les dix ans. Le gouvernement engage des personnes qui se rendent dans chaque maison de chaque ville de chaque État et posent des questions sur les habitants et leurs caractéristiques démographiques.

Auparavant, le recensement était effectué manuellement. Cela a bien fonctionné lors de la fondation des États-Unis, mais à la fin du 19e siècle, l'échelle est devenue un problème. La totalisation des résultats de toutes les maisons pour obtenir les résultats des villes, des États et de l'ensemble du pays prenait beaucoup de temps. En raison de la croissance démographique, le recensement a commencé à prendre près de dix ans, ce qui signifie qu'au moment où une itération du recensement était effectuée, ses données étaient déjà considérées comme obsolètes, et il était temps d'en faire un nouveau. C'est là que l'automatisation est intervenue : une entreprise a mis au point une machine qui comptait les cartes dans un format spécifique, permettant ainsi au recensement de se dérouler à nouveau dans des délais raisonnables. Bien entendu, il existe de nombreux

autres problèmes qui ne peuvent être résolus manuellement, et la création d'un type de machine spécifique pour chacun d'entre eux n'est pas envisageable. C'est là que les logiciels entrent en jeu : nous pouvons désormais écrire un *code* qui effectue ces tâches à l'aide d'un seul type de machine. Soit dit en passant, la société qui a fabriqué la machine à compter les points survit encore aujourd'hui sous le nom d'IBM !

De nos jours, les logiciels sont omniprésents, et savoir écrire et maintenir des logiciels est donc une compétence très prisée. Les voitures contiennent des millions de lignes de code, même les "vieilles" qui fonctionnent à gaz plutôt qu'à l'électricité. Même les appareils électroménagers tels que les machines à laver ou les barbecues utilisent des logiciels suffisamment complexes pour nécessiter des mises à jour régulières afin de corriger les bugs !

Au début des années 2000, les entreprises ayant le plus de capitalisation de la planète étaient principalement actives dans les secteurs du gaz, du pétrole, de la santé, de la banque et d'autres services de ce type. Aujourd'hui, c'est l'inverse : seule une petite fraction des entreprises avec le plus de valeur s'occupe de services spécifiques tels que la santé, et la grande majorité d'entre elles s'occupent de logiciels, de matériel ou des deux.

Les logiciels sont également un outil essentiel pour la science. Les scientifiques ont développé des modèles de cerveau dans des logiciels, permettant des expériences qui ne seraient pas réalisables sur de vrais cerveaux. Un autre exemple est la technique de communication cerveau-texte qui utilise du matériel pour obtenir des signaux directement du cerveau humain et des logiciels pour les traiter, permettant aux personnes paralysées de communiquer en pensant à écrire avec leur main, qui envoie des signaux du cerveau, même si leur corps ne peut pas bouger !

Il est tentant de penser qu'il existe déjà suffisamment de logiciels, mais c'est loin d'être le cas. Par exemple, au début de la pandémie de coronavirus, la Suisse n'a pas pu suivre les cas assez rapidement parce que les cas devaient être remplis sur des formulaires papier et faxés au gouvernement fédéral. Ce manque d'automatisation a été un problème majeur pour réagir assez rapidement à une pandémie afin d'aider les citoyens.

Pourquoi le génie logiciel est-il important ?

Nous venons de voir à quel point les logiciels sont importants, mais pourquoi faut-il suivre un cours sur le *génie* logiciel ? Il s'agit avant tout de *confiance* : permettre aux gens de croire que le logiciel fera ce qu'ils veulent faire de la manière dont ils le veulent. Les utilisateurs devraient pouvoir compter sur les logiciels pour les tâches quotidiennes, même celles qui pourraient les blesser ou les tuer si elles étaient mal exécutées.

Prenons à nouveau l'exemple des voitures. Les voitures modernes sont dotées de capacités de conduite autonome, qui sont alimentées par un logiciel utilisant des données provenant de capteurs matériels. Si ce logiciel comporte des bugs qui provoquent des erreurs, des accidents peuvent se produire. Ces accidents diminuent la confiance des utilisateurs dans le logiciel, ce qui peut rendre l'ensemble du logiciel de conduite autonome inutile en amenant les gens à refuser de l'utiliser.

Les vaisseaux spatiaux sont un autre type de véhicule auquel les utilisateurs doivent faire confiance. Lorsque la NASA a délégué une partie de la construction du logiciel à Boeing et a vérifié le résultat, leur enquête a révélé que des "problèmes systémiques" étaient à l'origine d'une erreur logicielle. En d'autres termes, cette erreur n'était pas le fait d'une seule personne, mais le résultat de mauvaises pratiques dans l'ensemble de l'entreprise.

Avant que la NASA ne dispose de machines, elle employait des "calculateurs" : des personnes dont le travail consistait à effectuer des calculs. L'un de ces calculateurs était Katherine Johnson, qui effectuait des calculs pour l'alunissage, entre autres opérations. Lorsque l'astronaute John Glenn devait se mettre en orbite autour de la Terre à bord de la capsule Friendship 7, il a d'abord refusé de voler parce que la NASA avait utilisé des machines pour effectuer les calculs, ce en quoi il n'avait pas confiance. Il a donc demandé à la NASA que Johnson vérifie les calculs de la machine, en déclarant fameusement : "Si elle dit qu'ils sont bons, alors je suis prêt à partir". C'est ce qu'elle a fait, et la mission a été un succès.

Pourrions-nous avoir une Katherine Johnson pour vérifier tous les logiciels que nous écrivons ? Malheureusement, la *complexité* des logiciels modernes rend cette tâche infaisable. Considérons le morceau de code suivant,

qui pourrait se trouver dans la fonction d'accélération d'une voiture auto-conduite :

```
# Accélérer, sauf si la vitesse est déjà 100
if speed >= 100:
  speed = 100
else:
  speed = speed + 1
```

Ce code divise le programme en deux chemins : l'un dans lequel la vitesse était au moins égale à 100 et a été plafonnée, et l'autre dans lequel la vitesse était inférieure à 100 et a été augmentée. Une instruction ”if” a permis de doubler le nombre de chemins. S'il y avait une autre déclaration de ce type après celle-ci, il y aurait alors quatre chemins. Une autre, et il y en aurait huit, et ainsi de suite. La mission lunaire Apollo 11 contenait environ 150 000 lignes de code. Si seulement 1 % de ces lignes sont des instructions ”if”, cela conduit à 2^1500 chemins. Et Apollo 11 est minuscule par rapport aux normes modernes ; le système d'exploitation Windows compte des dizaines de millions de lignes de code. Il n'y a pas assez d'atomes dans l'univers pour répertorier tous les chemins possibles du programme !

L'ingénierie d'un logiciel fiable va toutefois au-delà du problème impossible de la vérification de chaque chemin dans le programme. Prenons l'exemple d'un programme qui, à partir d'une liste d'étudiants et de leurs notes, envoie à chacun d'entre eux un courrier électronique leur annonçant leur note. Il s'agit là d'un bon exemple de logiciel automatisant une tâche qui serait longue et sujette à des erreurs si elle était effectuée manuellement. Le logiciel transforme une ligne de saisie telle que ”Alice, 9/10” en un courriel envoyé à Alice pour l'informer de sa note. Cela fonctionne-t-il toujours si, au lieu d'avoir un nom utilisant des lettres anglaises comme Alice, l'élève a un nom en caractères cyrilliques, arabes, ou chinois par exemple ? Que se passe-t-il si l'élève a un nom à l'américaine comme ”Bob, Jr”, qui comprend une virgule qui est également utilisée par le logiciel pour séparer les noms et les notes dans l'entrée ? Que se passe-t-il si, après l'envoi des premiers courriels, l'ordinateur sur lequel le logiciel est exécuté perd sa connexion Internet ? Les courriels sont-ils perdus ? L'exécution répétée du logiciel entraîne-t-elle l'envoi de courriels en double ? Le logiciel peut-il gérer la notification aux utilisateurs dans une autre langue que l'anglais ? Par exemple, en français, les salutations sont souvent spécifiques au genre, l'anglais ”Dear Alice”/”Dear Bob” devenant le français ”*Chère* Alice”/”*Cher* Bob”. Le logiciel peut-il gérer cela ? Même s'il gère tous les problèmes susmentionnés, comment une autre personne peut-elle s'en assurer ? Le responsable du logiciel peut affirmer que cela fonctionne, mais ce n'est pas suffisant pour risquer d'envoyer des notes erronées à vos étudiants. Et si quelqu'un d'autre lui fait confiance et souhaite ajouter une fonctionnalité, comment cette fonctionnalité peut-elle être réintégrée dans la version ”principale” du logiciel ? L'envoi d'une version par courrier électronique fonctionne si les développeurs sont peu nombreux, mais cette méthode n'est pas adaptée à des dizaines de personnes apportant des modifications susceptibles d'entrer en conflit. Rappelons que Windows contient des dizaines de millions de lignes de code ; une personne seule, ou même une équipe de plusieurs dizaines de personnes, n'est pas suffisante pour développer de grands projets logiciels.

Le but de ce cours

Ce cours a pour but de transformer les *étudiants* en *ingénieurs*, en les initiant aux concepts du monde réel et à leurs applications, afin qu'ils passent de l'*écriture de code* au *développement de logiciels*.

Un aspect clé de cette démarche est de s'éloigner des exercices théoriques avec des solutions bien définies et de s'orienter vers des exercices du monde réel avec des solutions discutables. Par exemple, un étudiant peut rendre un travail de codage et recevoir une note reflétant ”la qualité” de sa solution. Mais dans le monde réel, un ingénieur soumet un *projet* et reçoit des *réactions* de la part des utilisateurs. Ce retour d'information peut inclure des désaccords sur le problème même que le logiciel est censé résoudre, y compris des changements d'opinion de la part du client qui nécessitent des modifications du logiciel même si la solution de l'ingénieur était ”bonne” d'un point de vue théorique.

Une analogie utile consiste à imaginer la construction d'un avion. Si un étudiant rend un avion ”complet à 95 %”, il peut s'attendre à obtenir une note de 95 %. Mais si un ingénieur présente un avion ”complet à 95 %”, le résultat dépend fortement des 5 % manquants. Si les sièges ne sont pas aussi confortables qu'ils pourraient l'être, ou si la vitesse maximale est un peu inférieure à ce qu'elle devrait être, le client peut toujours être

satisfait. Mais si l'aile n'est qu'à moitié terminée, l'avion ne peut pas voler, même s'il est "complet à 95 %", et le client le rejettera !

Dans les cours précédents, vous avez appris à écrire du code. Dans ce cours, vous apprendrez les autres étapes clés du développement d'un logiciel :

- **Exigences** : comment déterminer les besoins des utilisateurs et comment les traduire en logiciels.
- **Conception** : comment concevoir un logiciel correctement, afin de rendre le logiciel plus facile à écrire et à maintenir.
- **Gestion de versions** : comment assurer le suivi du logiciel et de ses modifications.
- **Test** : comment tester un logiciel de manière automatisée, pour donner confiance aux autres que le logiciel fait ce qu'il doit faire, et comment utiliser les tests pour aider l'ensemble du processus de développement.
- **Débuggage** : comment utiliser les outils et les techniques modernes pour trouver et corriger les bugs d'un logiciel.
- **Performance** : comment concevoir et écrire des logiciels efficaces, et comment trouver et résoudre les problèmes de performance.
- **Travail d'équipe** : comment effectuer toutes les tâches susmentionnées au sein d'une équipe, à l'échelle des bases de code des logiciels modernes.

Outils modernes

Heureusement, il n'est pas nécessaire de repartir de zéro chaque fois que vous ou votre équipe souhaitez concevoir un logiciel. En fait, si vous deviez écrire des logiciels en partant de rien et sans aide extérieure à chaque fois, vous ne parviendriez jamais à terminer un projet de taille raisonnable.

Les ingénieurs logiciels utilisent des dépôts de paquets tels que Maven Central, la NuGet Gallery, ou le NPM Registry pour réutiliser le code existant. Si vous souhaitez que votre logiciel réessaie automatiquement des opérations qui ont échoué, quelqu'un d'autre l'a probablement déjà fait ; vous n'avez pas besoin de passer des semaines à écrire une nouvelle bibliothèque, à concevoir des tests, à réfléchir aux cas limites, à la rendre suffisamment générique pour la réutiliser dans plusieurs projets, et ainsi de suite.

Lorsque les ingénieurs logiciels rencontrent des problèmes, ils ne les résolvent pas seuls. Bien que tout problème puisse être résolu avec suffisamment de temps, l'utilisation efficace de son temps est un objectif clé de l'ingénierie. Au lieu de cela, les ingénieurs logiciels utilisent des sites web tels que StackOverflow pour poser des questions et y répondre. Répondre à des questions est bénéfique pour celui qui y répond, car l'enseignement est un excellent moyen de revérifier sa compréhension d'un concept.

Exercice Considérez le code PHP suivant, même si vous n'avez jamais utilisé PHP auparavant :

```
class Context {
    protected $config;
    public function getConfig($key) {
        $cnf = $this->config;
        return $cnf::getConfig($key);
    }
}
```

Si vous essayez d'utiliser ce code, l'interpréteur PHP affichera l'erreur suivante : `syntax error, unexpected T_PAAMAYIM_NEKUDOTAYIM` (erreur de syntaxe, T_PAAMAYIM_NEKUDOTAYIM inattendu). Qu'est-ce qui ne va pas ? (Indice : ce n'est pas un problème que vous pouvez résoudre en regardant le code, utilisez des outils !)

En cherchant ce nom étrange dans votre moteur de recherche préféré, vous trouverez des questions existantes qui vous apprendront qu'il s'agit en fait de l'hébreu, la langue des auteurs de PHP, pour "double deux-points". L'erreur vient du fait qu'au lieu d'un double deux-points, le code devrait utiliser une flèche `->`.

Exercice Si un collègue vous dit qu'il a reçu un rapport de bug d'un utilisateur de la version "Win32" de votre application, nommée d'après l'ancienne interface de programmation de Windows, avec le code d'erreur 39, pourriez-vous dire ce qui a causé l'erreur ?

En cherchant "win32 error code 39" ou quelque chose de similaire dans votre moteur de recherche préféré, vous trouverez la documentation de Microsoft pour l'interface de programmation Win32, Vous y apprendrez que le code d'erreur 0x27, qui correspond au chiffre 39 en hexadécimal, signifie que le disque est plein. Vous pourrez alors discuter de la meilleure façon de gérer cette erreur, par exemple en affichant un message utile à l'utilisateur, ou peut-être en supprimant certains des fichiers temporaires de votre application et en réessayant.

Un mot sur la méthode de ce cours

Ce cours n'est pas un cours magistral "traditionnel" dans lequel vous écoutez un cours ou lisez des notes de cours de manière passive. La recherche sur l'enseignement et l'apprentissage a montré que l'interactivité améliore l'apprentissage, et il y aura donc des exercices fréquents dans les cours, tels que ceux mentionnés ci-dessus. Il y a également des exercices traditionnels que vous pouvez faire après les cours pour tester votre propre compréhension de la matière.

Infrastructure

Prérequis 1 : Installation de Git : Pour suivre cette partie, vous devez :

- Installer Git. Sous Windows, utilisez WSL car Git est principalement conçu pour Linux. Sur macOS, voir la documentation de git. Sous Linux, Git est peut-être déjà installé, ou utilisez la gestionnaire de paquets de votre distribution. Si vous avez installé Git avec succès, l'exécution de `git --version` dans la ligne de commande devrait afficher un numéro de version.
- Dites à Git qui vous êtes en lançant `git config --global user.name 'votre_nom'` avec votre nom et `git config --global user.email 'votre_email'`.
- Choisissez un éditeur que Git ouvrira pour écrire un résumé de vos changements avec `git config --global core.editor 'votre_editeur'`, puisque Git utilise par défaut `vi` qui est difficile à utiliser pour les nouveaux venus. Sous Windows avec WSL, vous pouvez utiliser `notepad.exe`, qui ouvrira le Notepad de Windows. Sous macOS, vous pouvez utiliser `open -e -W -n` qui ouvrira une nouvelle fenêtre TextEdit. Sous Linux, vous pouvez utiliser l'éditeur de texte graphique intégré à votre distribution, ou `nano`.

  Si vous utilisez Windows avec WSL, notez que l'exécution de `explorer.exe .` à partir de la ligne de commande Linux ouvrira l'explorateur de Windows dans le dossier de votre ligne de commande, ce qui est pratique.

  Si vous le souhaitez, vous pouvez définir le paramètre de configuration de Git `core.autocrlf` à `true` sur Windows et `input` sur Linux et macOS, pour que Git convertisse automatiquement les fins de lignes à la manière d'Unix (`\n`) et les séparateurs de lignes à la manière de Windows (`\r\n`).

Prérequis 2 : Création d'une clé SSH:

  En ligne de commande, exécutez `ssh-keygen` puis *appuyez sur Entrée 3x pour confirmer les questions, sans écrire d'autre texte*. Ensuite, exécutez `ls ~/.ssh/*.pub`, ce qui devrait afficher une longue chaîne de caractères qui est votre clé publique SSH. **Copiez** cette chaîne de caractères dans le presse-papiers.

Prérequis 3 : À l'EPFL, nous utiliserons l'instance GitLab de l'EPFL, vous devez :

- Vous connecter à https://gitlab.epfl.ch avec votre compte EPFL
- Dans les préférences de votre profil, ajoutez une clé SSH. Votre clé publique est celle que vous avez collé dans le presse papiers à l'étape précédente.

Prérequis 3 : Hors de l'EPFL, vous pouvez utiliser GitHub, vous devez:

- Créer un compte GitHub (vous n'êtes pas obligé d'utiliser un compte GitHub existant, vous pouvez en créer un uniquement pour ce cours si vous le souhaitez)
- Dans les préférences de votre profil, ajoutez une clé SSH. Votre clé publique est celle que vous avez collé dans le presse papiers à l'étape précédente.

Où stockez-vous votre code et comment le modifiez-vous ? Si vous écrivez votre propre logiciel, ce n'est pas un problème, car vous pouvez utiliser votre propre machine et modifier les fichiers que vous voulez quand vous le voulez. Mais si vous travaillez avec quelqu'un d'autre, cela devient problématique. Vous pouvez utiliser un service de cloud en ligne où vous stockez des fichiers et coordonnez qui modifie quel fichier et à quel moment. Vous pouvez vous envoyer par courrier électronique des modifications apportées à des ensembles de fichiers. Mais cela ne fonctionne pas aussi bien lorsque vous avez plus de personnes, et c'est complètement inutilisable lorsque vous avez des dizaines ou des centaines de personnes qui travaillent sur la même base de code. C'est là que l'infrastructure entre en jeu.

Objectifs

- Contraster les anciens et les nouveaux systèmes de *gestion de version*
- Organiser votre code avec le système de gestion de version *Git*.
- Rédiger des descriptions utiles des modifications du code
- Éviter les erreurs avec l'*intégration continue*

Qu'est-ce que la gestion de version ?

Avant de parler de la gestion du code à l'aide d'un système de gestion de version, il convient de définir certains termes.

Un *dépôt* (”repository” en anglais) est un emplacement dans lequel vous stockez une base de code, par exemple un dossier sur un serveur distant. Lorsque vous apportez un ensemble de modifications à un dépôt, vous *poussez* (”push”) des modifications. Lorsque vous récupérez les modifications apportées par d'autres personnes dans le dépôt, vous *tirez* (”pull”) des modifications.

Un ensemble de modifications est appelé *commit*. Un cimmit a quatre composantes principales : qui, quoi, quand et pourquoi. ”Qui” est l'auteur du commit, la personne qui a effectué les modifications. Le ”quoi” est le contenu du commit, les modifications elles-mêmes. ”Quand” est la date et l'heure à laquelle le commit a été effectué. Cette date peut être antérieure à la date à laquelle le commit a été poussé dans le dépôt. Le ”Pourquoi” est un message associé au commit qui explique pourquoi les modifications ont été apportées, par exemple en expliquant pourquoi il y avait un bug et pourquoi le nouveau code corrige le bug. Le ”pourquoi” est particulièrement important, car vous devrez souvent revenir sur d'anciennes modifications et comprendre pourquoi elles ont été effectuées.

Il arrive parfois qu'une modification entraîne des problèmes. Il se peut qu'une modification censée améliorer les performances introduise un bug. Les systèmes de gestion de version vous permettent d'inverser (”revert”) ce commit, ce qui crée un nouveau commit dont le contenu est l'inverse de celui d'origine. En d'autres termes, si le commit original a remplacé ”X” par ”Y”, un commit inversé remplace ”Y” par ”X”. Il est important de noter que le commit original n'est pas perdu ou détruit, mais qu'un nouveau commit est créé.

Les modifications sont rassemblées dans un *historique* des changements. Au départ, un dépôt est vide. Ensuite, quelqu'un ajoute du contenu dans un commit, puis plus de contenu dans un autre commit, et ainsi de suite. L'historique d'un dépôt contient donc toutes les modifications nécessaires pour passer de rien à l'état actuel. Certaines de ces modifications peuvent faire l'objet d'allers-retours, comme les commits inversés, ou les commits qui remplacent le code qu'un commit précédent a ajouté. À tout moment, n'importe quel développeur ayant accès au dépôt peut consulter l'historique complet pour savoir qui a effectué quelles modifications, quand et pourquoi.

Les systèmes de gestion de version de la première génération étaient essentiellement une couche d'automatisation sur la gestion manuelle des versions. Comme nous l'avons mentionné précédemment, si vous développez

avec quelqu'un d'autre, vous pouvez mettre vos fichiers quelque part et coordonner qui modifie quoi et quand. Un système de première génération vous aide à faire cela avec moins d'erreurs, mais utilise toujours fondamentalement le même modèle.

Avec la gestion de version de première génération, si Alice veut travailler sur le fichier A, elle réserve (”check out”) le fichier. À ce moment-là, le fichier est verrouillé : Alice peut le modifier, mais personne d'autre ne peut le faire. Si Bob veut également réserver le fichier A, le système rejettera sa tentative. Bob peut toutefois réserver le fichier B si personne d'autre ne l'utilise. Une fois qu'Alice a terminé son travail, elle crée un commit avec ses modifications et libère la réservation. À ce moment-là, Bob peut réserver le fichier A et y apporter ses modifications.

Les systèmes de gestion de version de première génération agissent donc comme des verrous à la granularité des fichiers. Ils empêchent les développeurs d'apporter des modifications parallèles au même fichier, ce qui permet d'éviter certaines erreurs mais n'est pas très pratique. Par exemple, Alice peut vouloir modifier la fonction X dans le fichier A, tandis que Bob veut modifier la fonction Y dans le fichier A. Ces modifications n'entreront pas en conflit, mais elles ne pourront toujours pas être effectuées en parallèle, car les verrous de la première génération de gestion de version sont sur des fichiers entiers.

Les développeurs ont abandonné les systèmes de première génération parce qu'ils voulaient mieux contrôler les *conflits*. Lorsque deux développeurs veulent travailler sur le même fichier en même temps, ils devraient pouvoir le faire, à condition qu'ils puissent ensuite *fusionner* (”merge”) leurs modifications dans une version unifiée. La fusion des modifications n'est pas toujours possible automatiquement. Si deux développeurs ont modifié la même fonction de manière différente, par exemple, ils devront probablement discuter pour décider quelles modifications doivent être conservées.

Les *branches* sont une autre fonctionnalité qui a du sens si un système peut gérer les conflits et les fusions. Parfois, les développeurs souhaitent travailler en parallèle sur plusieurs copies de la base de code. Par exemple, vous êtes peut-être en train de travailler sur des modifications visant à améliorer les performances, lorsqu'un client vous fait part d'un rapport de bug. Vous pourriez corriger le bug et créer un commit avec la correction et vos modifications de performance, mais le commit résultant n'est pas pratique. Si, par la suite, vous devez revenir sur les modifications de performance, par exemple, vous devrez également revenir sur la correction de bugs car elle se trouve dans le même commit. Au lieu de cela, vous créez une branche pour vos changements de performance, puis vous passez à une branche pour la correction des bugs, et vous pouvez travailler sur les deux en parallèle. Lorsque votre correction de bug est prête, vous pouvez la *fusionner* dans la branche ”principale” du dépôt, et il en va de même pour les changements de performance. Une utilisation courante des branches concerne les versions : vous pouvez par exemple publier la version 1.0 de votre logiciel et créer une branche représentant l'état du dépôt pour cette version. Vous pouvez alors travailler sur la future version 2.0 dans la branche ”principale”. Si un client signale un bug dans la version 1.0, vous pouvez passer à la branche de la version 1.0, corriger le bug et publier la correction, puis reprendre votre travail sur la version 2.0. Vos modifications pour la version 1.0 n'ont pas affecté votre branche principale, car vous les avez effectuées dans une autre branche.

Dans les logiciels modernes, le processus habituel de création de branches consiste à créer une branche à partir de la branche principale du dépôt, puis ajouter des commits à la branche pour corriger un bug, ajouter une fonctionnalité ou effectuer toute autre tâche pour laquelle la branche a été créée, et demander ensuite à un collègue de le réviser. Si le collègue demande des modifications, comme l'ajout de commentaires de code, vous pouvez ajouter un commit à la branche avec ces modifications. Une fois que votre collègue est satisfait, vous pouvez fusionner les commits de la branche dans la branche principale. Vous pouvez ensuite créer une autre branche pour travailler sur quelque chose d'autre, et ainsi de suite. Vos collègues travaillent eux aussi sur leurs propres branches. Ce flux de travail permet à chacun de pousser les commits qu'il souhaite sur sa branche sans entrer en conflit avec les autres, même si son travail n'est pas encore tout à fait terminé. Souvent, il est judicieux d'écraser les commits d'une branche en un seul commit et de fusionner le commit résultant dans la branche principale. Cela permet de combiner toutes les modifications de la branche en un seul commit propre dans l'historique de la branche principale, plutôt que d'avoir un tas de commits qui font quelques petites modifications chacun mais qui n'ont aucun sens l'un sans l'autre.

Dans le cas de branches représentant des versions, il est parfois nécessaire d'appliquer les mêmes modifications

à plusieurs branches. Par exemple, en développant la version 2.0 dans la branche principale, vous pouvez trouver un bug et vous rendre compte que ce bug existe également dans la version 1.0. Vous pouvez faire un commit corrigeant le bug dans la version 2.0, et ensuite "*cherry pick*" le commit dans la branche pour la version 1.0. Tant que la modification n'entre pas en conflit avec d'autres modifications apportées à la branche de la version 1.0, le système de contrôle de la version peut copier votre commit de correction de bug dans une commit pour une autre branche.

Les systèmes de gestion de version de deuxième génération avaient pour but de permettre aux développeurs de gérer les conflits. Alice peut travailler sur le fichier A sans avoir besoin de le verrouiller, et Bob peut également travailler sur le fichier A en même temps. Si Alice apporte ses modifications en premier, le système les acceptera, et lorsque Bob voudra ensuite appliquer ses modifications, deux choses pourront se produire. Il est possible que les modifications soient fusionnées automatiquement, par exemple parce qu'elles concernent deux parties différentes du fichier. L'autre possibilité est que les modifications soient contradictoires et doivent être fusionnées manuellement. Bob doit alors choisir ce qu'il faut faire, éventuellement en demandant à Alice, et produire une version "fusionnée" du fichier qui peut être transmise.

Le principal inconvénient du gestion de version de deuxième génération est sa centralisation. Les développeurs travaillent avec un dépôt unique, hébergé sur un serveur. La validation des modifications nécessite une connexion Internet à ce serveur. C'est un problème si le serveur est en panne, si le développeur se trouve dans un endroit où il n'a pas accès à Internet, ou pour toute autre raison qui empêche le développeur d'atteindre le serveur.

Les systèmes de gestion de version de troisième génération sont axés sur la décentralisation. Chaque machine possède son propre dépôt. Il ne s'agit pas d'une "sauvegarde" ou d'une "réplique" d'un dépôt "principal", mais simplement d'un autre clone du dépôt. Les développeurs peuvent effectuer des modifications localement sur leur propre dépôt, puis transférer ces modifications vers d'autres clones du dépôt, par exemple sur un serveur. Les développeurs peuvent également avoir plusieurs branches localement, avec des commits différents dans chacune d'entre elles, et pousser tout ou partie de ces branches vers d'autres clones du dépôt. Tout cela fonctionne tant que les dépôts ont des historiques compatibles. En d'autres termes, il n'est pas possible d'apporter une modification à un dépôt qui n'est pas basé sur le même historique que le dépôt local.

Dans la pratique, les équipes se mettent d'accord sur un dépôt "principal" vers lequel elles vont toutes envoyer des commits, et travaillent localement sur leur clone de ce dépôt. Bien que, du point de vue du système de gestion de version, tous les clones du dépôt soient égaux, il est pratique pour les développeurs de se mettre d'accord sur un seul endroit où tout le monde place ses modifications.

Le principal système de gestion de version utilisé aujourd'hui est *Git*. Git a été inventé par Linus Torvalds, qui a inventé Linux, parce qu'il était fatigué des problèmes posés par le précédent système de gestion de version qu'il utilisait pour Linux. Il existe également d'autres systèmes de gestion de version de troisième génération, tels que Mercurial et Bazaar, mais Git est de loin le plus utilisé.

De nombreux développeurs utilisent des sites web publics pour héberger le clone du dépôt "principal" de leurs projets. Le plus connu aujourd'hui est GitHub, qui utilise Git mais n'y est pas techniquement lié. GitHub ne se contente pas de stocker un clone de dépôt, mais peut également héberger une liste de "problèmes" ("issues") pour le dépôt, tels que les bugs et les demandes de fonctionnalités, ainsi que d'autres données telles qu'un wiki pour la documentation. Il existe également d'autres sites web présentant des caractéristiques similaires, tels que GitLab et BitBucket, bien qu'ils ne soient pas aussi populaires.

Un exemple de projet développé sur GitHub est le runtime .NET, qui est développé principalement par des employés de Microsoft et entièrement sur GitHub. Les conversations sur les bugs, les demandes de fonctionnalités et les révisions de code se déroulent au grand jour, sur GitHub.

Comment utiliser Git ?

Maintenant que nous avons vu la théorie, passons à la pratique ! Vous allez créer un dépôt, y apporter quelques modifications et le publier en ligne. Nous verrons ensuite comment contribuer à un dépôt en ligne existant.

Git possède quelques commandes de base quotidiennes que nous allons voir maintenant, et de nombreuses commandes avancées que nous n'aborderons pas ici. Vous pouvez toujours rechercher des commandes sur Internet, qu'elles soient basiques ou avancées. Vous finirez par vous souvenir des principes de base après les avoir suffisamment utilisés, mais il n'y a aucune honte à chercher ce qu'il faut faire.

Nous utiliserons Git en ligne de commande pour ce tutoriel, car il fonctionne de la même manière partout. Cependant, pour les tâches quotidiennes, vous préférerez peut-être utiliser des interfaces graphiques telles que GitKraken, GitHub Desktop, ou le support Git de votre IDE favori.

Commencez par créer un dossier et *initialiser* un dépôt dans ce dossier :

```
~$ mkdir exemple
~$ cd exemple
~/exemple$ git init
```

Git vous dira que vous avez initialisé un dépôt Git vide dans `~/example/.git/`. Ce dossier `.git/` est un dossier spécial que Git utilise pour stocker les métadonnées. Il ne fait pas partie du dépôt lui-même, même s'il se trouve dans le dossier du dépôt.

Créez un fichier :

```
$ echo 'Hello' > hello.txt"
```

Nous pouvons maintenant demander à Git ce qu'il pense qu'il se passe :

```
$ git status
...
Fichiers non suivis :
        hello.txt
```

Git nous dit qu'il voit que nous avons ajouté `hello.txt`, mais ce fichier n'est pas encore suivi. C'est-à-dire que Git ne l'inclura pas dans un commit à moins que nous ne le demandions explicitement. C'est exactement ce que nous allons faire :

```
$ git add -A
```

Cette commande demande à Git d'inclure toutes les modifications actuelles dans le dépôt lors du prochain commit. Si nous apportons d'autres modifications, nous devrons demander que ces nouvelles modifications soient également suivies. Mais pour l'instant, demandons à Git ce qu'il en pense :

```
$ git status
...
Changements à commit :
        nouveau fichier : hello.txt
```

Maintenant, Git sait que nous voulons inclure ce fichier dans un commit. Faisons donc exactement cela :

```
$ git commit
```

Cela ouvrira un éditeur de texte dans lequel vous pourrez saisir le message de validation. Comme nous l'avons vu précédemment, le message de validation doit être une description de la raison pour laquelle les changements ont été effectués. Souvent, le tout premier commit d'un dépôt met en place la structure de base du fichier en tant que commit initial, vous pourriez donc écrire `Commit initial mettant en place le fichier` ou quelque chose de similaire. Vous obtiendrez alors un résultat comme celui-ci :

```
[...] Commit initial.
 1 fichier modifié, 1 insertion(+)
 create mode 100644 hello.txt
```

Git répète le message de commit que vous avez mis, ici `Commit initial.`, et vous dit ensuite quels changements ont eu lieu. Ne vous inquiétez pas de ce `mode 100644`, c'est plus un détail d'implémentation.

Modifions maintenant les choses en ajoutant une ligne :

```
$ echo 'Goodbye' >> hello.txt
```

Nous pouvons demander à git les détails des changements que nous avons effectués :

```
$ git diff
```

Cela affichera une liste détaillée des différences entre l'état du dépôt à partir du dernier commit et l'état actuel du dépôt, c'est-à-dire que nous avons ajouté une ligne disant `Goodbye`.

Ajoutons les modifications que nous venons d'apporter :

```
$ git add -A
```

Que se passe-t-il si nous demandons à nouveau une liste de différences ?

```
$ git diff
```

...Rien ! Pourquoi ? Parce que `diff` montre par défaut les différences qui ne sont pas suivies pour le prochain commit. Il existe trois états pour les modifications de fichiers dans Git : modifié, suivi (”staged”) et committed. Par défaut, les changements sont modifiés, puis avec `git add -A` ils sont suivis, et avec `git commit` ils sont validés. Nous avons utilisé `-A` avec `git add` pour signifier ”tous les changements”, mais nous pourrions en fait n'ajouter que des changements spécifiques, comme des fichiers spécifiques ou même des parties de fichiers.

Pour voir les changements suivis, nous devons les demander :

```
$ git diff --staged
```

Nous pouvons maintenant valider nos modifications. Comme il s'agit d'un petit commit qui ne nécessite pas beaucoup d'explications, nous pouvons utiliser `-m` pour écrire le message de commit directement dans la commande :

```
$ git commit -m 'Say goodbye'
```

Vous pouvez afficher l'historique avec `log` :

```
$ git log
```

Le résultat est assez long, il peut être raccourci en une ligne par commit avec l'argument `--oneline` :

```
$ git log --oneline
```

Essayons maintenant les branches, en créant une branche et en y basculant :

```
git switch -c feature/today
```

La barre oblique dans le nom de la branche n'a rien de spécial pour Git, il s'agit seulement d'une convention de nommage courante pour distinguer le but des différentes branches. Par exemple, vous pouvez avoir des branches nommées `feature/delete-favorites` ou `bugfix/long-user-names`. Mais vous pouvez aussi nommer votre branche `delete-favorites` ou `bugfix/long/user/names` si vous le souhaitez, tant que tous ceux qui utilisent le dépôt se mettent d'accord sur une convention pour les noms.

Modifiez maintenant la seule ligne du fichier, en remplaçant par exemple ”Hello” par ”Hello today”. Ensuite, ajoutez vos modifications et validez-les :

```
$ git add -A && git commit -m 'Change greeting'
```

Vous remarquerez que Git vous dit qu'il y a `1 insertion (+), 1 suppression (-)`. C'est un peu bizarre, nous avons changé une ligne, pourquoi y a-t-il deux changements ? La raison en est que Git considère les modifications à la granularité des lignes. Lorsque vous éditez une ligne, Git voit cela comme ”vous avez supprimé la ligne qui était là, et vous avez ajouté une nouvelle ligne”. Le fait que les lignes ”supprimées” et ”ajoutées” soient similaires n'est pas pertinent.

Si vous avez déjà utilisé Git, vous avez peut-être entendu parler du `-a` de `git commit`, qui pourrait remplacer le `git add -A` explicite dans notre cas. La raison pour laquelle nous ne l'utilisons pas ici, et la raison pour

laquelle vous devriez être prudent si vous l'utilisez, est que `-a` ne fait qu'ajouter des changements à des fichiers existants. Il n'ajoute pas de modifications aux nouveaux fichiers ou aux fichiers supprimés. Il est donc très facile d'oublier accidentellement d'inclure certains fichiers nouveaux ou supprimés dans le commit, et de devoir alors effectuer un autre commit avec ces seuls fichiers, ce qui est ennuyeux.

Quoi qu'il en soit, nous avons fait un commit sur notre branche `feature/today`. Si nous voulons nous assurer que nous sommes bien sur cette branche, nous pouvons le demander à Git :

```
$ git branch
```

Cela produira une liste de branches, avec un astérisque `*` à côté de celle sur laquelle nous nous trouvons.

Passons maintenant à notre branche principale. Selon votre version de Git, cette branche peut avoir des noms différents, donc regardez la sortie de la commande précédente et utilisez le bon, comme `master` ou `main` :

```
$ git switch main
```

Pour voir ce qui se passe lorsque deux commits entrent en conflit, apportons une modification à notre fichier `hello.txt` qui entre en conflit avec l'autre branche que nous venons de créer. Par exemple, remplacez "Hello" par "Hello everyone". Ensuite, suivez la modification et validez-la comme précédemment.

A ce stade, nous avons deux branches, notre branche principale et `feature/today`, qui ont divergé : elles ont toutes deux un commit qui n'est pas dans l'autre. Demandons à Git de fusionner les branches, c'est-à-dire d'ajouter les commits de la branche spécifiée à la branche courante :

```
$ git merge feature/today
```

Git commencera de manière optimiste avec `Fusion automatique de hello.txt`, mais cela échouera rapidement avec un `Conflit de fusion dans hello.txt`. Git nous demandera de corriger les conflits et de "commit" le résultat manuellement.

A quoi ressemble `hello.txt` maintenant ?

```
$ cat hello.txt
<<<<<<< HEAD
Bonjour à tous
=======
Bonjour aujourd'hui
>>>>>>> feature/today
Au revoir
```

Prenons le temps de comprendre. La dernière ligne n'a pas changé, car elle ne fait pas partie du conflit. La première ligne a été étendue pour inclure les deux versions : entre les `<<<` et `===` se trouve la version dans `HEAD`, c'est-à-dire la "tête", le dernier commit, dans la branche courante. En effet, sur notre branche principale, la première ligne était "Bonjour à tous". Entre le `===` et le `>>>` se trouve la version dans `feature/today`. Ce que nous devons faire, c'est fusionner manuellement les changements, c'est-à-dire éditer le fichier pour remplacer le conflit comprenant les lignes `<<`, `===`, et `>>>` par les changements fusionnés que nous voulons. Par exemple, nous pourrions nous retrouver avec un fichier contenant ce qui suit :

```
$ cat hello.txt
Bonjour à tous
Au revoir
```

C'est une façon de fusionner le fichier. Nous aurions également pu choisir une seule des deux lignes. Ou peut-être voulons-nous encore un autre changement, nous pourrions avoir `Hello hello` à la place. Git ne s'en préoccupe pas, il veut seulement que nous décidions quelle sera la version fusionnée.

Une fois que nous avons effectué nos modifications de fusion, nous devons ajouter les modifications et effectuer un commit comme précédemment :

```
$ git add -A && git commit -m 'Merge'
```

Très bien. Attendez, non, en fait, pas si bien que ça. C'est un message de commit assez mauvais. Il est beaucoup trop court et pas assez descriptif. Heureusement, *parce que nous n'avons pas encore publié nos changements sur un autre clone du dépôt*, nous pouvons apporter des modifications à nos commits ! C'est comme la chute d'un arbre qui ne fait aucun bruit s'il n'y a personne pour l'entendre. Si personne ne l'entend, c'est qu'il ne s'est pas produit. Nous pouvons modifier notre commit maintenant, et lorsque nous le pousserons vers un autre clone, ce dernier ne verra que notre commit modifié. Cependant, si nous avions déjà poussé notre commit vers un clone, notre commit serait visible, nous ne pourrions donc plus le modifier car le clone serait confus par un commit changeant puisque les commits sont supposés être immuables.

Pour modifier notre commit, ce qui ne devrait être fait que si le commit n'a pas encore été poussé, nous le ”modifions” :

```
$ git commit --amend -m 'Fusionner la branche feature/today'
```

Nous n'avons modifié ici que le message de commit, mais nous pourrions également modifier le contenu du commit, c'est-à-dire les modifications elles-mêmes.

Parfois, nous apportons des modifications que nous ne voulons pas vraiment, par exemple des modifications temporaires pendant que nous débugguons un code. Effectuons un ”mauvais” changement :

```
$ echo 'asdf' >> hello.txt
```

Nous pouvons restaurer le fichier dans l'état où il se trouvait lors du dernier commit afin d'annuler cette modification :

```
$ git restore hello.txt
```

C'est fait ! Nos modifications temporaires ont disparu. Vous pouvez également utiliser `.` pour restaurer tous les fichiers du répertoire actuel, ou de tout autre chemin. Cependant, gardez à l'esprit que ”disparu” signifie réellement ”disparu”. C'est comme si nous n'avions jamais modifié le fichier, puisqu'il est maintenant dans l'état où il se trouvait après le dernier commit. N'utilisez pas `git restore` à moins que vous ne vouliez vraiment perdre vos changements.

Il arrive que nous ajoutions accidentellement des fichiers dont nous ne voulons pas. Il se peut qu'un script se soit détraqué ou que nous ayons copié des fichiers par accident. Par exemple, si vous créez un fichier par erreur :

```
$ echo 'asdf' > mistake.txt
```

Nous pouvons demander à Git de ”nettoyer” le dépôt, c'est-à-dire de supprimer tous les fichiers et répertoires non suivis. Cependant, comme cela va supprimer des fichiers, nous ferions mieux de l'exécuter d'abord en mode ”dry run” en utilisant `-n` :

```
$ git clean -fdn
```

Ceci affichera une liste de fichiers qui *seraient* supprimés si nous n'avions pas inclus `-n`. Si nous sommes d'accord avec la suppression proposée, faisons-la :

```
$ git clean -fd
```

Maintenant notre `mistake.txt` a disparu.

Enfin, avant de publier notre dépôt, une dernière chose : gardez à l'esprit que Git ne suit que les *fichiers*, pas les *dossiers*. Git ne garde trace des dossiers que s'ils font partie du chemin d'accès d'un fichier.

Ainsi, si nous créons un dossier et demandons à Git ce qu'il voit, il nous dira qu'il n'y a rien, car le dossier est vide :

```
$ mkdir folder
$ git status
```

Si vous avez besoin d'inclure un dossier ”vide” dans un dépôt Git pour une raison quelconque, vous devez y ajouter un fichier vide afin que Git puisse suivre le dossier en tant que partie de ce fichier.

Publions maintenant notre dépôt. Allez sur l'instance GitLab de l'EPFL ou GitHub et créez un dépôt. Vous pouvez le rendre public ou privé, mais ne créez pas de fichiers tels que des fichiers "Read Me" ou quoi que ce soit d'autre, juste un dépôt vide.

Ensuite, suivez les instructions pour un dépôt existant à partir de la ligne de commande. Copiez et collez les commandes qui vous sont données. Ces commandes ajouteront le dépôt nouvellement créé en tant que "remote" à votre dépôt local, c'est-à-dire un autre clone du dépôt que Git connaît. Puisque ce sera le seul remote, ce sera aussi le remote par défaut. Le remote par défaut est traditionnellement nommé `origin`. Les commandes données pousseront également vos modifications vers ce serveur distant. Une fois les commandes exécutées, vous pouvez rafraîchir la page de votre dépôt et voir vos fichiers.

Maintenant, faites un changement dans votre `hello.txt`, suivez le changement, et livrez-le. Vous pouvez ensuite synchroniser le commit avec le clone du dépôt en ligne :

```
$ git push
```

Vous pouvez également récupérer les changements du dépôt en ligne :

```
$ git pull
```

Cette commande n'a ici aucun effet, puisque personne d'autre n'utilise ce dépôt. Dans un scénario réel, d'autres développeurs disposeraient également d'un clone du dépôt sur leur machine et utiliseraient le même serveur distant par défaut. Ils apportaient leurs modifications et vous les récupéreriez.

Il est important de noter que `git pull` ne synchronise que la branche courante. Si vous souhaitez synchroniser les commits d'une autre branche, vous devez d'abord `git switch` vers cette branche.

De même, `git push` ne synchronise que la branche courante, et si vous créez une nouvelle branche, vous devez lui indiquer où pousser avec `-u` en passant à la fois le nom distant et le nom de la branche :

```
$ git switch -c exemple
$ git push -u origin exemple
```

Publier votre dépôt en ligne est une bonne chose, mais il y a parfois des fichiers que vous ne voulez pas publier. Par exemple, les fichiers binaires compilés à partir du code source dans le dépôt ne devraient probablement pas se trouver dans le dépôt, car ils peuvent être recréés facilement et ne feraient qu'occuper de l'espace. Les fichiers contenant des données sensibles telles que des mots de passe ne doivent pas non plus se trouver dans le dépôt, surtout s'il est public. Simulons un fichier sensible :

```
$ echo '1234' > password.txt
```

Nous pouvons dire à Git de faire comme si ce fichier n'existait pas en ajoutant une ligne avec son nom dans un fichier spécial appelé `.gitignore` :

```
$ echo 'password.txt' >> .gitignore
```

Maintenant, si vous essayez `git status`, il vous dira que `.gitignore` a été créé mais pas `password.txt` puisque vous avez dit à Git de l'ignorer.

Vous pouvez également ignorer des répertoires entiers. Notez que cela ne fonctionne que pour les fichiers qui n'ont pas encore été livrés au dépôt. Si vous avez déjà fait un commit dans lequel `password.txt` existe, ajouter son nom à `.gitignore` n'ignorera que les changements futurs, pas ceux passés. Si vous poussez accidentellement sur un dépôt public un commit avec un fichier contenant un mot de passe, vous devez supposer que le mot de passe est compromis et le changer immédiatement. Il existe des robots qui analysent les dépôts publics à la recherche de mots de passe qui ont été accidentellement inclus dans un commit, et ils trouveront votre mot de passe si vous le mettez dans un dépôt public, même pendant quelques secondes.

Maintenant que vous avez vu les bases de Git, il est temps de contribuer à un projet existant ! Vous le ferez par le biais d'une *pull request* (terminologie GitHub) / *merge request* (terminologie GitLab), qui est une demande adressée aux responsables d'un projet existant pour qu'ils intègrent vos modifications dans leur projet. Du point de vue de Git, il s'agit simplement de synchroniser les modifications entre les clones d'un dépôt.

Allez sur https://gitlab.epfl.ch/solal.pirelli/hello (pour l'EPFL) ou https://github.com/sweng-example/hello (pour les autres) et cliquez sur le bouton "Fork". Un *fork* est un clone du dépôt sous votre propre nom d'utilisateur, dont vous avez besoin ici parce que vous n'avez pas d'accès en écriture au dépôt original et que vous ne pouvez donc pas y apporter de modifications. Au lieu de cela, vous allez pousser les changements vers votre fork, sur lequel vous avez un accès en écriture, et ensuite demander aux mainteneurs du dépôt original d'accepter le changement. Vous pouvez également créer des branches à l'intérieur d'un fork, car un fork est simplement un autre clone du dépôt. En règle générale, si vous êtes un collaborateur d'un projet, vous utiliserez une branche dans le dépôt principal du projet, tandis que si vous êtes une personne extérieure souhaitant proposer une modification, vous créerez d'abord un fork.

Maintenant que vous avez une version forkée du projet, cliquez sur le bouton "Code" et copiez l'URL SSH, qui devrait commencer par `git@`. Ensuite, demandez à Git de créer un clone local de votre fork, bien que vous deviez d'abord retourner dans votre répertoire d'origine, car la création d'un dépôt dans un dépôt pose des problèmes :

```
$ cd ~
$ git clone git@...
```

Git va cloner votre fork localement, ce qui vous permettra d'effectuer une modification, de la valider et de la pousser vers votre fork. Une fois que c'est fait, si vous allez sur votre fork, il devrait y avoir une bannière au-dessus du code vous indiquant que la branche de votre fork est en avance d'un commit par rapport à la branche principale du dépôt original. Utilisez cette bannière pour confirmer que vous voulez ouvrir une pull/merge request, et écrivez une description pour celle-ci.

Félicitations, vous avez apporté votre première contribution à un projet open source !

La meilleure façon de s'habituer à Git est de l'utiliser souvent. Utilisez Git même pour vos propres projets, même si vous n'avez pas l'intention d'utiliser des branches. Vous pouvez utiliser des dépôts privés sur GitHub, GitLab, etc. comme sauvegardes, de sorte que même si votre ordinateur tombe en panne, vous ne perdrez pas votre code.

Il existe de nombreuses fonctionnalités avancées dans Git qui peuvent être utiles dans certains cas, comme `bisect`, `blame`, `cherry-pick`, `stash`, et bien d'autres. Lisez la documentation officielle ou trouvez des tutoriels avancés en ligne pour en savoir plus si vous êtes curieux !

Comment écrire de bons messages de commit ?

Imaginez que vous soyez archéologue et que vous deviez comprendre ce qui s'est passé dans le passé en vous basant uniquement sur des dessins à moitié effacés, des fossiles et des traces. Vous finirez par trouver ce qui a pu se produire pour provoquer tout cela, mais cela prendra du temps et vous ne saurez pas si votre supposition est correcte. Ne serait-ce pas bien s'il existait à la place un journal que quelqu'un aurait rédigé, décrivant tout ce qu'il a fait d'important et pourquoi il l'a fait ?

C'est à cela que servent les messages de commit : garder une trace de ce que vous faites et de la raison pour laquelle vous l'avez fait, afin que d'autres personnes le sachent même des années après. Les messages de commit sont utiles aux personnes qui examinent votre code avant de l'approuver pour le fusionner dans la branche principale, ainsi qu'à vos collègues qui recherchent des bugs plusieurs mois après l'écriture du code. Dans ce contexte, vos collègues incluent le "futur vous". Même si les changements vous semblent "évidents" ou "clairs" au moment où vous les effectuez, quelques mois plus tard, vous ne vous souviendrez plus de la raison pour laquelle vous avez agi de la sorte.

Le format typique d'un message de validation est un résumé d'une ligne suivi d'une ligne vide et d'autant de lignes que nécessaire pour les détails. Par exemple, voici un bon message de commit :

```
Correction de l'ajout de favoris sur les petits téléphones

L'écran des favoris comportait trop de boutons empilés sur la même ligne.
Sur les téléphones à petit écran, il n'y avait pas assez d'espace pour les afficher tous,
et le bouton "ajouter" était hors de vue.
```

```
Cette modification ajoute une logique permettant d'utiliser plusieurs rangées de boutons si nécessaire.
```

Comme nous l'avons vu précédemment, l'écrasement des commits est une option lors de la fusion de votre code dans la branche principale, de sorte que tous les commits d'une branche n'ont pas besoin d'avoir des messages aussi détaillés. Parfois, un commit se résume à "Corriger une coquille" ou "Ajouter un commentaire selon le feedback des collègues". Ces modifications ne sont pas importantes pour comprendre les changements, leurs messages seront donc supprimés une fois que la branche sera réduite à un seul commit lors de la fusion.

Le résumé d'une ligne est utile pour avoir une vue d'ensemble de l'histoire sans avoir à en voir tous les détails. Vous pouvez le voir sur des dépôts en ligne tels que GitHub, mais aussi localement. Git dispose d'une commande `log` pour afficher l'historique, et `git log --oneline` n'affichera que le résumé d'une ligne de chaque commit.

Un bon résumé doit être court et à l'impératif. Par exemple :

- "Corriger le bug #145"
- "Ajouter une version HD du fond d'écran"
- "Supporter Unicode 14.0"

Les détails doivent décrire *ce que* les changements font et *pourquoi* vous les avez faits, mais pas *comment*. Il est inutile de décrire comment, car le message de commit est associé au contenu du commit, et celui-ci décrit déjà la manière dont vous avez modifié le code.

Comment éviter de fusionner du code buggué ?

La fusion de code buggué dans la branche principale d'un dépôt est une gêne pour tous les contributeurs de ce dépôt. Ils devront corriger le code avant de faire le travail qu'ils veulent réellement faire, et ils ne le corrigeront peut-être pas tous de la même manière, ce qui entraînera des conflits.

Idéalement, nous n'accepterions les demandes de retrait que si le code résultant compile, est "propre" selon les normes de l'équipe et a été testé. Chaque équipe a une idée différente de ce qu'est un code "propre", ainsi que de ce que signifie le terme "test", qui peut être manuel, automatisé, effectué sur une ou plusieurs machines, etc.

Lorsque l'on travaille dans un IDE, il existe généralement des options de menu permettant d'analyser le code pour en vérifier la propreté, le compiler, l'exécuter et lancer des tests automatisés si les développeurs en ont écrit. Cependant, tout le monde n'utilise pas le même IDE, ce qui signifie qu'ils peuvent avoir des définitions différentes de ce que ces opérations signifient.

Le principal problème lié à l'utilisation d'opérations dans un IDE pour vérifier les propriétés du code est que les humains font des erreurs. Dans les projets de grande envergure, les erreurs humaines sont fréquentes. Par exemple, il n'est pas raisonnable de s'attendre à ce que des centaines de développeurs n'oublient jamais, ne serait-ce qu'une fois, de vérifier que le code se compile et s'exécute. Vérifier les erreurs de base est également une mauvaise utilisation du temps des gens. L'examen du code devrait porter sur la logique du code, et non sur la validité syntaxique de chaque ligne, ce qui est du ressort du compilateur.

Nous aimerions plutôt *automatiser* les étapes nécessaires à la vérification du code. Cela se fait à l'aide d'un *système de build*, tel que CMake pour C++, MSBuild pour C#, ou Gradle pour Java. Il existe de nombreux systèmes de build, dont certains prennent en charge plusieurs langues, mais ils offrent tous fondamentalement la même fonctionnalité : l'automatisation de tâches. Un système de build peut invoquer le compilateur sur les bons fichiers avec les bons drapeaux pour compiler le code, et invoquer le binaire résultant pour exécuter le code, et même effectuer des opérations plus complexes telles que le téléchargement de dépendances sur la base de leur nom si elles n'ont pas déjà été téléchargées.

Les systèmes de build sont configurés avec du code. Ils disposent généralement d'un langage déclaratif personnalisé intégré dans un autre langage tel que le XML. Voici un exemple de code de construction pour MSBuild :

```
<Projet Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="Microsoft.Z3" Version="4.10.2" />
  </ItemGroup>
</Projet>
```

Ce code indique à MSBuild que (1) il s'agit d'un projet .NET, qui est le runtime typiquement associé à C#, et (2) qu'il dépend de la librairie `Microsoft.Z3`, en particulier de sa version `4.10.2`. On peut alors lancer MSBuild avec ce fichier à partir de la ligne de commande, et MSBuild compilera le projet après avoir téléchargé la librairie dont il dépend, si elle n'a pas déjà été téléchargée. Dans ce cas, le nom de la librairie est associé à une librairie réelle en recherchant le nom sur NuGet, le catalogue de librairies associé à MSBuild.

Les systèmes de build suppriment la dépendance à l'égard d'un IDE pour la construction et l'exécution du code, ce qui signifie que chacun peut utiliser l'éditeur de son choix à condition d'utiliser le même système de build. La plupart des IDE peuvent utiliser le code du système de build comme base pour leur propre configuration. Par exemple, le fichier ci-dessus peut être utilisé tel quel par Visual Studio pour configurer un projet.

Les systèmes de build permettent aux développeurs de construire, d'exécuter et de vérifier leur code n'importe où. Mais il faut bien qu'il soit quelque part, alors quelle(s) machine(s) doivent-ils utiliser ? Une fois de plus, l'utilisation de la machine spécifique d'un développeur n'est pas une bonne idée, car les développeurs personnalisent leur machine en fonction de leurs préférences personnelles. Les machines utilisées par les développeurs peuvent ne pas être représentatives des machines sur lesquelles le logiciel fonctionnera réellement lorsqu'il sera utilisé par les clients.

De la même manière que nous avons défini les constructions à l'aide de code via un système de build, nous pouvons définir les environnements à l'aide de code ! Voici un exemple de code de définition de l'environnement pour le système de conteneurs Docker, que vous n'avez pas besoin de comprendre :

```
FROM node:12-alpine
RUN apk add python g++ make
COPY . .
RUN yarn install
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Ce code indique à Docker d'utiliser l'environnement de base `node:12-alpine`, qui a Node.js préinstallé sur un environnement Linux Alpine. Ensuite, Docker doit exécuter `apk add` pour installer des paquets spécifiques, y compris `make`, un système de build. Docker doit alors copier le répertoire courant à l'intérieur du conteneur, et lancer `yarn install` pour invoquer le système de build `yarn` de Node.js afin de pré-installer les dépendances. Le fichier indique également à Docker la commande à exécuter lors du démarrage de cet environnement et le port HTTP à exposer au monde extérieur.

La définition d'un environnement à l'aide de code permet aux développeurs d'exécuter et de tester leur code dans des environnements spécifiques qui peuvent être personnalisés pour correspondre aux environnements des clients. Les développeurs peuvent également définir des environnements multiples, par exemple pour s'assurer que leur logiciel peut fonctionner sur différents systèmes d'exploitation, ou sur des systèmes d'exploitation dans différentes langues.

Nous avons utilisé le terme ”machine” pour désigner l'environnement dans lequel le code s'exécute, mais dans la pratique, il est peu probable qu'il s'agisse d'une machine physique, car cela serait inefficace et coûteux. Les ”pull requests” et les ”pushes” sont assez rares étant donné que les ordinateurs modernes peuvent effectuer des milliards d'opérations par seconde. Approvisionner une machine exclusivement pour un projet serait un gaspillage.

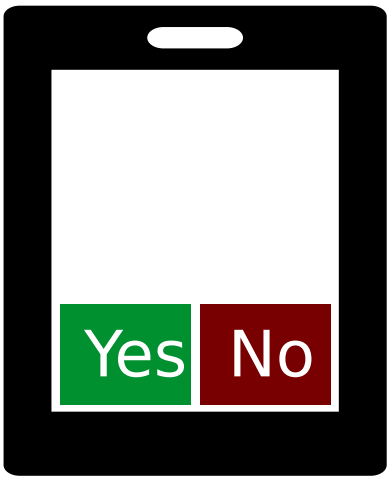
Au lieu de cela, les constructions automatisées utilisent des *machines virtuelles* ou des *conteneurs*. Une machine virtuelle est un programme qui émule une machine entière en son sein. Par exemple, il est possible d'exécuter une machine virtuelle Ubuntu sur Windows. Du point de vue de Windows, la machine virtuelle

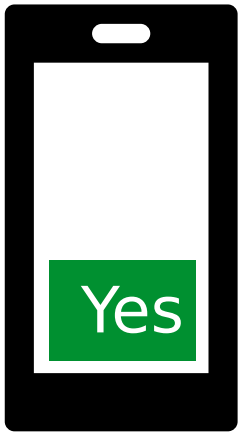
n'est qu'un programme parmi d'autres. Mais pour les programmes qui s'exécutent dans la machine virtuelle, c'est comme s'ils s'exécutaient sur du vrai matériel. Cela permet de partitionner les ressources : une seule machine physique peut faire tourner plusieurs machines virtuelles, surtout si ces dernières ne sont pas toutes occupées en même temps. Il isole également les programmes s'exécutant dans la machine virtuelle, ce qui signifie que même s'ils tentent de casser le système d'exploitation, le monde extérieur à la machine virtuelle n'est pas affecté. Cependant, les machines virtuelles ont des frais généraux, en particulier lorsqu'elles sont nombreuses. Même si 100 machines virtuelles exécutent toutes la même version de Windows, par exemple, elles doivent toutes exécuter une instance distincte de Windows, y compris le noyau Windows. C'est là qu'interviennent les *conteneurs*. Les conteneurs sont une forme légère de machines virtuelles qui partagent le noyau du système d'exploitation hôte au lieu d'inclure leur propre noyau. Il y a donc moins de duplication des ressources, au prix d'un moindre isolement. En règle générale, les services qui permettent à quiconque de télécharger du code utiliseront des machines virtuelles pour l'isoler autant que possible, tandis que les services privés peuvent utiliser des conteneurs puisqu'ils font confiance au code qu'ils exécutent.

L'utilisation de systèmes de compilation et de machines virtuelles pour compiler, exécuter et vérifier automatiquement le code chaque fois qu'un développeur apporte des modifications est appelée *intégration continue*, et il s'agit d'une technique clé dans le développement des logiciels modernes. Lorsqu'un développeur ouvre une demande d'extraction, l'intégration continue peut effectuer les vérifications configurées, par exemple tester que le code se compile et qu'il passe une analyse statique. La fusion peut alors être bloquée si l'intégration continue ne réussit pas. Ainsi, personne ne peut accidentellement fusionner du code cassé dans la branche principale, et les développeurs qui examinent les demandes d'extraction n'ont pas besoin de vérifier manuellement que le code fonctionne.

Il est important de noter que la réussite ou l'échec d'une opération spécifique d'intégration continue signifie qu'il existe une machine sur laquelle le code réussit ou échoue. Il est possible qu'un code fonctionne parfaitement sur la machine du développeur qui l'a écrit, mais qu'il échoue lors de l'intégration continue. Une réponse courante est ”mais ça marche sur ma machine !”, mais cela n'a rien à voir. L'objectif d'un logiciel n'est pas de fonctionner sur la machine du développeur, mais de fonctionner pour les utilisateurs.

Les problèmes liés à l'intégration continue proviennent généralement de différences entre les machines des développeurs et les machines virtuelles configurées pour l'intégration continue. Par exemple, un développeur peut tester une application téléphonique sur son propre téléphone, avec un scénario de test consistant à ”ouvrir la page 'créer un article' et cliquer sur le bouton 'non'”, ce qu'il peut faire sans problème. Mais leur environnement d'intégration continue peut être configuré avec un émulateur de téléphone doté d'un petit écran avec peu de pixels, et la façon dont l'application est écrite signifie que l'on ne peut pas l'utiliser. le bouton ”non” n'est pas visible :





.

Le code ne fonctionne donc pas dans l'environnement d'intégration continue, non pas à cause d'un problème d'intégration continue, mais parce que le code ne fonctionne pas sur certains téléphones. Le développeur

devrait corriger le code pour que le bouton "Non" soit toujours visible, éventuellement sous le bouton "Oui" avec une barre de défilement si nécessaire.

Exercice : Ajouter l'intégration continue Retournez au dépôt en ligne que vous avez créé, et ajoutez l'intégration continue !

Les instructions différent selon le site que vous utilisez.

GitLab CI/CD GitLab inclut un service d'intégration continue appelé GitLab CI/CD, qui est gratuit pour une utilisation de base. Voici un fichier de base que vous pouvez utiliser, qui doit être nommé `.gitlab-ci.yml` :

```yaml
stages:
  - hello

hello:
  stage: hello
  script:
    - echo "Hello World"
```

Après avoir ajouté ce fichier au dépôt GitLab et attendu quelques secondes, vous devriez voir un cercle jaune à côté du commit indiquant que votre action est en cours d'exécution, que vous pouvez également voir dans l'onglet "Build > Pipelines" du dépôt.

...**cependant, cette action ne s'exécutera pas car l'instance GitLab de l'EPFL n'a pas de "runners" disponibles pour nous**.

Il s'agit d'une action très basique qui se contente de cloner le dépôt et d'imprimer du texte. Dans un scénario réel, vous devriez au moins invoquer un système de build. GitLab CI/CD est puissant, comme vous pouvez le lire sur la documentation.

GitHub Actions GitHub inclut un service d'intégration continue appelé GitHub Actions, qui est gratuit pour une utilisation de base. Voici un fichier de base que vous pouvez utiliser, qui doit être nommé `.github/workflows/example.yml` :

```yaml
on: push
jobs:
  example:
    runs-on: ubuntu-latest
      steps:
        - uses: actions/checkout@v3
        - run: echo "Hello!"
```

Après avoir ajouté ce fichier au dépôt GitHub et attendu quelques secondes, vous devriez voir un cercle jaune à côté du commit indiquant que votre action est en cours d'exécution, que vous pouvez également voir dans l'onglet "Actions" du dépôt. Il s'agit d'une action très basique qui se contente de cloner le dépôt et d'imprimer du texte. Dans un scénario réel, vous devriez au moins invoquer un système de build. GitHub Actions est puissant, comme vous pouvez le lire sur la documentation.

Le contrôle des versions, l'intégration continue et d'autres tâches de ce type étaient généralement appelés "opérations" et étaient effectués par une équipe distincte de l'équipe de "développement". Cependant, de nos jours, ces concepts se sont combinés en "DevOps", dans lequel la même équipe fait les deux, ce qui permet aux développeurs de configurer plus facilement exactement les opérations qu'ils souhaitent.

Résumé

Dans ce cours, vous avez appris :

- Les systèmes de gestion de version et les différences entre la première, la deuxième et la troisième génération.
- Git : comment l'utiliser pour des scénarios de base, et comment écrire de bons messages de commit.
- Intégration continue : systèmes de build, machines virtuelles et conteneurs.

Vous pouvez maintenant consulter les exercices !

Conception

Imaginez que, pour afficher "Hello, World !" à l'écran, vous deviez apprendre comment tout fonctionne. Vous devriez tout apprendre sur les voyants LED au niveau de la physique théorique. Vous devriez ensuite lire des milliers de pages de fiches techniques sur les processeurs pour savoir quel code écrire en assembleur et comment communiquer avec des périphériques externes.

Au lieu de cela, vous pouvez écrire `print("Hello, World!")` dans un langage tel que Scala ou Python, et le tour est joué. Le runtime du langage fait tout le travail à votre place, avec l'aide de votre système d'exploitation, qui contient lui-même des pilotes pour le matériel. Même ces pilotes ne connaissent pas les voyants LED, car l'écran lui-même expose une interface pour afficher les données que les pilotes utilisent. Python lui-même n'est pas non plus un monolithe : il contient des sous-modules tels qu'un tokenizer, un parseur et un interpréteur.

Malheureusement, il n'est pas facile d'écrire de grandes bases de code de manière claire, et c'est là que la conception entre en jeu.

Objectifs

À l'issue de ce cours, vous devriez être capable de :

- Écrire du code orienté objet
- Choisir entre héritage et composition
- Appliquer la modularité et l'abstraction en pratique

Comment encapsuler du code derrière une interface ?

Il ne s'agit ici pas d'interface utilisateur, mais bien d'interface machine : comment est-ce que des parties différentes de votre code peuvent se parler, sans avoir besoin de connaître tous les détails ?

Avec des fonctions, vous pouvez faire des groupes, mais cela devient vite beaucoup :

```
def animal_describe(kind): ...
def animal_draw(kind, canvas): ...
def animal_name(kind): ...
def str_replace(s, old, new): ...
def str_capitalize(s): ...
```

Ces fonctions pourraient se partager des variables globales. Par exemple, une méthode `animal_pet` pourrait augmenter une valeur stockée dans une table globale définissant à quel point chaque animal est content. Cette table pourrait ensuite être lue par la fonction qui décrit un animal.

Mais il est facile de modifier une variable globale de manière non prévue par l'auteur des autres fonctions, ce qui rend le code complexe à modifier. De plus, avec un éditeur de code, si vous écrivez `animal_` et que votre éditeur vous propose les 50 fonctions et variables commençant par ce nom, comment sauriez-vous lesquels vous devez utiliser et lesquels sont des détails d'implémentation ?

L'idée principale derrière la solution est de grouper des fonctions et des variables dans des *objets*. Les fonctions sur un objet sont appelées des *méthodes*. Chaque méthode a accès à l'*instance* de l'objet sur laquelle elle est appelée. Ainsi, un programme orienté objet crée des objets et appelle des méthodes dessus. Chaque appel peut potentiellement altérer l'état d'un objet, ce qui peut changer le comportement des méthodes appelées plus tard.

On utilise des objets en déclarant leur classe :

```
class Animal:
  def describe(self): ...
  def draw(self, canvas): ...
  def name(self): ...
```

On voit déjà qu'au lieu d'avoir des fonctions potentiellement éparses parmi le code source, elles sont toutes au même endroit, avec `self` indiquant l'instance de la classe sur laquelle elles opérent.

Note

En Python comme dans d'autres langages, il faut distinguer le code qui définit des classes et des fonctions du code qui les exécute. Par exemple :

```
class Thing:
  def method(self): ...

def function(): ...

print("Hello")
```

Ce code définit une classe `Thing` avec une méthode `method`, ainsi qu'une fonction `function` qui ne fait pas partie d'une classe, puis exécute `print("Hello")`. Le code à l'intérieur de `method` et de `fonction` n'est pas exécuté, puisqu'elles ne sont pas appelée.

On crée une instance en utilisant le nom de la classe puis les arguments de création. Par défaut, il n'y a pas à donner d'argument :

```
a = Animal() # crée une instance de Animal et la stocke dans la variable 'a'
b = Animal() # de même, dans la variable 'b'

print(a == b) # écrit False, ce sont des instances différentes !
```

Si nécessaire, on peut définir un *constructeur* explicite qui peut prendre des paramètres :

```
class Animal:
  def __init__(self, name):
    ...

a = Animal("Chat") # invoque __init__ avec `self` étant la nouvelle instance et `name` étant `"Chat"`
```

Le constructeur peut par exemple stocker un paramètre comme *état* de la classe :

```
class Animal:
  def __init__(self, kind):
    self.kind = kind

a = Animal("Chat")
print(a.kind) # Chat
```

Une partie de l'état d'une classe est souvent *privé*, c'est-à-dire que seule la classe peut l'utiliser, ce qui est la base de l'encapsulation. Par exemple, pour garder une trace de si un animal est content, vous pouvez initialiser un champ privé dans le constructeur :

```
class Animal:
  def __init__(self, kind):
    self.kind = kind
    self._happy = True
```

Ce champ `_happy` peut ensuite être modifié dans des méthodes telles que ”donner à manger à l'animal”, ”caresser l'animal”, ”emmener l'animal chez le vétérinaire”...

Comme il commence par un souligné `_`, la convention en Python est que personne ne doit l'utiliser hors de la classe. Vous pouvez l'utiliser quand même si vous le voulez, mais c'est une mauvaise idée, et la présence d'un souligné `_` vous indique immédiatement que vous faites quelque chose de bizarre.

Pour utiliser l'état défini dans le constructeur, ainsi que modifier l'état d'un objet et agir en général, on définit des *méthodes* :

```
class Animal:
  def describe(self):
    return "Un " + self.kind

a = Animal("Chat")
print(a.describe()) # Un Chat (`self` dans la méthode se réfère à `a` ici)
```

Par analogie avec l'état privé, vous pouvez déclarer des méthodes privées :

```
class Animal:
  def _increase_happiness(self):
    ...
```

Il est également possible de déclarer des méthodes avec certains noms spéciaux, comme `__add__`, qui peuvent être utilisées comme des opérateurs, comme `+`. Référez vous à la documentation.

Nous avons vu comment déclarer une méthode pour un type de données, mais parfois vous avez besoin de plusieurs fonctions selon le type exact :

```
def cat_describe(): ...
def dog_describe(): ...
def giraffe_describe(): ...
```

Ceci se fait en programmation orientée objet grâce à l'**héritage** :

```
class Animal: ...

class Cat(Animal): ...

class Dog(Animal): ...
```

On dit ici qu'un `Cat` *est un* `Animal`. La ”sous-classe” `Cat` hérite des méthodes de la ”super-classe” `Animal` et peut redéfinir certaines méthodes :

```
class Animal:
  def describe(self): return "Un animal"
  def happy(self): return self._happy

class Cat(Animal):
  def describe(self): return "Un chat"
```

Les appels de méthodes sont *dynamiques*, c'est-à-dire qu'ils dépendent du type exact de l'instance sur laquelle ils sont appelés :

```
def example(a: Animal):
  print(a.describe())
```

```
example(Cat()) # "Un chat"
```

Dans cet exemple, même si nous avons donné le type `Animal` au paramètre `a`, puisque nous passons un `Cat`, c'est la méthode `Cat.describe` qui est appelée.

Si besoin, vous pouvez appeler la méthode de la super-classe spécifiquement avec la syntaxe `super().`:

```
class Dog(Animal):
  def describe(self):
    return "Un chien est " + super().describe()

example(Dog()) # "Un chien est Un animal"
```

Exercice, partie 1 Ouvrez le fichier `interactions.py` dans le dossier d'exercices pendant le cours.

Comme indiqué en commentaire, déplacez le code interagissant avec l'utilisateur dans une nouvelle `class UI`.

Par exemple :

```
class UI:
    def show(self, text):
        print(text)

    def input(self, prompt):
        result = None
        while not result:
            print(prompt)
            result = input()
        return result
```

Le reste du code devient maintenant beaucoup plus clair :

```
ui = UI()
ui.show('Bonjour !')
name = ui.input('Quel est votre nom ?')
ui.show('Bienvenue, ' + name + '!')
hobby = ui.input('Quel est votre passe-temps préféré ?')
ui.show(hobby + ', quelle bonne idée !')
```

Exercice, partie 2 Gardez votre solution de la partie 1, et ouvrez le fichier `interactions-tkinter.py` dans le dossier d'exercices pendant le cours.

Les fonctions dans ce fichier permettent d'afficher du texte et de demander du texte à l'utilisateur à l'aide de `tkinter`, sorte de boîte à outils graphique pour des interfaces utilisateur basiques en Python.

Écrivez une classe `GraphicalUI` avec la même interface que `UI`, donc les mêmes noms de méthodes, pour que vous puissiez réutiliser la logique de la partie 1 tout en changeant la manière dont le code interagit avec l'utilisateur.

Notez qu'en Python, contrairement aux des langages statiquement typés, il n'y a pas besoin d'avoir une super-classe commune pour utiliser deux classes de la même manière. Si un objet marche comme un canard et fait coin-coin comme un canard, c'est un canard, ce qui donne le nom ”duck typing” en anglais. Par exemple :

```
class Duck:
  def quack(self): print("Quack")
```

```
class Sheep:
  def quack(self): print("Baaa???")
```

Un `Duck` peut `quack`, mais un `Sheep` a apparemment aussi appris à `quack`, donc toute méthode s'attendant à un `Duck` peut aussi utiliser un `Sheep`. Ce qui ne veut pas forcément dire que c'est une bonne idée !

Enfin, avant de passer à la suite, une astuce Python : si vous avez besoin d'une classe juste pour grouper des valeurs, comme par exemple un nom et un age appartenant à une personne, vous pouvez utiliser `dataclass` pour ne pas avoir à écrire le constructeur `__init__` à la main :

```
from dataclasses import dataclass

@dataclass
class Person:
    name: str
    age: int

p = Person("Alice", 7)
print(p.name)
```

Nous ne parlerons pas plus de `dataclass`, mais c'est bon à savoir.

Comment utiliser les objets en pratique ?

Nous venons de voir comment écrire des classes et utiliser l'héritage et les attributs en Python, mais qu'en est-il de la pratique ? Comment choisir ce qui doit être public ou privé, ce qui doit hériter d'une autre classe... ?

Déjà, concernant la distinction public/privé, préférez les méthodes privées et attributs privés par défaut. Il est facile de changer dans la direction "privé => public", car personne d'autre ne peut utiliser les choses privées, donc donner au reste du code la possibilité d'utiliser quelque chose qui était auparavant privé ne pose pas de problème. Mais l'inverse n'est pas vrai : changer dans la direction "public => privé" peut causer beaucoup de problèmes, car tout le code qui utilisait ce qui était avant public et est maintenant privé ne fonctionne plus. Si vous vous rendez compte qu'une méthode n'aurait pas du être publique car elle permet de casser l'encapsulation fournie par sa classe, par exemple, vous devez changer tout le code hors de la classe qui l'utilise. Dans le cas où votre code est utilisé par d'autres, par exemple si vous avez publié votre code sur un dépôt de paquets afin que d'autres puissent l'utiliser, les utilisateurs ne seront pas content que leur code ne marche plus avec votre nouvelle version qui rend une méthode privée alors qu'elle était publique.

Notons aussi que le fait qu'une classe a des méthodes publiques et privées ne signifie pas qu'on peut toujours la remplacer par une implémentation différente qui fournisse "la même chose", à cause des performances. Par exemple, une carte de la Terre implémentée avec la projection de Mercator peut très facilement fournir l'opération "quel angle dois-je utiliser pour naviguer entre deux points", car conserver les angles est justement le but de cette projection. Vous pourriez changer l'implémentation de la carte pour utiliser une autre projection, mais si calculer un angle est beaucoup plus lent avec votre projection différente, l'implémentation ne sera pas forcément utilisable en pratique avec le niveau de performance attendu.

Exercice Vous implémentez un réseau social et avez besoin de stocker des images. Pour l'instant, vous stockez ces images sur le disque local de votre machine.

Lesquelles de ces méthodes devraient être préfixées d'un `_` pour indiquer qu'elles sont privées ?

```
class PictureStorageOnDisk:
  def get_picture(self, id): ...
  def get_file_name(self, id): ...
  def get_all_pictures(self): ...
```

`get_picture` est le but principal de la classe donc doit rester public.

`get_file_name` est un détail d'implémentation. Si vous stockiez les fichiers dans une base de données, ou sur un site externe, le concept même de "nom de fichier" n'aurait pas de sens, cette méthode doit donc être privée.

`get_all_pictures` est discutable. Cela est potentiellement utile pour le reste de l'application, mais selon la manière dont les images sont stockées, cette opération pourrait être extrêmement inefficace, par exemple en téléchargeant des milliers d'images. Il vaut mieux la laisser en privé jusqu'à ce qu'il y ait un vrai besoin.

Il est facile de céder à la tentation du "juste au cas où..." et de rendre ses interfaces extrêmement générales. Le stockage d'images de l'exercice que vous venez de faire pourrait être un stockage de "données" prenant non seulement leur identifiant mais aussi leur type, un Booléen indiquant si un cache local doit être utilisé, et bien d'autres paramètres. En interne, c'est peut-être comme cela que ce stockage sera implémenté. Mais il vaut mieux garder une interface publique spécifique, qui facilite l'utilisation de la classe.

Un autre choix que vous devez faire en écrivant des classes concerne l'héritage. Un chat "est un" animal, un participant à un évènement "est une" personne, un sponsor de cet évènement "est une" société. Par contre, un chat "a une" tête, un participant "a une" adresse email, un sponsor "a un" compte bancaire.

Il est donc normal d'écrire `class Chat(Animal)`, mais pas `class Chat(Tête)`. Ce n'est pas parce qu'on peut utiliser la tête d'un chat pour le caresser qu'un chat est plus généralement une tête. L'exemple du sponsor est flagrant : oui, un sponsor vous fournit de l'argent pour votre évènement, mais vous ne pouvez pas créer un compte chez votre sponsor pour y verser ou y retirer de l'argent à votre guise !

Gardez à l'esprit le principe de substitution de Liskov, nommé d'après Barbara Liskov, informaticienne et pionnière de l'abstraction et de l'encapsulation dans les langages de programmation. Si `X` est un `Y`, alors partout où l'on peut utiliser un `Y`, on peut également utiliser un `X`. On peut donc utiliser un `Chat` partout où l'on peut utiliser un `Animal` en général. Mais on ne peut pas utiliser un `Participant` partout où l'on peut utiliser une `AdresseEmail`. Les participants ne veulent peut-être même pas que vous connaissiez leur adresse !

Un exemple connu est celui de la classe `Stack` en Java, qui est censé représenter une "pile" où l'on peut déposer des objets en mode "premier arrivé, dernier sorti". Si vous déposez `1, 2, 3`, vous retirez `3, 2, 1` dans l'ordre. En interne, cette classe pourrait utiliser une représentation similaire à celle d'une "liste" comme la `list` en Python. Mais comme Java est un des premiers langages orientés objet à être devenu populaire, sa librairie standard contient des erreurs de débutant. Dans ce cas-ci, `Stack` *hérite* de la classe pour les listes au lieu d'en *contenir* une. Donc n'importe qui peut ajouter ou supprimer des objets à n'importe quel index de la pile, ce qui détruit l'encapsulation.

Enfin, gardez à l'esprit que les "types de données" en théorie ne correspondent pas forcément aux types que vous pouvez utiliser en pratique. Par exemple, un `âge` peut être un `int` en Python, mais cela ne veut pas pour autant dire que `-500` est un âge ! Quand vous réfléchissez au types de vos données, souvenez-vous que vous pouvez être plus précis dans la documentation et avec du code qui vérifie les valeurs que les types généraux que vous avez à disposition. Certains langages vous permettent d'ailleurs d'être plus précis que Python et de définir vous-même des types comme "les entiers de 0 à 9 inclus".

Les types de données sont parfois définis en fonction d'autres types. "Une liste", implicitement, c'est une liste de quelque chose, par exemple une `list[int]`, une `list[str]`, ou même une `list[list[int]]`. Cela mène à une intersection intéressante avec l'héritage. Sachant qu'un `Chat` est un `Animal`, est-ce qu'une `Image[Chat]` est une `Image[Animal]` ? Oui, clairement. Est-ce que de la `NourriturePour[Chat]` est plus généralement de la `NourriturePour[Animal]` ? Non ! Tous les animaux ne peuvent pas manger tout ce que les chats peuvent manger. Par contre, si vous avez de la `NourriturePour[Animal]` en général, c'est de la `NourriturePour[Chat]` puisque si quelque chose convient à tous les animaux, cela convient clairement aux chats.

Et nos listes ? Est-ce qu'une `list[Chat]` est une `list[Animal]` ? Si c'était le cas, on pourrait écrire du code comme cela :

```python
def add_tiger(lst: list[Animal]): ...

lst = [giraffe1, giraffe2]
add_tiger(lst) # oups, on ajoute un tigre dans une liste de girafes, pauvres girafes
```

Est-ce donc l'inverse ? Une `list[Animal]` est-elle une `list[Chat]` ? Non plus, si on retire un élément de la liste, c'est un `Animal` mais pas forcément un `Chat`. Il n'y a donc aucune relation d'héritage entre `list[Chat]` et `list[Animal]`.

Formellement, on parle de "variance" :

- La **covariance**, c'est l'usage en sortie : une image de chat est une image d'animal
- La **contravariance**, c'est l'usage en entrée : la nourriture pour animal est de la nourriture pour chat
- L'**invariance**, c'est ni l'un ni l'autre : une liste de chats et une liste d'animaux ne peuvent pas être utilisées l'une pour l'autre

Comment concevoir de grands systèmes logiciels ?

Barbara Liskov, que nous venons de voir ci-dessus, a un jour fait remarquer que *la technique de base dont nous disposons pour gérer la complexité des logiciels est la modularité.*

La modularité consiste à diviser et subdiviser les logiciels en unités indépendantes qui peuvent être maintenues séparément et réutilisées dans d'autres systèmes : les modules. Chaque module possède une interface, qui est ce que le module expose au reste du système. Le module abstrait certains concepts et présente cette abstraction au monde. Le code qui utilise le module n'a pas besoin de savoir ni de se soucier de la manière dont l'abstraction est mise en oeuvre, mais seulement de son existence.

Par exemple, il n'est pas nécessaire de s'y connaître en menuiserie ou en textiles pour comprendre comment utiliser un canapé. Certains canapés peuvent même être personnalisés par les clients, qui peuvent par exemple choisir d'y ajouter un compartiment de rangement ou un lit convertible, car les canapés sont composés de sous-modules.

En programmation, l'interface d'un module est généralement appelée "API", abréviation de "Application Programming Interface" (interface de programmation d'application). Les API contiennent des objets, des fonctions, des erreurs, des constantes, etc. Ce n'est pas la même chose que le concept d'"interface" dans les langages de programmation tels que Java. Dans cette leçon, nous aborderons la notion générale d'interface, et non la mise en oeuvre spécifique de ce concept dans un langage particulier.

Prenons l'exemple de la fonction Python suivante :

```python
def compute(expr: str) -> int:
  # ...
```

Elle peut être considérée comme un module dont l'interface est la signature de la fonction. Les utilisateurs de ce module n'ont pas besoin de savoir comment le module calcule les expressions, par exemple en renvoyant `4` pour `2 + 2`. Ils ont seulement besoin de comprendre son interface.

Une interface similaire peut être écrite dans une technologie différente, telle que COM de Microsoft :

```
[uuid(a03d1424-b1ec-11d0-8c3a-00c04fc31d2f)]
interface ICalculator : IDispatch {
  HRESULT Compute([in] BSTR expr,
                  [out] long* result);
};
```

Il s'agit de l'interface d'un composant COM, conçue pour être utilisable et implémentable dans différents langages. Elle définit fondamentalement le même concept que celle de Python, à l'exception d'une manière différente de définir les erreurs (exceptions vs codes `HRESULT`) et de renvoyer les données (valeurs de retour vs paramètres `[out]`). Tout le monde peut utiliser ce module COM grâce à son interface, sans avoir à connaître ou à se soucier de la manière dont il est implémenté et dans quel langage.

Un autre type d'interface inter-programmes est HTTP, qui peut être utilisé via des frameworks serveur :

```
@Get("/api/v1/calc")
String compute(@Body String expr) {
  // ...
}
```

L'interface de ce serveur HTTP est `HTTP GET /api/v1/calc`, sachant que le seul paramètre doit être transmis dans le corps et que la valeur renvoyée sera une chaîne de caractères. Le nom de la méthode Java ne fait pas partie de l'interface, car il n'est pas exposé au monde extérieur. De même, le nom du paramètre `expr` n'est pas exposé non plus.

Ces trois interfaces peuvent être utilisées dans un seul système qui combine trois modules : un serveur HTTP qui appelle en interne un composant COM qui appelle en interne une fonction Python. Le serveur HTTP n'a même pas besoin de savoir que la fonction Python existe s'il passe par le composant COM, ce qui simplifie son développement.

Cependant, cela nécessite une certaine discipline dans l'application des modules. Si la fonction Python crée un fichier sur le disque local et que le serveur HTTP décide d'utiliser ce fichier, la modularité est rompue. Ce problème existe également au niveau des fonctions. Considérons la même fonction Python que ci-dessus, mais cette fois avec une fonction supplémentaire :

```
def compute(expr: str) -> int:
  # ...

def useReversePolishNotation():
  # ...
```

La deuxième fonction sert à configurer le comportement de la première. Cependant, elle crée une dépendance entre deux modules qui utilisent `compute`, car ils doivent s'accorder sur l'utilisation ou non de `useReversePolishNotation`. Si un module tente d'utiliser `compute` en supposant la notation standard, mais qu'un autre module du système a choisi d'utiliser la notation polonaise inversée, le premier échouera.

Un autre problème courant avec les modules est l'exposition volontaire d'informations non nécessaire, généralement parce que cela simplifie l'implémentation à court terme. Par exemple, une classe `User` avec un `name` et une `favoriteFood` devrait-elle également avoir un attribut booléen `fetchedFromDatabase` ? Cela peut avoir du sens dans une implémentation spécifique, mais le concept de suivi des aliments préférés des utilisateurs n'a absolument rien à voir avec une base de données. Les programmeurs écrivant du code qui utilise cette classe `User` devraient connaître les bases de données pour comprendre le concept d'utilisateur du système, et les responsables de la maintenance de la classe `User` elle-même ne pourraient plus modifier l'implémentation pour la rendre indépendante des bases de données, car l'interface impose un lien entre les deux concepts. De même, au niveau du paquet, un paquet `calc` pour une application de calculatrice avec une classe `User` et une classe `Calculator` ne devrait probablement pas avoir de classe `UserInterfaceCheckbox`, car il s'agit d'un concept de niveau beaucoup plus bas.

Exercice À quoi ressemblerait l'interface d'une classe `Student`...

- ... pour une application ”compagnon” de campus ?
- ... pour un système de gestion des cours ?
- ... pour un système d'authentification ? En quoi diffèrent-elles et pourquoi ?

Une application pour le campus pourrait considérer les étudiants comme ayant un nom et des préférences, par exemple s'ils souhaitent que les menus végétariens s'affichent en premier ou dans quel ordre afficher les cours de l'utilisateur.

Mais cette application ne se soucie pas de savoir si l'étudiant a payé ses frais pour le semestre en cours, ce qui pourrait intéresser un système de gestion des cours, ni de la filière dans laquelle il est inscrit.

Ni l'application ni le système de gestion des cours ne doivent connaître le mot de passe de l'utilisateur, ni même le concept de mot de passe, car l'utilisateur peut se connecter à l'aide de données biométriques ou d'une authentification à deux facteurs. Ces concepts sont ceux qui intéressent le système d'authentification pour les étudiants.

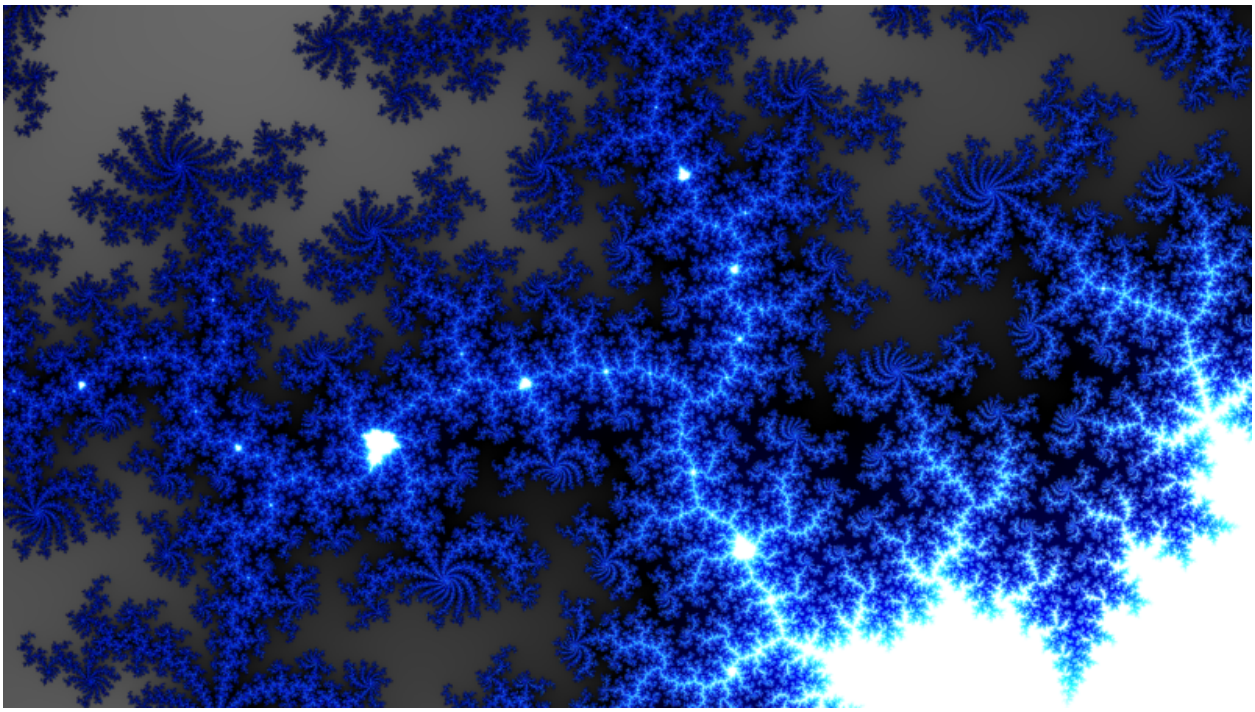
Qu'est-ce que la modularité exige dans la pratique ?

Il est trop facile d'écrire des systèmes logiciels dans lesquels chaque « module » est un mélange hétéroclite de concepts, les modules dépendant les uns des autres sans schéma clair. La maintenance d'un tel système nécessite de lire et de comprendre la majeure partie du code du système, ce qui n'est pas adapté aux grands systèmes. Nous avons vu les avantages théoriques et les pièges de la modularité, voyons maintenant comment concevoir des systèmes modulaires dans la pratique.

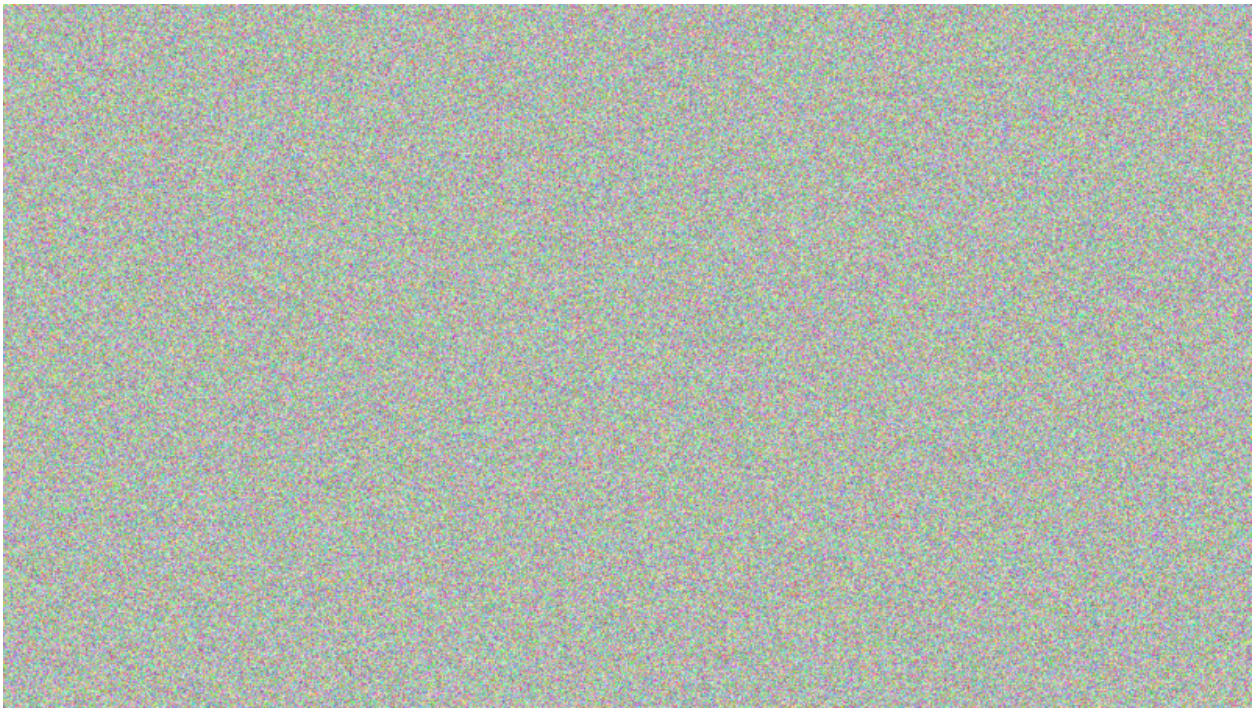
Nous aborderons la régularité des interfaces, le regroupement et la stratification des modules, ainsi que l'organisation des modules par niveau d'abstraction.

Régularité

Prenons l'exemple d'une fractale telle que celle-ci :



Cette image peut sembler complexe, mais comme il s'agit d'une fractale, elle est très régulière. Elle peut être formellement définie à l'aide d'une courte équation mathématique et d'une courte phrase. Comparez-la à cette image :



Il s'agit d'un bruit aléatoire. Il ne présente aucune régularité. La seule façon de le décrire est de décrire chaque pixel tour à tour, ce qui prend beaucoup de temps.

L'idée que les choses doivent être régulières et pouvoir être décrites en quelques mots s'applique également au code. Considérons l'extrait suivant du package `java.util` de Java :

```
class Stack {
  /** Renvoie la position basée sur 1 où se trouve un objet dans cette pile. */
  int search(Object o);
}
```

Pour une raison quelconque, `search` renvoie une position basée sur 1, alors que tous les autres index en Java sont basés sur 0. Ainsi, toute description de `search` doit inclure cette information, et un coup d'oeil rapide au code qui utilise `search` peut manquer un bug si l'index est accidentellement utilisé comme s'il était basé sur 0.

Il convient de suivre le ”principe de moindre surprise”, c'est-à-dire que les choses doivent se comporter de la manière dont la plupart des gens s'y attendent, et donc ne pas déroger aux règles courantes. Un autre exemple tiré de Java est la méthode `equals` de la classe `URL`. On pourrait s'attendre à ce que, comme toute autre vérification d'égalité en Java, `URL::equals` vérifie les champs des deux objets, ou peut-être un sous-ensemble de ceux-ci. Cependant, ce qu'elle fait en réalité, c'est vérifier si les deux URL renvoient à la même adresse IP. Cela signifie que le résultat dépend du fait que les deux URL pointent vers la même adresse IP à ce moment précis, et même du fait que la machine sur laquelle le code est exécuté dispose d'une connexion Internet. La résolution des adresses IP prend également du temps, ce qui est beaucoup plus lent que les méthodes equals habituelles qui vérifient l'égalité des champs.

Une manière plus formelle d'appréhender la régularité est la complexité de Kolmogorov : combien de mots faut-il pour décrire quelque chose ? Par exemple, la fractale ci-dessus a une faible complexité de Kolmogorov, car elle peut être décrite en très peu de mots. On peut écrire un petit programme informatique pour la produire. En comparaison, le bruit aléatoire ci-dessus a une complexité de Kolmogorov élevée, car il ne peut être décrit qu'avec de nombreux mots. Un programme permettant de le produire doit générer chaque pixel individuellement. Tout module dont la description doit inclure ”et...” ou ”sauf...” a une complexité de Kolmogorov plus élevée que nécessaire.

Groupes

Qu'ont en commun les classes suivantes ? `Map<K, V>`, `Base64`, `Calendar`, `Formatter`, `Optional<T>`, `Scanner`, `Timer`, `Date`.

Pas grand-chose, n'est-ce pas ? Pourtant, elles se trouvent toutes dans le même package `java.util` de la bibliothèque standard Java. Ce n'est pas une bonne interface de module : elle contient un ensemble d'éléments sans rapport les uns avec les autres ! Si vous voyez qu'un programme Java dépend de `java.util`, cela ne vous apprend pas grand-chose, car il s'agit d'un module très vaste.

Et les classes suivantes ? `Container`, `KeysView`, `Iterable`, `Sequence`, `Collection`, `MutableSet`, `Set`, `AsyncIterator`.

C'est beaucoup plus simple : ce sont toutes des collections, et elles se trouvent effectivement dans le module collections de Python. Malheureusement, ce module s'appelle `collections.abc`, car c'est un acronyme amusant pour "abstract base classes" (classes de base abstraites), ce qui n'est pas un très bon nom pour un module. Mais au moins, si vous voyez qu'un programme Python dépend de `collections.abc`, après avoir recherché le nom, vous savez maintenant qu'il utilise des structures de données.

L'importance de regrouper les éléments connexes explique pourquoi les variables globales posent un tel problème. Si plusieurs modules accèdent tous à la même variable globale, ils forment alors tous un seul module, car un programmeur doit comprendre comment chacun d'entre eux utilise la variable globale pour pouvoir les utiliser. Le regroupement effectué par les variables globales est accidentel et a donc peu de chances de produire des groupes utiles.

Couches

Vous connaissez peut-être déjà les couches de la pile réseau : la couche application utilise la couche transport, qui utilise la couche réseau, et ainsi de suite jusqu'à la couche physique au niveau le plus bas. La couche application n'utilise pas directement la couche réseau et ne sait même pas qu'il existe une couche réseau. La couche réseau ne sait pas non plus qu'il existe une couche application ou une couche transport.

Séparer un système en couches est un moyen de définir les dépendances entre les modules de manière minimale et gérable, de sorte que la maintenance d'un module puisse être effectuée sans connaissance de la plupart des autres modules.

Il peut y avoir plusieurs modules à un niveau donné : par exemple, une application peut utiliser des modules mobiles et serveurs, qui forment la couche inférieure à celle dans laquelle est le module "application". Le module serveur lui-même peut dépendre d'un module d'authentification et d'un module de base de données, qui forment la couche inférieure, et ainsi de suite.

Ainsi, la couche `N` dépend *uniquement* de la couche `N-1`, et le contexte pour l'implémentation de la couche `N` est l'interface de la couche `N-1`. En construisant des couches dans une pile, on minimise le contexte de l'implémentation de chaque couche.

Cependant, il est parfois nécessaire qu'une couche prenne des décisions en fonction d'une logique de niveau supérieur, comme le type de comparaison à utiliser dans une fonction de tri. Donner des connaissances sur les éléments de niveau supérieur dans la fonction de tri romprait la stratification et rendrait plus difficile la maintenance de la fonction de tri. Au lieu de cela, on peut injecter une fonction de « comparaison » comme paramètre de la fonction de tri :

```
def sort(items, less_than):
  ...
  if less_than(items[i], items[j]):
    ...
```

Les couches de niveau supérieur peuvent ainsi transmettre une fonction de comparaison de niveau supérieur, et la fonction de tri n'a pas besoin de dépendre explicitement d'une couche supérieure, ce qui résout le problème. Cela peut également être fait avec des objets en transmettant d'autres objets comme paramètres de constructeur, et même avec des paquets dans les langages qui le permettent, tels que Ada.

La stratification explique également la différence entre l'héritage, tel que `class MyApp(MobileApp)`, et la composition, telle que `class MyApp: def __init__(self, app: MobileApp): ...`. Le premier nécessite que `MyApp` expose toute l'interface de `MobileApp` en plus de la sienne, tandis que le second permet à `MyApp` de choisir ce qu'il souhaite exposer et d'utiliser éventuellement `MobileApp` pour implémenter son interface :

Dans la plupart des cas, la composition est le choix approprié pour éviter d'exposer des détails non pertinents aux couches de niveau supérieur. Cependant, l'héritage peut être utile si les modules se trouvent logiquement dans la même couche, comme `LaserPrinter` et `InkjetPrinter` qui héritent tous deux de `Printer`.

Niveaux d'abstraction

Un câble à fibre optique offre un niveau d'abstraction très bas, qui traite la lumière pour transmettre des bits individuels. Le protocole Ethernet offre un niveau d'abstraction plus élevé, qui traite les adresses MAC et transmet des octets. Une application mobile offre un niveau d'abstraction encore plus élevé, qui traite les requêtes et les réponses pour transmettre des informations telles que les menus quotidiens des cafétérias.

Si vous deviez mettre en oeuvre une application mobile et que vous ne disposiez que d'un câble à fibre optique, vous passeriez la plupart de votre temps à réimplémenter des abstractions intermédiaires, car il est difficile de définir une demande pour le menu du jour en termes de bits individuels.

Mais si vous deviez mettre en oeuvre une extension de câble à fibre optique et que vous ne disposiez que de l'abstraction de haut niveau des menus quotidiens de la cafétéria, vous ne seriez pas en mesure de faire votre travail. L'abstraction de haut niveau est pratique pour les opérations de haut niveau, mais elle cache volontairement les détails de bas niveau.

Lorsque vous concevez un module, réfléchissez à son niveau d'abstraction : où se situe-t-il dans le spectre des abstractions de bas niveau à haut niveau ? Si vous fournissez une abstraction d'un niveau supérieur à celui qui est nécessaire, les autres ne pourront pas faire leur travail car ils ne pourront pas accéder aux informations de bas niveau dont ils ont besoin. Si vous fournissez une abstraction d'un niveau inférieur à celui qui est nécessaire, les autres devront passer beaucoup de temps à réinventer la roue de haut niveau par-dessus votre abstraction de bas niveau.

Vous n'êtes pas toujours obligé de choisir un seul niveau d'abstraction : vous pouvez en exposer plusieurs. Par exemple, une bibliothèque peut exposer un module pour transmettre des bits sur fibre optique, un module pour transmettre des paquets Ethernet et un module pour effectuer des requêtes de haut niveau. En interne, le module de requête de haut niveau peut utiliser le module Ethernet, qui peut utiliser le module de fibre optique. Ou pas ; vos clients n'ont aucun moyen de le savoir, et ils n'ont aucune raison de s'en soucier, tant que vos modules fournissent des implémentations fonctionnelles des abstractions qu'ils exposent.

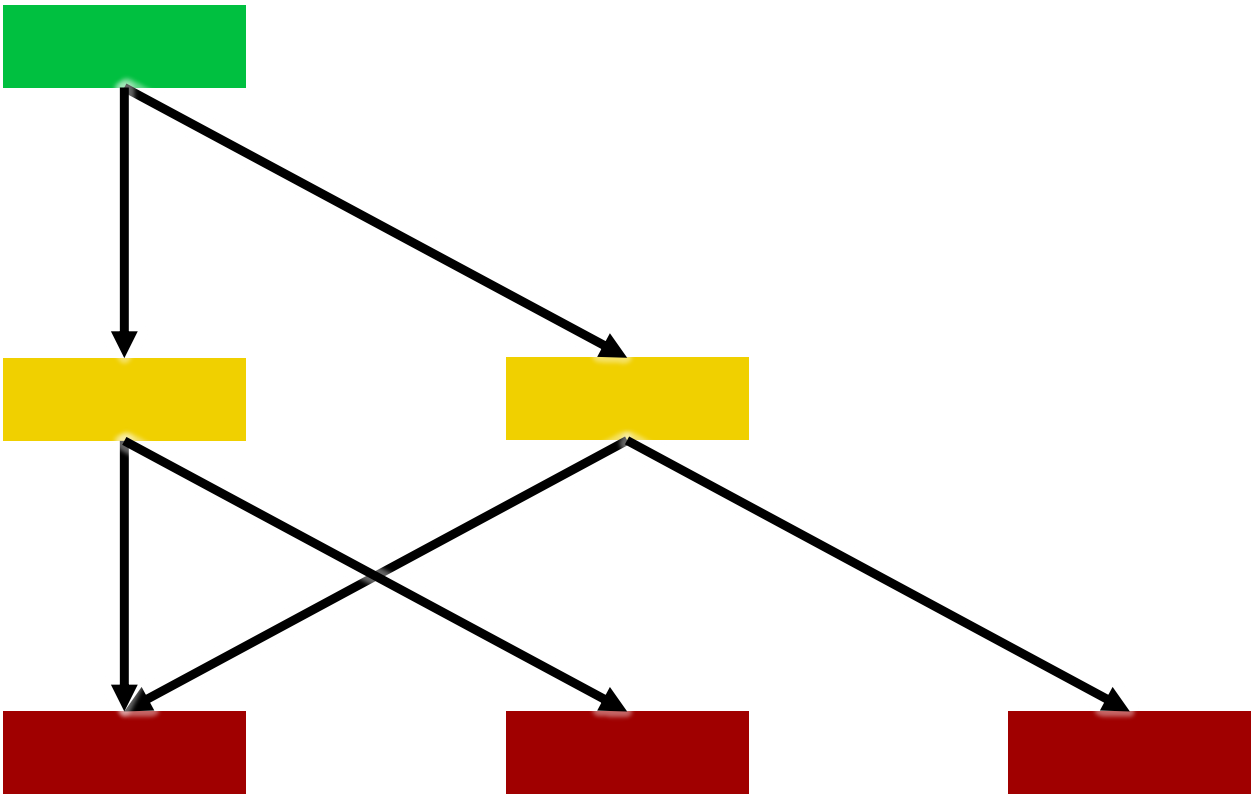
Un exemple concret de niveaux d'abstraction différents est l'affichage d'un triangle à l'aide d'un GPU, qui est l'équivalent graphique de l'impression du texte ”Hello, World !”. À l'aide d'une API de haut niveau telle que GDI, l'affichage d'un triangle nécessite environ 10 lignes de code. Vous pouvez créer un objet fenêtre, créer un objet triangle, configurer les couleurs de ces objets et les afficher. À l'aide d'une API de bas niveau telle que OpenGL, l'affichage d'un triangle nécessite environ 100 lignes de code, car vous devez traiter explicitement certains détails de bas niveau. En utilisant une API de niveau encore plus bas telle que Vulkan, l'affichage d'un triangle nécessite environ 1'000 lignes de code, car vous devez gérer explicitement tous les concepts GPU de bas niveau que même OpenGL abstrait. Chaque partie du pipeline graphique doit être configurée dans Vulkan. Mais cela ne fait pas de Vulkan une ”mauvaise” API, seulement une API qui n'est pas adaptée aux tâches de haut niveau telles que l'affichage de triangles. Au contraire, Vulkan et les API similaires telles que Direct3D 12 sont destinées à être utilisées pour les moteurs de jeux et autres abstractions ”intermédiaires” qui

fournissent elles-mêmes des abstractions de plus haut niveau. Par exemple, OpenGL peut être implémenté comme une couche au-dessus de Vulkan. Sans ces abstractions de bas niveau, il serait impossible de mettre en oeuvre efficacement des abstractions de haut niveau. En effet, les performances ont été la principale motivation pour la création d'API telles que Vulkan.

Lors de la mise en oeuvre d'une abstraction par dessus une abstraction de niveau inférieur, il est important d'éviter les ”fuites d'abstraction”. Une fuite d'abstraction se produit lorsqu'un détail de bas niveau ”fuite” d'une abstraction de haut niveau, obligeant les utilisateurs de l'abstraction à comprendre et à gérer des détails de bas niveau qui ne les concernent pas. Par exemple, si la fonction permettant d'afficher le menu du jour a la signature `def showMenu(date, useIPv4)`, toute personne souhaitant écrire une application qui affiche des menus doit réfléchir explicitement à la question de savoir si elle souhaite utiliser IPv4, un détail de niveau inférieur qui ne devrait pas être pertinent dans ce contexte. Notez que le terme ”fuite d'abstraction” n'est pas lié à une ”fuite de sécurité”, bien que les deux soient des fuites.

Récapitulatif

Concevez des systèmes de manière à ce que chaque module dispose d'une API standard qui fournit une abstraction cohérente. Structurez vos modules en couches afin que chaque module ne dépende que des modules de la couche immédiatement inférieure, et que les couches soient classées par niveau d'abstraction. Par exemple, voici une conception dans laquelle le module vert fournit une abstraction de haut niveau et dépend des modules jaunes, qui fournissent des abstractions de niveau inférieur, et qui dépendent eux-mêmes des modules rouges et de leurs abstractions de niveau inférieur :



Une façon de procéder au niveau des fonctions individuelles consiste à écrire d'abord le module de haut niveau, avec une interface de haut niveau, et une implémentation qui utilise des fonctions que vous n'avez pas encore écrites. Par exemple, pour une méthode qui prédit la météo :

```
def predictWeather(date):
    past = getPastWeather(date)
    temps = extractTemperatures(past)
```

```
    return predict(temps)
```

Après cela, vous pouvez implémenter `getPastWeather` et d'autres, eux-mêmes en termes d'interfaces de niveau inférieur, jusqu'à ce que vous implémentiez vous-même le niveau le plus bas ou que vous réutilisiez le code existant pour cela. Par exemple, `getPastWeather` sera probablement implémenté avec une bibliothèque HTTP, tandis que `extractTemperature` sera défini en fonction du format de `past`.

Exercice Regardez `calc.py` dans les exercices pendant le cours. Il mélange toutes sortes de concepts. Modularisez-le ! Réfléchissez aux modules dont vous avez besoin et à la manière dont vous devriez concevoir le système dans son ensemble. Tout d'abord, à quoi ressemblera le code de haut niveau ?

Créez une fonction pour obtenir les entrées de l'utilisateur, une pour les analyser en un format interne, une pour évaluer ce format et une pour afficher le résultat ou l'absence de résultat. La fonction d'évaluation peut utiliser en interne une autre fonction pour exécuter chaque opérateur individuellement, de sorte que tous les opérateurs se trouvent au même endroit et soient indépendants de l'analyse des entrées. Vous pouvez donc avoir la structure suivante :

```
graph TD
    A[main]
    B[getInput]
    C[parseInput]
    D[compute]
    E[execute]
    F[display]
    A --> B
    A --> C
    A --> D
    A --> F
    D --> E
```

À ce stade, vous vous demandez peut-être jusqu'où aller dans la modularisation. Vos programmes doivent-ils être composés de milliers de petits modules empilés sur des centaines de couches ? Probablement pas, car cela poserait des problèmes de maintenabilité, tout comme le fait d'avoir un seul grand module pour tout. Mais où s'arrêter ?

Il n'existe pas de mesure objective unique pour déterminer la taille d'un module, mais voici quelques règles empiriques. Vous pouvez estimer la taille à l'aide du nombre de chemins logiques dans un module. Combien de choses différentes le module peut-il faire ? Si vous obtenez plus d'une douzaine, le module est probablement trop grand. Vous pouvez estimer la complexité à l'aide du nombre d'entrées pour un module. De combien d'éléments le module a-t-il besoin pour fonctionner ? Si ce nombre est supérieur à quatre ou cinq, le module est probablement trop grand.

Souvenez-vous de l'acronyme YAGNI, qui signifie "You Aren't Gonna Need It" : vous n'en aurez pas besoin). Vous pourriez diviser un module en trois parties encore plus petites qui pourraient théoriquement être réutilisées individuellement, mais en aurez-vous besoin ? Non ? Vous n'en aurez pas besoin, alors ne le faites pas. Vous pourriez fournir dix paramètres différents pour un module afin de configurer chaque détail de ce qu'il fait, mais en aurez-vous besoin ? Non ? Vous n'en aurez pas besoin, alors ne le faites pas.

Une façon de discuter des conceptions avec vos collègues consiste à utiliser des diagrammes tels que les diagrammes de classes UML, dans lesquels vous dessinez des modules avec leurs données et leurs opérations et vous les reliez pour indiquer les relations de composition et d'héritage. Gardez à l'esprit que l'objectif est de discuter de la conception du système, et non de respecter des conventions spécifiques. Tant que tout le monde s'accorde sur la signification de chaque élément du diagramme, le fait de respecter ou non une convention spécifique telle que l'UML n'a aucune importance.

Méfiez-vous du phénomène connu sous le nom de ”cargo cult programming”. L'idée d'un ”culte du cargo” trouve son origine dans des îles isolées utilisées comme bases par les soldats américains pendant les guerres. Ces îles abritaient des populations indigènes qui n'avaient aucune idée de ce qu'étaient les avions, mais qui avaient remarqué que lorsque les soldats faisaient des gestes spécifiques impliquant du matériel militaire, des avions-cargos remplis de provisions atterrissaient sur les îles. Ils ont naturellement émis l'hypothèse que s'ils pouvaient reproduire ces mêmes gestes, d'autres avions atterriraient ! Bien sûr, de notre point de vue, nous savons que cela était inutile, car ils avaient inversé la corrélation : les soldats faisaient des gestes d'atterrissage parce qu'ils savaient que des avions allaient arriver, et non l'inverse. Mais les autochtones ne le savaient pas et essayaient de faire atterrir des avions-cargos. Certaines de ces croyances ont perduré plus longtemps qu'elles n'auraient dû, et leur équivalent moderne en programmation sont les ingénieurs qui conçoivent leur système ”parce que certaines grandes entreprises, comme Google ou Microsoft, le font de cette façon”, sans savoir ni comprendre pourquoi ces grandes entreprises le font ainsi. En général, les grands systèmes des grandes entreprises ont des contraintes qui ne s'appliquent pas à la grande majorité des systèmes, comme le traitement de milliers de requêtes par seconde ou la nécessité de fournir des garanties de disponibilité extrêmes.

Comment atténuer l'impact des défauts ?

Que doit-il se passer lorsqu'une partie d'un système rencontre un problème ?

Margaret Hamilton, qui a écrit avec son équipe le logiciel permettant de mettre des vaisseaux spatiaux en orbite et d'envoyer des hommes sur la Lune, a raconté dans une conférence comment elle a tenté de persuader les responsables d'ajouter une fonctionnalité de sécurité à un vaisseau spatial. Un jour, elle avait emmené sa fille au travail, et celle-ci avait essayé le simulateur de vaisseau spatial. À sa surprise, sa fille avait réussi à faire crasher le logiciel fonctionnant dans le simulateur. Le logiciel n'était pas capable de supporter le lancement d'une opération alors que le vaisseau spatial était censé se trouver dans une phase de vol complètement différente. Hamilton a tenté de convaincre ses responsables que le logiciel devait être capable de supporter de telles erreurs, mais comme elle le rappelle : ”(les responsables) ont répondu : 'Cela n'arrivera jamais, les astronautes sont bien entraînés, ils ne font pas d'erreurs'... lors de la mission suivante, Apollo 8, c'est exactement ce qui s'est produit... il a fallu des heures pour récupérer (les données)”

L'absence de vérification de cette condition était une erreur, c'est-à-dire que l'équipe qui a programmé le logiciel a choisi de ne pas tenir compte d'un problème qui pouvait se produire dans la pratique. D'autres types d'erreurs consistent à oublier de traiter un cas de défaillance ou à écrire un code qui ne fait pas ce que le programmeur pense qu'il fait.

Les erreurs provoquent des défauts dans le système, qui peuvent être déclenchés par des entrées externes, comme un astronaute appuyant sur le mauvais bouton. Si les défauts ne sont pas traités, ils provoquent des échec, ce que nous voulons éviter.

Les erreurs sont inévitables dans tout système de grande envergure, car les systèmes impliquent des humains et les humains sont faillibles. ”Ne faites pas d'erreurs” n'est pas une solution réaliste. Il est même difficile d'envisager tous les cas de défaillance possibles ; pensez au ”Chat provoquant le blocage de l'écran de connexion” dans Ubuntu. Qui aurait pu imaginer qu'un utilisateur non malveillant puisse saisir des milliers de caractères dans un champ de saisie de nom d'utilisateur ?

Pour empêcher les échecs, il faut donc empêcher les défauts de se propager dans le système, c'est-à-dire atténuer leur impact. Nous verrons quatre façons d'y parvenir, toutes basées sur des modules : isoler, réparer, réessayer et remplacer.

L'effort à fournir pour tolérer les défauts dépend de l'enjeu. Un petit script que vous avez écrit vous-même pour récupérer des dessins animés doit être tolérant aux erreurs réseau temporaires, mais ne nécessite pas de techniques de récupération avancées. En revanche, la barrière sur la Tamise qui empêche les inondations massives doit être résistante à de nombreux défauts possibles.

Isoler

Au lieu de faire planter l'ensemble d'un logiciel, il est préférable d'isoler le défaut et de ne faire planter qu'un seul module, aussi petit et proche que possible de la source du défaut. Par exemple, les navigateurs Web modernes isolent chaque onglet dans son propre module, et si le site Web à l'intérieur de l'onglet pose un problème, seul cet onglet doit planter, et non l'ensemble du navigateur. De même, les systèmes d'exploitation isolent chaque programme de manière à ce que seul le programme plante s'il présente un défaut, et non l'ensemble du système d'exploitation.

Cependant, l'isolation ne doit être effectuée que si le reste du programme peut fonctionner raisonnablement sans le module défaillant. Par exemple, si le module responsable du dessin de l'interface globale du navigateur plante, le reste du navigateur ne peut pas fonctionner. En revanche, le plantage d'un seul onglet du navigateur est acceptable, car l'utilisateur peut toujours utiliser les autres onglets.

Réparer

Il arrive parfois qu'un module passe dans un état inattendu en raison d'un défaut. Il est alors possible de le ”réparer” en le ramenant à un état connu. Cela ne signifie pas le faire passer de l'état inattendu à un autre état quelconque, car le module ne sait même pas où il se trouve, mais remplacer l'intégralité de l'état du module par un état ”de secours” spécifique dont on sait qu'il fonctionne. Un exemple intéressant de cela est la salle ”secrète” dans le jeu vidéo *The Legend of Zelda: A Link to the Past*. Si le joueur parvient à mettre le jeu dans un état inconnu, par exemple en passant trop rapidement d'une zone à l'autre de la carte pour que le jeu puisse suivre, le jeu reconnaît qu'il est confus et place le joueur dans une pièce spéciale, en faisant comme si c'était intentionnel et que le joueur avait trouvé une zone secrète.

Une forme grossière de réparation consiste à ”l'éteindre et le rallumer”, par exemple en redémarrant un programme ou en redémarrant le système d'exploitation. L'état immédiatement après le démarrage est connu pour fonctionner, mais cela ne peut être caché aux utilisateurs et constitue plutôt un moyen de contourner un défaut.

Cependant, ne procédez à une réparation que si l'état d'un module peut être entièrement restauré à un état connu pour fonctionner. Réparer seulement une partie d'un module risque de créer un monstre de Frankenstein qui ne fera qu'aggraver le problème.

Réessayer

Tous les défauts ne sont pas permanentes. Certains défauts proviennent de causes externes qui peuvent se résoudre sans votre intervention, et il est donc souvent judicieux de réessayer. Par exemple, si la connexion Internet d'un utilisateur tombe en panne, toute requête Web effectuée par votre application échouera. Mais il est probable que la connexion soit rapidement rétablie, par exemple parce que l'utilisateur se trouvait temporairement dans un endroit où la connectivité mobile était faible, comme un tunnel. Ainsi, réessayer plusieurs fois avant d'abandonner évite d'afficher des échecs inutiles à l'utilisateur. Le nombre de tentatives et le délai d'attente avant de réessayer dépendent de vous, du système et du contexte.

Cependant, ne réessayez que si la requête est *idempotente*, c'est-à-dire si le fait de la répéter plusieurs fois a le même effet que de la faire une seule fois. Par exemple, retirer de l'argent d'un compte bancaire n'est pas une requête idempotente. Si vous réessayez parce que vous n'avez pas obtenu de réponse, mais que la requête a en fait atteint le serveur, l'argent sera retiré deux fois.

Vous ne devez également réessayer que lorsque vous rencontrez des problèmes qui sont récupérables, c'est-à-dire pour lesquels réessayer a une chance de réussir, car ils proviennent de circonstances indépendantes de votre volonté qui pourraient se résoudre d'elles-mêmes. Par exemple, ”pas d'Internet” est récupérable, tout comme ”l'imprimante démarre et n'est pas encore prête”. En revanche, des problèmes tels que ”le nom d'utilisateur souhaité est déjà pris” ou ”le code contient un bug qui divise par zéro” ne sont pas récupérables, car réessayer mènera au même problème.

Remplacer

Il existe parfois plusieurs façons d'effectuer une tâche, et certaines d'entre elles peuvent servir de rechange, en remplaçant le module principal en cas de problème. Par exemple, si un lecteur d'empreintes digitales ne parvient pas à reconnaître le doigt d'un utilisateur parce que celui-ci est trop humide, un système d'authentification pourrait demander un mot de passe à la place.

Cependant, ne remplacez que si vous disposez d'une alternative aussi robuste et testée que l'originale. Le module "de rechange" ne doit pas être un ancien code qui n'a pas été exécuté depuis des années, mais doit être traité avec le même soin et le même niveau de qualité que le module principal.

Résumé

Dans ce cours, vous avez appris :

- La programmation orientée objet : classes, méthodes, héritage vs composition, co/contra/invariance
- L'abstraction et la modularité : régularité, groupes, couches, niveaux, "fuites" de niveau
- Atténuer l'impact des défauts : isoler, réparer, réessayer, remplacer

Vous pouvez maintenant consulter les exercices !

Tests

Il est tentant de penser que les tests ne sont pas nécessaires si l'on écrit "simplement" du code correct dès le premier essai et si l'on relit son code avant de l'exécuter.

Mais dans la pratique, cela ne fonctionne pas. Les êtres humains commettent des erreurs en permanence. Même Ada Lovelace, qui a écrit un algorithme correct pour calculer les nombres de Bernoulli. pour le "Moteur analytique" de Charles Babbage, a commis une faute de frappe en intervertissant deux variables dans la transcription du code de son algorithme. Et elle a eu tout le temps de le vérifier, puisque le moteur analytique était une proposition de Babbage qui n'a pas été construite ! Le "tout premier programme" contenait déjà une coquille.

La vérification assistée par ordinateur est une option moderne, mais elle nécessite beaucoup de temps. Si Ada Lovelace avait vécu au XXIe siècle, elle aurait pu écrire une preuve et demander à un ordinateur de la vérifier, en s'assurant que la preuve est correcte tant que le programme de vérification est lui-même correct. Cela peut fonctionner dans la pratique, mais actuellement au prix d'un effort important de la part des développeurs. Le noyau du système d'exploitation seL4, par exemple, a nécessité 200 000 lignes de preuve pour ses 10 000 lignes de code (Klein et al., "seL4 : formal verification of an OS kernel"). Une telle méthode aurait pu fonctionner pour Ada Lovelace, une aristocrate disposant de beaucoup de temps libre, mais elle n'est pas réaliste pour les programmeurs de tous les jours.

Une autre option moderne consiste à laisser les utilisateurs faire le travail, dans le cadre d'une version "bêta" ou d'un "accès anticipé". Les utilisateurs ont la possibilité d'utiliser un programme avant tout le monde, au prix de rencontrer des bugs et de les signaler, ce qui fait d'eux des testeurs. Cependant, cela ne fonctionne que si le programme est suffisamment intéressant, comme un jeu, alors que la plupart des programmes existants sont conçus comme des outils internes pour un public restreint qui ne voudra probablement pas faire de bêta-tests. En outre, elle n'élimine pas complètement les bugs. Le jeu "New World" d'Amazon, malgré une période de "bêta ouverte", publié avec de nombreux problèmes, notamment un délai de 7 jours avant de réapparaître en cas de mort.

Avons-nous vraiment besoin de tests ? Quelle est la pire conséquence d'un bug ? Dans certains cas, le pire n'est pas si grave, comme par exemple un bug dans un jeu en ligne. Mais imaginez un bug dans le système d'inscription aux cours d'une université, laissant les étudiants incertains de s'ils sont inscrits à un cours ou non. Pire, un bug dans une banque peut faire apparaître ou disparaître de l'argent. Pire encore, les bugs peuvent être mortels, comme dans le cas de la machine de radiothérapie Therac-25 qui a tué certains patients.

Objectifs

Après ce cours, vous devriez être en mesure de :

- Comprendre les bases des *tests automatisés*
- Évaluer les tests avec la *couverture du code*
- Identifier *quand* écrire quels tests
- Adapter le code pour permettre des tests précis

Qu'est-ce qu'un test ?

Les tests se déroulent essentiellement en trois étapes :

1. Mettre en place le système
2. Effectuer une action
3. Vérifier le résultat

Si le résultat est celui que nous attendons, nous sommes convaincus que le système fait ce qu'il faut dans ce cas. Cependant, la ”confiance” n'est pas une garantie. Comme l'a dit Edsger W. Dijkstra, ”Les tests peuvent être utilisés pour démontrer la présence de bugs, mais jamais pour démontrer leur absence”.

La manière la plus simple de tester est le test manuel. Un humain exécute manuellement le processus ci-dessus. Cela a l'avantage d'être facile, puisqu'il suffit d'effectuer les actions que l'on attendrait de toute façon des utilisateurs. Elle permet également un certain degré de subjectivité : le résultat doit ”avoir l'air correct”, mais il n'est pas nécessaire de définir formellement ce que signifie ”correct”.

Cependant, les tests manuels présentent de nombreux inconvénients. Ils sont lents : imaginez que vous exécutiez manuellement une centaine de tests. Ils sont également sujets aux erreurs : à chaque essai, les chances qu'un être humain oublie d'exécuter une étape exécute une étape incorrectement ou ne remarque pas que quelque chose ne va pas augmentent. La subjectivité des tests manuels est aussi souvent un inconvénient : deux personnes peuvent ne pas être d'accord sur ce qui est ”bon” ou ”mauvais” pour un test donné. Enfin, il est également difficile de tester les cas limites. Imaginez que vous testiez qu'une application météo affiche correctement les chutes de neige lorsqu'elles se produisent alors qu'il fait actuellement beau chez vous.

Pour éviter les problèmes liés aux tests manuels, nous nous concentrerons sur les tests automatisés. Le workflow fondamentalement le même, mais il est automatisé :

1. Le code met en place le système
2. Le code effectue une action
3. Le code vérifie le résultat

Ces étapes sont souvent appelées ”arrange”, ”act”, et ”assert” en anglais.

Les tests automatisés peuvent être exécutés rapidement, car les ordinateurs sont beaucoup plus rapides que les humains et n'oublient pas les étapes et n'exécutent pas le code incorrectement. Cela ne signifie pas que les tests automatisés sont toujours corrects : si le code décrivant le test est erroné, le résultat du test n'a pas de sens. Les tests automatisés sont également plus objectifs : la personne qui rédige le test sait exactement ce qui sera testé. Enfin, les tests automatisés permettent de tester les cas limites en simulant l'environnement du système testé, par exemple le serveur de prévisions météorologiques d'une application météo.

Les tests automatisés présentent également d'autres avantages : les tests peuvent être écrits une fois et utilisés pour toujours, partout, même sur des implémentations différentes. Par exemple, la spécification CommonMark pour les outils utilisant le format Markdown comprend de nombreux exemples utilisés comme tests, ce qui permet à chacun d'utiliser ces tests pour vérifier son propre outil. Si quelqu'un remarque un bug qui n'a pas été couvert par les tests standard, il peut suggérer un test qui couvre ce bug pour la prochaine version de la spécification. Ce test peut ensuite être utilisé par tous les autres. Le nombre de tests ne cesse de croître avec le temps et peut atteindre des quantités énormes, comme la suite de tests SQLite, qui compte actuellement plus de 90 millions de lignes de tests.

D'un autre côté, les tests automatisés sont plus difficiles que les tests manuels car il faut passer du temps à écrire le code de test, ce qui inclut une définition formelle du "bon" comportement.

Comment écrire des tests automatisés ?

Nous utiliserons principalement Python comme exemple, mais les tests automatisés fonctionnent de la même manière dans la plupart des langages.

L'idée principale est que chaque test est une méthode, et qu'un échec du test est indiqué par le fait que la méthode lance une exception. Si la méthode ne lance pas d'exception, le test est réussi.

Une façon de le faire en utilisant Python est la suivante :

```
def test_1plus1():
  assert add(1, 1) == 2
```

Si `add(1, 1)` retourne `2`, alors l'assertion ne fait rien, la méthode se termine, et le test est considéré comme réussi. Mais s'il renvoie un autre nombre, l'assertion lève une `AssertionError`, qui est une sorte d'exception, et le test est considéré comme échoué.

...ou, du moins, c'est ainsi que cela devrait être, mais les assertions Python peuvent être désactivées, malheureusement. Cette méthode ne fait donc pas forcément ce que l'on veut.

On pourrait imiter la déclaration `assert` avec un `if` et un `raise` :

```
def test_1plus1():
  if add(1, 1) != 2:
    raise AssertionError()
```

Il s'agit d'une implémentation pratique d'un test, que nous pourrions exécuter. Cependant, si le test échoue, il n'y a pas de message d'erreur, puisque nous n'en avons pas mis lors de la création de `AssertionError`. Par exemple, si le test échoue, qu'est-ce que `add(1, 1)` a réellement retourné ? Il serait bon de le savoir. Nous pourrions écrire du code pour stocker le résultat dans une variable, tester cette variable, puis créer un message pour l'exception incluant cette variable. Ou nous pouvons utiliser le module `unittest` de Python pour le faire à notre place :

```
import unittest
class TestsAdd(unittest.TestCase):
  def test_1plus1(self):
    self.assertEqual(add(1, 1), 2)
```

`unittest` trouve toutes les classes héritant de `unittest.TestCase` et les exécute, nous libérant ainsi de la nécessité d'écrire le code pour le faire nous-mêmes, et lance des exceptions dont le message comprend la valeur "attendue" et la valeur "obtenue".

Tip

Vous pouvez utiliser Hamcrest pour écrire des assertions en plus de `unittest` afin de les rendre plus lisibles, surtout les messages d'erreur :

```
from hamcrest import assert_that, equal_to
import unittest
class TestsAdd(unittest.TestCase):
  def test_1plus1(self):
    assert_that(add(1, 1), equal_to(2))
```

La partie `equal_to` est un "matcher" Hamcrest, qui décrit la valeur attendue. `equal_to` est le plus simple, il correspond à une seule valeur, mais nous pouvons en utiliser de plus sophistiqués :

```
values = [...]
assert_that(values, any_of(empty(), contains(1, 2)))
```

Si cette assertion échoue, le message d'exception de Hamcrest indique "Expected : (an empty collection or a sequence containing [<1>, <2>]) but : was <[42]>".

Parfois, nous devons tester qu'un morceau de code "échoue" dans certaines circonstances, par exemple en validant correctement les arguments et en levant une exception si un argument a une valeur invalide. C'est à cela que sert `assertRaises` :

```
class TestRaises(unittest.TestCase):
  def test_raises(self):
    with self.assertRaises(ZeroDivisionError):
        y = 1 / 0
```

Si la fonction ne lève pas d'exception, ou lève une exception d'un autre type, le test échoue.

Pour voir plus de méthodes de test, consultez la documentation de `unittest`.

Exercice C'est à vous de jouer ! Ouvrez le dossier d'exercices pendant le cours et testez `fonctions.py`. Commencez par tester des valeurs valides pour `fibonacci`, puis testez qu'il rejette les valeurs invalides.

Vous pouvez tester `fibonacci` pour des nombres tels que 1 et 10, et tester qu'il lève une exception pour les nombres inférieurs à `0`.

Faut-il tester beaucoup de choses avec une seule méthode ou avoir plusieurs petites méthodes de test ? Pensez à ce à quoi ressembleront les résultats des tests si vous combinez plusieurs tests dans une seule méthode. Si la méthode de test échoue, vous n'obtiendrez qu'un message d'exception concernant le premier échec de la méthode, et vous ne saurez pas si le reste de la méthode de test réussira. Le fait d'avoir de grandes méthodes de test signifie également que la fraction de tests réussis est moins représentative de l'exactitude globale du code. À l'extrême, si vous écrivez toutes les assertions dans une seule méthode, un seul bug dans votre code entraînerait 0 % de réussite des tests. Vous devriez donc préférer les petites méthodes de test qui testent chacune un concept "logique", qui peut nécessiter une ou plusieurs assertions. Cela ne signifie pas qu'il faille copier-coller de gros blocs de code entre les tests ; il faut plutôt partager le code en utilisant des fonctions que `unittest` appelle si elles sont présentes :

```
@classmethod def setUpClass(cls): ...
@classmethod def tearDownClass(cls): ...
def setUp(self): ...
def tearDown(self): ...
```

Comment tester les méthodes privées ? On ne teste *jamais* les méthodes privées ! Sinon, les tests doivent être réécrits à chaque fois que l'implémentation change. Revenons à l'exemple de SQLite : le code serait impossible à modifier si un changement dans les détails de l'implémentation nécessitait de modifier ne serait-ce qu'une fraction des 90 millions de lignes de tests.

Quelles sont les normes à respecter pour le code de test ? Les mêmes que pour le reste du code. Le code de test doit se trouver dans le même dépôt de gestion de version que le reste du code, et doit être examiné au même titre que le reste du code lorsque des modifications sont apportées. Cela signifie également que les tests doivent avoir des noms appropriés, pas `test_1` ou `test_feature_works` mais des noms spécifiques qui donnent des informations dans une vue d'ensemble des tests tels que `test_name_can_include_thai_characters`. Évitez les noms qui utilisent des descriptions vagues telles que "correctement", "fonctionne" ou "valide".

Quelle mesure peut-on utiliser pour évaluer les tests ?

Qu'est-ce qui fait un bon test ? Lors de l'examen d'une modification de code, comment savoir si les tests existants sont suffisants ou s'il faut en ajouter ou en réduire le nombre ? Lors de l'examen d'un test, comment savoir s'il est utile ?

Il existe de nombreuses façons d'évaluer les tests ; nous nous concentrerons ici sur la plus courante, la *couverture* (”coverage” en anglais). La couverture des tests est définie comme la fraction de code exécutée par les tests par rapport à la quantité totale de code. Sans tests, elle est de 0%. Avec des tests qui exécutent chaque partie du code au moins une fois, elle est de 100 %. Mais qu'est-ce qu'une ”partie du code” ? Quelle devrait être la mesure exacte de la couverture ?

Une façon naïve de le faire est la couverture de *lignes*. Prenons l'exemple suivant :

```
int getFee(Package pkg) {
  if (pkg == null) throw ...;
  int fee = 10;
  if (pkg.isHeavy()) fee += 10;
  if (pkg.isInternational()) fee *= 2;
  return fee;
}
```

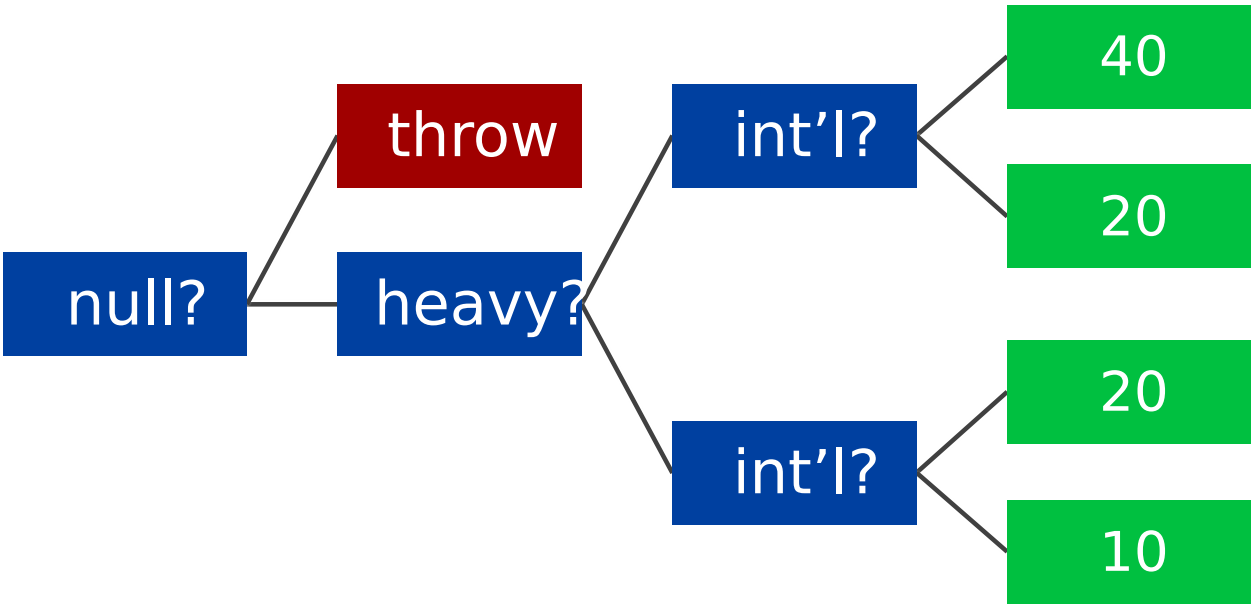
Un seul test avec un paquet non nul qui est à la fois lourd et international couvrira toutes les lignes. Cela peut sembler une bonne chose puisque la couverture est à 100 % et facile à obtenir, mais ce n'est pas le cas. Si le `throw` était sur une ligne différente au lieu d'être sur la même ligne que le `if`, la couverture de la ligne ne serait plus de 100%. Il n'est pas judicieux de définir une mesure de couverture qui dépende du formatage du code.

Au lieu de cela, la mesure la plus simple pour la couverture des tests est la couverture des *instructions*. Dans notre exemple, l'instruction `throw` n'est pas couverte mais toutes les autres le sont, et cela ne change pas en fonction du formatage du code. Il n'en reste pas moins que le fait d'atteindre une couverture de près de 100 % sur la base d'un seul test pour le code ci-dessus semble erroné. Il y a trois instructions ”if”, indiquant que le code effectue différentes actions en fonction d'une condition, mais nous avons ignoré les blocs ”else” implicites dans ces ”ifs”.

Une forme plus avancée de couverture est la couverture de *branches* : la fraction des choix de branche qui sont couverts. Pour chaque branche, telle qu'une instruction ”if”, une couverture à 100 % de la branche implique de couvrir les deux choix. Dans le code ci-dessus, la couverture de branche pour notre seul exemple de test est de 50% : nous avons couvert exactement la moitié des choix.

Il est possible d'atteindre 100 % en effectuant deux tests supplémentaires : un colis nul et un colis qui n'est ni lourd ni international.

Mais prenons un peu de recul et réfléchissons à ce que notre exemple de code peut faire :



Le code comporte cinq chemins, dont un qui échoue. Pourtant, avec la couverture des branches, nous pourrions crier victoire après seulement trois tests, laissant deux chemins inexplorés. C'est là qu'intervient la couverture des *chemins*. La couverture des chemins est la forme la plus avancée de la couverture, qui compte la fraction des chemins exécutés dans le code. Nos trois tests couvrent 60% des chemins, soit 3 sur 5. Nous pouvons atteindre 100 % en ajoutant des tests pour les deux chemins non couverts : un paquet lourd mais non international et l'inverse.

La couverture des chemins semble très intéressante en théorie. Mais dans la pratique, cette solution est souvent irréalisable, comme le montre l'exemple suivant :

```
while (true) {
  var input = getUserInput();
  if (input.length() <= 10) break;
  tellUser("Pas plus de 10 caractères");
}
```

La couverture maximale des chemins que l'on peut obtenir pour ce code est *zéro*. En effet, il existe un nombre infini de chemins : la boucle peut s'exécuter une fois, ou deux fois, ou trois fois, et ainsi de suite. Comme on ne peut écrire qu'un nombre fini de tests, la couverture des chemins est bloquée à 0%.

Même en l'absence de boucles infinies, la couverture des chemins est difficile à obtenir dans la pratique. Avec seulement 5 instructions `if` indépendantes, on doit écrire 32 tests. Si 10% des lignes de code sont des instructions "if", un programme de 5 millions de lignes comporte plus de chemins qu'il n'y a d'atomes dans l'univers. Et 5 millions de lignes, c'est bien moins que ce que certains programmes ont en pratique, comme les navigateurs Web.

Il y a donc un compromis entre la faisabilité et la confiance dans la couverture. La couverture des instructions est généralement facile à obtenir mais ne donne pas beaucoup de confiance, alors que la couverture des chemins peut être impossible à obtenir dans la pratique mais donne beaucoup de confiance. La couverture des branches est une solution intermédiaire.

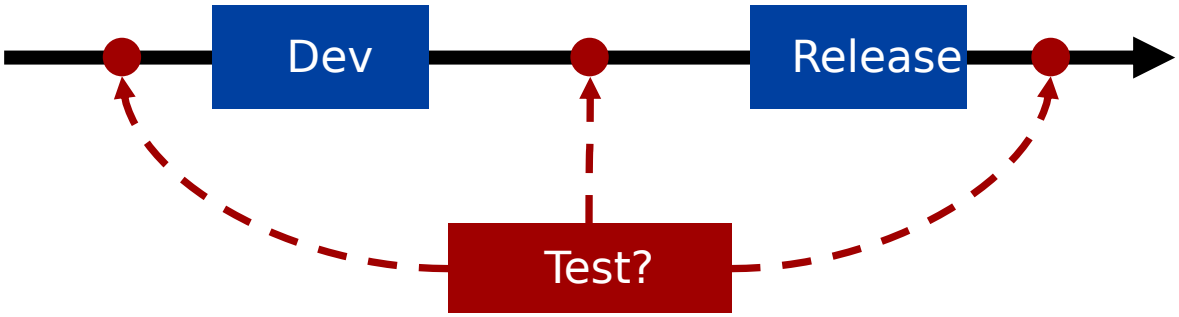
Il est important de noter que la couverture n'est pas tout. Nous pourrions couvrir 100 % des chemins de notre fonction `getFee` ci-dessus avec 5 tests, mais si ces 5 tests ne vérifient pas réellement la valeur renvoyée par la fonction, ils ne sont pas utiles. La couverture est une mesure qui devrait vous aider à décider si des tests supplémentaires seraient utiles, mais elle ne remplace pas l'examen humain.

Exercice Exécutez vos tests de l'exercice précédent avec la couverture. Vous pouvez le faire soit à partir de la ligne de commande, soit à partir de votre IDE préféré, qui devrait avoir une commande ”exécuter les tests avec couverture” à côté de ”exécuter les tests”.

Quand tester ?

Jusqu'à présent, nous avons supposé que les tests étaient écrits après le développement, avant que le code ne soit publié. C'est pratique, car le code testé existe déjà. Mais elle présente le risque de reproduire les erreurs trouvées dans le code : si un ingénieur n'a pas pensé à un cas limite lors de l'écriture du code, il est peu probable qu'il y pense en écrivant les tests immédiatement après. Il est également trop tard pour corriger la conception : si un cas de test révèle que le code ne fonctionne pas parce que sa conception nécessite des modifications fondamentales, cela devra probablement être fait rapidement sous pression, ce qui conduira à une conception sous-optimale.

Si nous simplifions le cycle de vie d'un produit en le réduisant à son développement et à sa mise sur le marché, il y a trois moments où nous pouvons effectuer des tests :



Celui du milieu est celui que nous avons déjà vu. Les deux autres peuvent sembler étranges à première vue, mais ils ont de bonnes raisons d'exister.

Les tests avant le développement sont communément appelés **développement piloté par les tests**, ”test-driven development” en anglais ou *TDD* en abrégé, parce que les tests ”pilotent” le développement, en particulier la conception du code. Dans la méthode TDD, on écrit d'abord les tests, puis le code. Après avoir écrit le code, on peut exécuter les tests et corriger les bugs. Cela oblige les programmeurs à réfléchir avant de coder, au lieu d'écrire la première chose qui leur vient à l'esprit. Il fournit un retour d'information instantané pendant l'écriture du code, ce qui peut être très gratifiant : écrivez une partie du code, exécutez les tests, et certains tests réussissent maintenant ! Cela donne une sorte d'indication de l'état d'avancement. Il n'est pas non plus trop tard pour corriger la conception, puisque celle-ci n'existe pas encore.

Le principal inconvénient du TDD est qu'il nécessite un investissement en temps plus important et qu'il peut même conduire à des dépassements de délais. En effet, le code testé doit être écrit indépendamment des tests écrits. Si l'on passe trop de temps à écrire des tests, il ne restera plus assez de temps pour écrire le code. Lorsque les tests sont effectués après le développement, ce n'est pas un problème car il est toujours possible d'arrêter d'écrire des tests à tout moment, puisque le code existe déjà, au prix d'un nombre réduit de tests et donc d'une confiance moindre dans le code. Un autre inconvénient du TDD est que la conception doit être connue à l'avance, ce qui est bien pour développer un module en fonction des exigences du client, mais pas pour prototyper, par exemple, un code de recherche. Il ne sert à rien d'écrire une suite de tests complète pour un programme si l'objectif même de ce programme change le lendemain après réflexion.

Voyons maintenant un exemple de TDD étape par étape. Vous êtes ingénieur logiciel et vous développez une application pour une banque. Votre première tâche consiste à mettre en oeuvre le retrait d'argent d'un compte. La banque vous dit que ”les utilisateurs peuvent retirer de l'argent de leur compte bancaire”. Il vous reste donc une question à poser à la banque : ”Un compte bancaire peut-il avoir un solde inférieur à zéro ?”. La banque répond ”non”, ce n'est pas possible.

Vous commencez par écrire un test :

```
@Test void canWithdrawNothing() {
  var account = new Account(100);
  assertThat(account.withdraw(0), is(0));
}
```

Le constructeur `new Account` et la méthode `withdraw` n'existent pas, vous créez donc un code ”squelette” qui est juste suffisant pour que les tests *compilent*, mais pas encore pour qu'ils passent :

```
class Account {
  Account(int balance) { }
  int withdraw(int amount) { throw new UnsupportedOperationException("TODO"); }
}
```

Vous pouvez maintenant ajouter un autre test pour la question ”solde inférieur à zéro” que vous aviez :

```
@Test void noInitWithBalanceBelow0() {
  assertThrows(IllegalArgumentException.class, () -> new Account(-1));
}
```

Ce test ne nécessite pas d'autres méthodes dans `Account`, donc vous continuez avec un autre test :

```
@Test void canWithdrawLessThanBalance() {
  var account = new Account(100);
  assertThat(account.withdraw(10), is(10));
  assertThat(account.balance(), is(90));
}
```

Cette fois, vous devez ajouter une méthode `balance` à `Account`, avec le même contenu temporaire que `withdraw`. Encore une fois, le but est de faire compiler les tests, pas de les faire passer. Vous ajoutez ensuite un dernier test pour les retraits partiels :

```
@Test void partialWithdrawIfLowBalance() {
  var account = new Account(10);
  assertThat(account.withdraw(20), is(10));
  assertThat(account.balance(), is(0));
}
```

Vous pouvez maintenant exécuter les tests... et constater qu'ils échouent tous ! C'est normal, puisque vous n'avez rien implémenté. Vous pouvez maintenant implémenter `Account` et exécuter les tests à chaque fois que vous faites un changement jusqu'à ce qu'ils passent tous.

Enfin, vous retournez voir votre client, la banque, et lui demandez ce qu'il y a lieu de faire. Il vous donne une autre exigence qu'il avait oublié : la banque peut bloquer des comptes et le retrait d'un compte bloqué n'a pas d'effet. Vous pouvez maintenant traduire cette exigence en tests, en ajoutant du code si nécessaire pour que les tests se compilent, puis en implémentant le code. Une fois que vous aurez terminé, vous reviendrez vers le client, et ainsi de suite jusqu'à ce que votre demande réponde à toutes les exigences.

Exercice A vous de jouer ! Dans le dossier d'exercices pendant le cours vous trouverez `people_counter.py`, qui est documenté mais non implémenté. Écrivez d'abord des tests, puis implémentez le code et corrigez votre code s'il ne passe pas les tests, à la manière TDD. Il faut d'abord réfléchir aux tests à écrire, puis les écrire et enfin implémenter le code.

Vous pourriez avoir cinq tests : le compteur s'initialise à zéro, la méthode ”increment” incrémente le compteur, la méthode ”reset” met le compteur à zéro, la méthode ”increment” n'incrémente pas au-delà du maximum, et le maximum ne peut être inférieur à zéro.

Les tests effectués *après* le déploiement sont communément appelés **tests de régression**. L'objectif est de s'assurer que les anciens bugs ne réapparaissent pas.

Lorsqu'on est confronté à un bug, l'idée est d'abord d'écrire un test défaillant qui reproduit le bug, puis de corriger le bug, et enfin d'exécuter à nouveau le test pour montrer que le bug est corrigé. Il est essentiel d'exécuter le test avant de corriger le bug afin de s'assurer qu'il échoue réellement. Dans le cas contraire, le test pourrait ne pas reproduire le bug et ne passerait après la ”correction” du bug que parce qu'il passait déjà avant, ne fournissant ainsi aucune information utile.

Rappelons l'exemple de SQLite : toutes ces 90 millions de lignes de code montrent qu'une très longue liste de bugs possibles n'apparaîtra plus dans aucune version future. Cela ne signifie pas qu'il n'y a plus de bugs, mais que la plupart des bugs courants ont été supprimés, et que les autres sont très probablement des cas de figure inhabituels que personne n'a encore rencontrés.

Comment peut-on tester des modules entiers ?

Jusqu'à présent, nous avons vu des tests pour des fonctions pures, qui ne dépendent pas d'autres code. Il est utile de les tester pour s'assurer de leur exactitude, mais tous les programmes ne sont pas des fonctions pures.

Considérons la fonction suivante :

```
/** Télécharge le livre avec l'ID donné
 * et l'imprime sur la console. */
void printBook(String bookId);
```

Comment pouvons-nous tester cela ? Tout d'abord, la fonction renvoie `void`, c'est-à-dire rien, alors que pouvons-nous tester ? La documentation mentionne également le téléchargement de données, mais d'où vient cette fonctionnalité ?

Nous pourrions tester cette fonction en lui transmettant un numéro d'identification de livre que nous savons valide et en vérifiant le résultat. Cependant, ce livre pourrait un jour être retiré ou son contenu mis à jour, ce qui invaliderait notre test.

En outre, les tests qui dépendent de l'environnement, comme le dépôt de livres utilisé par cette fonction, ne peuvent pas facilement tester les cas limites. Comment tester ce qui se passe si le contenu du livre est malformé ? Ou si la connexion Internet est interrompue après le téléchargement de la table des matières mais avant le téléchargement du premier chapitre ?

On pourrait concevoir des tests *de bout en bout* pour cette fonction : exécuter la fonction dans un environnement personnalisé, tel qu'une machine virtuelle dont les demandes de réseau sont interceptées, et analyser sa sortie à partir de la console, ou éventuellement la rediriger vers un fichier. Si les tests de bout en bout sont utiles, ils nécessitent beaucoup de temps et d'efforts, et constituent une infrastructure qui doit être maintenue.

Au lieu de cela, attaquons-nous à la cause première du problème : l'entrée et la sortie de `printBook` sont *implicites*, alors qu'elles devraient être *explicites*.

Rendons d'abord l'entrée explicite en concevant une interface pour les requêtes HTTP :

```
// Note: Ceci est du Java.
// En Python, il n'y a pas besoin d'interface, on peut directement déclarer une classe qui est la "vrai

interface HttpClient {
  String get(String url);
}
```

Nous pouvons alors donner un `HttpClient` comme paramètre à `printBook`, qui l'utilisera au lieu de faire des requêtes HTTP lui-même. Cela rend l'entrée explicite, et permet également au code `printBook` de se concentrer sur la tâche qu'il est censé accomplir plutôt que sur les détails des requêtes HTTP.

Notre fonction `printBook` avec une entrée explicite ressemble donc à ceci :

```
void printBook(
  String bookId,
  HttpClient client
);
```

Ce processus consistant à rendre les dépendances explicites et à les transmettre en tant qu'entrées est appelé **injection de dépendances**.

Nous pouvons ensuite le tester avec toutes les réponses HTTP que nous voulons, y compris les exceptions, en créant un faux client HTTP pour les tests :

```
var fakeClient = new HttpClient() {
  @Override
  public String get(String url) { ... }
}
```

```
class FakeHttpClient():
  # ... même nom de méthode que le "vrai" client HTTP...`
```

Pendant ce temps, dans le code de production, nous implémenterons les requêtes HTTP dans une classe `RealHttpClient` afin que nous puissions appeler `printBook(id, new RealHttpClient(...))`.

Nous pourrions rendre la sortie explicite de la même manière, en créant une interface `ConsolePrinter` que nous passerions en argument à `printBook`. Cependant, nous pouvons modifier la méthode pour qu'elle renvoie le texte à la place, ce qui est souvent plus simple :

```
String getBook(
  String bookId,
  HttpClient client
);
```

Nous pouvons maintenant tester le résultat de `getBook`, et dans le code de production l'écrire sur la console.

Adapter le code en injectant des dépendances et en rendant les sorties explicites nous permet de tester davantage de code avec des tests ”simples” plutôt qu'avec des tests complexes de bout en bout. Si les tests de bout en bout restent utiles pour s'assurer que les dépendances sont respectées et que les résultats sont utilisés correctement, les tests manuels pour les scénarios de bout en bout fournissent déjà un degré raisonnable de confiance. Par exemple, si le code n'imprime pas du tout sur la console, un humain le remarquera certainement.

Ce type de modifications du code peut être effectué de manière récursive jusqu'à ce que seul le ”code de collage” entre les modules et les primitives de bas niveau restent non testables. Par exemple, une classe ”client UDP” peut prendre en paramètre une interface ”client IP”, de sorte que la fonctionnalité UDP puisse être testée. La mise en oeuvre de l'interface ”client IP” peut elle-même prendre en paramètre une interface ”client de données”, de sorte que la fonctionnalité IP puisse être testée. Les implémentations de l'interface ”client de données”, comme Ethernet ou Wi-Fi, devront probablement être testées de bout en bout puisqu'elles ne dépendent pas elles-mêmes d'autres logiciels locaux.

Exercice A vous de jouer ! Dans le dossier d'exercices pendant le cours vous trouverez `joker.py`, qui n'est pas facile à tester dans son état actuel. Modifiez-le pour le rendre testable, écrivez des tests et modifiez `app.py` pour qu'il corresponde aux changements de `Joker` et préserve la fonctionnalité du programme original. Commencez par écrire une interface pour un client HTTP, implémentez-la en déplaçant le code existant, et utilisez-la dans `Joker`. Ensuite, ajoutez des tests.

Les changements nécessaires sont similaires à ceux que nous avons discutés plus haut, y compris l'injection d'une dépendance `HttpClient` et le fait que la fonction retourne du texte.

Si vous avez le temps et l'envie, essayez un exercice optionnel : implémentez les `TODO` dans `app_test.py`.

Dans des langages fortement typés comme Java, si vous avez besoin d'écrire beaucoup de dépendances de test différentes, vous pouvez trouver utiles les frameworks de *mocking* tels que Mockito pour Java. Ces frameworks vous permettent d'écrire un faux `HttpClient`, par exemple, comme ceci :

```
// En Python on pourrait juste déclarer une classe avec la même méthode, mais en Java puisqu'il faut un
var client = mock(HttpClient.class);
when(client.get(anyString())).thenReturn("Hello");
// il existe également des méthodes permettant de lever une exception, de vérifier que des appels spéci
```

Il existe d'autres types de tests dont nous n'avons pas parlé dans ce cours, tels que les tests de performance, les tests d'accessibilité, les tests d'utilisabilité, etc. Nous verrons certains d'entre eux dans les cours suivants.

Résumé

Dans ce cours, vous avez appris :

- Les tests automatisés, leurs fondements, quelques bonnes pratiques, et comment adapter le code pour le rendre testable
- La couverture du code comme moyen d'évaluer les tests, y compris la couverture des instructions, la couverture des branches et la couverture des chemins.
- Quand écrire des tests, y compris les tests après le développement, le TDD et les tests de régression

Vous pouvez maintenant consulter les exercices !

Exigences

Nous n'écrivons pas du code pour le plaisir d'écrire du code : nous développons des logiciels pour *aider les gens à accomplir des tâches*. Il peut s'agir de "n'importe qui sur Terre" pour les logiciels largement utilisés tels que GitHub, ou de "personnes qui font un travail spécifique" pour les applications internes, ou même une seule personne dont la vie peut être aidée par un logiciel. Pour savoir *quel* logiciel développer, nous devons connaître les besoins des utilisateurs : leurs *exigences*.

Objectifs

Après cette partie, vous devriez être en mesure de :

- Définir les besoins des utilisateurs
- Formaliser les exigences en *personas* et en *récits utilisateurs*
- Comprendre les exigences *implicites* et _explicites

Quels sont les besoins ?

Une exigence est quelque chose dont les utilisateurs ont besoin. Par exemple, un utilisateur peut vouloir se déplacer d'un endroit à un autre. Il peut alors utiliser une voiture. Mais peut-être l'utilisateur a-t-il d'autres exigences, comme le fait de ne pas vouloir ou de ne pas pouvoir conduire, auquel cas un train pourrait répondre à ses besoins. Les utilisateurs peuvent également avoir des exigences plus spécifiques, comme le fait de vouloir un moyen de transport à faible émission de carbone, et doivent se déplacer entre des endroits éloignés des transports publics, auquel cas une voiture électrique alimentée par une électricité bas-carbone pourrait être une solution.

Les exigences ne concernent pas les détails de la mise en oeuvre. Un type spécifique de moteur électrique ou un type spécifique d'acier pour les portières de la voiture ne sont pas des exigences de l'utilisateur. Toutefois, les utilisateurs peuvent avoir des besoins qui amènent les concepteurs du système à faire de tels choix.

Il arrive que l'on distingue les exigences "fonctionnelles" des exigences "non fonctionnelles", ces dernières étant également connues sous le nom d'"attributs de qualité". Les exigences "fonctionnelles" sont celles

qui sont directement liées aux caractéristiques, tandis que les exigences ”non fonctionnelles” comprennent l'accessibilité, la sécurité, les performances, etc. La distinction n'est pas toujours claire, mais elle est parfois faite.

La définition des besoins commence généralement par une discussion avec les utilisateurs. C'est plus facile si le logiciel a un petit nombre d'utilisateurs bien définis, comme un logiciel spécialement conçu pour qu'une personne puisse faire son travail au sein d'une entreprise. C'est plus difficile si le logiciel a un grand nombre d'utilisateurs mal définis, comme un moteur de recherche ou une application pour écouter de la musique.

Il est important, lorsque l'on écoute ce que disent les utilisateurs, de garder à l'esprit ce dont ils ont *besoin* plutôt que ce qu'ils *veulent*. Ce que les utilisateurs déclarent vouloir est fortement influencé par ce qu'ils connaissent et utilisent déjà. Par exemple, une personne habituée à voyager à cheval et qui doit traverser une montagne peut demander un ”cheval volant”, alors qu'il s'agit en fait de traverser la montagne, et donc un train avec tunnel ou un avion répondrait à leurs besoins. De même, avant l'avènement du téléphone intelligent moderne, les utilisateurs auraient demandé des vieux téléphones pour la simple raison qu'ils ne savaient même pas qu'un téléphone intelligent était possible, même si aujourd'hui ils préfèrent leur téléphone intelligent à leur ancien téléphone.

Les souhaits des utilisateurs peuvent être ambigus, surtout lorsqu'ils sont nombreux. Si vous sélectionnez les cellules contenant ”100” et ”200” dans un tableur tel que Microsoft Excel et que vous élargissez la sélection, que se passe-t-il ? Excel doit-il remplir les nouvelles cellules avec ”300”, ”400”, etc. ? Doit-il répéter ”100” et ”200” ? Que se passe-t-il si vous développez une sélection contenant la cellule unique ”Salle 120” ? Doit-elle être répétée, ou doit-elle devenir ”Salle 121”, ”Salle 122”, etc. ? Il n'y a pas de réponse parfaite à ces questions, car toute réponse laissera certains utilisateurs insatisfaits, mais un développeur doit faire un choix.

Ne pas écouter les besoins des utilisateurs peut coûter cher, comme Microsoft l'a constaté avec Windows 8. L'interface utilisateur de Windows 8 était une réinvention majeure de l'interface Windows, utilisant un ”écran de démarrage” plutôt qu'un menu, avec des applications en plein écran qui pouvaient être empilées plutôt que déplacées. Il s'agissait d'un changement radical par rapport au Windows que les utilisateurs connaissaient, et ce fut un échec commercial. Microsoft a dû rétablir le menu Démarrer et abandonner le concept d'applications plein écran en mosaïque. Cependant, Apple a ensuite fait quelque chose de similaire pour une nouvelle version de son système d'exploitation pour iPad, et cela a plutôt bien fonctionné, peut-être parce que les utilisateurs ont des attentes différentes sur les ordinateurs de bureau et les tablettes.

Comment formaliser les exigences ?

Une fois que vous avez discuté avec un grand nombre d'utilisateurs et obtenu de nombreuses idées sur les besoins des utilisateurs, comment consolider ce retour d'information en éléments exploitables ? C'est la raison d'être de la formalisation des exigences, et nous discuterons de la formalisation du *qui* avec les ”personas” et du *quoi* avec les ”user stories”.

À qui s'adresse le logiciel ? La réponse à cette question est parfois évidente, car le logiciel est destiné à un petit groupe d'utilisateurs spécifiques, mais la plupart des grands logiciels sont destinés à de nombreuses personnes. En fait, les grands logiciels comptent généralement beaucoup trop d'utilisateurs pour qu'un processus personnalisé puisse être mis en place. Au lieu de cela, vous pouvez utiliser des *personas* pour représenter des groupes d'utilisateurs.

Un persona est une personne *abstraite* qui représente un groupe d'utilisateurs similaires. Par exemple, dans une application musicale, une persona pourrait être Alice, une étudiante, qui utilise l'application pendant son trajet dans les transports publics. Alice n'est pas une personne réelle, et elle n'a pas besoin de caractéristiques spécifiques telles qu'une couleur de cheveux ou une nationalité. Au lieu de cela, Alice est une représentation abstraite des nombreuses personnes qui pourraient utiliser l'application et qui ont toutes des caractéristiques similaires du point de vue de l'application, à savoir qu'elles utilisent l'application dans les transports publics pour se rendre à l'école, à l'université ou dans un autre lieu similaire. Les exigences d'Alice pourraient conduire à des fonctionnalités telles que le téléchargement de podcasts à l'avance à la maison et l'écoute avec l'écran éteint. Une autre persona pour la même application pourrait être Bob, un retraité, qui utilise l'application pour cuisiner et faire le ménage. Bob n'est pas non plus une personne réelle, mais il représente

un groupe d'utilisateurs potentiels qui ne sont pas très au fait des dernières technologies et qui souhaitent utiliser l'application pendant qu'ils effectuent des tâches à la maison.

Bien qu'il soit possible de créer des personas pour de nombreux groupes d'utilisateurs potentiels, elles ne seront pas toutes retenues. Toujours pour l'exemple de l'application musicale, un autre personnage pourrait être Carol, une ”hacker” qui veut écouter de la musique piratée. Carol aurait besoin de fonctionnalités telles que le chargement de morceaux de musique existants dans l'application et le contournement de la protection des droits d'auteur. L'application est-elle destinée à des personnes comme Carol ? C'est aux développeurs d'en décider.

Un dernier mot sur les personas : évitez de trop les abstraire. Les personas sont utiles parce qu'elles représentent des personnes réelles d'une manière utile au développement. Si vos personas finissent par ressembler à ”Jean, un utilisateur qui utilise l'application” ou ”Jeanne, une personne qui a un téléphone”, elles ne seront pas utiles. De même, si vous savez déjà qui est exactement dans un groupe d'utilisateurs, il n'est pas nécessaire de l'abstraire. ”Sam, un administrateur système” n'est pas une persona utile si votre application n'a qu'un seul administrateur système : utilisez plutôt la personne réelle.

Exercice Quels pourraient être des personas pour une application de chat vidéo ?

Anne, une responsable qui s'adresse fréquemment à son équipe tout en travaillant à distance.

Basil, un retraité qui souhaite dialoguer par vidéo avec ses petits-enfants pour rester en contact.

Carlos, un médecin qui doit parler à des patients dans le cadre d'une installation de télémédecine.

Que peuvent faire les utilisateurs ? Après avoir défini à qui s'adresse le logiciel, il faut décider des fonctionnalités à mettre en place. Les récits utilisateurs sont un outil utile pour formaliser les caractéristiques basées sur les exigences, y compris qui veut la caractéristique, quelle est la caractéristique, et quel est le contexte. Le contexte est essentiel, car la même fonctionnalité peut être mise en oeuvre de manière extrêmement différente en fonction du contexte. Par exemple, ”envoyer des courriers électroniques contenant des informations” est une fonction qu'un système logiciel peut avoir. Si les utilisateurs souhaitent archiver des informations, les courriers électroniques doivent contenir des informations très détaillées, mais leur heure d'arrivée n'a que peu d'importance. Si le contexte est que les utilisateurs souhaitent recevoir une notification dès qu'un événement se produit, les courriels doivent être envoyés immédiatement, et il convient de faire très attention à ce qu'ils ne se retrouvent pas dans un filtre anti-spam. Si le contexte est que les utilisateurs souhaitent partager des données avec leurs amis qui n'utilisent pas le logiciel, les courriels doivent avoir une conception claire qui ne contient que les informations pertinentes afin qu'ils puissent être facilement transférés.

Il existe de nombreux formats pour les récits utilisateurs ; dans ce cours, nous utiliserons le format en trois parties ”*en tant que... je veux... pour...*”. Ce format inclut l'utilisateur qui souhaite la fonctionnalité, qui peut être un persona ou un rôle spécifique, la fonctionnalité elle-même et un contexte expliquant pourquoi l'utilisateur souhaite cette fonctionnalité. Par exemple, ”En tant qu'étudiant, je veux regarder des enregistrements de cours, afin de pouvoir rattraper mon retard après une maladie”. Ce récit utilisateur permet aux développeurs de créer une fonctionnalité qui est réellement utile à la personne : il ne serait pas utile à cet étudiant, par exemple, de créer une fonction d'enregistrement des cours qui soit une archive accessible une fois le cours terminé, car on peut supposer que l'étudiant ne sera pas malade pendant toute la durée du cours.

Revenons à notre exemple d'application musicale et considérons le récit utilisateur ”En tant qu'Alice, je souhaite télécharger des podcasts à l'avance, afin d'économiser des données mobiles”. Cela signifie que l'application doit télécharger l'intégralité du podcast à l'avance, mais Alice dispose toujours de données mobiles, elle ne veut simplement pas en utiliser trop. Un récit similaire dans un contexte différent pourrait être ”En tant que navetteur en voiture, je souhaite télécharger des podcasts à l'avance, afin de pouvoir utiliser l'application sans données mobiles”. Ce récit conduit à une autre fonctionnalité : maintenant, l'application ne peut pas du tout utiliser les données mobiles, parce que le navetteur n'a tout simplement pas de données à certains moments de son trajet.

Pour évaluer les récits utilisateurs, rappelez-vous l'acronyme ”INVEST” :

- **I**ndépendant : le récit doit être autonome.
- **N**égociable : le récit n'est pas un contrat strict mais peut évoluer.
- **V**aluable (ayant une valeur) : le récit doit apporter une valeur évidente à quelqu'un.
- **E**stimable : les développeurs doivent être en mesure d'estimer le temps nécessaire à la mise en oeuvre du récit.
- ***S**mall* (petit) : le récit doit être d'une taille raisonnable et ne pas être un énorme récit ”fourre-tout”.
- **T**estable : les développeurs doivent être en mesure de savoir quels sont les critères d'acceptation du récit sur la base de son texte, afin de pouvoir tester leur mise en oeuvre.

Les récits trop difficiles à comprendre et surtout trop vagues échoueront au test ”INVEST”.

Exercice Quels récits utilisateurs pourriez-vous utiliser pour une application de chat vidéo ?

En tant qu'Anne, je veux voir mon calendrier dans l'application, afin de pouvoir planifier des réunions sans entrer en conflit avec mes autres engagements.

Comme Basil, je veux lancer une réunion à partir d'un message que mes petits-enfants m'envoient, afin de ne pas avoir à passer du temps à configurer l'application.

En tant qu'adepte de la protection de la vie privée, je veux que mes chats vidéo soient chiffrés de bout en bout, afin que mes données ne soient pas divulguées en cas de piratage des serveurs de l'application.

Exercice Lesquels des éléments suivants constituent des bons récits utilisateur et pourquoi ?

1. En tant qu'utilisateur, je veux me connecter rapidement pour ne pas perdre de temps.
2. En tant que titulaire d'un compte Google, je souhaite me connecter avec mon adresse Gmail.
3. En tant que cinéphile, je souhaite afficher les films recommandés en haut sur un fond gris foncé avec un défilement horizontal.
4. En tant que lecteur occasionnel, je veux savoir où je me suis arrêté la dernière fois, afin de poursuivre ma lecture.
5. En tant que développeur, je souhaite améliorer l'écran de connexion, afin que les utilisateurs puissent se connecter avec des comptes Google.

1 est trop vague, 2 est acceptable car la raison est implicite et évidente, 3 est beaucoup trop spécifique, 4 est excellent, et 5 est terrible car il concerne les développeurs et non les utilisateurs.

Comment pouvons-nous développer à partir des exigences ?

Vous avez écouté les utilisateurs, vous les avez abstraits dans des personas et leurs exigences dans des récits utilisateurs, et vous avez développé une application sur cette base. Vous êtes convaincu que vos personas aimeraient votre application et que votre mise en oeuvre répond aux besoins définis par les récits des utilisateurs. Après avoir dépensé beaucoup de temps et d'argent, vous disposez maintenant d'une application que vous pouvez présenter à de vrais utilisateurs... et ils ne l'aiment pas. Ce n'est pas du tout ce qu'ils avaient imaginé. Qu'est-ce qui n'a pas fonctionné ? Ce que vous venez de faire, en demandant aux utilisateurs leur avis sur l'application, est une *validation* : vous vérifiez si ce que vous avez spécifié correspond à ce que veulent les utilisateurs. Ceci est différent de la *vérification*, qui vérifie que votre application fait correctement ce que vous avez spécifié.

L'une des clés de la réussite d'un logiciel est de procéder à une validation précoce et fréquente, plutôt que de l'attendre jusqu'à la fin. Si ce que vous construisez ne correspond pas aux attentes des utilisateurs, vous devez le savoir le plus tôt possible, au lieu de gaspiller des ressources à construire quelque chose que personne n'utilisera. Pour effectuer cette validation, vous devrez construire le logiciel d'une manière qui puisse être décrite par les utilisateurs, en utilisant un *vocabulaire commun* avec eux et en les *intégrant* dans le processus afin qu'ils puissent donner leur avis. Voyon donc ces deux façons de procéder.

Tout d'abord, vous avez besoin d'un vocabulaire commun avec vos utilisateurs, comme le résume Eric Evans dans son livre de 2003 "Domain-Driven Design". Envisagez la fabrication de bonbons : vous pourriez demander à l'avance aux gens s'ils veulent des bonbons contenant du NaCl, $C_{24}\ H_{36}\ O_{18}$, $C_{36}\ H_{50}\ O_{25}$, $C_{125}\ H_{188}\ O_{80}$, et ainsi de suite. Il s'agit d'une définition précise que vous pourriez donner à des chimistes, qui créeraient ensuite le produit en question. Toutefois, il est peu probable qu'une personne ordinaire comprenne ce que c'est avant que vos chimistes ne le créent réellement. Au lieu de cela, vous pourriez demander aux gens s'ils veulent des bonbons au "caramel salé". C'est moins précis, car on peut imaginer différentes sortes de caramel salé, mais beaucoup plus compréhensible pour les utilisateurs finaux. Vous n'avez pas besoin de créer des biscuits au caramel salé pour que les gens vous disent s'ils aiment l'idée ou non, et une discussion sur le bonbon que vous proposez sera beaucoup plus fructueuse si l'on utilise le terme "caramel salé" plutôt qu'une formule chimique.

Dans son livre, Evans propose quelques termes spécifiques qui peuvent être enseignés aux utilisateurs, tels que "entité" pour les objets dont l'identité est liée à une propriété spécifique, "objet valeur" pour les objets qui sont des agrégats de données sans identité distincte, etc. L'important n'est pas de savoir quels termes exacts vous utilisez, mais l'idée qu'il faut concevoir un logiciel d'une manière qui puisse être facilement décrite aux utilisateurs.

Prenons l'exemple de ce qu'un utilisateur pourrait appeler un "service de connexion", qui identifie les utilisateurs par ce qu'ils appellent des "adresses électroniques". Un programmeur habitué à l'aspect technique des choses pourrait appeler cela une `PrincipalFactory`, puisque "principals" est une façon d'appeler une identité d'utilisateur, et qu'une "factory" est un objet qui crée des objets. Les identifiants pourraient être techniquement appelés "ID". Cependant, si l'on demande à un utilisateur "Que doit-il se passer lorsque l'ID n'est pas trouvé ? Le principal renvoyé par la fabrique doit-il être `null` ?", l'utilisateur sera perplexe. La plupart des utilisateurs ne connaissent aucun de ces termes. Au lieu de cela, si l'objet dans le code est nommé `LoginService`, et prend en compte des objets de type `Email` pour identifier les utilisateurs, le programmeur peut maintenant demander à l'utilisateur "Que se passe-t-il lorsque l'e-mail n'est pas trouvé ? Le processus de connexion doit-il échouer ?" et obtenir une réponse utile.

L'utilisation d'un vocabulaire *correct* pour discuter avec les gens passe aussi par l'utilisation d'un vocabulaire *spécifique*. Prenons le terme "personne". Si vous demandez à des employés d'une université ce qu'est une "personne", ils marmonneront une vague réponse sur le fait que les gens ont un nom et un visage, parce qu'ils n'ont pas affaire à des "personnes" en général dans leur travail. Ils s'occupent plutôt de types de personnes spécifiques. Par exemple, les gens du service financiers ont affaire à des "employés" et à des "contractants", et pourraient volontiers vous enseigner exactement ce que sont ces concepts, en quoi ils diffèrent, quels types d'attributs ils possèdent, quelles opérations sont effectuées sur leurs données, etc. Evans parle de "contexte délimité" : dans un domaine d'activité spécifique, certains mots ont une signification particulière, qui doit se refléter dans la conception. Imaginez que vous essayiez de faire en sorte qu'un auditeur financier, un employé de cafétéria et un professeur se mettent d'accord sur ce qu'est une "personne" et discutent de l'ensemble de l'application en utilisant une définition de la "personne" qui inclut tous les attributs possibles. Cela prendrait une éternité et tout le monde s'ennuierait. Au lieu de cela, vous pouvez parler à chaque personne séparément, représenter ces concepts séparément dans le code et effectuer des opérations pour les relier entre eux par le biais d'une identité commune, comme une fonction `get_employee(email)`, une fonction `get_student(email)`, et ainsi de suite.

Exercice Quel vocabulaire utiliseriez-vous pour discuter d'un système d'inscription aux cours dans une université (ou de tout autre système que vous utilisez fréquemment), et quels scénarios de test pourriez-vous définir ?

Un système d'inscription aux cours pourrait avoir des étudiants, qui sont des entités identifiées par une adresse électronique de l'université, qui disposent d'une liste de cours et de notes associées à ces cours, et les professeurs, qui sont associés aux cours qu'ils enseignent et peuvent les modifier. Les cours eux-mêmes peuvent avoir un nom, un code, une description et un nombre de crédits.

Certains scénarios de test pourraient être les suivants : "Étant donné qu'un utilisateur est déjà inscrit à un cours, quand l'utilisateur essaie de s'inscrire à nouveau, alors cela n'a aucun effet", ou "Étant donné qu'un

professeur est responsable d'un cours, quand le professeur fixe la note d'un étudiant dans le cours, alors la note de cet étudiant est mise à jour".

Quels sont les besoins implicites ?

Les ingénieurs en logiciel conçoivent des systèmes pour toutes sortes de personnes, de toutes les régions du monde, avec toutes sortes de besoins. Souvent, certaines personnes ont des exigences qui sont *implicites*, mais qui sont tout aussi nécessaires que les exigences dont elles vous parleront explicitement. Concrètement, nous verrons la *traduction* ("localization" en anglais), l'*internationalisation* et l'*accessibilité*. Ils sont parfois abrégés en "l10n", "i18n" et "a11y", chacun ayant conservé sa lettre de début et de fin, les lettres restantes étant réduites à leur nombre. Par exemple, il y a 10 lettres entre "l" et "n" dans "localization", d'où "l10n".

La *traduction* est importante. Les utilisateurs s'attendent à ce que tous les textes des programmes soient rédigés dans leur langue, même s'ils n'y pensent pas explicitement. Au lieu de `print("Hello " + user)`, par exemple, votre code devrait utiliser une constante qui peut être modifiée en fonction de la langue. Il est tentant d'avoir une constante `HELLO_TEXT` avec la valeur `"Hello"` pour l'anglais, mais cela ne fonctionne pas pour toutes les langues car le texte peut aussi venir après le nom d'utilisateur, et pas seulement avant. Le code pourrait donc utiliser une fonction qui se charge d'envelopper le nom de l'utilisateur avec le bon texte : `print(hello_text(user))`.

La traduction peut sembler simple, mais elle implique également une vérification des hypothèses de votre interface utilisateur et de votre logique. Par exemple, un bouton qui peut contenir le texte "Log in" en anglais peut ne pas être assez large lorsque le texte est le français "Connexion". Un champ de texte suffisamment large pour contenir chacun des mots de "capitaine de la navigation à vapeur du Danube" pourrait déborder sur l'allemand "Donaudampfschifffahrtsgesellschaftkapitän". Vos fonctions qui fournissent du texte traduit peuvent avoir besoin de plus d'informations que vous ne le pensez. Les noms anglais n'ont pas de genre grammatical, donc tous les noms peuvent utiliser le même texte, mais le français en a deux, l'allemand en a trois, et le swahili en a dix-huit ! Les noms anglais ont une forme "singulière" et une forme "plurielle", tout comme de nombreuses langues telles que le français, mais ce n'est pas universel ; le slovène, par exemple, a une terminaison pour 1, une pour 2, une pour 3 et 4, et une pour 5 et plus.

Les traductions peuvent comporter des bugs, tout comme les logiciels. Par exemple, la traduction allemande du jeu Champions World Class Soccer comporte un bug dans lequel "shootout" n'est pas traduit par "schiessen", comme il se doit, mais plutôt à "scheissen", qui a une signification totalement différente bien qu'il n'y ait qu'une lettre d'écart. Un autre cas de problèmes allemands se présente dans le jeu Grandia HD : manquer une attaque affiche le texte "Fräulein", qui est bien la traduction du mot anglais "Miss" mais dans un contexte totalement différent.

La traduction n'est pas quelque chose que vous pouvez faire seul, à moins que vous ne traduisiez vers votre langue maternelle, car personne ne connaît toutes les caractéristiques de toutes les langues. Tout comme les autres parties d'un logiciel, la traduction doit être testée, en l'occurrence par des locuteurs natifs.

L'*internationalisation* est une question d'éléments culturels autres que la langue. Prenons l'exemple suivant :

Que voyez-vous ? Comme vous lisez un texte en français, vous pensez peut-être qu'il s'agit d'une illustration d'un t-shirt qui passe de sale à propre dans une machine à laver. Mais quelqu'un dont la langue maternelle se lit de droite à gauche, comme l'arabe, pourrait voir le contraire : un t-shirt qui passe de propre à sale dans la machine. C'est un peu dommage, mais c'est ainsi que fonctionne la communication humaine : les gens ont des attentes implicites qui sont parfois contradictoires. Les logiciels doivent donc s'adapter. Un autre exemple est de mettre "Vincent van Gogh" dans une liste de personnes classées par nom de famille. Vincent

est-il sous ”G” pour ”Gogh” ou sous ”V” pour ”van Gogh” ? Cela dépend de la personne à qui l'on s'adresse : les néerlandais attendent le premier, les belges le second. Les logiciels doivent s'adapter, sinon au moins l'un de ces deux groupes sera confus. L'utilisation d'un format de date spécifique à une culture est un autre exemple : ”10/01” signifie des dates très différentes pour un habitant des États-Unis et pour un habitant du reste du monde.

Les noms des personnes constituent un élément important lié à l'internationalisation. De nombreux systèmes logiciels sont construits sur la base d'hypothèses étranges concernant les noms des personnes, telles que ”ils sont entièrement composés de lettres”, ou ”les noms de famille ont au moins 3 lettres”, ou simplement ”les noms de famille sont quelque chose que tout le monde a”. En général, les programmeurs croient beaucoup de choses fausses sur les noms, ce qui entraîne de nombreuses difficultés pour les personnes dont le nom n'est pas conforme à ces hypothèses, comme les personnes dont le nom comporte des traits d'union ou des apostrophes, ou encore les personnes dont le nom ne comporte qu'une ou deux lettres, les personnes issues de cultures qui n'ont pas de notion de nom de famille, etc. Rappelez-vous que *le nom d'une personne n'est jamais invalide.*

Tout comme la traduction, l'internationalisation n'est pas quelque chose que vous pouvez faire seul. Même si vous êtes originaire de la région que vous visez, cette région est susceptible de contenir de nombreuses personnes issues de cultures différentes. L'internationalisation doit être testée.

L'*accessibilité* est une propriété que possède un logiciel lorsqu'il peut être utilisé par tout le monde, même par les personnes qui n'entendent pas, ne voient pas, n'ont pas de mains, etc. Les fonctions d'accessibilité comprennent les sous-titres, la synthèse vocale, la dictée, les claviers à une main, etc. Les frameworks d'interface utilisateur ont généralement des fonctions d'accessibilité documentées en même temps que leurs autres fonctions. Ils sont utiles non seulement aux personnes souffrant d'un handicap permanent, mais aussi à celles qui sont temporairement incapables d'utiliser une partie de leur corps. Par exemple, les sous-titres sont pratiques pour les personnes qui se trouvent dans des trains bondés et qui ont oublié leurs écouteurs. La synthèse vocale est pratique pour les personnes qui font autre chose pendant qu'elles utilisent une application sur leur téléphone. Les fonctions conçues pour les personnes sans mains sont pratiques pour les parents qui tiennent leur bébé.

L'accessibilité n'est pas seulement une bonne chose d'un point de vue moral : elle est souvent exigée par la loi, en particulier pour les logiciels destinés aux agences gouvernementales. C'est également une bonne chose d'un point de vue égoïste : une application plus accessible a plus de clients potentiels.

Toutes les exigences sont-elles éthiques ?

Nous avons parlé des exigences des utilisateurs en partant du principe que vous devez faire tout ce dont les utilisateurs ont besoin. Mais est-ce toujours le cas ?

Parfois, les exigences sont en conflit avec votre éthique d'ingénieur, et ces conflits ne doivent pas être ignorés. Ce n'est pas parce qu'”un ordinateur le fait” que la tâche est acceptable ou qu'elle sera exécutée sans aucun biais. L'ordinateur peut être impartial, mais il exécute un code écrit par un être humain, qui reproduit les hypothèses et les préjugés de cet être humain, ainsi que toutes sortes de problèmes dans les données sous-jacentes.

Par exemple, il y a eu de nombreuses variantes d'”algorithmes pour prédire si quelqu'un sera un criminel”, utilisant des caractéristiques telles que les visages des personnes. Si quelqu'un passait dans une classe et disait à chaque élève ”tu es un criminel” ou ”tu n'es pas un criminel”, on pourrait raisonnablement être contrarié et soupçonner toutes sortes de mauvaises raisons pour ces décisions. Il en va de même pour un ordinateur : il n'existe pas d'algorithme magique ”impartial” ou ”objectif” et, en tant qu'ingénieur logiciel, vous devez toujours être conscient de l'éthique.

Résumé

Dans cette conférence, vous avez appris à :

- Définir et formaliser les besoins des utilisateurs à l'aide d'exigences, de personas et de récits d'utilisateurs.

- Comprendre les exigences implicites telles que la traduction, l'internationalisation, l'accessibilité et l'éthique

Vous pouvez maintenant consulter les exercices !

Debugging

> **Prérequis** : Vous êtes *fortement* encouragé, mais pas strictement obligé, d'utiliser un IDE pour les exercices sur le debugging dans cette conférence. Vous pouvez également utiliser le debugger en ligne de commande intégré à Python, `pdb`, mais il est beaucoup moins pratique qu'une interface utilisateur graphique.

L'écriture du code n'est qu'une partie du génie logiciel ; vous passerez souvent plus de temps à *lire* du code qu'à en écrire. Il peut s'agir de comprendre un morceau de code afin de pouvoir l'étendre, ou de "debugger" un code existant en trouvant et en corrigeant les bugs. Ces tâches sont beaucoup plus faciles si vous écrivez votre code en tenant compte de la lisibilité et de la debuggabilité, et si vous savez comment utiliser les outils de debugging.

Objectifs

Après ce cours, vous devriez être en mesure de :

- Développer un code _lisible
- Utiliser un *debugger* pour comprendre et debugger le code
- Isoler la *cause principale* d'un bug
- Développer du code *débuggable*

Qu'est-ce qui rend le code lisible ?

Jetez un coup d'oeil à `planets.py` dans le dossier d'exercices pendant le cours. Le trouvez-vous facile à lire et à comprendre ? Probablement pas. Avez-vous remarqué que le tri ne fonctionne pas parce que le code utilise une fonction qui renvoie la liste triée, mais n'utilise pas sa valeur de retour, plutôt qu'un tri sur place ? Il est beaucoup plus difficile de repérer ce bug lorsqu'il faut faire appel à toute sa puissance cérébrale pour lire le code. (Vous pouvez consulter plus de problèmes concrets dans la solution)

Malheureusement, le code difficile à lire ne se trouve pas uniquement dans les exercices des notes de cours. Prenons l'exemple suivant, extrait du cadre ScalaTest :

```
package org.scalatest.tools;

objet Runner {
  def doRunRunDaDoRunRun(...) : Unit
}
```

Que fait cette méthode ? Le nom de la méthode ne vous aidera pas. Il s'agit d'une référence à une chanson de 1963. Bien sûr, c'est une référence amusante, mais ne préféreriez-vous pas lire un nom qui vous explique ce que fait la méthode ? Pire, la méthode comporte 21 paramètres. Vous pouvez bien sûr comprendre ce que fait la méthode si vous y consacrez suffisamment de temps, mais est-ce nécessaire ?

Parlons de cinq composantes de la lisibilité du code : le nom, la documentation, les commentaires, la mise en forme et la cohérence.

Noms

Tout d'abord, un exemple de *nommage* sans même utiliser de code : les erreurs de mesure. Si vous mesurez la présence de quelque chose qui est absent, il s'agit d'une erreur de type I. Si vous mesurez l'absence de quelque chose qui est présent, il s'agit d'une erreur de type II. Ce type d'erreurs se produit constamment, par

exemple dans les tests de détection de virus. Cependant, il est facile d'oublier quel est le type `I` et quel est le type `II`. Et même si vous vous en souvenez, une faute de frappe dupliquant un caractère peut complètement changer le sens d'une phrase. A la place, vous pouvez utiliser `faux positif` et `faux négatif`. Ces noms sont plus faciles à comprendre, plus faciles à retenir et plus résistants aux fautes d'orthographe.

Les noms dans le code peuvent également faire la différence entre un code facile à comprendre et un code difficile à analyser mentalement. Une variable nommée `isUserOnline` en Java est très bien... tant qu'il s'agit d'une variable booléenne "l'utilisateur est-il en ligne". S'il s'agit d'un nombre entier, c'est beaucoup plus confus. Et si vous écrivez Python à la place, c'est aussi un problème puisque Python utilise des soulignés pour séparer les mots, alias "snake case", donc cela devrait être `is_user_online`. Une variable nommée `can_overwrite_the_data` est un nom assez long, ce qui n'est pas un problème en soi, mais est redondant : bien sûr que nous écrasons des "données", le nom est trop vague.

Les noms ne concernent pas uniquement les noms de variables, de méthodes ou de classes. Considérons l'appel de méthode suivant :

```
split("abc", "a", true, true)
```

Que fait cette méthode ? Bonne question. Que se passerait-il si l'appel ressemblait plutôt à ceci, avec des constantes ou des enums ?

```
split("abc", "a",
      SplitOptions.REMOVE_EMPTY_ENTRIES,
      CaseOptions.IGNORE_CASE)
```

Il s'agit du même appel de méthode, mais nous avons donné des noms au lieu de valeurs, et la signification est maintenant plus claire. Ces constantes ne sont cependant qu'une solution de contournement à l'absence de paramètres nommés en Java. En Scala, et dans d'autres langages comme le C#, vous pourriez appeler la méthode comme suit :

```
split("abc",
      separator = "a",
      removeEmptyEntries = true,
      ignoreCase = true)
```

Le code est maintenant beaucoup plus lisible grâce aux noms explicites, sans avoir à écrire du code supplémentaire.

La *notation hongroise* est un exemple de bonne volonté en matière de nommage. Charles Simonyi, un ingénieur hongrois de Microsoft, a eu la bonne idée de commencer les noms de ses variables par le type de données précis qu'elles contenaient, par exemple `xPosition` pour une position sur l'axe des X, ou `cmDistance` pour une distance en centimètres. Cela signifie que n'importe qui pouvait facilement repérer l'erreur dans une formule telle que `distanceTotal += speedLeft / timeLeft`, puisque diviser la vitesse par le temps ne donne pas une distance. C'est ce qu'on a appelé la "notation hongroise", car dans la Hongrie natale de Simonyi, les noms s'écrivent avec le nom de famille en premier, par exemple "Simonyi Károly". Malheureusement, un autre groupe au sein de Microsoft n'a pas bien compris l'objectif de Simonyi et a plutôt pensé qu'il s'agissait du type de variable. Ils ont donc écrit des noms de variables tels que `cValue` pour une variable `char`, `lIndex` pour un index `long`, et ainsi de suite, ce qui rend les noms plus difficiles à lire sans ajouter plus d'informations que celles déjà présentes dans le type. On l'a appelé "Systems Hungarian", parce que le groupe faisait partie de la division des systèmes d'exploitation, et malheureusement, cette notation a fait son chemin dans les API Windows "Win32", qui étaient les principales API de Windows jusqu'à récemment. De nombreuses API Windows ont reçu des noms difficiles à lire à cause d'un malentendu ! Une fois de plus, le nommage n'est pas le seul moyen de résoudre ce problème, mais seulement l'un d'entre eux. Dans le langage de programmation F#, vous pouvez déclarer des variables avec des unités, comme `let distance = 30<cm>`, et le compilateur vérifiera que les comparaisons et les calculs ont un sens compte tenu des unités.

Exercice Les noms suivants semblent tous assez raisonnables, pourquoi sont-ils médiocres ?

- `pickle` (en Python)
- `binhex` (en Python)
- `LinkedList<E>` (dans `java.util` de Java)
- `vector<T>` (dans `std` de C++)
- `SortedList` (dans `System.Collections` en C#)

`pickling` est une façon plutôt étrange de se référer à la sérialisation et à la désérialisation des données pour les "préserver".

`binhex` semble être le nom de quelques utilitaires binaires et hexadécimaux, mais il s'agit en fait d'un module qui gère un ancien format Mac.

Une liste chaînée et une liste doublement chaînée ne sont pas la même chose, mais Java nomme la seconde comme s'il s'agissait de la première.

Un vecteur a une signification spécifique en mathématiques ; le vecteur de C++ est en fait un tableau redimensionnable.

`SortedList` est un nom acceptable pour une classe de liste triée. Mais la classe portant ce nom est un tableau associatif ("map") !

Dans l'ensemble, les noms sont un outil permettant de rendre le code clair et succinct, ainsi que cohérent avec d'autres bouts de code, de sorte que les lecteurs n'aient pas à penser explicitement aux noms.

Documentation

La *documentation* est un outil permettant d'expliquer *ce que fait* un morceau de code.

La documentation est le principal moyen pour les développeurs d'en savoir plus sur le code qu'ils utilisent. Lorsqu'ils écrivent du code, les développeurs consultent les commentaires de la documentation, généralement dans un IDE sous forme d'infobulles. Les commentaires de la documentation doivent donc décrire succinctement ce que fait une classe ou une méthode, y compris les informations dont les développeurs sont susceptibles d'avoir besoin, comme le fait de savoir si elle lève des exceptions, ou s'il exige que ses entrées soient dans un format spécifique.

Commentaires

Les commentaires sont un outil qui permet de savoir pourquoi un morceau de code fait ce qu'il fait. Il est important de noter que les commentaires ne doivent pas indiquer *comment* un morceau de code fait ce qu'il fait, car cette information existe déjà dans le code lui-même.

Malheureusement, tous le code ne se documente pas lui-même. Les commentaires sont un moyen d'expliquer un code délicat. Parfois, le code doit être écrit d'une manière qui semble excessivement compliquée ou erronée parce que le code contourne un problème dans son environnement, tel qu'un bug dans une bibliothèque, ou un compilateur qui ne produit un code assembleur rapide que dans des conditions spécifiques.

Voici un bon exemple de commentaire, tiré d'une ancienne version de la bibliothèque `libjsound` du kit de développement Java :

```
/* Solution de contournement : L'application 32bit sur linux 64bit obtient un échec d'assertion en essa
   Jusqu'à ce que le problème soit corrigé dans ALSA (bug 4807), signalons l'absence de périphériques m
if (jre32onlinux64) { return 0 ; }
```

Il s'agit d'un excellent exemple de commentaire : il explique quel est le problème externe, quelle est la solution choisie et renvoie à un identifiant pour le problème externe. De cette manière, un futur développeur peut rechercher ce bug dans ALSA, un système audio Linux, et vérifier s'il a été corrigé entre-temps, de sorte que le code qui contourne le bug puisse être supprimé.

Les commentaires sont un moyen d'expliquer le code au-delà de ce que le code lui-même peut faire, ce qui est souvent nécessaire même si, dans l'idéal, cela ne devrait pas l'être. Cette explication peut être destinée aux personnes qui examineront votre code avant d'accepter les modifications que vous proposez, ou aux collègues qui liront votre code lorsqu'ils travailleront sur la base de code des mois plus tard. N'oubliez pas que l'un de ces "collègues" est probablement "vous-même dans le futur". Même si vous pensez qu'un morceau de code est clair à l'heure actuelle, vous serez reconnaissant à l'avenir des commentaires qui expliquent les parties non évidentes.

Formatage

Le formatage du code a pour but de faciliter la lecture du code. On ne remarque pas un bon formatage, mais on remarque un mauvais formatage, et il est plus difficile de se concentrer.

Voici un exemple concret de mauvais formatage :

```
if (!update(&context, &params))
  goto fail;
  goto fail;
```

Avez-vous repéré le problème ? Le code donne l'impression que le deuxième `goto` est redondant, parce qu'il est formaté de cette façon. Mais c'est du C. Le second `goto` est en fait en dehors de la portée du `if`, et est donc toujours exécuté. Il s'agit d'un véritable bug qui a déclenché une vulnérabilité dans les produits Apple.

Certains langages imposent au moins une certaine cohérence de formatage, comme Python et F#. Mais comme vous l'avez vu avec l'exercice `planets.py` plus tôt, cela ne signifie pas qu'il est impossible de mal formater son code.

Cohérence

Devriez-vous utiliser `camelCase` ou `snake_case` pour vos noms ? 4 ou 8 espaces pour l'indentation ? Ou peut-être des tabulations ? Autant de questions.

C'est à cela que servent les *conventions*. Toute l'équipe décide, une fois pour toutes, de ce qu'il faut faire. Chaque membre de l'équipe accepte alors les décisions et bénéficie d'un style de code cohérent sans avoir à y penser explicitement.

Lors du choix d'une convention, il faut se méfier d'un problème courant appelé "bikeshedding" en anglais, venant du mot pour un abri à vélos. Le nom vient de l'histoire qui l'illustre : une réunion du conseil municipal a deux points à l'ordre du jour, l'entretien d'une centrale nucléaire et la construction d'un abri à vélos. Le conseil approuve rapidement la maintenance nucléaire, qui est très coûteuse, car tous conviennent que cette maintenance est nécessaire pour continuer à fournir de l'électricité à la ville. Ensuite, le conseil passe une heure à discuter des plans de l'abri à vélos, qui est très bon marché. N'est-il pas encore trop cher ? Il est certainement possible de réduire un peu le coût, un abri à vélos devrait être encore moins cher. Doit-il être bleu ou rouge ? Ou peut-être gris ? Combien de vélos doit-il contenir ? Il est facile de passer beaucoup de temps sur de petites décisions qui, en fin de compte, n'ont pas beaucoup d'importance, parce qu'il est facile de se concentrer sur elles. Mais ce temps devrait être consacré à des décisions plus importantes qui ont plus d'impact, même si elles sont plus difficiles à discuter.

Une fois que vous vous êtes mis d'accord sur une convention, vous devez utiliser des outils pour la faire respecter, et non des efforts manuels. Il existe des outils en ligne de commande, tels que `clang-format` pour le code C, ainsi que des outils intégrés aux IDE, qui peuvent être configurés pour s'exécuter à chaque fois que vous enregistrez un fichier. Ainsi, vous n'avez plus à réfléchir aux préférences de l'équipe, les outils le font pour vous.

Comment peut-on debugger efficacement un programme ?

Votre programme vient de recevoir un rapport de bug de la part d'un utilisateur : quelque chose ne fonctionne pas comme prévu. Que se passe-t-il alors ? La présence d'un bug implique que le comportement du code ne

correspond pas à l'intention de la personne qui l'a écrit.

Le but du *debugging* est de trouver et de corriger la *cause principale* du bug, c'est-à-dire le problème dans le code qui est à l'origine du bug. Il s'agit d'une différence par rapport aux *symptômes* de l'insecte, de la même manière que les symptômes d'une maladie telle que la perte d'odorat sont différents d'une cause profonde telle qu'un virus.

Vous trouverez la cause principale en posant à plusieurs reprises la question ”pourquoi ?” lorsque vous observez des symptômes, jusqu'à ce que vous atteigniez la cause principale. Par exemple, disons que vous avez un problème : vous êtes arrivé en retard en classe. Pourquoi ? Parce que vous vous êtes réveillé tard. Pourquoi ? Parce que votre alarme n'a pas sonné. Pourquoi ? Parce que votre téléphone n'avait plus de batterie. Pourquoi ? Parce que vous avez oublié de brancher votre téléphone avant de vous coucher. Voilà la cause du problème. Si vous vous étiez arrêté à, disons, ”votre alarme n'a pas sonné”, et que vous aviez essayé de résoudre le problème en ajoutant un deuxième téléphone avec une alarme, vous auriez simplement eu deux alarmes qui n'auraient pas sonné, car vous oublieriez de brancher également le deuxième téléphone. Mais maintenant que vous savez que vous avez oublié de brancher votre téléphone, vous pouvez vous attaquer à la cause principale, par exemple en plaçant un post-it au-dessus de votre lit pour vous rappeler de charger votre téléphone. En théorie, on peut continuer à demander ”pourquoi ?”, mais cela ne sert plus à rien au bout de quelques fois. Dans cet exemple, la ”vraie” cause profonde est peut-être que vous oubliez souvent des choses, mais vous ne pouvez pas y remédier facilement.

À un niveau élevé, le debugging comporte trois étapes : reproduire le bug, l'isoler, et le debugger.

Reproduire le bug signifie trouver les conditions dans lesquelles il apparaît :

- Quel environnement ? Est-ce sur des systèmes d'exploitation spécifiques ? À des heures précises de la journée ? Dans des langues spécifiques ?
- Quelles sont les mesures à prendre pour découvrir le bug ? Il peut s'agir d'une situation aussi simple que ”ouvrir le programme, cliquer sur le bouton 'login', le programme se bloque”, ou d'une situation plus complexe, telle que la création de plusieurs utilisateurs ayant des propriétés spécifiques et effectuant ensuite une séquence de tâches qui déclenchent un bug.
- Quel est le résultat attendu ? En d'autres termes, que devrait-il se passer s'il n'y avait pas de bug ?
- Quel est le résultat réel ? Il peut s'agir simplement d'un ”plantage”, ou de quelque chose de plus complexe, comme ”les résultats de la recherche sont vides alors que la base de données contient un élément correspondant à la recherche”
- Pouvez-vous reproduire le bug à l'aide d'un test automatisé ? Cela permet de vérifier plus facilement et avec moins d'erreurs si vous avez corrigé le bug ou non.

Isoler le bug signifie trouver approximativement l'origine du bug. Par exemple, si vous désactivez certains modules de votre programme en commentant le code qui les utilise, le bug apparaît-il toujours ? Pouvez-vous déterminer quels modules sont nécessaires pour déclencher le bug ? Vous pouvez également isoler en utilisant le contrôle de version : le bug existe-t-il dans un commit précédent ? Si c'est le cas, qu'en est-il d'un commit encore plus ancien ? Pouvez-vous trouver le commit qui a introduit le bug ? Si vous pouvez trouver le commit qui a introduit le bug, et que ce commit est suffisamment petit, vous avez considérablement réduit la quantité de code que vous devez examiner.

Enfin, une fois que vous avez reproduit et isolé le bug, il est temps de le debugger : voir ce qui se passe et comprendre pourquoi. Vous pouvez le faire à l'aide d'instructions d'impression :

```
printf("size : %u, capacity : %u\n", size, capacity);
```

Cependant, les instructions d'impression ne sont pas pratiques. Vous devez écrire potentiellement beaucoup d'instructions, surtout si vous voulez voir les valeurs d'une structure de données ou d'un objet. Il se peut même que vous ne puissiez pas voir les valeurs d'un objet s'il s'agit de membres privés, auquel cas vous devez ajouter une méthode à l'objet uniquement pour imprimer certains de ses membres privés. Vous devez également supprimer manuellement ces instructions après avoir corrigé le bug. En outre, si vous vous rendez compte pendant l'exécution du programme que vous avez oublié d'imprimer une valeur spécifique, l'utilisation d'impressions vous oblige à arrêter le programme, à ajouter une impression et à l'exécuter à nouveau, ce qui

est lent.

Au lieu d'utiliser des instructions d'impression, utilisez un outil conçu pour le debugging : un debugger !

Debugger

Un debugger est un outil, généralement intégré à un IDE, qui vous permet d'interrompre l'exécution d'un programme où vous le souhaitez, d'inspecter l'état du programme et même de le modifier, et, d'une manière générale, de voir ce que fait réellement un morceau de code sans avoir à le modifier.

Une remarque sur les debuggers et les outils en général : certaines personnes pensent que le fait de ne pas utiliser un outil et de faire les choses "à la dure" les rendent meilleurs ingénieurs. C'est complètement faux, c'est l'inverse : connaître les outils disponibles et les utiliser correctement est essentiel pour être un bon ingénieur. Tout comme vous ignorez les personnes qui vous disent de ne pas prendre une lampe de poche et une bouteille d'eau lorsque vous partez en randonnée dans une grotte, ignorez les personnes qui vous disent de ne pas utiliser un debugger ou tout autre outil que vous jugez utile.

Les debuggers fonctionnent également pour les logiciels qui s'exécutent sur d'autres machines, comme un serveur, à condition que vous puissiez y lancer un outil de debugging, vous pouvez exécuter le debugger graphique sur votre machine pour debugger un programme sur une machine distante. Il existe également des debuggers en ligne de commande, tels que `jdb` pour Java ou `pdb` pour Python, bien qu'ils ne soient pas aussi pratiques car vous devez entrer manuellement des commandes pour, par exemple, voir quelles sont les valeurs des variables.

Le seul prérequis des debuggers est un fichier contenant des *symboles de debugging* : le lien entre le code source et le code compilé. En d'autres termes, lorsque le programme exécute une ligne spécifique de code assembleur, de quelle ligne s'agit-il dans le code source ? Quelles variables existent à ce moment-là et dans quels registres du CPU se trouvent-elles ? Cela n'est bien sûr pas nécessaire pour les langages interprétés tels que Python, pour lesquels vous disposez de toute façon du code source. Il est techniquement possible d'utiliser un debugger sans symboles de debugging, mais il faut alors comprendre soi-même comment les détails de bas niveau correspondent aux concepts de haut niveau, ce qui est fastidieux.

Examinons cinq questions clés que vous pouvez vous poser pendant le debugging et comment vous pouvez y répondre à l'aide d'un debugger.

Le programme atteint-il cette ligne ? Vous vous demandez peut-être si le bug se déclenche lors de l'exécution d'une ligne de code particulière. Pour répondre à cette question, utilisez un *point d'arrêt* ("breakpoint"), ce que vous pouvez généralement faire en cliquant avec le bouton droit de la souris sur une ligne et en sélectionnant "ajouter un point d'arrêt" dans le menu contextuel. Une fois que vous avez ajouté un point d'arrêt, exécutez le programme en mode debugging et l'exécution s'interrompra lorsque cette ligne sera atteinte. Les debuggers permettent généralement de définir des points d'arrêt plus avancés, tels que "ne s'arrêter que si une certaine condition est remplie" ou "s'arrêter une fois toutes les N fois que cette ligne est exécutée".

Vous pouvez même utiliser des points d'arrêt pour imprimer des choses au lieu de mettre l'exécution en pause. Attendez, ne venons-nous pas de dire que les impressions n'étaient pas une bonne idée ? La raison pour laquelle l'impression des points d'arrêt est préférable est que vous n'avez pas besoin de modifier le code, et donc pas besoin de revenir sur ces modifications plus tard, et vous pouvez changer ce qui est imprimé et où pendant que le programme est en cours d'exécution.

Quelle est la valeur de cette variable ? Vous avez ajouté un point d'arrêt, le programme s'y est rendu et a interrompu l'exécution. Dans un IDE, vous pouvez généralement passer votre souris sur une variable du code source pour voir sa valeur lorsque le programme est en pause, ainsi que pour afficher une liste de toutes les variables locales. Il s'agit notamment des valeurs contenues dans les structures de données telles que les tableaux et les membres privés des classes. Vous pouvez également exécuter du code pour poser des questions, comme `n % 12 == 0` pour voir si `n` est actuellement un multiple de 12, ou `IsPrime(n)` si vous avez une méthode `IsPrime` et que vous voulez voir ce qu'elle retourne pour `n`.

Et si ... ? Vous constatez qu'une variable a une valeur à laquelle vous ne vous attendiez pas et vous vous demandez si le bug disparaîtrait si elle avait une valeur différente. Bonne nouvelle : vous pouvez essayer exactement cela. Les debuggers ont généralement une sorte de fenêtre de "commande" où vous pouvez écrire des lignes telles que `n = 0` pour changer la valeur de `n`, ou `lst.add("x")` pour ajouter `"x"` à la liste `lst`.

Qu'est-ce qui va se passer ensuite ? L'état du programme semble correct, mais c'est peut-être la ligne suivante qui pose problème. Les commandes d'exécution pas à pas ("step") vous permettent d'exécuter le programme étape par étape, de sorte que vous pouvez examiner l'état du programme après l'exécution de chaque étape pour voir si quelque chose ne va pas. Les debuggers fournissent typiquement une option pour entrer dans n'importe quelle méthode appelée, une pour passer à la ligne suivante au lieu d'entrer dans une méthode, et une pour finir l'exécution de la méthode en cours. Certains debuggers disposent d'outils supplémentaires, tels qu'une exécution pas à pas "intelligente" qui ne rentre que dans les méthodes de plus de quelques lignes. En fonction du langage de programmation et du debugger, vous pouvez même changer le pointeur d'instruction pour la ligne de code de votre choix et modifier du code à la volée sans avoir à interrompre l'exécution du programme.

Notez que vous n'êtes pas obligé d'utiliser la souris pour lancer le programme et l'exécuter pas à pas : les debuggers disposent généralement de raccourcis clavier, tels que F5 pour lancer, F9 pour exécuter pas à pas, etc. et vous pouvez généralement les personnaliser. Ainsi, votre flux de travail consistera à appuyer sur une touche pour exécuter, à regarder l'état du programme après l'atteinte du point d'arrêt, puis à appuyer sur une touche pour avancer, à regarder l'état du programme, à avancer à nouveau, et ainsi de suite.

Comment sommes-nous arrivés ici ? Vous avez placé un point d'arrêt dans une méthode, le programme l'a atteint et a interrompu l'exécution, mais comment le programme a-t-il atteint cette ligne ? La *pile d'appel* est là pour répondre à cette question : vous pouvez voir quelle méthode vous a appelé, quelle méthode a appelé celle-ci, et ainsi de suite. De plus, vous pouvez voir l'état du programme au moment de cet appel. Par exemple, vous pouvez voir quelles valeurs ont été données comme arguments à la méthode qui a appelé la méthode qui a appelé la méthode dans laquelle vous vous trouvez actuellement.

Qu'est-ce qui s'est passé pour provoquer un crash ? Ne serait-il pas agréable de pouvoir voir l'état du programme au moment où un crash s'est produit sur la machine d'un utilisateur ? Eh bien, c'est possible ! Le système d'exploitation peut générer un *crash dump* qui contient l'état du programme lorsqu'il se plante, et vous pouvez charger ce crash dump dans un debugger, ainsi que le code source du programme et les symboles de debugging, pour voir quel était l'état du programme. C'est ce qui se passe lorsque vous cliquez sur "Signaler le problème à Microsoft" après le plantage de votre document Word. Notez que cela ne fonctionne que pour les plantages, et non pour les bugs du type "le comportement n'est pas celui auquel je m'attendais", car il n'existe aucun moyen automatisé de détecter ce type de comportement inattendu.

Le debugging en pratique

Lorsque vous utilisez un debugger pour trouver la cause principale d'un bug, vous ajoutez un point d'arrêt, exécutez le programme jusqu'à ce que l'exécution s'interrompe pour inspecter l'état, et apportez éventuellement des modifications au programme en fonction de vos observations, puis répétez le cycle jusqu'à ce que vous ayez trouvé la cause principale.

Cependant, il arrive que vous ne puissiez pas résoudre le problème tout seul et que vous ayez besoin d'aide. C'est tout à fait normal, surtout si vous débugz un code écrit par quelqu'un d'autre. Dans ce cas, vous pouvez demander de l'aide à un collègue, voire poster sur un forum tel que StackOverflow. Venez préparés, afin de pouvoir aider les autres à vous aider. Quels sont les symptômes ? Avez-vous un moyen facile de reproduire le bug ? Qu'avez-vous essayé ?

Parfois, vous commencez à expliquer votre problème à un collègue et, au cours de votre explication, une ampoule s'allume dans votre tête : voilà le problème ! Votre collègue vous regarde alors, heureux que vous ayez trouvé la solution, mais un peu agacé d'être interrompu. Pour éviter cette situation, commencez par la technique du "*rubber ducking*" (de l'anglais "rubber duck", canard en caoutchouc) : expliquez votre problème à un canard en caoutchouc, ou à tout autre objet inanimé. Parlez à l'objet comme s'il s'agissait d'une personne, en expliquant votre problème. La raison pour laquelle cela fonctionne est que lorsque nous

expliquons un problème à quelqu'un d'autre, nous expliquons généralement ce qui se passe réellement, plutôt que ce que nous souhaiterions qu'il se passe. Si vous ne comprenez pas le problème en l'expliquant à un canard, vous aurez au moins répété comment vous expliquerez le bug, et vous serez en mesure de mieux l'expliquer à un humain.

Il n'y a qu'une seule façon de s'améliorer en matière de debugging : s'entraîner. La prochaine fois que vous rencontrerez un bug, utilisez un debugger. Les premières fois, vous serez peut-être plus lent que si vous n'en aviez pas, mais ensuite votre productivité montera en flèche.

Exercice Regardez le code de `binarytree.py` le dossier d'exercices pendant le cours. Tout d'abord, lancez-le. Il se plante ! Utilisez un debugger pour ajouter des points d'arrêt et examiner ce qui se passe jusqu'à ce que vous en trouviez la cause et que vous corrigiez les bugs. Notez que le plantage n'est pas le seul bug.

Tout d'abord, il n'y a pas de cas de base pour la méthode récursive qui construit un arbre, vous devez donc en ajouter un pour gérer le cas `len(lst) == 0`.

Deuxièmement, les limites des sous-listes ne sont pas correctes : elles devraient être `0..mid` et `mid+1..len(lst)`.

Finalement, le constructeur utilise `l` deux fois, alors qu'il devrait mettre `right` à `r`. Cela ne se serait pas produit si le code avait utilisé de meilleurs noms !

Qu'est-ce qui rend un code débuggable ?

L'inventeur Charles Babbage a dit un jour à propos d'une de ses machines "*A deux reprises, on m'a demandé : Monsieur Babbage, si vous introduisez de mauvais chiffres dans la machine, les bonnes réponses en sortiront-elles ? Je ne suis pas en mesure d'appréhender correctement le genre de confusion d'idées qui pourrait provoquer une telle question.*"

Il devrait être clair qu'une entrée erronée ne peut pas conduire à une sortie correcte. Malheureusement, il arrive souvent qu'une entrée erronée conduise à une sortie erronée *silencieusement* : il n'y a aucune indication que quelque chose d'erroné s'est produit. Il est donc difficile de trouver la cause principale d'un bug. Si vous remarquez que quelque chose ne va pas, est-ce que c'est parce que l'opération précédente a mal fonctionné ? Ou parce que l'opération effectuée 200 lignes de code plus tôt a produit un résultat erroné qui est passé inaperçu jusqu'à ce qu'il cause finalement un problème ?

Margaret Hamilton, qui a inventé le terme "génie logiciel" pour donner une légitimité aux logiciels sur la base de son expérience du développement de logiciels pour les premières missions de la NASA, disait à propos de ses premiers travaux : "nous avons appris à passer plus de temps en amont [...] afin de ne pas perdre tout ce temps à debugger".

Nous verrons trois méthodes pour rendre le code débuggable : la programmation défensive, le logging et le code seulement pour debugging.

Programmation défensive

Les bugs vous attaquent et vous devez défendre votre code et vos données ! C'est l'idée derrière la *programmation défensive* : s'assurer que les problèmes qui surviennent en dehors de votre code ne peuvent pas corrompre votre état ou vous faire renvoyer des déchets. Ces problèmes peuvent être, par exemple, des bugs de logiciels, des entrées non valables saisies par des humains, des tentatives délibérées d'attaque du logiciel par des humains, ou même une corruption du matériel.

Il peut sembler étrange de s'inquiéter de la corruption du matériel, mais cela arrive plus souvent qu'on ne le pense ; Par exemple, une simple inversion de bit peut transformer `microsoft.com` en `microsfmt.com`, puisque `o` est généralement encodé comme `01101101` en binaire, qui peut se transformer en `01101111`, l'encodage pour `m`. Un logiciel qui a l'intention de communiquer avec `microsoft.com` pourrait donc finir par recevoir

des données de `microsfmt.com`, qui peuvent être sans rapport, spécifiquement préparées pour des attaques, ou une expérience pour voir dans quelle mesure cela se produit.

Au lieu de produire silencieusement des déchets, le code doit échouer le plus tôt possible. Plus une défaillance est proche de sa cause principale, plus il est facile de la debugger.

Les *assertions* sont l'outil clé pour éviter les échecs précoces. Une assertion est un moyen de vérifier si le code se comporte réellement de la manière dont l'ingénieur qui l'a écrit pense qu'il devrait le faire.

Par exemple, si un morceau de code trouve l'index d'une valeur dans un tableau, un ingénieur pourrait écrire ce qui suit immédiatement après avoir trouvé l'index :

```
if (array[index] != val) {
  throw new AssertionError(...);
}
```

Si cette vérification échoue, il doit y avoir un bug dans le code qui trouve `index`. L'étape "isoler le bug" du debugging est déjà réalisée par l'assertion.

Mais que faire en cas d'échec de la vérification ? Abandonner le programme ? Cela dépend du programme. En général, s'il existe un moyen de couper ce qui a causé le problème du reste du programme, c'est une meilleure idée. Par exemple, la *requête* actuelle pourrait échouer. Il peut s'agir d'une requête adressée à un serveur, par exemple, ou d'une opération déclenchée par un utilisateur qui appuie sur un bouton. Le logiciel peut alors afficher un message indiquant qu'une erreur s'est produite, ce qui permet à l'utilisateur de réessayer ou de faire autre chose. Cependant, certains échecs sont si graves qu'il n'est pas raisonnable de continuer. Par exemple, si le code qui charge la configuration du logiciel échoue à une assertion, il est inutile de continuer sans la configuration.

Une assertion qui doit être maintenue lors de l'appel d'une méthode est une *précondition* de cette méthode. Par exemple, une méthode `int pickOne(int[] array)` qui renvoie un des éléments du tableau a probablement comme précondition "le tableau n'est pas `null` et a au moins `1` élément". Le début de la méthode pourrait ressembler à ceci :

```
int pickOne(int[] array) {
  if (array == null || array.length == 0) {
    throw new IllegalArgumentException(...);
  }
  // ...
}
```

Si un morceau de code appelle `pickOne` avec un tableau nul ou vide, la méthode lèvera une `IllegalArgumentException`.

Pourquoi prendre la peine de vérifier cela explicitement alors que la méthode échouerait de toute façon si le tableau était nul ou vide, puisque la méthode déréférencera le tableau et indexera son contenu ? Le type d'exception levée indique *qui est responsable*. Si vous appelez `pickOne` et obtenez une `NullPointerException`, il est raisonnable de supposer que `pickOne` a un bug, car cette exception indique que le code de `pickOne` croit qu'une référence donnée est non nulle, puisqu'il la déréférence, alors qu'en pratique la référence est nulle. Cependant, si vous appelez `pickOne` et que vous obtenez une `IllegalArgumentException`, il est raisonnable de supposer que votre code a un bug, car cette exception indique que vous avez transmis un argument dont la valeur est illégale. Ainsi, le type d'exception vous aide à trouver l'endroit où se trouve le bug.

Une assertion qui doit être maintenue lors du retour d'une méthode est une *postcondition* de cette méthode. Dans notre exemple, la postcondition est "la valeur retournée est une valeur du tableau", ce qui est exactement ce que vous appelez `pickOne` pour obtenir. Si `pickOne` renvoie une valeur qui n'est pas dans le tableau, le code qui l'appelle produira n'importe quoi, parce que le code s'attendait à ce que `pickOne` satisfasse son contrat, ce qui n'a pas été le cas. Il n'est pas raisonnable d'insérer des assertions à chaque fois que l'on appelle une méthode pour vérifier que la valeur retournée est acceptable ; c'est plutôt à la méthode de vérifier qu'elle honore sa postcondition. Par exemple, la fin de `pickOne` peut ressembler à ceci :

```
int result = ...
if (!contains(array, result)) {
  throw new AssertionError(...);
}
return result;
```

De cette façon, si `result` a été calculé de façon incorrecte, le code échouera avant de corrompre le reste du programme avec une valeur invalide.

Certaines assertions sont à la fois des pré-conditions et des post-conditions pour les méthodes d'un objet : les *invariants d'objet*. Un invariant est une condition qui reste toujours valable de l'extérieur. Il peut être cassé au cours d'une opération, à condition que cela ne soit pas visible pour le monde extérieur car il est rétabli avant la fin de l'opération. Par exemple, considérons les champs suivants pour une pile :

```
classe Stack {
  private int[] values;
  private int top; // sommet de la pile dans `values`
}
```

Un invariant de cette classe est `-1 <= top < values.length`, c'est-à-dire que soit `top == -1`, ce qui signifie que la pile est vide, soit `top` pointe vers la valeur supérieure de la pile à l'intérieur du tableau. Une façon de vérifier les invariants est d'écrire une méthode `assertInvariants` qui les affirme et de l'appeler à la fin du constructeur et au début et à la fin de chaque méthode. Toutes les méthodes de la classe doivent préserver l'invariant afin qu'elles puissent également compter sur son maintien lorsqu'elles sont appelées. C'est une des raisons pour lesquelles l'encapsulation est si utile : si n'importe qui pouvait modifier `values` ou `top` sans passer par les méthodes de `Stack`, il n'y aurait aucun moyen de faire respecter cet invariant.

Considérons la méthode Java suivante :

```
void setWords(List<String> words) {
  this.words = words;
}
```

Cela semble trivialement correct, et pourtant, on peut l'utiliser de la manière suivante :

```
setWords(badWords);
badWords.add("Pas bien !");
```

Oups ! Maintenant l'état de l'objet qui contient `mots` a été modifié de l'extérieur de l'objet, ce qui pourrait briser les invariants que l'objet est supposé avoir.

Pour éviter cela et protéger l'état d'une personne, des *copies de données* sont nécessaires lorsqu'il s'agit de valeurs mutables :

```
void setWords(List<String> words) {
  this.words = new ArrayList<>(words);
}
```

De cette manière, aucun changement ne peut intervenir dans l'état de l'objet à son insu. Il en va de même pour les lectures, `return this.words` étant problématique et `return new ArrayList<>(this.words)` évitant le problème.

Encore mieux, si possible, l'objet pourrait utiliser une liste *immuable* pour les `mots`, comme la `List[A]` par défaut de Scala. Cela résout le problème sans nécessiter de copies de données, qui ralentissent le code.

Exercice Consultez le code de `stack.py` le dossier d'exercices pendant le cours, qui contient une classe `IntStack` et un exemple d'utilisation. Ajoutez du code à `IntStack` pour détecter les problèmes à temps, et corrigez les bugs que vous trouvez dans le processus. Tout d'abord, regardez ce que le constructeur doit faire.

Une fois cela fait, ajoutez un invariant et utilisez-le, ainsi qu'une condition préalable pour `push`. Ensuite, corrigez les bugs que vous trouvez.

Tout d'abord, le constructeur doit rejeter si `maxSize < 0`, puisque c'est invalide.

Deuxièmement, la pile doit avoir l'invariant `-1 <= top < len(values)`, comme nous l'avons vu plus haut.

Après avoir ajouté cet invariant, notez que `top -= 1` dans `pop` peut briser l'invariant puisqu'il est utilisé inconditionnellement. Il en va de même pour `top += 1` dans `push`. Elles doivent être modifiées pour ne modifier que `top` si nécessaire.

Pour permettre aux utilisateurs de `IntStack` d'appeler `push` en toute sécurité, on peut exposer une méthode `def isFull(self)`, et l'utiliser comme précondition de `push`.

Logging

Qu'est-ce qui s'est passé en production ? S'il y a eu un crash, vous pouvez obtenir un crash dump et l'ouvrir dans un debugger. Mais s'il s'agit d'un bug plus subtil où le résultat ”semble erroné” pour un humain, comment pouvez-vous savoir ce qui s'est passé pour produire ce résultat ?

C'est là qu'intervient le *logging* : l'enregistrement de ce qui se passe pendant l'exécution afin que le log puisse être lu au cas où quelque chose se serait mal passé. Une façon simple d'enregistrer les données est d'imprimer des instructions :

```
print("Demande : " + demande)
print("État : " + état)
```

Cette méthode fonctionne, mais n'est pas idéale pour de multiples raisons. Tout d'abord, le développeur doit choisir ce qu'il veut enregistrer au moment de l'écriture du programme. Par exemple, si l'enregistrement de chaque appel de fonction est considéré comme trop lourd pour un fonctionnement normal, le log des appels de fonction ne sera jamais enregistré, même si, dans certains cas, il pourrait être utile. Deuxièmement, le développeur doit choisir le mode d'enregistrement au moment de l'écriture du programme. Le log doit-il être imprimé sur la console ? Enregistré dans un fichier ? Les deux ? Certains événements devraient-ils envoyer un courriel aux développeurs ? Troisièmement, l'utilisation de la même fonction d'impression pour chaque log ne permet pas de distinguer ce qui est important de ce qui ne l'est pas.

Au lieu d'utiliser une fonction d'impression spécifique, les frameworks de logging fournissent l'abstraction d'un log avec plusieurs niveaux d'importance, comme le module de logging de Python :

```
logging.debug("Détail")
logging.info("Information")
logging.warning("Avertissement")
logging.error("Erreur")
logging.critical("Rien ne va plus")
```

Le nombre de niveaux de logging et leur nom changent dans chaque cadre de logging, mais le fait est qu'il y en a plusieurs et qu'ils n'impliquent rien quant à la destination du log.

Les ingénieurs peuvent écrire des appels de logging pour tout ce qu'ils pensent être utile, en utilisant le niveau de logging approprié, et décider *plus tard* ce qu'il faut logger et où le faire. Par exemple, par défaut, les logs ”debug” et ”info” peuvent ne même pas être stockés, car ils sont trop détaillés et pas assez importants. Mais s'il y a actuellement un bug subtil dans la production, on peut leur permettre de voir exactement ce qui se passe, sans avoir à redémarrer le programme. Il peut être judicieux d'enregistrer les erreurs en envoyant un courriel aux développeurs, mais s'il y a beaucoup d'erreurs, les développeurs en sont déjà conscients, ils peuvent décider d'enregistrer temporairement les erreurs dans un fichier, sans avoir à redémarrer le programme.

Il est important de penser à la protection de la vie privée lors de l'écriture du code de logging. L'enregistrement du contenu intégral de chaque requête, par exemple, peut conduire à l'enregistrement de mots de passe en texte clair pour une fonction de ”création d'utilisateur”. Si ce log est conservé dans un fichier et que le serveur est piraté, les attaquants disposeront d'un log détaillé de tous les mots de passe jamais définis.

Code seulement pour debugging

Qu'en est-il des contrôles de programmation défensifs et des logs qui sont trop lents pour être raisonnablement activés en production ? Par exemple, si un graphe a pour invariant "aucun noeud n'a plus de 2 arêtes", mais que le graphe a typiquement des millions de noeuds, que faire ?

C'est là qu'intervient le code *seulement pour debugging*. Les langages de programmation, leurs moteurs d'exécution et les frameworks offrent généralement des moyens d'exécuter le code uniquement en mode debug, par exemple lors de l'exécution de tests automatisés.

Par exemple, en Python, on peut écrire un bloc `if __debug__:`, qui ne s'exécutera que lorsque le code n'est pas optimisé.

Il est important de vérifier ce que signifient les termes "debugging" et "optimisé" dans un contexte donné. Par exemple, en Python, `__debug__` est `True` par défaut, à moins que l'interpréteur ne soit lancé avec le commutateur `-O`, pour "optimize". En Java, les assertions sont du code de debugging uniquement, mais elles sont désactivées par défaut, et peuvent être activées avec le commutateur `-ea`. Scala a plusieurs niveaux de code de debugging seulement qui sont tous activés par défaut mais qui peuvent être désactivés sélectivement avec le commutateur `-Xelide-below`.

Plus important encore, avant d'écrire du code de debugging uniquement, réfléchissez bien à ce qu'il est "raisonnable" d'activer en production compte tenu de la charge de travail que vous avez. Passer une demi-seconde à vérifier un invariant est acceptable dans un morceau de code qui prendra des secondes à s'exécuter parce qu'il effectue de nombreuses requêtes web, par exemple, même si une demi-seconde représente beaucoup de temps de manière absolue.

Gardez à l'esprit ce que Tony Hoare, l'un des pionniers de l'informatique et en particulier des langages de programmation et de la vérification, a dit un jour dans son "Hints on Programming Language Design" : "*Il est absurde de procéder à des contrôles de sécurité élaborés sur des cycles de debugging, alors qu'aucune confiance n'est accordée aux résultats, puis de les supprimer lors de la production, lorsqu'un résultat erroné peut s'avérer coûteux ou désastreux. Que penserait-on d'un passionné de voile qui porte son gilet de sauvetage lorsqu'il s'entraîne sur la terre ferme, mais l'enlève dès qu'il prend la mer ?*".

Résumé

Dans ce cours, vous avez appris :

- Comment écrire un code lisible : noms, formatage, commentaires, conventions
- Comment debugger un code : reproduire des bugs, utiliser un debugger
- Comment écrire du code débuggable : programmation défensive, logging, code uniquement pour debugging

Vous pouvez maintenant consulter les exercices !

Performance

Produire des résultats corrects n'est pas le seul objectif d'un logiciel. Les résultats doivent être produits dans un délai raisonnable, sinon les utilisateurs risquent de ne pas attendre la réponse et d'abandonner. Accélérer les logiciels peut considérablement améliorer l'expérience utilisateur, par exemple en produisant des jeux vidéo plus jolis. La vitesse peut également avoir des conséquences plus directes, comme l'a découvert Google lorsqu'il a constaté qu'un délai supplémentaire de 0,5 seconde pour afficher les résultats de recherche entraînait une perte de 20 % des revenus publicitaires (http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html).

Objectifs

À l'issue de cette leçon, vous devriez être capable de :

- Définir des objectifs de performance

- Créer des benchmarks pour mesurer la performance
- Concevoir et implémenter des systèmes rapides
- Connaître des compromis de performance courants

Comment définir les objectifs de performance ?

De nombreux facteurs doivent être pris en compte lors de la définition des objectifs. Quelle unité utiliser ? Quelle mesure ? Quel degré de variabilité est acceptable ? Quel doit être le degré de formalité de l'objectif ? Examinons chacun de ces points tour à tour.

Selon l'opération que vous mesurez, l'unité de mesure peut varier considérablement. La restauration des données à partir de bandes magnétiques utilisées pour le stockage à long terme peut prendre des heures si un camion doit se rendre au lieu de stockage, récupérer les bandes, les apporter au centre de données et les copier. Dans ce cas, le fait que l'opération prenne 3 ou 10 heures a probablement de l'importance, mais le fait qu'elle dure 3 heures et 10 minutes ou 3 heures et 12 minutes n'a probablement pas d'importance. La sauvegarde d'une base de données d'un serveur à un autre peut prendre quelques minutes. Là encore, le fait que cela prenne 4 ou 15 minutes a de l'importance, mais le fait que cela prenne 4 minutes et 3 secondes ou 4 minutes et 10 secondes n'a probablement pas d'importance. Nous pouvons descendre à des unités de plus en plus petites. Le téléchargement d'un fichier volumineux vers un serveur se mesure en secondes. Le chargement d'une page Web se mesure en millisecondes. La requête effectuée d'un serveur à un autre au sein d'un centre de données pour servir cette page Web prend quelques microsecondes. Les calculs sur un seul serveur prennent des nanosecondes, car les processeurs modernes effectuent des milliards d'opérations par seconde.

Si vous ne savez pas exactement combien de temps dure une nanoseconde, consultez l'exemple intéressant de Grace Hopper (la vidéo comporte des sous-titres en français transcrits manuellement). Grace Hopper était une pionnière dans le domaine des langages de programmation. Avec son équipe, elle a conçu COBOL, le premier langage de programmation utilisant des commandes similaires à celles de la langue anglaise. Comme elle le rappelait : ”J'avais un compilateur qui fonctionnait, mais personne ne voulait s'en servir. On me disait que les ordinateurs ne pouvaient faire que des calculs arithmétiques”. Heureusement, elle a ignoré les détracteurs, sinon nous ne programmerions pas dans des langages de haut niveau aujourd'hui !

Une fois que nous avons défini une unité de temps, il existe deux indicateurs clés pour mesurer les performances. Le débit est le nombre de requêtes traitées par unité de temps, et la latence est le temps nécessaire pour traiter une requête. Dans un système qui traite une requête à la fois, le débit est l'inverse de la latence moyenne. Dans un système qui traite plusieurs requêtes à la fois, cette corrélation n'existe pas. Par exemple, un coeur d'un processeur double coeur peut traiter une requête en 10 secondes tandis qu'un autre coeur a traité 3 requêtes pendant ce temps. Le débit est donc de 4 requêtes par 10 secondes, mais la latence varie selon les requêtes. Nous ne voulons souvent prendre en compte que les requêtes réussies, car il est rarement utile de traiter rapidement une réponse d'erreur. Le débit des requêtes réussies est appelé ”goodput”, combinaison de ”good” (bon) et ”throughput” (débit).

Il existe bien sûr d'autres mesures de performance que le temps. La quantité de mémoire utilisée par un programme peut avoir son importance, tout comme l'espace de stockage. Par exemple, un programme qui s'exécute rapidement mais qui nécessite plus de RAM que votre ordinateur n'en dispose ne vous est d'aucune utilité, et vous feriez mieux d'utiliser un programme plus lent qui peut réellement fonctionner sur votre machine. L'efficacité énergétique est également un indicateur important, qui peut se traduire par l'autonomie de la batterie des appareils mobiles tels que les smartphones. Les utilisateurs ne sont pas susceptibles de vouloir un programme légèrement plus rapide qui épuise la batterie de leur téléphone.

En théorie, nous aimerions que les performances soient les mêmes pour toutes les demandes et tous les utilisateurs, mais dans la pratique, c'est rarement le cas. Certaines requêtes nécessitent fondamentalement plus de travail que d'autres, comme commander une pizza avec 10 garnitures par rapport à une pizza avec seulement du fromage et des tomates. Certaines requêtes passent par des chemins de code différents parce que les ingénieurs ont fait des compromis, comme commander une pizza sans gluten qui prend plus de temps car au lieu de disposer d'une cuisine séparée sans gluten, le restaurant doit nettoyer sa seule cuisine. Certaines requêtes sont en concurrence avec d'autres pour obtenir des ressources, comme commander une pizza juste

après qu'une grande table ait passé une commande dans un restaurant qui ne dispose que d'un seul four à pizza.

Il existe de nombreux sous-indicateurs couramment utilisés pour traiter les statistiques : moyenne, médiane, 99e centile, et même les performances pour la première demande uniquement, c'est-à-dire en incluant le temps nécessaire au démarrage du système. Le choix de la mesure à cibler dépend de votre cas d'utilisation et doit être défini en collaboration avec les clients. Les percentiles élevés, tels que le 99e percentile, sont pertinents pour les grands systèmes dans lesquels de nombreux composants sont impliqués dans chaque requête, comme le décrit Google dans son article "Tail at Scale". Certains systèmes s'intéressent en fait au "100e centile", c'est-à-dire au pire temps d'exécution possible, car les performances peuvent être un objectif de sécurité. Par exemple, lorsqu'un pilote d'avion donne un ordre, l'avion doit réagir rapidement. Il serait catastrophique que l'avion exécute la commande plusieurs minutes après que le pilote l'ait donnée, même si l'exécution est sémantiquement correcte. Pour de tels systèmes, le temps d'exécution dans le pire des cas est généralement surestimé par une analyse manuelle ou automatisée. Certains systèmes ne tolèrent aucune variabilité, c'est-à-dire qu'ils exigent une exécution en temps constant. C'est généralement le cas des opérations cryptographiques, dont le timing ne doit révéler aucune information sur les clés secrètes ou les mots de passe.

Même si les utilisateurs peuvent vous donner leur perception subjective de ce qui est "rapide" et de ce qui est "lent", les clients ont besoin de définitions formelles à inscrire dans un contrat. Les performances sont généralement définies en termes d'indicateurs de niveau de service ou "SLI" ("Service Level Indicator"), qui sont des mesures telles que la "latence médiane des requêtes", les objectifs de niveau de service ou "SLO" ("Service Level Objective"), qui définissent des buts pour ces mesures, tels que "moins de 50 ms", et les accords de niveau de service ou "SLA" ("Service Level Agreement"), qui ajoutent des conséquences aux objectifs, telles que "ou nous remboursons 20 % des dépenses du client pour le mois". Par exemple, la chaîne de pizzerias Domino's a mené une campagne marketing affirmant que votre pizza serait gratuite si elle n'arrivait pas dans les 30 minutes suivant votre commande. Le délai de réception de la pizza est le SLI, 30 minutes est le SLO, et l'offre d'une pizza gratuite est le SLA. Ce SLA spécifique s'est avéré être une mauvaise idée, car il a entraîné une augmentation des accidents de voiture pour leurs chauffeurs. Les SLA peuvent même être codifiés dans la loi, comme l'ordonnance suisse sur la Poste (https://www.fedlex.admin.ch/eli/cc/2012/586/fr#art_32) qui fixe des délais spécifiques pour les lettres et les colis, les pourcentages de livraisons qui doivent respecter ces délais et des instructions à la Poste suisse pour mandater un observateur externe pour ces mesures.

Comment mesurer la performance ?

Le processus de mesure des performances est appelé "benchmarking" et consiste en une forme de test des performances. Comme les tests, les benchmarks sont généralement automatisés et posent des problèmes tels que la manière d'isoler un composant spécifique plutôt que de toujours évaluer l'ensemble du système. Les benchmarks pour des fonctions individuelles sont généralement appelés "microbenchmarks", tandis que les benchmarks pour des systèmes entiers sont des benchmarks "de bout en bout".

En théorie, le benchmarking est un processus simple : démarrer un chronomètre, effectuer une action, arrêter le chronomètre. Mais dans la pratique, il existe de nombreux aspects délicats. Quelle est la précision du chronomètre ? Quelle est la résolution du chronomètre ? Dans quelle mesure l'opération est-elle variable ? Combien de mesures faut-il prendre pour que les résultats soient valables ? Faut-il écarter les valeurs aberrantes ? Quelle est la meilleure API sur chaque système d'exploitation pour chronométrer les actions ? Comment éviter les effets de démarrage ? Et les effets de mise en cache ? Les optimisations du compilateur invalident-elles le benchmark en modifiant le code ?

Vous ne devriez jamais écrire votre propre code de benchmarking. À moins que vous ne deveniez un jour un expert en benchmarking, auquel cas vous saurez que vous pouvez enfreindre cette règle. Utilisez plutôt un framework de benchmarking existant, tel que JMH pour Java, BenchmarkDotNet pour .NET ou pytest-benchmark pour Python.

Le processus général de benchmarking est le suivant :

1. Choisissez le code que vous souhaitez benchmarker

2. Choisissez la *référence* par rapport à laquelle vous souhaitez effectuer le benchmark, qui peut être un autre morceau de code ou un niveau de performance absolu
3. Choisissez les entrées pour votre benchmark, en fonction des entrées réalistes ou courantes pour votre logiciel
4. Écrivez et exécutez votre benchmark

Même lorsque vous utilisez un cadre de benchmarking, vous devez garder à l'esprit certains faits importants. Tout d'abord, les optimisations ne sont pas toujours vos alliées. Prenons l'exemple de la fonction Python suivante :

```
lst = ...
def append():
    lst + [0]
```

Si vous évaluez la performance de `append`, vous risquez de ne rien évaluer du tout, car Python pourrait remarquer que le résultat de `+` n'est pas utilisé et, comme il n'a aucun effet secondaire, ne pas l'exécuter du tout, comme si vous aviez écrit `def append(): pass`. Il est possible de contourner ce problème, par exemple en renvoyant le résultat de `+`. Mais même si vous faites cela, aviez-vous l'intention de ne comparer que `+` ? Car cette méthode crée également une liste à un élément, et le temps de cette opération sera inclus dans le résultat. Vous préférerez donc peut-être écrire ceci à la place :

```
lst = ...
lst2 = ...
def append():
    return lst + lst2
```

Il est facile d'écrire accidentellement un code de benchmark qui n'a aucun sens à cause d'un problème subtil. Demandez toujours à quelqu'un de vérifier votre code de benchmark et fournissez le code chaque fois que vous communiquez les résultats. Essayez de petites variations du code pour voir si vous obtenez de petites variations dans les résultats du benchmark. Reproduisez les résultats précédents, si vous en avez, pour vous assurer que votre configuration globale fonctionne.

Exercice Maintenant que vous savez comment effectuer un benchmark, ouvrez le fichier `benchmarking.py` dans les exercices pendant le cours et suivez les instructions du ReadMe dans le dossier.

Commencez par exécuter les benchmarks. Les résultats correspondent-ils à votre intuition ?

Ensuite, ajoutez de nouvelles méthodes de benchmark pour tester les mêmes opérations, mais avec une liste chaînée, puis exécutez les benchmarks. Une fois encore, les résultats correspondent-ils à vos attentes ?

Enfin, écrivez un benchmark dans lequel la `list` intégrée à Python est clairement plus rapide.

<détails>

Il est normal que l'ajout d'une valeur au début d'une `list` soit beaucoup plus lent, car la liste doit être copiée dans un nouveau tableau afin d'ajouter un élément au début, alors que la plupart des ajouts peuvent réutiliser le tableau existant qui n'est pas encore plein.

Une liste chaînée permet d'ajouter rapidement au début, car seuls quelques liens doivent être mis à jour.

Un benchmark simple qui est beaucoup plus rapide consiste à accéder à l'élément au milieu ou à la fin de la liste. C'est presque instantané avec une liste de tableaux, alors qu'il faut parcourir de nombreux noeuds avec une liste chaînée.

Avant d'utiliser vos nouvelles connaissances pour écrire des tonnes de benchmarks, voici quelques points importants à garder à l'esprit.

Méfiez-vous de trop vous concentrer sur des mesures intermédiaires qui peuvent être représentatives ou non de l'ensemble. Par exemple, des chercheurs ont découvert que certains ramasseurs de déchets pour Java avaient optimisé le temps passé dans le ramasse-miettes au détriment du temps nécessaire à l'exécution globale d'une application. En essayant de minimiser à tout prix le temps de GC, certains GC ont en fait augmenté le temps d'exécution global.

Méfiez-vous du "problème des petites entrées": il est facile de passer à côté d'algorithmes peu performants en effectuant des tests de performance uniquement avec de petites entrées. Par exemple, si vous triez une liste de 10 éléments, deux algorithmes de tri quelconques seront très proches l'un de l'autre, même si vous comparez un algorithme très inefficace comme le "bubble sort" ou "tri à bulles" (https://fr.wikipedia.org/wiki/Tri_%C3%A0_bulles) à un algorithme très efficace comme le "quicksort" ou "tri rapide" (https://fr.wikipedia.org/wiki/Tri_rapide).

Faites attention à la charge de travail que vous utilisez, en particulier lorsque vous mesurez des frameworks conçus pour être invoqués parallèlement aux opérations des utilisateurs. Par exemple, des chercheurs ont découvert qu'un framework réseau pouvait atteindre un débit plus élevé qu'un autre lorsqu'il exécutait une fonction réseau "no-op" qui se contentait de transférer des paquets, mais que le second était bien plus performant lorsqu'il exécutait une fonction réseau "réelle"qui inspectait le trafic et décidait quels paquets laisser passer. En effet, le framework et le code de la fonction réseau se disputaient des ressources telles que les caches du processeur.

Un autre exemple intéressant de charge de travail est ce problème dans le framework .NET. Un benchmark simple qui génère des chaînes de caractères et les place dans une carte montre d'excellentes performances pour le `HashMap` de Java par rapport au `Dictionary` de .NET, qui effectuent tous deux la même tâche. Il s'avère que la raison en est que la fonction de hachage de chaîne par défaut de Java est très différente de celle de .NET, ce qui signifie que dans le cas spécifique de chaînes très similaires, `HashMap` semble plus rapide, mais que `Dictionary` offre des performances similaires lorsqu'il utilise des chaînes aléatoires.

Enfin, n'oubliez pas qu'il n'est pas utile de comparer et d'améliorer un code qui est déjà suffisamment rapide. Bien que les micro-benchmarks puissent être addictifs, vous devriez consacrer votre temps à des tâches qui ont un impact sur les utilisateurs finaux. Accélérer une opération de 100 ms à 80 ms peut sembler formidable, mais si les utilisateurs se soucient uniquement que la tâche prenne moins d'une seconde, ils auraient probablement préféré que vous consacriez votre temps à ajouter des fonctionnalités ou à corriger des bugs.

Comment concevoir des systèmes rapides ?

La conception de systèmes rapides est une forme d'ingénierie, tout comme la conception de systèmes corrects et sécurisés relève de l'ingénierie. Il est important de ne pas aborder les performances sous l'angle de "trucs et astuces". Connaître les techniques de manipulation de bits de bas niveau peut parfois s'avérer utile, mais c'est la conception d'un système qui a le plus grand impact sur ses performances. L'idée de "performance = fonctionnalité" est une mauvaise approche. Il n'est pas facile d'"ajouter de la performance" à un système existant, de la même manière qu'il est difficile d'"ajouter de l'exactitude" ou d'"ajouter de la sécurité" à un système incorrect ou non sécurisé.

Vous voulez que votre système se situe à la frontière de Pareto des mesures de performance que vous avez choisies. Par exemple, pour un niveau de débit donné, il existe de nombreux systèmes avec des latences moyennes différentes, et vous souhaitez obtenir le meilleur. Il est inutile d'avoir un système moins performant qu'il pourrait l'être. D'un autre côté, il existe de nombreuses combinaisons de débit et de latence moyenne, et il n'est pas évident de déterminer laquelle constitue le bon compromis sur la frontière de Pareto.

Nous avons utilisé le "débit" et la "latence moyenne" dans l'exemple ci-dessus, mais il existe de nombreux axes différents. Certains d'entre eux ne sont pas liés aux performances, mais à d'autres préoccupations telles que la maintenabilité et la lisibilité du code. Il est souvent préférable d'optimiser la maintenabilité si le code n'a pas besoin d'être aussi rapide que possible, par exemple. Et même au sein d'une mesure de performance telle que la latence, vous devrez trouver un compromis entre des statistiques telles que "le débit pour les types de requêtes X et Y" et d'autres telles que "une latence globale de 99,99%". Comme il est difficile de visualiser

l'utilisation de nombreux axes (https://en.wikipedia.org/wiki/10-cube), il est généralement préférable de se concentrer sur deux ou trois mesures spécifiques lorsqu'on traite des performances.

Nous verrons trois préoccupations générales en matière de performances : le flux de données, la spécialisation et l'utilisation efficace des outils disponibles. Ces trois aspects tournent autour de la conception de systèmes qui ne font que le nécessaire, sans opérations supplémentaires ni abstractions trop générales, car le code le plus rapide est celui qui n'a pas besoin d'exister.

Le flux de données est le facteur le plus important dans la performance globale d'un système. Un système qui traite les données par étapes, en les transférant d'un module à l'autre et en exécutant éventuellement plusieurs instances d'une étape en parallèle, sera beaucoup plus rapide qu'un système qui copie les données partout, envoie les données par cycles entre les modules et ne peut pas être parallélisé en raison de dépendances complexes entre les étapes. C'est l'une des principales raisons pour lesquelles les structures de données immutables sont considérées comme une bonne pratique d'ingénierie : vous n'avez pas besoin de les copier et vous pouvez les traiter en parallèle si nécessaire. Il est facile d'écrire accidentellement du code qui copie des données entre les modules, ou pire, de concevoir un système tel que des copies de données sont nécessaires pour implémenter son interface publique.

Prenons l'exemple des tampons réseau. Dans les réseaux traditionnels, l'application demande au système d'exploitation de recevoir les données. Le système d'exploitation demande alors à la carte réseau de recevoir les données. Une fois que la carte a reçu les données, elle remplit un tampon fourni par le système d'exploitation, qui copie ensuite ce tampon dans un tampon fourni par l'application. Si la copie du réseau vers un tampon est nécessaire pour stocker les données, la copie du tampon du système d'exploitation vers le tampon de l'application est une opération inutile. Elle est nécessaire dans les systèmes réseau traditionnels, car l'application est autorisée à transmettre n'importe quel tampon, même ceux qui ne répondent pas aux exigences de la carte réseau, par exemple ceux qui ne sont pas assez grands. Les nouvelles API réseau ”zéro copie”, telles que DPDK, exigent en revanche que les applications allouent les tampons d'une manière spécifique. L'application peut alors transmettre le tampon au système d'exploitation, qui demande à la carte réseau d'utiliser directement le tampon sans le copier. En réfléchissant à cet exemple, vous remarquerez peut-être qu'il existe ici une forme de fuite d'abstraction : dans le modèle ”zéro copie”, l'application doit connaître les exigences de bas niveau de la carte réseau. Il s'agit d'un compromis entre la maintenabilité et les performances, connu sous le nom de ”contournement de couche”. Les machines virtuelles modernes fonctionnent de la même manière : il serait extrêmement lent pour l'hôte d'intercepter chaque instruction que le client souhaite exécuter, c'est pourquoi seules certaines instructions importantes sont interceptées et la plupart des instructions sont exécutées directement.

La spécialisation est un autre aspect important des performances du système. Écrire le système le plus polyvalent possible conduit généralement à de mauvaises performances sans apporter de réels avantages, car peu de systèmes ont besoin de traiter une très grande variété de demandes. Par exemple, si vous voulez peindre un mur, vous choisirez un grand rouleau à peinture, et non un petit pinceau. Le petit pinceau est utile pour tous les types de peinture, car vous ne pouvez pas utiliser un rouleau à peinture pour les petites tâches, mais il est extrêmement inefficace pour peindre un mur. Pourtant, de nombreux systèmes finissent par faire l'équivalent de peindre un mur avec un petit pinceau, car ils sont ”surdimensionnés” pour être extrêmement généraux sans que cela soit nécessaire.

Un exemple intéressant de généralité est l'instruction ”a += b” dans différents langages de programmation, étant donné, par exemple, ”a = 0, b = 42”. Dans un langage compilé comme Java, si `a` et `b` sont des `int`, cette addition est compilée en une seule instruction d'assemblage, telle que `add eax, ebx` en assemblage x86. En effet, le compilateur sait exactement quelle opération doit être effectuée : l'addition de nombres entiers 32 bits. En revanche, dans un langage interprété comme Python, l'interpréteur doit d'abord déterminer ce que sont `a` et `b`, trouver une méthode qui implémente `+=` ou `+`, puis l'exécuter. Même après avoir déterminé qu'il s'agit d'entiers, Python a encore du travail à faire, car les entiers Python peuvent avoir une longueur illimitée et ne pas tenir dans un seul registre CPU. il faut donc encore plus de code pour gérer le cas où le résultat de l'addition tient dans un registre et le cas où il ne tient pas. Si tout ce que l'utilisateur voulait, c'était additionner deux petits nombres, l'utilisation de Python est très inefficace.

L'utilisation efficace des outils disponibles est le dernier aspect clé des performances du système que nous

allons aborder. À un niveau élevé, cela semble simple. Utilisez un marteau si vous voulez enfoncer un clou, et non le manche d'un pinceau, même s'il est techniquement possible d'utiliser un pinceau avec suffisamment de patience. Aussi évident que cela puisse paraître, il existe de nombreux systèmes qui, par inadvertance, font l'équivalent d'enfoncer des clous avec un pinceau.

Prenons l'exemple du "problème de requête N+1", illustré par cet exemple :

```
var cars = getAllCars(); // par exemple, "SELECT id FROM cars"
for (var car : cars) {
  var plate = getPlate(car); // par exemple, "SELECT plate FROM cars WHERE id = ..."
  // ...
}
```

Ce code effectue une requête pour lister les voitures, puis une requête pour chacune des N voitures afin d'obtenir son numéro d'immatriculation. Il y a donc `N+1` requêtes, même si la base de données est parfaitement capable de renvoyer toutes les voitures et leurs plaques d'immatriculation en une seule requête. Ce problème est rarement aussi évident que dans cet exemple de trois lignes. En général, un module à usage général appelle un autre module à usage général, qui peut être fusionné en un module efficace et spécialisé.

Prenons un autre exemple : vous devez écrire un système en Python. Vous pourriez utiliser l'implémentation par défaut, appelée CPython, qui vous permet de faire tout ce que vous voulez. Mais vous souhaiterez peut-être obtenir des performances supérieures en utilisant le compilateur "juste à temps"PyPy, qui prétend être trois fois plus rapide que CPython. Cependant, PyPy n'est pas 100 % compatible avec CPython, car certains cas particuliers ne se comportent pas de la même manière dans les deux. Si vous tenez compte de ces cas limites lors de la conception de votre système, vous pouvez vous assurer que celui-ci est compatible avec PyPy, alors que la mise à niveau d'un système complexe utilisant des fonctionnalités Python avancées pour utiliser PyPy est beaucoup plus difficile.

Une fois que vous avez épuisé toutes les options générales pour une conception efficace du système, il est temps de faire des compromis. Quels sont les cas les plus courants ? Quelles requêtes peuvent prendre plus de temps que d'autres ? Quels sont les indicateurs importants ? Personne ne peut répondre à ces questions pour tous les systèmes possibles.

Comment écrire du code rapide ?

Maintenant que nous avons discuté de la conception de systèmes rapides, voyons comment écrire du code rapide au niveau des fonctions et des classes individuelles. Le processus général est le suivant :

1. Écrire un code correct, car un système incorrect qui produit des réponses rapides n'a aucun intérêt
2. Définissez un objectif de performance, afin de savoir quand vous arrêter
3. Trouvez le "goulot d'étranglement", c'est-à-dire la partie qui prend le plus de temps
4. Accélérez ce goulot d'étranglement
5. Répétez le processus jusqu'à ce que vous ayez atteint votre objectif

Commençons par un exemple simple pour discuter de l'amélioration de la latence et du débit : une pizzeria où les serveurs passent les commandes au cuisinier, qui prépare la pâte, ajoute les garnitures, cuit la pizza, puis la met dans une assiette et la remet au serveur. Comme cet exemple traite une seule demande à la fois, le débit et la latence sont liés par une équation simple : "débit = 1 / latence". Autrement dit, réduire le temps nécessaire à la préparation d'une pizza augmente directement le nombre de pizzas préparées par seconde.

Il existe plusieurs options simples pour réduire la latence dans notre exemple de pizzeria, comme former le cuisinier pour devenir plus rapide ou augmenter la température du four pour cuire les pizzas plus rapidement. Il est également possible d'augmenter le débit de nombreuses façons, par exemple en embauchant davantage de cuisiniers ou en utilisant n'importe quelle méthode permettant de réduire la latence. Cependant, le goulot d'étranglement de la pizzeria est probablement le four. Il faut plus de temps pour cuire une pizza que pour ajouter les garnitures ou la couper. Il est inutile d'embaucher beaucoup plus de cuisiniers s'ils doivent tous attendre leur tour pour utiliser le four. De manière plus générale, il est inutile de discuter des performances des parties du système qui ne constituent pas un goulot d'étranglement, car leurs performances ne sont pas

essentielles. Par exemple, s'il faut 90 secondes pour cuire une pizza, il n'est pas particulièrement utile de gagner 0,5 seconde lors de l'ajout du fromage sur la pizza, ou de gagner 10 nanosecondes dans le logiciel qui envoie la commande à la cuisine.

Cet exemple illustre la loi d'Amdahl, qui stipule que l'accélération globale obtenue en optimisant un composant est limitée par la part de ce composant dans le temps d'exécution global. Par exemple, si une fonction prend 2 % du temps d'exécution total, il n'est pas possible d'accélérer le système de plus de 2 % en optimisant cette fonction, même si celle-ci est rendue beaucoup plus rapide.

Il est important de garder à l'esprit que les causes d'une mauvaise performance sont généralement quelques goulots d'étranglement importants dans le système, qui peuvent être détectés et corrigés à l'aide d'un profileur. La fonction moyenne dans une base de code donnée est suffisamment rapide par rapport à ces goulots d'étranglement. Il est donc important de profiler avant d'optimiser afin de s'assurer que le temps de l'ingénieur est bien utilisé. Par exemple, un amateur a réussi à réduire les temps de chargement de 70 % dans un jeu en ligne en trouvant deux goulots d'étranglement et en les corrigeant. Il s'agissait dans les deux cas d'algorithmes inefficaces qui pouvaient être remplacés par des versions plus efficaces sans compromettre la maintenabilité.

L'identification de ces goulots d'étranglement nécessite toujours des mesures. L'intuition est souvent trompeuse en matière de performances. Voir les questions en ligne telles que Pourquoi l'impression de 'B' est-elle nettement plus lente que l'impression de '#'?" ou "Pourquoi le fait de changer 0.1f en 0 ralentit-il les performances de 10 fois ?". Un autre exemple concret est cette demande d'extraction .NET Runtime qui permet d'accélérer de 25 % la version de débuggage du runtime pour une entrée complexe, en utilisant une méthode plus spécialisée pour écrire les messages de journalisation qui ne nécessite pas de verrouillage global. On ne s'attendrait pas à ce qu'un runtime volumineux soit ralenti par un seul verrou utilisé dans les impressions de débuggage, mais cette pull request a eu un impact plus important que n'importe quelle modification apportée au compilateur ou à la bibliothèque standard du runtime.

Alors, comment mesurer le temps nécessaire à l'exécution des différentes parties du code ? La réponse est le "profiling". Il existe deux principaux types de profiling : l'instrumentation, qui ajoute du code au système pour enregistrer le temps avant et après chaque opération, et l'échantillonnage, qui arrête périodiquement le programme pour prélever un échantillon des opérations en cours à ce moment-là. Un profileur par échantillonnage est moins précis, car il peut manquer certaines opérations si le programme ne les exécute jamais au moment où le profileur effectue son échantillonnage, mais il a également beaucoup moins de surcharge qu'un profileur instrumentant. Il existe de nombreux profileurs pour chaque plateforme, et certains IDE sont livrés avec leur propre profileur.

Discutons maintenant de quelques corrections de performances courantes : choisir les structures de données et les algorithmes appropriés, accélérer les cas courants et éviter le travail inutile.

La complexité des structures de données et des algorithmes peut être exprimée à l'aide de la notation "grand O", qui définit la complexité asymptotique d'une opération. Par exemple, l'ajout en tête d'une liste basée sur un tableau est en "O(N)", où "N" est la taille de la liste, car cela nécessite de copier tous les éléments dans un nouveau tableau, tandis que la recherche de la valeur d'une liste basée sur un tableau à un index spécifique est en "O(1)", car elle prend un temps constant quel que soit l'index recherché. À l'inverse, une liste chaînée a une préposition O(1) et une requête O(N). Ainsi, selon l'opération requise dans un morceau de code, le choix entre ces deux listes peut faire une énorme différence. De même, certains algorithmes sont tout simplement meilleurs que d'autres. Le tri à bulles est simple, mais toujours en O(N^2), tandis que le tri rapide est généralement en O(N log N), on peut donc remplacer le premier par le second et améliorer presque toujours les performances.

On peut souvent accélérer les cas courants rencontrés par un système au détriment des cas rares. Par exemple, si le profilage révèle que 90 % des requêtes adressées à un système concernent des données, tandis que les 10 % restants modifient des données, le système peut être accéléré dans la pratique en choisissant des structures de données et des algorithmes qui accélèrent les requêtes de données, même si les modifications deviennent plus lentes.

Mais le code le plus rapide est celui qui n'a pas besoin d'être exécuté du tout. Parfois, les opérations peuvent

être combinées pour éviter d'effectuer un travail inutile. Prenons par exemple le code suivant :

```
sortedLst = sorted(lst)
return sortedLst[0]
```

Le tri est, dans le meilleur des cas, `O(N log N)`. La requête est alors, dans le meilleur des cas, `O(1)`. Mais ce que ce code fait réellement, c'est obtenir le plus petit élément de la liste, c'est-à-dire `min(lst)`, ce qui peut être fait en `O(N)` en parcourant tous les éléments un par un et en gardant trace du minimum. Il est inutile de discuter du choix de l'algorithme de tri à utiliser ou du type de liste permettant une requête plus rapide lorsque ces deux opérations peuvent être remplacées par une opération plus simple.

Enfin, il est important de prendre du recul lorsqu'on est confronté à un code lent et de réfléchir à sa conception. La plupart du temps, les gains ou pertes de performances les plus importants sont déterminés par la conception de haut niveau, comme nous l'avons vu plus haut. On ne peut pas ”optimiser” le tri à bulles ligne par ligne pour obtenir quelque chose d'aussi rapide que le tri rapide, car leurs conceptions sont fondamentalement différentes, ce qui conduit à des performances fondamentalement différentes.

Exercice Essayez quelques profils de base. Ouvrez `profiling.py` dans les exercices pendant le cours et suivez les instructions du ReadMe dans le dossier.

Les résultats du profilage correspondent-ils à vos attentes ?

Comment pourriez-vous l'accélérer ?

<détails>

Créer une chaîne caractère par caractère comme le fait le projet est extrêmement lent. Chaque fois qu'un nouveau caractère est ajouté, Python copie la chaîne de caractères entière, car les chaînes de caractères en Python sont immuables.

Au minimum, vous pouvez garder les caratères dans une liste et ne la convertir en une chaîne de caratères qu'une seule fois, pour éviter toutes ces copies. Idéalement, vous devriez travailler ligne par ligne plutôt que caractère par caractère.

Quels sont les compromis courants en matière de performances ?

Une fois que vous avez épuisé toutes les améliorations générales en matière de performances dans la conception et la mise en oeuvre, il est temps de faire des compromis. Nous aborderons quatre compromis importants, qui ne sont en aucun cas les seuls :

- Recalculer ou mettre en cache
- Chargement immédiat vs chargement différé
- Exécution immédiate vs différée
- Batching ou streaming

Recalculer ou mettre en cache

Au lieu de recalculer une réponse à chaque requête, vous pouvez mettre en cache les réponses. Cela fonctionne de la même manière que dans la vie réelle. Lorsque vous lisez un livre, vous le gardez sur une table à proximité, et vous avez probablement aussi une bibliothèque avec les livres que vous aimez. Pour les nouveaux livres, ou pour ceux que vous avez lus il y a longtemps et que vous n'avez pas conservés, vous allez à la librairie. Ce serait un énorme gaspillage d'acheter un livre chaque fois que vous voulez lire une partie d'un chapitre pour ensuite le jeter. Cependant, il serait également inutile de conserver tous les livres que vous avez lus dans votre bibliothèque, car vous manqueriez rapidement d'espace.

Vous pourriez donc mettre en place une mise en cache comme dans ce pseudocode Python :

```
cache = {}
def getContents(id):
  res = cache.get(id)
  if res is None:
      cache[id] = res = ...
  return res
```

Il est important de toujours évaluer si l'utilisation d'un cache en vaut la peine. Pour les tâches qui impliquent uniquement des calculs et aucune entrée/sortie, il est souvent plus rapide de recalculer le résultat à chaque fois que d'accéder à la RAM qui contient un résultat précédemment calculé, car les processeurs sont extrêmement rapides et la RAM l'est beaucoup moins en comparaison.

Si vous disposez d'un cache, vous pouvez également précharger les réponses, connu comme ”prefetching”, c'est-à-dire effectuer les requêtes que vous pensez que l'utilisateur fera plus tard afin que la réponse soit prête au moment où vous en aurez besoin. Les sites web de vente en ligne qui vous indiquent ”les clients qui ont acheté cet article ont également acheté...” en sont un exemple. Le site web a consacré une puissance de calcul supplémentaire à la recherche d'articles connexes, car il estime que vous êtes susceptible de vouloir ces articles.

Cependant, la mise en cache n'est pas aussi simple qu'il y paraît. Vous devez décider du nombre de réponses à conserver dans le cache, si les réponses expirent et, le cas échéant, après combien de temps, s'il faut précharger et, le cas échéant, dans quelle mesure, etc. Par exemple, si vous concevez une application de prévisions météorologiques, vous ne souhaitez probablement pas fournir des réponses mises en cache pour ”quel temps fait-il aujourd'hui” qui datent de plusieurs heures, car les prévisions auront changé.

Dans l'ensemble, par rapport au recalcul, la mise en cache réduit la latence et augmente le débit pour les requêtes courantes, mais elle augmente également l'utilisation de la mémoire et diminue la maintenabilité du code, car il y a plus de code et surtout une logique plus complexe.

Chargement hâtif ou paresseux

Au lieu d'effectuer de manière *hâtive* le travail qui pourrait être nécessaire, vous pouvez effectuer *avec paresse* uniquement le travail strictement nécessaire. Par exemple, les moteurs de recherche ne vous donnent pas une page avec tous les résultats de votre requête sur l'ensemble d'Internet, car vous trouverez très probablement ce que vous cherchez parmi les premiers résultats. Au lieu de cela, les résultats suivants ne sont chargés que si vous en avez besoin. Le chargement différé réduit l'utilisation des ressources pour les requêtes qui s'avèrent finalement inutiles, mais nécessite plus de travail pour les requêtes qui sont réellement nécessaires mais qui n'ont pas été exécutées tôt.

Ainsi, vous pouvez utiliser une requête ”recherche paginée” qui demande N éléments à partir de l'index M au lieu d'une requête ”recherche tout” qui renvoie tous les éléments de la base de données. Vous pouvez également choisir de n'initialiser un composant que lors de sa première utilisation, plutôt qu'au démarrage du programme :

```
def doStuff():
  if not _initialized:
    init()
    _initialized = True
  ...
```

Cependant, le chargement différé ajoute une certaine complexité, car vous devez gérer des problèmes tels que la marche à suivre lorsque deux requêtes vers le même composant non initialisé se produisent en parallèle. Est-il acceptable de l'initialiser deux fois ? Devriez-vous disposer d'un moyen de ”verrouiller” l'accès afin que la deuxième requête attende que l'initialisation effectuée par la première requête soit terminée ? En pratique, vous pouvez souvent utiliser des modules existants pour cela, tels que `Lazy<T>` dans .NET.

Dans l'ensemble, par rapport au chargement immédiat, le chargement différé réduit la quantité de travail pour les requêtes inutiles, mais augmente la quantité de travail pour les requêtes nécessaires, et réduit une

fois de plus la maintenabilité du code en raison d'une logique de plus en plus complexe.

Exécution immédiate ou différée

Au lieu d'envoyer immédiatement une requête à un serveur lorsqu'un utilisateur clique sur un bouton et d'attendre la réponse du serveur, vous pouvez différer la réponse en affichant un message "chargement en cours..." et en lançant une tâche en arrière-plan qui contacte le serveur et met à jour l'interface utilisateur une fois qu'elle a reçu une réponse. Cela garantit que votre application reste réactive même si le serveur est lent à répondre.

Un autre exemple de ce phénomène se produit dans votre IDE : lorsque vous tapez le début d'une ligne, tel que "System.out.", l'IDE vous propose des entrées possibles, telles que "println", et peut même les trier en fonction de leur probabilité d'occurrence. Cependant, si votre IDE bloquait toutes les opérations de l'interface utilisateur pendant qu'il calculait la liste des entrées suivantes, la saisie serait extrêmement lente. Au lieu de cela, chaque lettre saisie lance une tâche en arrière-plan pour trouver ce que vous pourriez taper ensuite, et si une lettre suivante invalide le résultat de la tâche précédente, une nouvelle tâche est lancée. Cela implique alors de gérer des tâches simultanées pour s'assurer que les résultats n'apparaissent pas dans le désordre, par exemple en proposant "println" même si vous avez tapé "fl" parce que vous vouliez "flush".

Il est également important de gérer les annulations, c'est-à-dire de ne pas continuer à effectuer des tâches qui ne sont plus nécessaires. Par exemple, si l'utilisateur demande la météo, mais quitte l'application pendant le chargement des données, le code qui analyse la réponse du serveur n'a pas besoin d'être exécuté une fois que le serveur a répondu, car le résultat n'est plus nécessaire.

Dans l'ensemble, par rapport à l'exécution immédiate, l'exécution différée augmente la réactivité, mais augmente également la latence, car il y a plus de travail à faire, et réduit une fois de plus la maintenabilité du code.

Batching ou streaming

Au lieu de télécharger un film entier avant de le regarder, vous pouvez le diffuser en streaming : votre appareil chargera le film par petits morceaux, et pendant que vous regardez les premiers morceaux, les suivants peuvent être téléchargés en arrière-plan. Cela réduit considérablement la latence, mais diminue également le débit, car il y a plus de requêtes et de réponses pour la même quantité de données globales.

Par exemple, en Python, le traitement par lots consiste à renvoyer une `list` qui contient tous les éléments, tandis que le streaming consiste à renvoyer un "itérateur" qui récupère les éléments un par un. Certains langages disposent même d'outils pour faciliter le streaming, tels que `yield return` en C# ou `yield` en Python :

```
def get_zeroes():
  while True:
    print("Yielding")
    yield 0
```

Notez que ce flux est infini, ce qui serait impossible à réaliser avec un lot. Si vous l'appelez de la manière suivante :

```
for n in get_zeroes():
  print("n = ", n)
```

Le résultat sera "Yielding", puis "n = 0", puis "Yielding" à nouveau, puis "n = 0" à nouveau, et ainsi de suite indéfiniment. La méthode `get_zeroes` ne s'exécute que jusqu'à `yield`, puis rend le contrôle à la boucle pour chaque itération.

Si vous appeliez `sum(get_zeroes())`, votre programme se bloquerait indéfiniment, car il tenterait d'additionner une séquence infinie. Vous devez plutôt utiliser des méthodes qui s'arrêtent tôt si votre séquence est susceptible d'être infinie, par exemple, `next(x for x in stream if x > 5)`, qui examine le premier

élément du flux et, s'il est supérieur à `5`, le renvoie sans examiner le reste. Sinon, il examine le deuxième élément, vérifie s'il est supérieur à `5`, et ainsi de suite.

Dans l'ensemble, par rapport au traitement par lots, le streaming réduit la latence, mais diminue également le débit.

Exercice Optimisons une application pour améliorer sa réactivité. Utilisez le fichier `tradeoffs.py` dans le dossier d'exercices pendant le cours, en commençant par l'exécuter. Comme vous pouvez le constater, l'affichage des résultats prend un certain temps. En examinant le code, vous pouvez voir qu'un lot entier de résultats est chargé en une seule fois.

Suivez les trois exercices indiqués dans les commentaires du code : diffusez les résultats, mettez-les en cache et ajoutez une prélecture.

<détails>

Pour le streaming, convertissez `getNews` en :

```
for n in range(count):
    yield getNewsItem(index + n)
```

Pour ajouter un cache, créez une nouvelle fonction et utilisez-la dans `getNews`:

```
CACHE = {}
def getNewsItemCached(idx):
  result = CACHE.get(idx)
  if result is None:
      CACHE[index] = result = getNewsItem(idx)
  return result
```

Pour le prefetching, commencez par re-définir une version de `getNews` qui charge tout, puis utilisez-la :

```
def getNewsList(index, count):
    return list(getNews(index, count))

threading.Thread(target=getNewsList, args=[index + BATCH_SIZE, BATCH_SIZE]).start()
```

Résumé

Dans cette leçon, vous avez appris :

- Définir les objectifs : latence, débit et variabilité
- Optimisation : conception, benchmarking, profiling
- Compromis courants : Cache, paresse, exec. différée, streaming

Vous pouvez maintenant consulter les exercices !

Interfaces graphiques et APIs

Quand on programme pour soi-même, il est souvent plus simple de créer une interface en ligne de commande basique. De plus, beaucoup de petits scripts et programmes simples ne communiquent pas avec le reste du monde, ou le font via des fichiers téléchargés manuellement.

Mais pour développer un produit logiciel, il est en général nécessaire d'avoir une interface *graphique*, et de communiquer avec des services externes via des *APIs*, pour "Application Programming Interfaces". De plus, il est souvent nécessaire d'extraire la logique réutilisable d'une application pour pouvoir développer plusieurs interfaces, par exemple mobile et bureau, et pour réutiliser certains modules dans différentes applications.

Objectifs

Après ce cours, vous devriez être en mesure de :

- Concevoir des interfaces utilisateur graphiques
- Organiser le code avec des design patterns
- Découpler l'UI, la logique, et les strategies réutilisables
- Interagir avec des services externes via des APIs

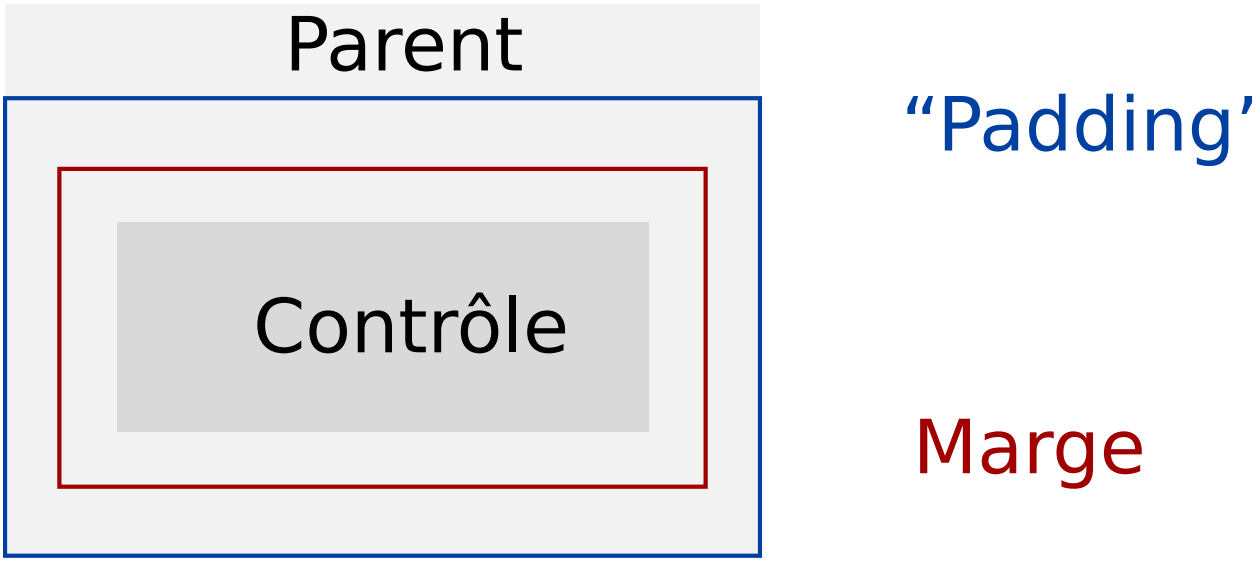
Comment concevoir une interface graphique ?

Il y a en gros deux types de contrôles dans une interface graphique : les entrées et les sorties. Les entrées incluent des entrées de texte, de nombres, de fichiers, et d'autres types de données, ainsi que des exécutions de fonctions via, par exemple, des boutons. Certaines entrées sont plus spécifiques, par exemple ”addresse email” et non simplement ”texte”. Les sorties incluent de l'affichage de texte, d'image, de son, et d'autres types de données.

En général, tant les entrées que les sorties ont des *propriétés* communes, telles que leur visibilité (visible, caché), leur opacité, la couleur du texte s'il y en a, la couleur de fond, et bien d'autres. De plus, les contrôles dans une interface utilisateur fournissent des *évènements*, c'est-à-dire un moyen d'appeler des fonctions spécifiques quand quelque chose se passe, comme ”clic”, ”double clic”, ou ”clic droit”.

Les interfaces utilisateurs sont en général modélisées à l'aide de concepts orientés objets, soit l'héritage et la composition. Par exemple, une classe de base représente tous les contrôles, et a des sous-classes telles que ”boîte d'entrée de texte”, ”label”, et ”image”. La boîte d'entrée de texte peut elle-même avoir une sous-classe pour entrer un mot de passe, qui montre des astérisques au lieu du mot de passe que l'utilisateur écrit. De plus, des contrôles en contiennent d'autres, comme par exemple une liste d'onglets qui chacun peut contenir d'autres contrôles. Un onglet qui affiche une image n'*est pas* une image, mais il *a* une image, c'est donc de la composition et non de l'héritage.

Avant de passer à un exemple, voyons concepts généraux qui sont présents dans les interfaces graphiques quel que soit le langage ou framework. D'abord, les marges, typiquement le ”padding” qui est une marge à l'intérieur d'un contrôle parent et la marge à l'extérieur d'un contrôle enfant :



Ensuite, la mise en page. Il existe énormément de types de mise en page, comme les grilles dans lesquels chaque contrôle occuppe une ou plusieurs cellules, l'entassement horizontal ou vertical, et bien d'autres :

Exemple en Python : `tkinter`

Le framework `tkinter` en Python est intégré à la librairie standard, raison pour laquelle nous allons l'utiliser comme exemple. Chaque framework d'interface graphique fait certaines tâches de manière différente, mais les concepts généraux sont les mêmes.

Créez un fichier Python et commencez par importer `tkinter` :

```
import tkinter
```

Ensuite, créez une fenêtre principale, donnez-lui un nom, et définissez sa taille :

```
# Créee la fenêtre principale
root = tkinter.Tk()
# Optionnel : Augmente la taille des contrôles Tkinter, très petits par défaut
root.tk.call('tk', 'scaling', 2.5)
# Définit le titre et la taille de la fenêtre principale, ainsi que sa position à l'écran
root.title("Exercice")
root.geometry("500x700+200+200")
```

Vous pouvez maintenant afficher la fenêtre :

```
# Lance la boucle principale Tkinter : affichage + attente d'entrée utilisateur
root.mainloop()
```

Cela lance la fenêtre, qui est évidemment vide.

Le reste du code de cet exemple doit apparaître **avant l'appel à `mainloop`**.

Commençons par afficher du texte à l'aide d'un `Label` :

```
label = tkinter.Label(root, text = "Hello, World!")
label.grid(row=0, column=0)
```

Si vous lancez le code, vous devriez voir une fenêtre avec le texte ”Hello, World!” en haut à gauche. Nous avons placé le label sur la ligne 0 et colonne 0, ce qui ne change rien pour l'instant puisqu'il n'y a rien d'autre.

Ajoutons donc une entrée textuelle, le contrôle `Entry` :

```
entry = tkinter.Entry(root)
entry.grid(row=0, column=1)
```

Si vous lancez le code, vous verrez une boîte de texte directement à droite du texte "Hello, World!", puisque l'`Entry` est sur la colonne 1 mais toujours sur la ligne 0.

Maintenant, un bouton :

```
bt = tkinter.Button(root, text="Click", padx=60, pady=5)
bt.grid(row=1, columnspan=2, pady=20)
```

Ce bouton existe mais ne fait rien. Remarquez que avec `columnspan=2`, notre bouton s'étend ("spans" en anglais, d'où le nom de la propriété) sur deux colonnes, et avec `pady=20`, nous ajoutons de la marge à gauche et à droite du texte à l'intérieur.

Pour ajouter une commande, on peut `bind` l'évènement `<Button-1>` à une fonction qui affiche une fenêtre avec le texte de l'entrée :

```
import tkinter.messagebox as msg
def onclick(e):
  msg.Message().show(message=entry.get(), type=msg.OK)
bt.bind("<Button-1>", onclick)
```

Lancez le code, écrivez du texte dans l'entrée textuelle, et cliquez sur le bouton. Vous pouvez également changer l'évènement, par exemple `<Button-2>` est le clic droit, vous pouvez trouver des listes en ligne de tous les évènements.

Ajoutons un autre type d'entrée, une boîte à cocher, souvent appelée "checkbox", que `tkinter` appelle `Checkbutton` :

```
ck = tkinter.Checkbutton(root, text="Check me")
ck.grid(row=2)
```

Il existe bien d'autres types d'entrées, comme une échelle :

```
scale = tkinter.Scale(root, from_=0, to=100, orient="h")
scale.grid(row=3, columnspan=2)
```

Ou même un choix de fichier. Cette fois, déclarons la commande directement dans la création d'un `Button`. Notez que dans ce cas la fonction ne prend aucun paramètre :

```
from tkinter import filedialog
def openfile():
  path = filedialog.askopenfilename(title="Choose", filetypes=(("Text", "*.txt"),("All", "*.*")))
  # ...ici on pourrait faire quelque chose avec `path`...
bt2 = tkinter.Button(root, text="File", command=openfile)
bt2.grid(row=4)
```

Il serait possible d'écrire du code pour, par exemple, détecter les changements de texte dans l'entrée textuelle via un évènement, et les répercuter ailleurs. Heureusement, les frameworks d'interface graphique ont en général des concepts de liaison entre contrôles et variables qui le font pour nous. `tkinter` a des classes de variables comme `StringVar`, et les contrôles prennent ces variables comme propriétés.

Par exemple, une variable textuelle et un label qui l'affiche, que nous pouvons synchroniser avec l'entrée :

```
v = tkinter.StringVar()
vlabel = tkinter.Label(root, textvariable=v)
vlabel.grid(row=5, columnspan=2)
entry.configure(textvariable=v)
```

Relancez l'application, écrivez du texte, et vous verrez que les changements sont immédiatement répércutés sur le nouveau label.

Il existe d'autre type de variables, comme un Booléen, que nous pouvons synchroniser avec la boîte à cocher :

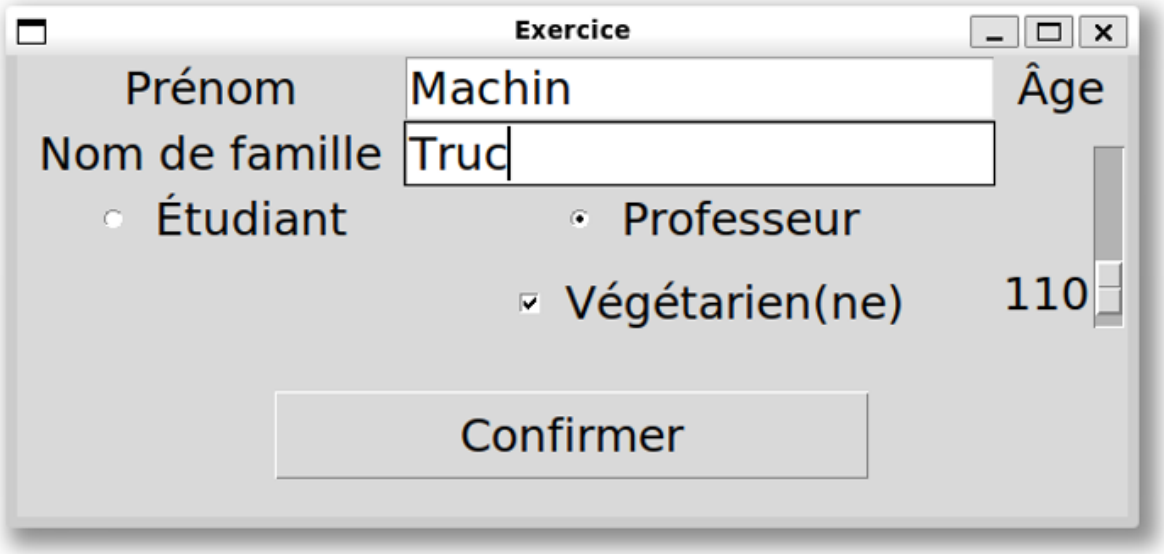
```
b = tkinter.BooleanVar()
blabel = tkinter.Label(root, textvariable=b)
blabel.grid(row=2, column=1)
ck.configure(variable=b)
```

Pour certains contrôles, le fait d'avoir une variable est en fait requis pour un fonctionnement correct, comme des boutons à choix unique dit ”radio button” :

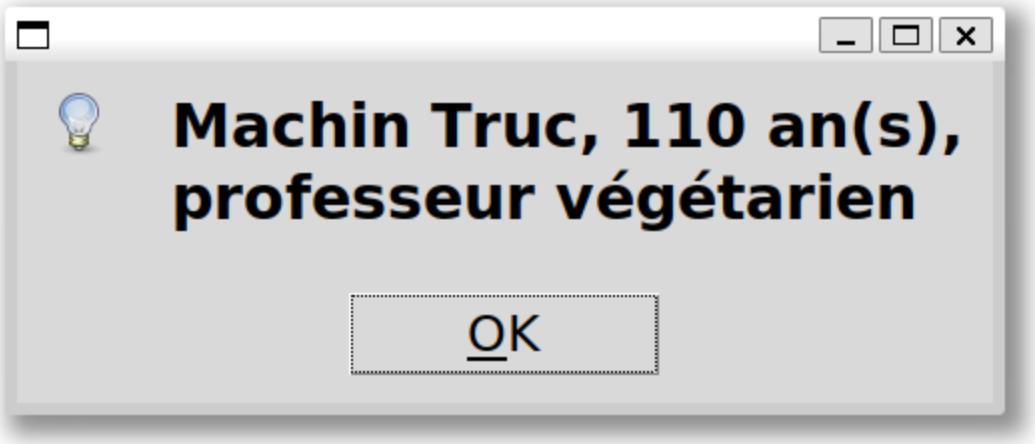
```
i = tkinter.IntVar()
rb1 = tkinter.Radiobutton(root, text="First", variable=i, value=1)
rb2 = tkinter.Radiobutton(root, text="Second", variable=i, value=2)
rb3 = tkinter.Radiobutton(root, text="Third", variable=i, value=3)
rb1.grid(row=6)
rb2.grid(row=7)
rb3.grid(row=8)
```

(Le nom ”radio button” vient des anciennes radios dans les voitures, qui possédaient plusieurs boutons pour choisir des fréquences prédéfinies. Seul un bouton peut être choisi à la fois, puisque la radio ne peut capter qu'une fréquence à la fois.)

Exercice Créez un formulaire ressemblant à ceci :



Quand l'utilisateur clique sur ”Confirmer”, l'application doit afficher l'information dans une nouvelle fenêtre :



Vous pouvez également consulter la documentation `tkinter` pour rendre votre fenêtre plus jolie.

Une solution est disponible ici.

Comment réutiliser des concepts dans plusieurs systèmes ?

Lors de la conception d'un système, le contexte est souvent le même que dans les systèmes précédents, tout comme les exigences des utilisateurs. Par exemple, ”traverser un cours d'eau” est une exigence et un contexte courants qui conduisent à la solution naturelle ”construire un pont”. Si chaque ingénieur concevait le concept d'un pont à partir de zéro chaque fois que quelqu'un avait besoin de traverser un cours d'eau, chaque pont ne serait pas très bon, car il ne bénéficierait pas des connaissances accumulées lors de la construction des ponts précédents. Au lieu de cela, les ingénieurs disposent de plans pour différents types de ponts, les sélectionnent en fonction des spécificités du problème et proposent des améliorations lorsqu'ils en trouvent.

En génie logiciel, ces types de plans sont appelés *”design patterns”*, ou ”patrons de conception” en pur français, et sont si courants qu'on en oublie parfois leur existence. Prenons par exemple la boucle suivante :

```
for item in items:
    # ...
```

Cette construction ”for” semble tout à fait normale et standard, mais elle n'a pas toujours existé. Par exemple, le langage Java ne l'a introduite que dans Java 1.5, en même temps que l'interface `Iterable<T>`, afin que chaque collection ne fournisse plus sa propre méthode d'itération. Elle était autrefois connue sous le nom de ”design pattern Iterator”, mais elle est aujourd'hui tellement courante dans les langages de programmation modernes que nous ne la considérons plus explicitement comme un patron de conception.

Les design patterns sont des plans, pas des algorithmes. Une design pattern n'est pas un bout de code que vous pouvez copier-coller, mais une description générale de ce à quoi peut ressembler la solution à un problème courant. Vous pouvez le considérer comme le nom d'un plat plutôt que sa recette. Vous avez du poisson ? Vous pourriez préparer du poisson avec des légumes et du riz, ce qui est une combinaison saine. La sauce soja est également une bonne idée pour accompagner le plat. La manière dont vous cuisinez le poisson ou les légumes que vous choisissez dépendent de vous.

Il existe de nombreuses patterns et encore plus de descriptions en ligne. Dans ce cours, nous verrons des modèles permettant de séparer l'interface utilisateur d'un programme, la logique métier qui est au coeur du programme et les stratégies réutilisables dont le programme a besoin, telles que le fait de réessayer lorsqu'une requête échoue.

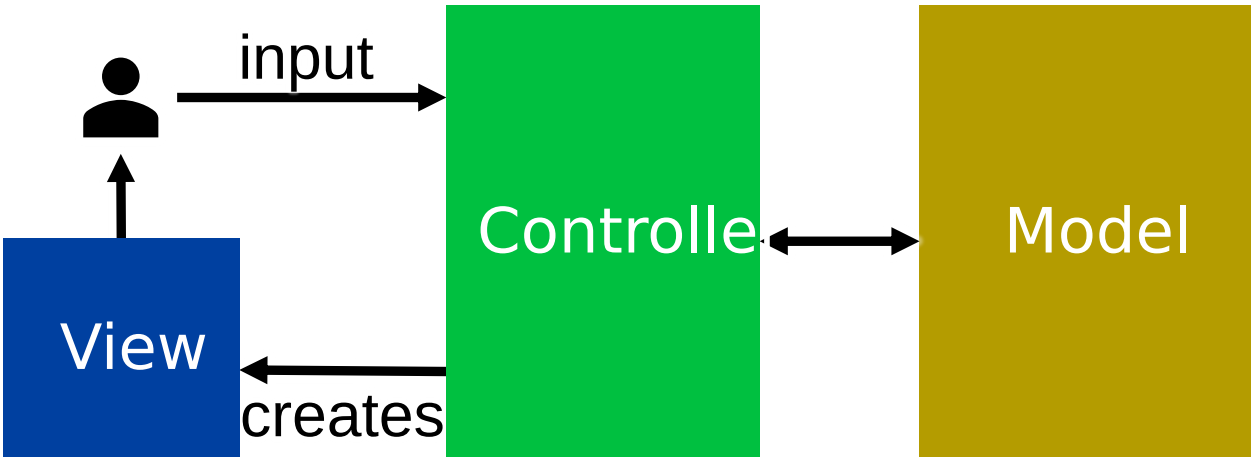
Le problème résolu par les design patterns pour les interfaces utilisateur est courant : les ingénieurs logiciels doivent écrire du code pour des applications qui fonctionneront sur différents types de systèmes, tels que les applications de bureau et les applications mobiles. Cependant, écrire le code une fois par plateforme ne serait pas viable : la plupart du code serait copié-collé. Toute modification devrait être répliquée sur le code de toutes les plateformes, ce qui conduirait inévitablement à une désynchronisation d'une des copies.

Au lieu de cela, nous voulons pouvoir écrire une seule fois la logique centrale de l'application et n'écrire du code différent par plateforme que pour l'interface utilisateur. Cela signifie également que les tests peuvent être écrits par rapport à la logique sans être liés à une interface utilisateur spécifique. C'est une exigence pratique pour toute application de grande envergure. Par exemple, Microsoft Office compte des dizaines de millions de lignes de code ; il serait tout à fait impossible de dupliquer ce code dans Office pour Windows, Mac, Android, le web, etc.

La logique métier est généralement appelée *"Model"* ("modèle") et l'interface utilisateur *"View"* ("vue"). Nous voulons éviter de les coupler, nous avons donc naturellement besoin d'un élément intermédiaire qui communique avec les deux, mais lequel ?

Model-View-Controller (MVC)

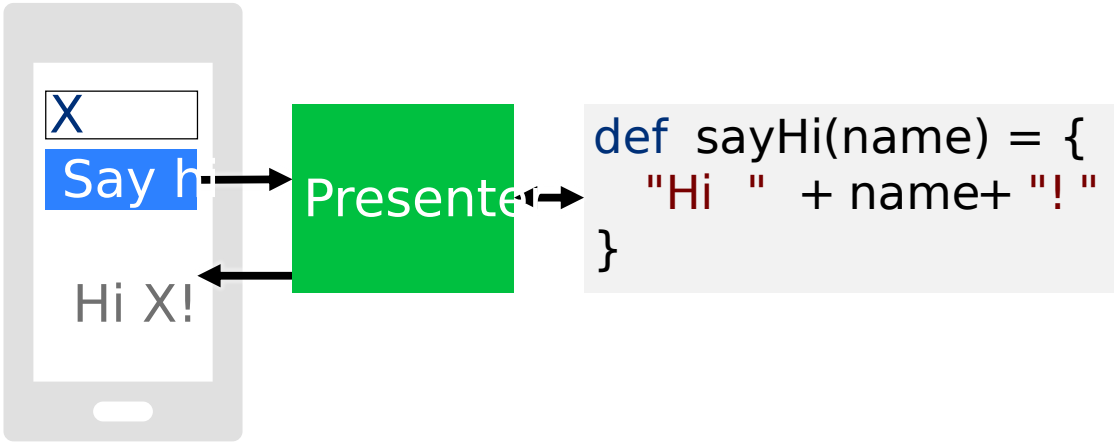
Dans le modèle MVC, la vue et le modèle sont gérés par un *"Controller"* ("contrôleur"), avec lequel les utilisateurs interagissent. Un utilisateur soumet une requête au contrôleur, qui interagit avec le modèle et renvoie une vue à l'utilisateur :



Par exemple, dans un site web, le navigateur de l'utilisateur envoie une requête HTTP au contrôleur, qui finit par créer une vue à partir des données du modèle, et la vue est rendue en HTML. La vue et le modèle sont découplés, ce qui est une bonne chose, mais il y a aussi des inconvénients. Premièrement, les utilisateurs ne communiquent généralement pas directement avec les contrôleurs, en dehors du web. Deuxièmement, créer une nouvelle vue à partir de zéro à chaque fois n'est pas très efficace.

Model-View-Presenter (MVP)

Dans le modèle MVP, la vue et le modèle sont médiatisés par un *"Presenter"* ("présentateur"), mais la vue gère directement les entrées de l'utilisateur. Cela correspond à l'architecture de nombreuses interfaces utilisateur : les utilisateurs interagissent directement avec la vue, par exemple en touchant un bouton sur l'écran d'un smartphone. La vue informe ensuite le présentateur de l'interaction, qui communique avec le modèle si nécessaire, puis indique à la vue ce qu'elle doit mettre à jour :



Cela résout deux des problèmes du modèle MVC : les utilisateurs n'ont pas besoin de connaître le module intermédiaire, ils peuvent interagir avec la vue à la place, et la vue peut être modifiée de manière incrémentale.

Exercice Maintenant, à votre tour. Ouvrez `weather.py` dans le dossier d'exercices pendant le cours et créez un `Model` et une `View` pour séparer le code de l'application.

Le `FakeModel` reprend le code utilisant `random`, mais retourne juste la météo au lieu d'écrire la météo et le message ”Météo actuelle” sur la console.

La `ConsoleView` utilise `print` et `input`, ainsi qu'une boucle infinie.

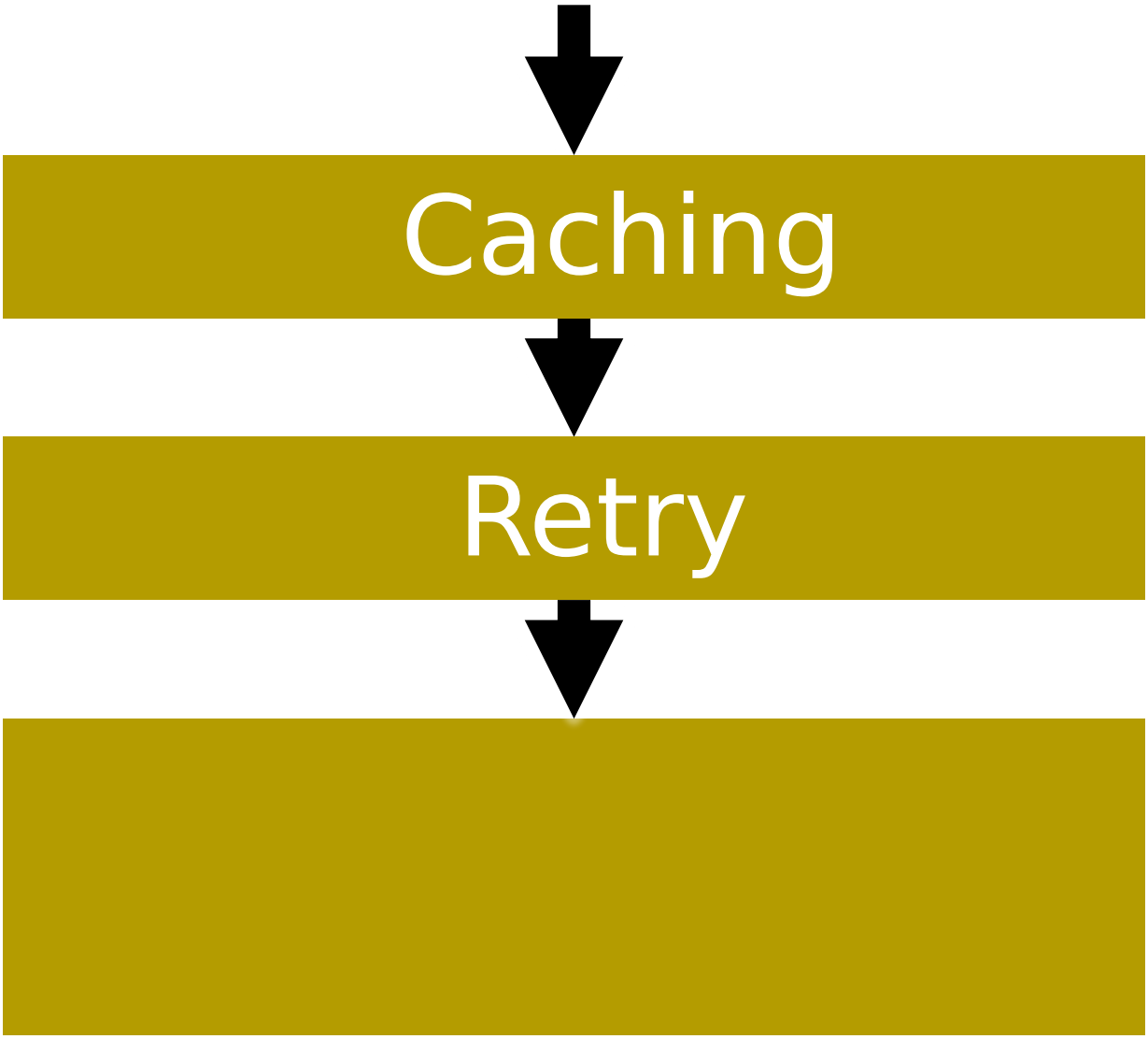
Consultez la solution pour voir les détails.

Middleware

Vous avez écrit une application en utilisant une design pattern d'interface utilisateur pour séparer votre logique métier et votre interface utilisateur, mais vous recevez maintenant une demande d'un client : les données peuvent-elles être mises en cache afin qu'une connexion Internet ne soit pas nécessaire ? De plus, lorsqu'il n'y a pas de données en cache, l'application peut-elle réessayer si elle ne parvient pas à se connecter immédiatement ?

Vous pourriez intégrer cette logique dans votre contrôleur, votre présentateur ou votre ViewModel, mais cela la lierait à une partie spécifique de votre application. Vous pourriez l'intégrer dans un modèle, mais au prix de rendre ce module plus complexe, car il contiendrait plusieurs concepts orthogonaux.

C'est là qu'intervient le modèle *middleware*, également appelé *decorator*. Un middleware fournit une couche qui expose la même interface que la couche inférieure, mais ajoute des fonctionnalités :



Un middleware peut ”court-circuiter” une requête s'il souhaite répondre directement au lieu d'utiliser les couches inférieures. Par exemple, si un cache contient des données récentes, il peut renvoyer ces données sans demander à la couche inférieure les données les plus récentes.

Un exemple concret de middleware est celui des minifiltres du système de fichiers Windows, qui sont des middlewares pour le stockage qui effectuent des tâches telles que la détection de virus, la journalisation ou la réplication vers le cloud. Cette conception permet aux programmes d'ajouter leur propre filtre dans la pile d'E/S Windows sans interférer avec les autres. Les programmes tels que Google Drive n'ont pas besoin de connaître l'existence d'autres programmes tels que les antivirus.

Exercice Reprennez votre `weather.py`, et ajoutez une fonctionnalité : si la météo est `"???"`, l'app doit réessayer.

Commencez par créer un nouveau `Model` qui prend un `Model` en paramètre de constructeur et délégue `getForecast`, puis écrivez une implémentation de `getForecast` qui réessaye si nécessaire.

Consultez la solution pour voir les détails.

Le MVP présente certains inconvénients. Tout d'abord, la vue contient désormais des variables, car elle est mise à jour de manière incrémentielle. Cela ajoute davantage de code à la vue, alors que l'un de nos objectifs initiaux était d'en réduire autant que possible. Ensuite, l'interface entre la vue et le présentateur est souvent liée à des actions spécifiques que la vue peut effectuer dans un contexte donné, tel qu'une application console, et il est difficile de rendre la vue générique pour de nombreux formats.

Model-View-ViewModel (MVVM)

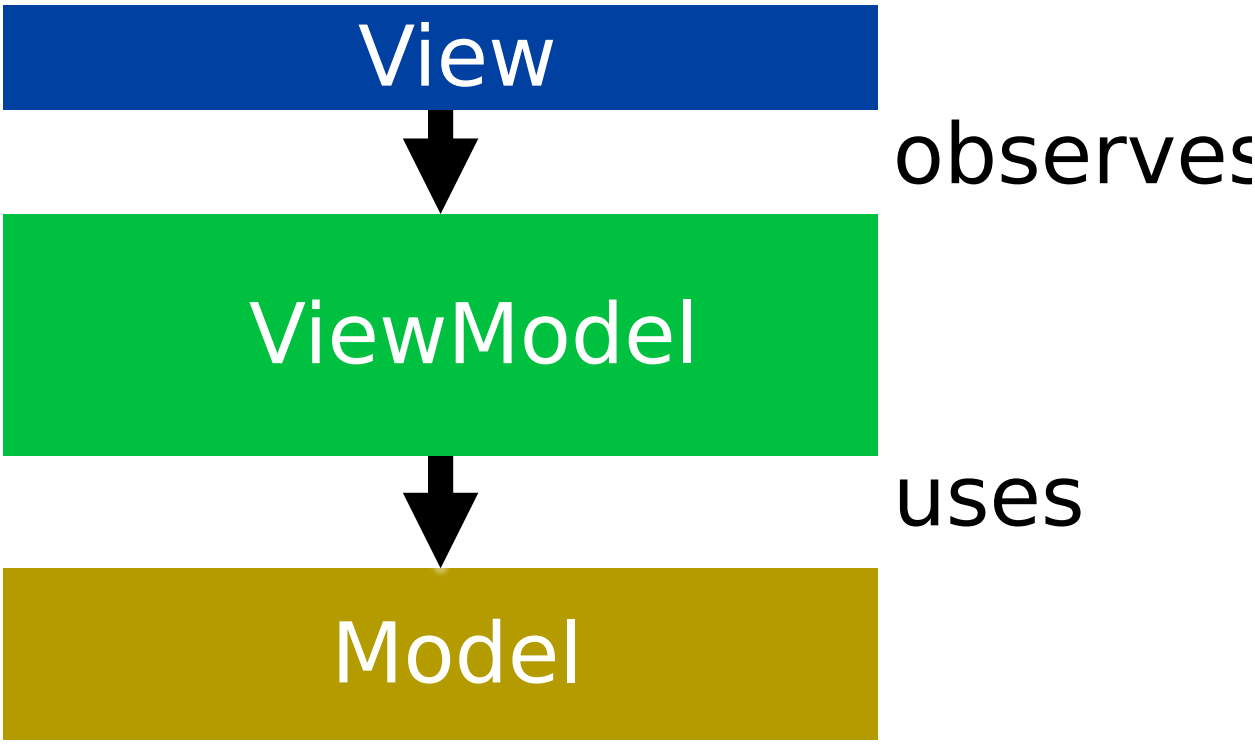
Prenons un peu de recul avant de décrire le modèle suivant. Qu'est-ce qu'une interface utilisateur, au juste ?

- Les données à afficher,
- les commandes à exécuter...

... et c'est tout ! Du moins, si on voit ça d'un très haut niveau.

L'idée clé derrière "MVVM" est que la vue doit *observer* les changements de données à l'aide de la design pattern Observer. Ainsi, le module intermédiaire, le "ViewModel", doit seulement être une interface utilisateur indépendante de la plateforme qui expose les données, les commandes et une implémentation du modèle Observer pour permettre aux vues d'observer les changements.

Le résultat est un système clairement structuré, dans lequel la vue contient peu de code et est superposée au ViewModel, qui conserve l'état et utilise lui-même le modèle pour mettre à jour son état lorsque des commandes sont exécutées :



La vue observe les changements et se met à jour automatiquement. Elle peut choisir d'afficher les données comme elle le souhaite, car le ViewModel ne lui indique pas comment se mettre à jour, mais seulement ce qu'elle doit afficher.

La vue est conceptuellement une fonction du ViewModel : elle peut être entièrement calculée à partir de ce dernier à chaque fois, ou elle peut se modifier progressivement à des fins d'optimisation. Cela est utile pour les plateformes telles que les smartphones, sur lesquelles les applications fonctionnant en arrière-plan doivent utiliser moins de mémoire : la vue peut simplement être détruite, car elle peut être entièrement recréée à

partir du ViewModel chaque fois que cela est nécessaire. Le MVVM permet également la réutilisation du code que nous souhaitons obtenir, car différentes plateformes ont besoin de vues différentes, mais du même Model et du même ViewModel, et le ViewModel contient les variables, ce qui rend les vues plus petites.

Attention : ce n'est pas parce que vous pouvez utiliser toutes sortes de design patterns que vous devez le faire. Si vous n'en avez pas besoin, ne le faites pas. Sinon, vous risquez de vous retrouver avec une « implémentation d'AspectInstanceFactory qui localise l'aspect à partir de BeanFactory à l'aide d'un nom de bean configuré », au cas où quelqu'un souhaiterait bénéficier de cette flexibilité. Vraiment !

Commment utiliser des services externes ?

Vous souhaitez communiquer avec un serveur externe. Mais comment ? Quel transport utiliser pour votre requête et pour recevoir une réponse ? Quels formats pour ces dernières ? Quelle sécurité ?

Regardons d'abord le transport. Que mettre entre la couche ”application”, comme une app pour téléphone, tout en haut et la couche ”réseau”, comme le Wi-Fi, tout en bas ? De nos jours, il y a 3 couches principales intermédiaires : HTTP, TCP ou UDP, et IP.

IP, pour ”Internet Protocol”, permet à deux machines de communiquer. Les paquets IPs contiennent leur adresse source et destination, la longueur du paquet, et d'autres métadonnées qui ne nous intéressent pas. Par-dessus IP, la couche la plus simple est UDP, ”User Datagram Protocol”, qui est un protocole simple sans notion de connexion. Les paquets UDPs contiennent les numéros de port source et destination, afin qu'une seule machine puisse héberger plusieurs services communiquants sur des ”ports” différents, la longueur du paquet, ce qui est redondant avec IP, et un ”checksum” permettant de vérifier l'intégrité du paquet. UDP n'offre aucune garantie quant à la réception des paquets, ou à leur ordre. Pour cela, il faut TCP, ”Transmission Control Protocol”, qui est aussi par-dessus IP et contient également des numéros de ports et un checksum. TCP offre en plus des métadonnées permettant aux deux côtés de la connexion de s'assurer que l'autre reçoit bien les paquets et dans le bon ordre, en renvoyant des paquets si nécessaire.

HTTP, ”HyperText Transfer Protocol”, est le protocole principal utilisé pour la communication sur le Web, par-dessus TCP. Une requête HTTP contient une ”méthode”, telle que GET ou POST, indiquant le type de requête, un ”chemin”, tel que `/example.txt`, et des métadonnées (dits ”headers”) permettant de communiquer des informations comme l'authentification. Les réponses HTTP contiennent des codes, tels que 100, 200, ou 404, chacun signifiant un type de réponse. Par exemple, ”200” signifie ”OK”, soit que la requête est valide et le serveur fournit une réponse. ”404” signifie ”Not Found”, soit que le chemin demandé par la requête est inconnu du serveur, malgré le fait que le reste de la requête est valide. HTTP supporte le chiffrement, qui est en théorie optionnel mais en pratique requis par presque tous les serveurs de nos jours. Beaucoup de serveurs utilisent des nouvelles versions de HTTP : HTTP/2 amène le ”multiplexing”, soit le fait qu'un client et un serveur peuvent communiquer plusieurs fichiers à la fois via la même connexion, et le ”server push”, avec lequel un serveur peut fournir des fichiers au client même sans requête correspondante si le serveur prévoit que le client en aura besoin. HTTP/3 est une évolution de HTTP/2 qui n'est plus basée sur TCP mais sur UDP, car les fonctionnalités de HTTP/2 et TCP étaient dupliquées et HTTP/3 est donc plus efficace.

En pratique, il faut se souvenir qu'une requête web contient une URL, une méthode, des métadonnées, et un contenu, et qu'une réponse contient un code de réponse, des métadonnées, et du contenu. Chaque langage de programmation vient avec une librairie standard qui en général contient un module HTTP, par exemple en Python :

```
import urllib.request
url = "https://example.org"
with urllib.request.urlopen(url) as resp:
    html = resp.read()
    print(html.decode("utf-8"))
```

Il existe des modules tiers en Python comme la librairie `requests` qui sont souvent recommandés comme meilleures alternatives, mais nous n'en parleront pas ici.

Après avoir choisi un transport comme HTTP, quel format utiliser pour une requête et une réponse ? Le plus

simple est le texte : `bonjour` est un message simple à écrire. Mais comment représenter, par exemple, la liste contenant `1`, `2` et `3` en tant que texte ? Ou un objet tel qu'une personne ayant pour nom `Alice` et pour âge `42` ? Il existe un standard pour communiquer des paires de clés et valeurs en HTTP, avec `&` pour séparer les paires et `=` pour séparer les clés des valeurs, par exemple `nom=Alice&age=42`. C'est suffisant pour des requêtes ou réponses simples. Mais il est pénible d'encoder, par exemple, une liste ou un objet contenant lui-même des objets avec uniquement des clés et valeurs.

Il existe donc des standards basés sur la notion d'objet, comme XML ("eXtensible Markup Language") :

```
<person>
  <name>Alice</name>
  <age>42</age>
</person>
```

et JSON ("JavaScript Object Notation") :

```
{
  "name": "Alice",
  "age": 42
}
```

De nos jours, on a tendance à préférer JSON, mais XML reste répandu car il existe énormément d'outils et de protocoles basés dessus. Il y a aussi d'autres formats plus complexes, qui demandent un schéma partagé par les deux parties communiquantes, comme Thrift, Protobuf, Cap'n Proto, etc. Nous n'en parleront pas ici.

Enfin, le HTML, utilisé pour écrire des sites Web, est aussi une sorte de format. Il est possible d'obtenir des données d'un site Web en obtenant une page prévue pour des humains et en analysant son code source. Mais contrairement aux pages prévues pour des machines avec des réponses en XML ou JSON, les pages HTML ont tendance à changer sans préavis, et à ne pas contenir les données dans un format pratique à analyser. Il n'est donc pas recommandé d'utiliser le HTML comme format, sauf s'il n'y a pas d'alternative.

Tout comme pour HTTP, les librairies standard contiennent typiquement des modules de sérialisation et déserialisation XML et JSON, par exemple en Python :

```
import json
s = json.dumps(...)
obj = json.loads(...)
```

Finalement, quelques détails sur la sécurité. À part le chiffrement, qui est en pratique obligatoire de nos jours, les services Web sont rarement complètement publics. Il y a typiquement des limites sur leur utilisation, tant d'un point de vue technique que légal. De plus, il faut souvent authentifier l'application qui utilise le service, ainsi que l'utilisateur de l'application. Souvent, les services ont une version gratuite qui limite strictement le nombre d'appels par heure, et qui ne peut être utilisée qu'à des fins personnelles et non commerciales. Nous n'entrerons pas plus en matière dans ce cours, mais en résumé, il vous est souvent nécessaire de vous enregistrer auprès d'un fournisseur pour obtenir une "clé d'API", que vous devez inclure dans vos requêtes, et d'exercer de la retenue dans vos requêtes, sans quoi vous recevrez une réponse vide avec un code de réponse indiquant que vous avez épuisé votre nombre de requêtes pour l'instant.

Voyons un exemple concret. En faisant une requête `HTTP`, méthode `GET`, à l'hôte `https://api.open-meteo.com` qui est une API météo gratuite avec le chemin `/v1/forecast` et la requête au format clés-valeurs `?latitude=46.516&longitude=6.6328&current=precipitation`, on obtient une réponse `HTTP 200` indiquant "OK" avec le contenu suivant :

```
{
  "latitude": 46.52,
  "longitude": 6.64,
  "generationtime_ms": 0.0147819519042969,
  "utc_offset_seconds": 0,
  "timezone": "GMT",
```

```
  "timezone_abbreviation": "GMT",
  "elevation": 461,
  "current_units": {
    "time": "iso8601",
    "interval": "seconds",
    "precipitation": "mm"
  },
  "current": {
    "time": "2025-10-19T14:45",
    "interval": 900,
    "precipitation": 0
  }
}
```

Ce qui nous intéresse est surtout la valeur `precipitation` dans `current`, qui est à `0` pour indiquer qu'il ne pleut pas.

Exercice Utilisez donc cette API de météo. Ouvrez `api.py` dans le dossier d'exercices pendant le cours et suivez les instructions.

Pour la partie 1, après avoir transformé le texte en JSON via `obj = json.loads(result)`, accédez à `result["current"]["precipitation"]`.

Pour la partie 2, ajoutez `,temperature_2m` à la valeur `current` (qui est actuellement juste `precipitation`), puis accédez à `result["current"]["temperature_2m"]`.

Exercice Maintenant, voici un exercice qui combine ce que vous avez appris.

Créez une interface graphique qui :

- Demande la latitude et longitude
- Donne le choix entre précipitation & température
- Affiche le résultat

N'hésitez pas à réutiliser le code que vous avez déjà écrit !

Cet exercice n'a pas de solution spécifique, tout dépend de vous.

Comment établir des fondations stables ?

On vous demande d'exécuter un ancien script Python écrit il y a longtemps par un collègue. Le script commence par `import simplejson`, puis utilise la bibliothèque `simplejson`. Vous téléchargez la bibliothèque, exécutez le script... et obtenez une erreur `NameError: name 'scanstring" is not defined`.

Malheureusement, comme le script ne précise pas la version de la bibliothèque attendue, vous devez maintenant la déterminer par essais et erreurs. Pour les crashs tels que des fonctions manquantes, cela peut être fait relativement rapidement en passant en revue les versions dans une recherche binaire. Cependant, il est également possible que votre script donne silencieusement un résultat erroné avec certaines versions de la bibliothèque. Par exemple, le script dépend peut-être d'une correction de bug effectuée à un moment précis, et l'exécution du script avec une version de la bibliothèque plus ancienne que celle-ci donnera un résultat incorrect, mais pas de manière évidente.

Les versions sont des versions spécifiques, testées et nommées. Par exemple, ”Windows 11” est une version, tout comme ”simplejson 3.17.6”. Les versions peuvent être plus ou moins spécifiques ; par exemple, ”Windows

11” est un nom de produit général avec un ensemble de fonctionnalités, et certaines fonctionnalités mineures ont été ajoutées dans des mises à jour telles que ”Windows 11 22H2”.

Les composants typiques d'une version comprennent un numéro de version majeur et mineur, parfois suivi d'un numéro de correctif et d'un numéro de build ; un nom ou parfois un nom de code ; une date de sortie ; et éventuellement d'autres informations.

Dans le cadre d'une utilisation classique, le changement du numéro de version majeur correspond à des modifications importantes et à l'ajout de nouvelles fonctionnalités, le changement du numéro de version mineur correspond à des modifications mineures et à des corrections, et le changement des autres éléments, tels que le numéro de correctif, correspond à des corrections mineures qui peuvent passer inaperçues pour la plupart des utilisateurs, ainsi qu'à des correctifs de sécurité.

Les schémas de versionnement peuvent être plus formels, comme le versionnement sémantique, un format couramment utilisé dans lequel les versions comportent trois composants principaux : `Major.Minor.Patch`. L'incrémentation du numéro de version majeur est réservée aux modifications qui rompent la compatibilité, le numéro de version mineur est réservé aux modifications qui ajoutent des fonctionnalités tout en restant compatibles, et le numéro de patch est réservé aux modifications compatibles qui n'ajoutent pas de fonctionnalités. Cependant, la définition des modifications ”compatibles” n'est pas objective. Certaines personnes peuvent considérer qu'une modification rompt la compatibilité même si d'autres la jugent compatible.

Voyons trois façons d'utiliser les versions : publier des versions, désigner des API publiques comme obsolètes si cela est vraiment nécessaire, et utiliser les versions de vos dépendances.

Publier des versions

Si vous autorisiez vos clients à télécharger votre code source et à le compiler quand ils le souhaitent, il serait difficile de savoir qui utilise quoi, et tout rapport de bug commencerait par un processus long et fastidieux visant à déterminer exactement quelle version du code est utilisée. En revanche, si un client indique qu'il utilise la version 5.4.1 de votre produit et rencontre un bug spécifique, vous pouvez immédiatement savoir à quel code cela correspond.

Fournir des versions spécifiques aux clients signifie offrir des garanties spécifiques, telles que ”la version X de notre produit est compatible avec les versions Y et Z du système d'exploitation” ou ”la version X de notre produit sera prise en charge pendant 10 ans avec des correctifs de sécurité”.

Vous n'avez pas à maintenir une seule version à la fois ; les produits ont généralement plusieurs versions sous support actif, comme Java SE.

En pratique, les versions sont généralement des branches différentes dans un dépôt. Si une modification est apportée à la branche ”principale”, vous pouvez alors décider si elle doit être transférée (”cherry-pick”) vers certaines des autres branches. Les correctifs de sécurité sont un bon exemple de modifications qui doivent être transférées vers toutes les versions encore prises en charge.

Désigner des API publiques comme obsolètes

Il arrive parfois que vous vous rendiez compte que votre base de code contient des erreurs graves qui entraînent des problèmes, et que vous souhaitiez les corriger d'une manière qui rompt techniquement la compatibilité, mais qui offre toujours une expérience raisonnable à vos clients. C'est à cela que sert le concept d'API ”obsolète”.

En déclarant qu'une partie de votre interface publique est obsolète, vous indiquez à vos clients qu'ils doivent cesser de l'utiliser et que vous pourriez même la supprimer dans une version future. L'obsolescence doit être réservée aux cas qui posent un réel problème, et non simplement une gêne. Par exemple, si les garanties fournies par une méthode spécifique obligent l'ensemble de votre base de code à utiliser une conception sous-optimale qui ralentit tout le reste, il peut être judicieux de supprimer cette méthode. Un autre bon exemple est celui des méthodes qui, en raison de subtilités sémantiques, facilitent accidentellement l'introduction de bugs ou de failles de sécurité.

Par exemple, la méthode `Thread::checkAccess` de Java a été rendue obsolète dans Java 17, car elle dépend du gestionnaire de sécurité Java, que très peu de personnes utilisent dans la pratique et qui limite l'évolution de la plate-forme Java, comme l'indique JEP 411.

Voici un exemple d'obsolescence moins raisonnable dans Python :

```
>>> from collections import Iterable
DeprecationWarning: Using or importing the ABCs
from "collections"
instead of from "collections.abc"
is deprecated since Python 3.3,
and in 3.10 it will stop working
```

Certes, avoir des classes dans le "mauvais" module n'est pas idéal, mais le coût de la maintenance de la rétrocompatibilité est faible. Casser tout le code qui s'attend à ce que les "Abstract Base Collections" se trouvent dans le "mauvais" module causerait probablement plus de problèmes que cela n'en vaut la peine.

Dans la pratique, rendre une fonctionnalité obsolète signifie réfléchir à la question de savoir si le coût vaut le risque, et si c'est le cas, utiliser la méthode de votre langage pour rendre une fonctionnalité obsolète, telle que `@Deprecated(...)` en Java ou `[Obsolete(...)]` en C#.

Utilisation des versions

L'utilisation de versions spécifiques de vos dépendances vous permet de disposer d'un environnement connu et fiable que vous pouvez utiliser comme base solide pour travailler. Vous pouvez mettre à jour les dépendances selon vos besoins, en utilisant les numéros de version comme indication du type de changements à attendre.

Cela ne signifie pas pour autant que vous devez vous fier à 100 % à toutes les dépendances pour respecter les directives de compatibilité telles que le versionnement sémantique. Même ceux qui s'efforcent de respecter ces directives peuvent commettre des erreurs, et la mise à jour d'une dépendance de la version 1.3.4 à la version 1.3.5 pourrait endommager votre code en raison d'une telle erreur. Mais au moins, vous savez que votre code fonctionnait avec la version 1.3.4 et vous pouvez y revenir si nécessaire. Le pire scénario, qui est malheureusement courant avec le code écrit il y a un certain temps, est celui où vous ne pouvez plus compiler le code car elle ne fonctionne pas avec les dernières versions de ses dépendances et que vous ne savez pas quelles versions sont requises. Vous devez alors passer beaucoup de temps à déterminer quelles versions fonctionnent et lesquelles ne fonctionnent pas, et à les noter afin que vous n'ayez pas à recommencer à l'avenir.

En pratique, pour gérer les dépendances, les ingénieurs logiciels utilisent des gestionnaires de paquets, tels que Gradle pour Java, qui gèrent les dépendances en fonction des versions :

```
testImplementation "org.junit.jupiter:junit-jupiter-api:5.8.1"
```

Cela permet d'obtenir facilement la bonne dépendance en fonction de son nom et de sa version, sans avoir à la rechercher manuellement en ligne. Il est également facile de mettre à jour les dépendances ; les gestionnaires de paquets peuvent même vous indiquer si une version plus récente est disponible. Vous devez toutefois faire attention aux versions "génériques" :

```
testImplementation "org.junit.jupiter:junit-jupiter-api:5.+"
```

Non seulement une telle version peut entraîner une rupture de votre code, car elle utilisera silencieusement une version plus récente lorsqu'elle sera disponible, qui pourrait contenir un bug qui perturbe votre code, mais vous devrez également passer du temps à déterminer quelle version vous utilisiez auparavant, car elle n'est pas consignée.

Pour les dépendances importantes telles que les systèmes d'exploitation, une façon simple de sauvegarder l'ensemble de l'environnement sous la forme d'une seule grande "version" consiste à utiliser une machine virtuelle ou un conteneur.Une fois qu'elle est construite, vous savez qu'elle fonctionne et vous pouvez distribuer votre logiciel sur cette machine virtuelle ou ce conteneur. Cela est particulièrement utile pour les logiciels qui doivent être conservés à l'identique pendant de longues périodes, tels que le code scientifique.

Résumé

Dans ce cours, vous avez appris :

- Les bases des interfaces graphiques, dont les entrées, sorties, et évènements
- Les ”design patterns”, y compris MVC, MVP, MVVM, et Middleware
- L'utilisation d'APIs, dont le transport, les formats, la sécurité, et les versions

Travail en équipe

Travailler seul signifie généralement concevoir une petite application, telle qu'une calculatrice. Pour concevoir des systèmes plus importants, il faut des équipes, composées non seulement d'ingénieurs, mais aussi de designers, de gestionnaires, de représentants de la clientèle, etc. Il existe différents types de tâches à accomplir, qui doivent être divisées et attribuées à différentes personnes : définition des exigences, conception, mise en oeuvre, vérification, maintenance, etc. Bien qu'il soit théoriquement possible de permettre à n'importe qui de modifier le code à tout moment, cela conduirait rapidement au chaos. De plus, vous souhaitez peut-être rendre votre projet « open source » et accepter des contributions extérieures.

Ce cours porte entièrement sur le travail d'équipe : qui fait quoi, quand et pourquoi ?

Objectifs

Après ce cours, vous devriez être en mesure de :

- Organiser le développement logiciel en équipe
- Comparer les méthodes de développement
- Utiliser les revues de code constructivement
- Comprendre le développement open source

Quelles sont les sources courantes de friction en équipe ?

La *friction* est une cause fréquente de retards et d'annulations de projets. Les membres d'une équipe qui sont capables d'obtenir des résultats individuellement échouent à le faire en équipe, car ils ne s'organisent pas efficacement.

N'oubliez pas qu'il n'y a pas de gagnants dans une équipe perdante. Il n'est pas utile de faire un travail « parfait » de manière isolée s'il ne s'intègre pas au reste du travail de l'équipe ou s'il y a d'autres tâches plus importantes à accomplir, comme aider un coéquipier. Un problème lié à la relation client n'est jamais dû à un seul membre de l'équipe, car cela implique que les personnes qui ont révisé le travail ont également commis des erreurs en ne détectant pas le problème. C'est l'équipe contre le problème, et non une personne contre le reste de l'équipe. Bien sûr, si la même personne continue à commettre les mêmes erreurs, l'équipe doit avoir une discussion.

Une cause particulièrement fréquente de retard est le phénomène « encore une petite chose... ». Un membre de l'équipe, chargé d'une tâche importante, ajoute sans cesse de nouvelles sous-tâches afin que le résultat global soit « meilleur » pris isolément. Mais comme la tâche est en retard, le résultat est en fait pire, car le reste de l'équipe est retardé en attendant ce résultat « meilleur ».

Lorsque vous estimez le temps nécessaire à la réalisation d'une tâche, gardez à l'esprit que le travail sur une tâche ne se limite pas au travail initial, mais qu'il implique également d'obtenir les commentaires des autres et d'intégrer ces commentaires. Si vous devez terminer en 3 jours, ne planifiez pas votre travail pour qu'il dure exactement 3 jours, car vous n'aurez alors pas le temps de le réviser et d'apporter les corrections nécessaires.

En général, il vaut mieux éviter les dépendances si possible, surtout lorsque plusieurs personnes dépendent d'une seule personne. Si cette personne n'est plus disponible ou est en retard pour une raison quelconque, les autres prendront du retard. Il peut être préférable de prévoir un peu de travail redondant plutôt que beaucoup de dépendances.

Pour mesurer le niveau de dépendance, utilisez le « *bus factor* » : combien de membres de l'équipe peuvent être renversés par un bus avant que le projet échoue ? C'est une façon plutôt morbide d'envisager les choses, on peut donc aussi penser aux vacances, aux maladies ou aux urgences personnelles. De nombreuses équipes ont un *bus factor* de 1, car il y a au moins un membre qui est la seule personne à connaître certaines tâches importantes, certains mots de passe, certains contacts externes, etc. Si ce membre quitte l'équipe, tombe malade ou est incapable de travailler pour une raison quelconque, l'équipe s'arrête de fonctionner car elle ne peut plus accomplir les tâches essentielles.

Même sans bus ni maladie, s'il n'y a qu'une seule personne qui « sait tout » et prend le temps de tout vérifier, comme un manager, le travail s'accumule pour être vérifié. Les membres de l'équipe ne pourront pas terminer leurs tâches à temps, car la vérification prend plus de temps que prévu. S'il est raisonnable qu'une seule personne ait le dernier mot sur certaines décisions, elle doit déléguer certaines vérifications et noter à l'avance les critères d'acceptation afin que les vérifications ne s'accumulent pas. Bien sûr, l'inverse, c'est-à-dire ne désigner personne comme responsable, n'est pas non plus une bonne idée, car cela conduit à un désalignement. Les membres de l'équipe effectuent toutes sortes de tâches sans consulter personne, certaines tâches doivent être abandonnées car elles ne sont pas nécessaires, d'autres sont dupliquées parce que deux membres de l'équipe n'étaient pas au courant, etc.

Une forme particulièrement problématique de désalignement se produit lorsqu'une personne travaille sur une tâche, la présente à l'équipe et se voit dire que la majeure partie de son travail doit être refaite car elle n'a pas fait ce que l'équipe souhaitait réellement. Cela peut par exemple se produire dans un projet open source sans workflow clair pour proposer de nouvelles fonctionnalités. Il doit exister un moyen pour les contributeurs potentiels, qu'ils soient membres ou non de l'équipe, d'obtenir rapidement un retour d'information sur l'acceptabilité de ce qu'ils souhaitent faire.

Comment une équipe peut-elle minimiser la friction ?

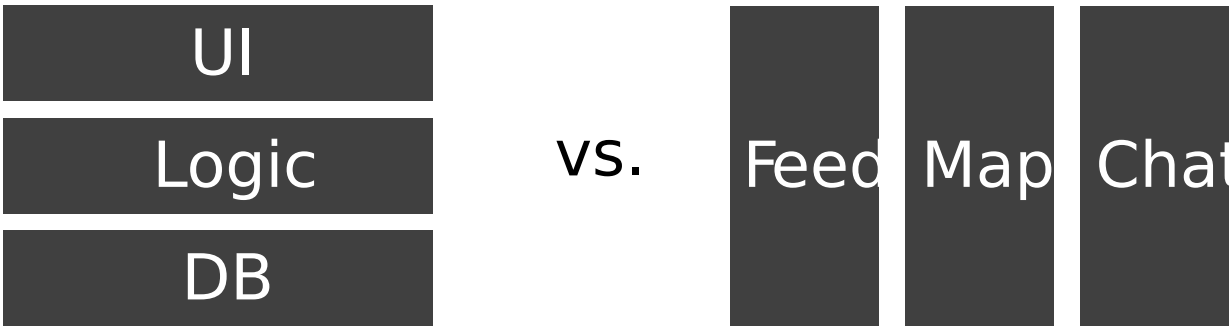
La réponse courte tient en trois points :

1. Communiquez
2. Communiquez
3. Communiquez Ce n'est qu'à moitié une plaisanterie. La communication est absolument essentielle. Personne ne peut lire vos pensées.

Une façon de communiquer au sein d'une équipe de développement logiciel consiste à envoyer du code pour qu'il soit révisé et mergé. Si vous fusionnez fréquemment votre code, le reste de l'équipe sait ce que vous avez fait et peut vous faire part de ses commentaires régulièrement. En revanche, si vous disposez d'une branche à longue durée de vie sur laquelle vous poussez du code sans demander de commentaires et sans essayer de le merge, le reste de votre équipe ne saura pas ce que vous faites, et vous risquez de vous rendre compte trop tard que votre travail n'est pas nécessaire ou qu'il fait double emploi avec celui de quelqu'un d'autre.

Au lieu d'ajouter « une chose de plus » à une tâche, faites le contraire : divisez-la en sous-tâches qui peuvent idéalement être effectuées en parallèle afin de pouvoir être attribuées à plusieurs personnes. Même si vous êtes le seul à travailler sur toutes les sous-tâches, fusionner de petits morceaux de code qui effectuent chacun une sous-tâche spécifique permet à votre équipe de rester plus facilement synchronisée.

Il existe conceptuellement deux façons de diviser le travail : soit « horizontalement » en termes de couches d'application, soit « verticalement » en termes de fonctionnalité de bout en bout :



Par exemple, l'ensemble du travail sur la base de données pourrait être attribué à une personne, l'ensemble du travail sur l'interface utilisateur à une autre, et ainsi de suite. Cependant, si la personne chargée de la base de données ne peut pas faire son travail, par exemple pour cause de maladie, alors quoi que fassent les autres membres de l'équipe, rien ne fonctionnera de bout en bout. De plus, si les mêmes personnes sont continuellement affectées aux mêmes couches, le bus factor devient 1. Au lieu de cela, les membres de l'équipe devraient être affectés à des fonctionnalités, par exemple une personne chargée de la connexion des utilisateurs, une autre chargée des messages d'information et une autre chargée du chat.

Pour les tâches qui nécessitent vraiment une coordination, telles que la définition des interfaces des modules d'une application, il est préférable d'affecter plusieurs personnes à la sous-tâche qui doit être coordonnée, puis de les laisser travailler en parallèle sur des sous-tâches dépendantes. Par exemple, au lieu que Alice conçoive la classe `User` et développe le flux utilisateur, puis que Bob réutilise cette classe pour développer le profil utilisateur, vous pouvez demander à Alice et Bob de conçoivent ensemble la classe `User`, puis travaillent en parallèle sur le flux et le profil. De cette façon, non seulement Bob peut commencer et donc terminer plus tôt, mais vous n'aurez pas à faire face à une situation où Bob doit repenser ce qu'Alice a fait parce qu'il ne lui a pas parlé de certaines fonctionnalités nécessaires pour le profil utilisateur.

Une limite importante de la coordination est la taille de l'équipe. Si vous avez une équipe de 3 personnes, vous n'avez besoin que de 3 canaux de communication individuels. Avec une équipe de 4 personnes, il y a 6 communications individuelles différentes. À mesure que la taille de l'équipe augmente, le nombre de paires individuelles augmente de manière quadratique, devenant de moins en moins efficace.

Une heuristique courante est l'équipe « deux pizzas », c'est-à-dire une équipe qui pourrait être nourrie avec deux grandes pizzas, soit 4 à 8 personnes. Au-delà, il vaut mieux se diviser en deux équipes qui travaillent sur des parties entièrement distinctes du produit final. Avec une équipe de taille raisonnable, vous pouvez avoir des discussions fréquentes afin que tout le monde soit synchronisé et que les gens n'aient pas à jeter le travail qui ne convient pas.

Une forme extrême de discussion fréquente est le « *pair programming* » : au lieu que deux programmeurs se coordonnent de manière asynchrone, par exemple par e-mail ou messagerie instantanée, ils peuvent tous deux s'asseoir devant le même ordinateur, l'un agissant comme le « pilote » chargé du clavier et de la souris, et l'autre comme le « passager » donnant un retour instantané. Cela peut également se faire à distance par vidéoconférence. Pour les tâches qui nécessitent beaucoup d'interactions, la programmation en binôme peut être extrêmement efficace. Certaines équipes l'apprécient tellement qu'elles imposent la programmation en binôme pour toutes les tâches, sans exception.

Une forme encore plus extrême de discussion fréquente est le « *mob programming* » : tous les membres de l'équipe sont devant le même écran, avec un « pilote » et tous les autres en tant que « passagers ». Cela peut être, par exemple, une alternative à une réunion de conception. Au lieu de discuter des conceptions potentielles, toute l'équipe peut rédiger la conception en une seule session de programmation en groupe.

Quelle est la méthode d'ingénierie traditionnelle ?

Avant d'aborder la manière dont les équipes modernes s'organisent, examinons la méthode d'ingénierie traditionnelle appliquée au développement logiciel.

Le travail à accomplir est divisé en phases, telles que « conception », « mise en oeuvre » et « test ». Le coût de la correction d'un défaut dans les exigences augmente à chaque étape. Par exemple, il est facile de mettre à jour les exigences pendant leur rédaction, tandis que la réécriture d'une exigence pendant les tests implique de refaire une grande partie du travail déjà effectué, par exemple lors de la mise en oeuvre. Il est donc intuitivement utile de consacrer du temps à s'assurer que les exigences sont correctes. De plus, le fait d'avoir une structure garantit que l'équipe saura quoi faire et quand, même si elle a peu d'expérience.

L'application de cette intuition à un petit programme, tel que celui destiné aux opérations internes d'une petite entreprise, conduit à un processus simple en deux étapes :

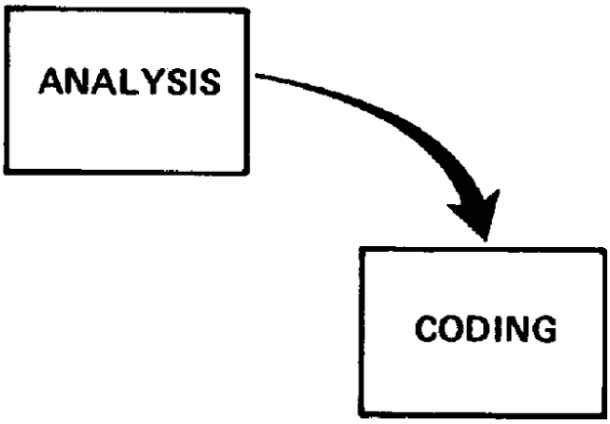


Figure 1. Implementation steps to deliver a small computer program for internal operations.

Cette figure est tirée de l'article influent de Winston W. Royce publié en 1970 et intitulé « Managing the Development of Large Software Systems » (Gérer le développement de grands systèmes logiciels). Il poursuit avec un schéma pour le développement d'un programme plus important :

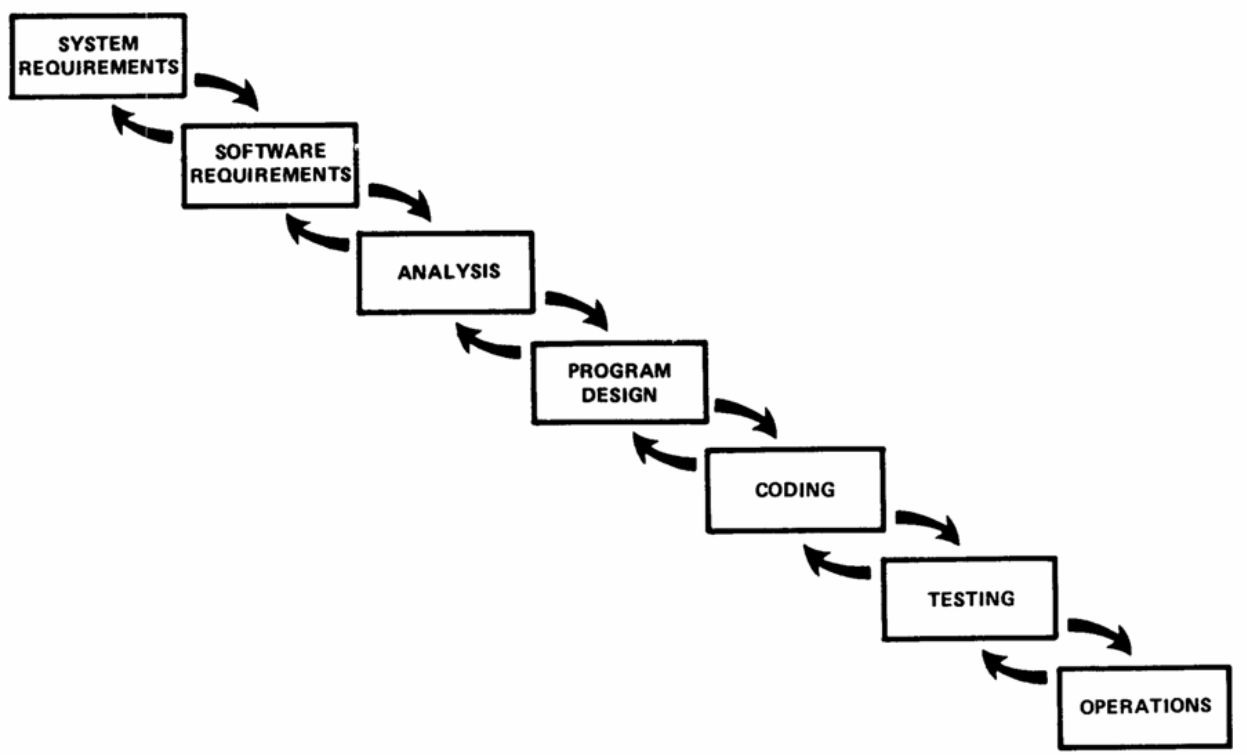


Figure 3. Hopefully, the iterative interaction between the various phases is confined to successive steps.

Notez que Royce précise explicitement qu'il ne s'agit pas d'une méthode particulièrement efficace pour développer des logiciels. Elle reste néanmoins un bon modèle pour la manière dont de nombreuses organisations développent des logiciels. Elle a été baptisée « Waterfall » (cascade) en raison de l'analogie avec le flux de l'eau : une fois qu'une étape est terminée, il est impossible de revenir en arrière, tout comme l'eau qui descend une cascade ne peut remonter.

Chaque étape de la méthode Waterfall se termine par des livrables. L'un est spécifique à l'étape, comme les exigences, la conception ou le code. Les autres sont la documentation et la validation par le client. Ce n'est que lorsque le résultat est documenté et que le client est satisfait que le développement passe à l'étape suivante.

Waterfall présente certains avantages. Tout d'abord, elle garantit que les exigences sont validées dès le début avec le client, car le développement ne peut se poursuivre qu'une fois que cela a été fait. Deuxièmement, elle impose une structure avec des objectifs clairs, car l'équipe sait toujours à quelle étape elle se trouve et quel est le livrable. Enfin, elle garantit que tout est documenté et qu'il n'y a pas de surprises, comme oublier d'effectuer une tâche qui a été retardée depuis longtemps, car ces tâches ne peuvent être retardées qu'au sein de chaque étape.

La méthode Waterfall est donc un bon choix pour les projets dont la technologie est stable, les exigences stables, l'équipe éventuellement inexpérimentée et qui impliquent une certaine bureaucratie externe imposant aux développeurs des étapes de type Waterfall. Il est assez courant que les organisations non techniques souhaitent modéliser le développement de projets techniques selon un schéma qui leur est familier, auquel cas la méthode Waterfall peut s'avérer tout à fait adaptée. Un exemple typique est celui des voyages spatiaux. Le développement d'un robot destiné à aller sur la Lune ou sur Mars est un bon cas d'utilisation de Waterfall, car les exigences sont connues et il n'y a aucune possibilité d'obtenir un feedback précoce, le lancement d'un seul robot étant extrêmement coûteux.

Cependant, Waterfall nécessite que toutes les exigences soient fixées très tôt et ne procède à la validation du produit qu'à un stade tardif, car il n'y a pas de produit fonctionnel avant la toute fin. Elle est donc peu

adaptée aux projets dont la technologie ou les exigences ne sont pas complètement stables, en particulier si l'équipe est expérimentée et qu'il n'y a pas trop de bureaucratie. Cette description correspond en fait à la plupart des projets ! La méthode Waterfall n'est généralement pas adaptée au développement logiciel moderne.

Un exemple couramment utilisé pour illustrer les inconvénients de la méthode Waterfall est le système de bagages de l'aéroport de Denver, une histoire qui a donné lieu à de nombreuses études de cas (par exemple, ici). Les travaux sur le système de bagages du nouvel aéroport de Denver ont commencé en juin 1991, la livraison étant prévue pour octobre 1993, date d'ouverture de l'aéroport. Après de nombreux retards dus au fait que le système automatisé de bagages n'était pas prêt, l'aéroport a ouvert ses portes en février 1995. Cependant, à ce moment-là, le système n'était utilisé que pour une partie des vols d'une seule compagnie aérienne, car il ne pouvait répondre à aucune des exigences initiales. Le développement a été entravé par des exigences trop complexes pour être mises en oeuvre de manière réaliste et par des retours tardifs qui ont entraîné des changements majeurs, deux symptômes d'un développement des exigences isolé du reste du processus. D'autres études de cas ont été rédigées, par exemple les difficultés rencontrées par Ericsson, là encore en raison de modifications tardives des exigences et de tests compromis pour respecter les délais du projet.

Winston W. Royce a conclu son article par un plan plus « réaliste » pour le développement de logiciels, qui est beaucoup plus complexe que les diagrammes originaux :

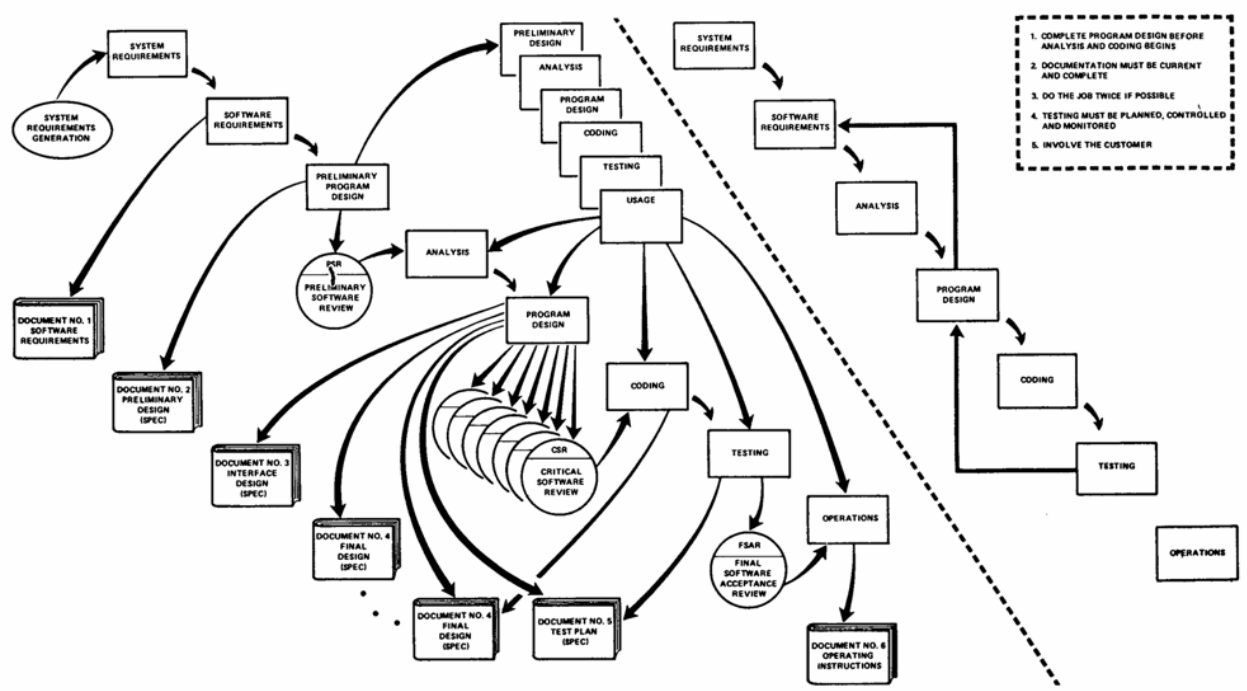


Figure 10. Summary

Nous n'aborderons pas ce diagramme plus en détail, mais il est important de rappeler que même l'article originel sur la méthode Waterfall ne recommandait pas réellement cette méthode pour le développement logiciel en général. Il existe de nombreuses autres méthodologies de développement logiciel, souvent inspirées des étapes de base de la méthode Waterfall. Par exemple, le « modèle en V » tente de représenter Waterfall avec davantage de liens entre la conception et les tests, et le « modèle en spirale » est conçu pour minimiser les risques en effectuant de nombreuses itérations du même cycle.

Exercice Pour lequel des projets suivants pourriez-vous utiliser Waterfall ? Pourquoi ou pourquoi pas ?

- Une application compagnon pour campus
- Une fusée étudiante pour une compétition interuniversitaire
- Le noyau Linux
- Une réécriture de Microsoft Word (pour obtenir des fonctionnalités similaires avec un code plus simple)

- Une application compagnon pour le campus est un très mauvais candidat, car ses besoins ne sont pas clairs, les fonctionnalités possibles étant extrêmement nombreuses, et il faudrait régulièrement recueillir les commentaires des utilisateurs finaux pendant le développement
- Un concours de fusées étudiantes est un bon candidat, pour les mêmes raisons que celles qui s'appliquent aux voyages spatiaux en général
- Le noyau Linux pourrait utiliser Waterfall pour des sous-projets spécifiques, tels que la conception d'un pilote logiciel pour un matériel spécifique, mais pour l'ensemble du projet, Waterfall ne fonctionnerait pas
- La réécriture d'un logiciel, ou une « version 2 » en général, peut être un bon cas d'utilisation pour Waterfall, car les exigences sont précisément connues

Comment les équipes modernes s'organisent-elles ?

En réaction aux lacunes des processus rigides de type Waterfall, les processus modernes tentent d'aller dans la direction opposée. Au lieu d'attendre les commentaires tardifs des utilisateurs sur le produit, recueillez leurs commentaires aussi souvent que possible. Au lieu d'un long cycle de développement au terme duquel le produit n'est utilisable qu'à la fin, prévoyez plusieurs cycles de développement courts qui aboutissent chacun à une amélioration utilisable. Au lieu d'une bureaucratie lourde autour des délais et de la documentation, optez pour une bureaucratie légère et concentrez-vous sur les résultats.

Ces principes ont été formalisés dans le "Manifeste agile", une déclaration rédigée par de nombreuses personnes intéressées par l'amélioration du développement de logiciels, qui comporte quatre valeurs fondamentales :

- Les individus et les interactions plutôt que les processus et les outils
- Les logiciels fonctionnels plutôt que la documentation exhaustive
- La collaboration avec les clients plutôt que la négociation de contrats
- Répondre au changement plutôt que suivre un plan Comme ils l'affirment, bien qu'ils accordent de l'importance aux éléments de droite, ils accordent davantage d'importance à ceux de gauche. Ils présentent en outre des principes qu'ils considèrent comme fondamentaux pour les logiciels agiles.

Il est important de noter la différence entre le développement « agile », principe général auquel fait référence ce manifeste, et les différentes formes d'« Agile » avec un A majuscule qui ont vu le jour au fil des ans. De nombreuses entités ont inventé leur propre version du développement agile, qu'elles ont formalisée en processus complexes qu'elles vendent aux entreprises comme une solution « Agile » magique à tous leurs problèmes de développement. En général, si cela nécessite des formations ou des certifications, il est peu probable qu'il s'agisse d'une méthode de développement agile.

La forme la plus courante de développement agile pratiquée aujourd'hui est *Scrum*, qui a été décrite par ses auteurs de la manière suivante :

> La philosophie déclarée et acceptée pour le développement de systèmes est que [le] processus de développement de systèmes est une approche bien comprise qui peut être planifiée, estimée et menée à bien. Il s'agit là d'un postulat erroné. Scrum affirme que le processus de développement de systèmes est un processus imprévisible et complexe qui ne peut être décrit que de manière approximative comme une progression globale. Scrum définit le processus de développement de systèmes comme un ensemble d'activités souples qui combine des outils et des techniques connus et fonctionnels avec le meilleur de ce qu'une équipe de développement peut concevoir pour construire des systèmes. Ces activités étant souples, des contrôles sont utilisés pour gérer le processus et les risques inhérents.

L'idée centrale est que nous ne pouvons véritablement contrôler le développement qu'à un niveau élevé, lorsque nous réfléchissons à la progression globale, car les étapes individuelles sont imprévisibles. Nous devons donc ajouter quelques garde-fous simples au niveau inférieur afin d'atténuer les risques à ce niveau.

Scrum se compose des éléments suivants :

- L'objectif du produit, qui est l'objectif vers lequel l'équipe travaille, par exemple un système logiciel spécifique pour un client
- Du « backlog » de produit, une liste ordonnée d'éléments à réaliser pour atteindre l'objectif produit, qui peut être élargie si nécessaire lorsque de nouveaux besoins apparaissent
- D'une séquence de « sprints », qui subdivisent le travail en unités de temps, chacune avec :
  - Son objectif, la partie du backlog de produit qui sera réalisée pour ce sprint et qui se transformera en « incréments » fonctionnels vers l'objectif produit
  - Son « backlog », une liste ordonnée de tâches à accomplir pour atteindre l'objectif du sprint Dans chaque sprint, en plus de son travail quotidien, l'équipe Scrum commence par une réunion de *Sprint Planning* pour définir le sprint, et termine par une *Sprint Review* orientée vers le client et une *Sprint Retrospective* pour décider des améliorations à apporter au processus.

Examinons maintenant chacun de ces éléments plus en détail.

L'équipe Scrum est composée d'un *Scrum Master*, qui facilite le processus, d'un *Product Owner*, qui représente les clients, et de développeurs. Il n'y a ni hiérarchie ni sous-équipes au sein d'une équipe Scrum. Le Scrum Master n'est pas un manager, mais plutôt un « leader serviteur », qui aide le reste de l'équipe en organisant et en animant les réunions souhaitées par l'équipe, en gérant les contraintes externes, etc. En général, comme ce travail ne représente pas une charge de travail à temps plein, le Scrum Master est également développeur. Le Product Owner est le seul représentant de tous les clients et gère les backlogs. Le Product Owner a le dernier mot sur ce qui est ajouté au backlog et dans quel ordre, et personne d'autre n'est autorisé à modifier directement le backlog. En particulier, les utilisateurs ne peuvent pas demander directement des fonctionnalités à des développeurs individuels ni modifier eux-mêmes le backlog, car cela conduirait au chaos.

Lorsque vous travaillez dans des équipes de développement logiciel, vous pouvez rencontrer d'autres rôles tels que « Tech Lead », « Engineering Manager », « Product Manager », etc. Ces rôles ont un objectif et peuvent être utiles dans certains cas, mais ils ne font pas partie de Scrum, même si de nombreuses organisations prétendent adhérer à Scrum tout en ayant de nombreux rôles sans rapport avec celui-ci et parfois même pas les rôles Scrum de base. Comme Scrum et le développement agile en général sont très populaires, il existe un écart important entre les définitions originales et ce que les gens font dans la pratique sous ces noms.

Un sprint dans Scrum consiste en un objectif et un backlog, réalisables dans un laps de temps donné. En général, tous les sprints ont la même durée, généralement comprise entre 1 et 4 semaines. Scrum étant agile, si un sprint n'est plus utile ou nécessite des changements majeurs, il n'est pas nécessaire de le poursuivre uniquement pour respecter le processus. Au contraire, le Product Owner peut modifier ou annuler un sprint si nécessaire.

La réunion de planification du sprint permet de définir le backlog du sprint afin de décider des incréments de travail que l'équipe souhaite réaliser au cours du sprint. Un incrément est une combinaison d'un élément du backlog du produit et d'une "*Definition of Done*", c'est-à-dire une déclaration claire qui définit si une tâche a été accomplie d'une manière qui nécessite un minimum d'interprétation. Au cours de la réunion de planification, l'équipe hiérarchise les tâches, par exemple en fonction de ce qui serait le plus utile pour l'utilisateur, de ce qui présente le moins de risques, etc. Les éléments du backlog de produit sont divisés en tâches sur lesquelles les développeurs individuels peuvent travailler, en estimant leur temps d'achèvement afin qu'aucune tâche ne soit trop longue et qu'aucun développeur n'ait trop ou trop peu de travail.

Voici, par exemple, quelques *bons* exemples de tâches, avec une définition implicite de ce qui est terminé :

- « *Ajouter un emplacement public aux profils des utilisateurs* »
- « *Permettre aux utilisateurs de trier les restaurants par type ou par note* »

Voici quelques *mauvais* exemples de tâches, qui sont beaucoup trop vagues :

- « *Mettre en place un profil utilisateur* » (que doit-il contenir ? Qu'est-ce qui est modifiable et qu'est-ce qui ne l'est pas ? Qu'est-ce qui est visible publiquement et qu'est-ce qui ne l'est pas ?)
- « *Générer un meilleur contenu de cours* » (que signifie « meilleur » ? Comment le contenu sera-t-il généré ?)

Idéalement, une seule tâche devrait représenter 1 à 2 jours de travail pour un seul développeur, avec des estimations de temps réalistes. Il est important d'éviter la mentalité « tout ira parfaitement bien » et de garder à l'esprit la nécessité de tester, de réviser le code et de s'adapter aux commentaires issus de la révision. Même si les tâches peuvent parfois être plus importantes que cela, plus une tâche est importante, plus elle risque d'être mal spécifiée et de causer plus tard des conflits de fusion.

La division des éléments du backlog en tâches relève davantage de l'art que de la science, et les développeurs s'améliorent à mesure qu'ils s'y exercent. Prenons par exemple l'élément « *Permettre aux utilisateurs de trouver des lieux d'intérêt à proximité* ». Nous pourrions le diviser en « *Permettre aux utilisateurs de définir ou de détecter automatiquement leur emplacement* », « *Importer les emplacements depuis la base de données* » et « *Trier les lieux par distance* ». Cela dépend bien sûr de la complexité de chaque élément dans son contexte. Il se peut que la détection automatique de l'emplacement soit difficile sur la plate-forme spécifique sur laquelle fonctionne l'application, et que la première tâche doive plutôt être divisée en deux tâches, par exemple.

Il est important de noter que les éléments du backlog de produit sont toujours traités de haut en bas, car le backlog est ordonné. Le Product Owner peut décider de réorganiser le backlog de produit en fonction des commentaires des clients.

Pour en revenir aux composants du sprint, pendant la partie active du sprint, les développeurs doivent se réunir pour une réunion quotidienne dite "*Daily Scrum*", souvent appelée « *standup* », car elle doit être suffisamment courte pour que la plupart des participants n'aient pas besoin de s'asseoir. Le standup peut se dérouler en personne, par vidéoconférence ou même par messagerie instantanée, selon les préférences de l'équipe. Au cours de la réunion standup, chaque développeur résume rapidement ce qu'il a fait la veille, ce sur quoi il va travailler aujourd'hui et s'il est bloqué par quelque chose.

Outre le fait de synchroniser l'équipe, l'objectif principal de la réunion standup est d'identifier les obstacles : les problèmes qui empêchent un développeur d'accomplir son travail. Tout ce qui ressemble à « Je ne peux pas continuer mon travail parce que... » est un obstacle, même si ce n'est pas formulé ainsi. En particulier, tout le monde ne sait pas forcément qu'il est bloqué ! Par exemple, quelqu'un peut supposer qu'il est normal qu'il ne puisse pas accomplir sa tâche de manière simple et déclarer qu'il cherche une méthode plus complexe qui prendra beaucoup plus de temps. Un coéquipier peut alors déclarer qu'il s'attendait vraiment à ce que la méthode simple fonctionne et suggérer aux deux personnes de se rencontrer peu après la réunion debout pour en discuter, dans l'idéal afin de débloquer leur collègue.

À la fin du sprint, lors de la réunion de revue du sprint, l'équipe présente son travail au propriétaire du produit, généralement sous la forme d'une démonstration du logiciel. Ce n'est pas nécessairement le seul moment où l'équipe peut faire une démonstration. Il est tout à fait acceptable de s'entretenir avec le propriétaire du produit pendant un sprint pour lui demander son avis sur une démonstration, par exemple. La revue sert à discuter des résultats liés au produit, tels que ce que l'équipe a appris sur la faisabilité et la hiérarchisation possible des tâches futures.

Enfin, il y a la rétrospective du sprint, qui porte sur le processus. L'équipe se réunit pour discuter de ce qui a fonctionné et de ce qui n'a pas fonctionné, dans une sorte d'« autopsie » du sprint. Un résultat courant d'une rétrospective est une liste de trois catégories : « Arrêter de faire..., Commencer à faire..., Continuer à faire... ». Comme Scrum est axé sur l'agilité, même le processus Scrum lui-même peut être modifié si nécessaire. Par exemple, l'équipe estime peut-être qu'il serait préférable de n'avoir une réunion debout qu'un jour sur deux et souhaite tester cette solution pendant un sprint.

Dans l'ensemble, n'oubliez pas que les processus agiles sont avant tout une question d'agilité. Si vous trouvez un bug mineur, vous pouvez et devez le corriger immédiatement. Il n'est pas nécessaire d'ouvrir un ticket officiel et de demander l'avis du Product Owner si vous pouvez le corriger en 10 minutes. Si vous rencontrez un problème inhabituel qui, selon vous, pourrait compromettre le développement futur, tel qu'une bibliothèque externe ne disposant pas des fonctions attendues par l'équipe pour les tâches futures, discutez-en rapidement avec l'équipe. Si vous avez une idée pour gagner du temps, par exemple un moyen de réaliser 80 % d'une tâche en 20 % du temps prévu, discutez-en rapidement avec l'équipe et le Product Owner. Ils seront peut-être d'accord et la définition de « terminé » pour votre tâche changera. Ne suivez pas le processus pour le simple

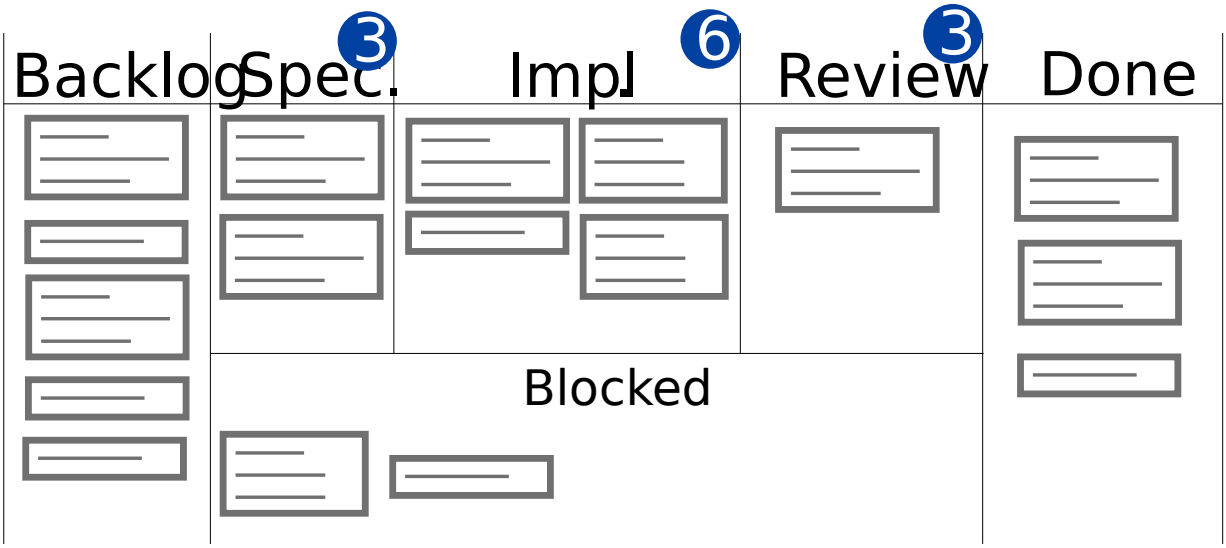
plaisir de suivre le processus.

Gardez également à l'esprit l'objectif global lorsque vous réfléchissez aux éléments du backlog et aux tâches. Par exemple, si vous souhaitez construire un grand manoir avec des murs en pierre renforcés, il est inutile de commencer par des murs en pierres sèches comme forme simple de mur, car vous aurez besoin de mortier pour construire la maison. Votre première tâche après avoir construit le mur en pierres sèches sera donc de le démanteler et de construire un mur avec du mortier cette fois-ci. Il faut de l'expérience et de l'expertise pour savoir quels « raccourcis » sont réellement des raccourcis et lesquels vous mènent sur la mauvaise voie.

Exercice Créez un objectif produit et un backlog produit, avec au moins 5 éléments, pour un projet que vous aimeriez réaliser.

Il peut s'agir de n'importe quoi, par exemple une application de podcasts, un gestionnaire de tournois d'échecs, un site web d'inscription à des événements, etc.

Si les itérations en sprints courts sont une bonne chose, les itérations en sprints extrêmement courts sont-elles extrêmement bonnes ? Peut-être. C'est l'idée centrale derrière le Kanban en tant que philosophie de développement logiciel. Au lieu de sprints explicites, le Kanban implique un tableau avec des colonnes pour les différents états des éléments. L'équipe maintient la colonne « Backlog » dans un ordre trié, puis déplace les éléments au fur et à mesure du développement :



Il est important de noter qu'il existe une limite pour chaque colonne, ce qui signifie qu'il ne peut y avoir plus d'un certain nombre d'éléments dans un statut donné à un moment donné. Par exemple, si vous souhaitez soumettre votre code à une revue pour un élément, mais que la colonne Revue est pleine, vous devez effectuer une revue de code pour l'un des éléments de cette colonne afin de libérer une place. Pour compenser cela, il existe une zone « bloquée » pour les éléments qui sont bloqués pour des raisons indépendantes de la volonté de l'équipe, par exemple parce qu'ils attendent une autre équipe.

Le concept de « tableau Kanban » est largement utilisé même en dehors du Kanban, souvent sans limite du nombre d'éléments par statut. Il est courant d'utiliser un tableau de type Kanban pour les projets Scrum, par exemple.

Comment fournir et recevoir du feedback utile sur du code ?

Que faire une fois que vous avez terminé une tâche de codage ? La merge directement dans la branche principale ? Malheureusement, les êtres humains sont imparfaits et nous commettons tous des erreurs. Il peut y avoir des bugs, des cas limites non traités, des tests manquants, de mauvais choix de conception, etc. Les interactions entre le code nouveau, modifié et existant peuvent cacher toutes sortes de problèmes. Par exemple, une nouvelle sous-classe peut provoquer le plantage du code existant, soit parce que la nouvelle classe ne respecte pas les postconditions de sa classe de base, soit parce que le code existant faisait des hypothèses plus fortes que les postconditions de la classe existante. Plus délicat encore, il peut manquer du code, par exemple pour gérer des cas limites spécifiques. Il est plus difficile de déterminer quel code n'a pas été ajouté mais devrait l'être que de déterminer si le code ajouté est correct.

Les revues de code réduisent la probabilité d'erreurs en faisant examiner le code par une autre personne. Ce n'est bien sûr pas un concept nouveau, puisqu'il existe dans tout autre processus de création d'un résultat, comme donner son avis sur le brouillon d'un livre. Et cela a commencé il y a longtemps, même dans le domaine du génie logiciel, avec Ada Lovelace suppliant Charles Babbage de cesser de modifier ses programmes, car il y introduisait des erreurs.

Si quelqu'un suggère des modifications au code et qu'une autre personne les examine, qui est responsable des bugs restants dans le code après avoir été merge sur la branche principale ? Il s'agit d'une question piège : la réponse est « toute l'équipe ». L'objectif n'est pas de blâmer les individus pour leurs erreurs humaines, car nous savons que celles-ci continueront de se produire même si quelqu'un d'autre lit le code, mais simplement moins souvent. Si la même personne commet systématiquement les mêmes erreurs, une intervention est nécessaire, mais sinon, l'objectif est de fournir un logiciel de qualité, et non de déterminer qui a commis le plus d'erreurs.

Une forme de revue est le ”pair programming” : si deux personnes travaillent sur un morceau de code, elles pourraient décider de renoncer au processus de revue, car le code a déjà été vérifié par une deuxième paire d'yeux. D'un autre côté, les deux personnes peuvent partager les mêmes préjugés et les mêmes angles morts si elles ont travaillé ensemble, les équipes doivent donc décider comment traiter le code programmé en binôme en termes de revue.

Les revues de code sont particulièrement importantes si vous recevez du code provenant d'une personne que vous ne connaissez pas ou en qui vous n'avez pas confiance, comme une proposition d'une personne sur Internet visant à fusionner du code dans votre projet accessible au public. Cette personne a peut-être écrit un excellent code, ou elle a peut-être commis de nombreuses erreurs. Elle pourrait même être malveillante !

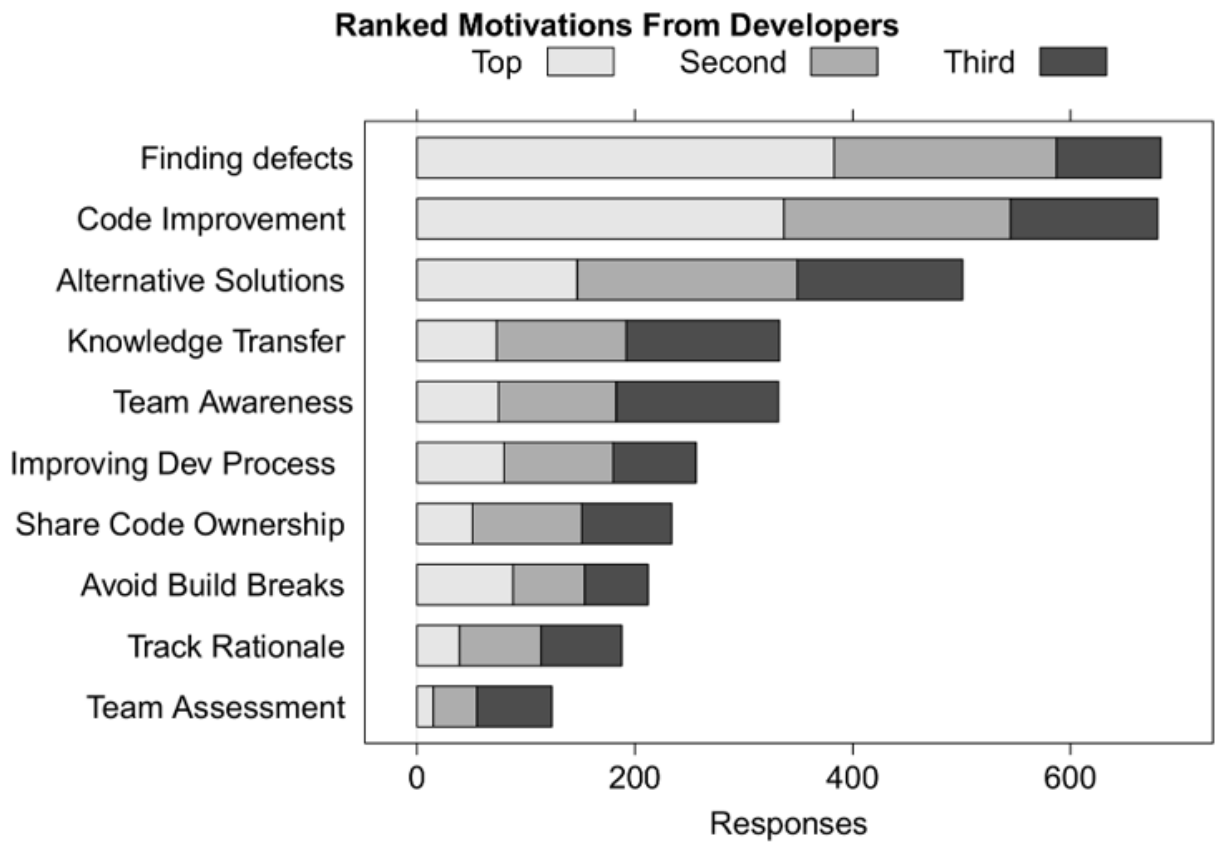
Alors, qu'est-ce qu'une revue de code ? Conceptuellement, c'est simple :

1. Lire le code
2. Écrire des commentaires sur le code
3. Accepter ou rejeter le code En général, le « rejet » à la dernière étape n'est pas définitif, mais s'apparente davantage à « veuillez corriger les problèmes que j'ai signalés et soumettre à nouveau le code pour une autre révision ».

Les révisions de code peuvent se faire par e-mail ou à l'aide d'outils en ligne tels que les pull requests de GitHub. Mais quel est exactement l'objectif, qui doit les effectuer et comment donner et recevoir de bonnes révisions de code ? Examinons ces quatre points tour à tour.

Objectifs de la revue du code

Au premier abord, l'objectif peut sembler se limiter à « trouver des bugs », mais en fait pas seulement. Heureusement, il existe des recherches empiriques sur la révision du code, dans lesquelles des chercheurs interrogent de nombreux ingénieurs logiciels afin de recueillir leur opinion. Voici, par exemple, les principales motivations des développeurs pour la révision, telles que recueillies par Bacchelli et Bird en 2013 :



Sans surprise, la recherche de défauts arrive en tête, suivie de près par l'amélioration du code, c'est-à-dire la proposition de meilleures méthodes même si le code soumis ne comporte pas de bugs, et les solutions alternatives, c'est-à-dire la proposition de meilleurs choix de conception. Par exemple, au lieu d'une solution de 100 lignes, la personne qui lit le code pourrait signaler qu'il existe déjà une méthode dans la base de code qui fait ce que le soumissionnaire du code voulait faire.

Vient ensuite le transfert de connaissances, autre objectif clé de la révision de code dans la pratique. Les revues de code ne sont pas un monologue, mais un dialogue entre l'auteur et le réviseur. N'oubliez pas le « bus factor » : vous ne voulez pas qu'un projet échoue parce que la seule personne qui connaissait une partie du code est malade ou quitte l'équipe. En imposant des révisions de code, vous garantissez qu'au moins deux personnes connaissent chaque partie du code. Cela peut également être une bonne occasion pour les ingénieurs seniors d'expliquer certaines choses aux ingénieurs juniors.

Auteurs de la revue de code

Qui devrait être le reliseur de code ? Si la recherche de défauts et la proposition de solutions alternatives nécessitent une connaissance approfondie du code modifié, comme l'ont constaté Bacchelli et Bird dans le même article, d'autres objectifs, tels que le transfert de connaissances, peuvent bénéficier de réviseurs moins expérimentés. Il y a un compromis évident : plus il y a de personnes qui relisent, plus il y a de commentaires et plus le transfert de connaissances est important, mais plus il faut de temps avant que le code puisse être mergé.

Dans la pratique, il est courant de désigner un « propriétaire » du code, c'est-à-dire une personne responsable de la partie du code en cours de modification. Cette procédure peut être mise en oeuvre à l'aide d'outils tels que le fichier `CODEOWNERS` de GitHub, qui exige que les pull requests soient approuvées par les personnes désignées comme propriétaires du code. Il peut être utile de demander à d'autres personnes de relire le code afin d'apprendre et de fournir des commentaires mineurs, mais cela n'est généralement pas obligatoire.

Réaliser une bonne revue de code

Examinons trois aspects de la revue de code : l'approche globale, les commentaires et certains facteurs humains.

Tout d'abord, afin d'aborder correctement la chose, vous devez prévoir du temps pour la réaliser, par exemple 10 à 15 minutes. À moins que le code ne soit particulièrement complexe, vous ne devriez pas avoir besoin de beaucoup plus de temps. Si vous avez besoin d'une heure pour comprendre une pull request, cela signifie soit que le code est trop confus, soit que la pull request devrait être divisée en plusieurs parties plus petites.

Vous pouvez lire le code en ligne sur la plateforme où la pull request est effectuée, ou localement en récupérant la branche et en l'ouvrant dans votre éditeur. Certains éditeurs sont même intégrés à des outils de revue de code, ce qui vous permet d'ajouter des commentaires depuis l'éditeur. Cependant, vous ne devriez généralement pas avoir besoin d'exécuter le code vous-même, c'est à cela que servent les tests automatisés et l'intégration continue.

Décidez quelles parties vous allez lire attentivement, parcourir rapidement et ignorer. Il peut être utile d'ignorer certains codes si vous faites confiance à la personne qui les a soumis et que les modifications ne sont pas intéressantes, comme une refonte qui a changé le nom d'une méthode ou l'ajout d'un fichier de ressources dont vous connaissez déjà le contenu. Si nécessaire, vous pouvez déléguer, par exemple parce que vous n'êtes pas sûr de votre capacité à examiner les modifications apportées à un module spécifique. Une fois votre examen terminé, documentez ces choix à l'aide de commentaires tels que « Je n'ai examiné que... parce que... » ou « Je te fais confiance... donc je n'ai pas lu... ».

La connaissance peut être acquise grâce à la discussion. Parfois, cette discussion consiste à souligner des faits, d'autres fois à poser des questions. Voici quelques exemples de **bons** types de commentaires :

- « *Je ne comprends pas pourquoi...* »
- « *Si* `x = -1` *ici, je pense que cela va planter, car...* »
- « *Pourrions-nous déplacer ce bout de code vers la classe X, afin que...* » Tous les commentaires ne doivent pas nécessairement être critiques ou signaler des problèmes. Il est judicieux de laisser des commentaires tels que « *C'est génial ! J'aime beaucoup...* » si vous pensez qu'une partie du code est bien faite !

Voici, en revanche, quelques exemples de **mauvais** commentaires et pourquoi :

- « *Je n'aime pas ça* »
  (Il ne s'agit pas de vos préférences personnelles, mais de celles de l'équipe)
- « *Améliore ça* »
  (Comment ? Ce n'est pas constructif)
- « *Ton code est stupide* »
  (Tout ce que cela fait, c'est mettre l'auteur sur la défensive, ce qui ne mène à rien)
- « *Faisons également cette modification sur ces 40 autres fichiers* »
  (Vous pouvez proposer de faire une autre pull request pour cela, mais n'étendez pas la portée d'une demande existante sauf si cela est vraiment nécessaire)
- « *Je l'aurais fait...* »
  (Quand quelqu'un d'autre effectue une tâche, il ne la fera jamais à 100 % comme vous le souhaitez)
- « *À mon avis...* »
  (Encore une fois, il s'agit de l'avis de l'équipe, vous devez étayer vos commentaires par des faits partagés, et non par des préférences personnelles)

Une façon de rendre vos commentaires plus exploitables consiste à les classer par catégorie, par exemple

- « ***Question** : Pourquoi devons-nous vérifier cette condition... ?* »
- « ***Bloqueur** : Cette condition doit inclure...* »
- « ***Mineur** : Peut-être qu'un meilleur nom pour cette méthode serait...* » De cette façon, l'auteur de la soumission sait ce qu'il doit traiter (= « bloqueurs »), ce qu'il peut choisir de prendre ou non (= « mineurs » / "*nitpicks*" en anglais) et quelles questions sont vraiment des questions par opposition à des questions rhétoriques.

N'oubliez pas de justifier vos commentaires, afin que l'auteur de la soumission dispose du contexte et puisse discuter de manière plus constructive, par exemple « *... conformément à nos conventions de formatage* » ou « *... comme nous en avons discuté jeudi dernier* ». Il se peut qu'il y ait eu un malentendu sur ce qui devait être fait jeudi dernier, par exemple. Dans ce cas, ajouter ce contexte vous permet de clarifier directement ce malentendu.

Gardez une vue d'ensemble. Vous examinez le code pour trouver des problèmes et des améliorations potentiels, mais aussi pour en apprendre davantage à son sujet. Demandez-vous si l'architecture est cohérente, quels tests pourraient manquer, etc. Il est inutile de laisser une révision composée uniquement de commentaires « mineurs ». Parfois, vous pouvez simplement laisser « *LGTM* », qui signifie « *Looks Good To Me* » (ça me semble correct), si vous pensez que le code est correct. Cela est particulièrement vrai pour les petites demandes de modification qui effectuent des tâches simples.

Enfin, n'oubliez pas les facteurs humains. Les revues de code ne visent pas à montrer qui est le plus intelligent, ce n'est pas « moi contre toi », mais « nous contre le problème ». Votre objectif global est de fournir un logiciel fonctionnel et vous collaborez avec la personne qui a soumis le code. N'écrivez pas de commentaires tels que « Tu fais... », mais concentrez-vous sur le code, par exemple « Le code fait... », afin de ne pas mettre les gens sur la défensive. Et n'oubliez pas que des malentendus culturels peuvent survenir. Par exemple, une personne originaire des États-Unis qui dit que le code est « *quite good* » veut dire qu'il est bon, mais un lecteur britannique pensera qu'il est plutôt mauvais, car les mots ont des significations différentes selon les cultures. Vous ne pouvez pas éviter les malentendus, mais vous pouvez garder leur possibilité à l'esprit afin de ne pas immédiatement faire une mauvaise interprétation et poser plutôt des questions pour clarifier les choses.

Bien recevoir une revue de code

Les révisions de code sont collaboratives, donc la personne qui soumet le code a également la responsabilité de faire de son mieux. Examinons trois aspects : aider les autres à vous aider, obtenir rapidement des commentaires et gérer les mauvaises révisions.

Commencez par résumer vos modifications dans la description de toute demande d'extraction que vous créez, en incluant tout ce que vous souhaitez signaler aux réviseurs potentiels. Réfléchissez aux questions « évidentes » qu'un réviseur pourrait se poser au départ, afin de pouvoir passer immédiatement à l'étape suivante de la conversation. Par exemple, vous pourriez dire certaines des choses suivantes :

- « *J'ai implémenté X comme décrit dans la tâche, mais cela m'a également obligé à modifier Y, car...* »
- « *J'ai utilisé le modèle Middleware, car...* »
- « *Je ne suis pas sûr de l'implémentation de...* »

Avant de demander à quelqu'un d'autre de réviser votre code, comme pour tout autre résultat que vous produisez, effectuez vous-même une révision afin de détecter les erreurs élémentaires. Peut-être avez-vous oublié de supprimer du code que vous aviez ajouté lors du débuggage. Peut-être avez-vous commenté une ligne et oublié de la décommenter. Peut-être avez-vous modifié un fichier par accident. Au lieu d'attendre que quelqu'un d'autre vous le signale, effectuez vous-même une révision pour le découvrir. Vous pouvez également laisser des commentaires sur votre propre pull request, de la même manière que vous pouvez laisser une description générale. Par exemple, vous pouvez laisser des commentaires tels que :

- « *À l'origine, je voulais écrire X ici, mais...* »
- « *Je ne suis pas sûr que ce soit le bon endroit pour cette méthode, car...* »
- « *Je prévois de réécrire ce fichier la semaine prochaine pour la fonctionnalité Y* » Ces commentaires doivent être utilisés lorsqu'ils sont importants pour la relecture, mais pas pour les futurs responsables de la maintenance. Si quelque chose doit être laissé à l'intention des responsables de la maintenance, laissez-le plutôt sous forme de commentaire dans le code.

Obtenez des commentaires dès le début, tout comme vous souhaitez obtenir rapidement les commentaires des utilisateurs dans le cadre d'un développement agile. Vous n'avez pas besoin d'attendre d'avoir entièrement mis en oeuvre une tâche pour obtenir des commentaires sur la conception ou les cas de test. Il est tout à fait acceptable de demander à un collègue « *pouvez-vous jeter un oeil [à la conception / aux tests / ...] ?* ». En

fait, des plateformes telles que GitHub ont un concept de pull requests « brouillons » (”*drafts*” en anglais) spécialement conçu à cet effet. Une pull request brouillon comporte une indication spéciale et ne peut pas être fusionnée, ce qui indique qu'elle a été ouverte uniquement pour obtenir des commentaires.

Si nécessaire, vous pouvez également diviser votre pull request en plusieurs. De la même manière que vous pouvez diviser une tâche avant de la commencer, si vous vous rendez compte après avoir terminé une partie de la tâche que la partie suivante est indépendante de la précédente, ouvrez une pull request pendant que vous travaillez sur la partie suivante. De cette façon, votre code sera relu plus rapidement et vous risquerez moins d'entrer en conflit avec la branche principale lors de la fusion.

Enfin, acceptez le fait que vous recevrez parfois de mauvaises relectures de code et que vous devrez les traiter de manière professionnelle et constructive. Tout le monde ne sait pas comment laisser une bonne revue, et certaines personnes veulent vous micromanager ou imposer leur opinion à tout le monde. Répondez à leurs commentaires en leur rappelant les justifications communes, telles que « *Conformément aux conventions de l'équipe...* », ou en faisant appel à quelqu'un d'autre pour vous aider, par exemple « *Demandons à X ce qu'il en pense* ». Si vous pensez que quelqu'un écrit des commentaires ”mineurs” sans les signaler comme tels, demandez-lui si les corrections demandées par ce commentaire peuvent attendre. Il répondra probablement oui, puis oubliera qu'il vous a demandé de les faire.

Exercice Effectuez une revue du code de cette pull request. Qu'en pensez-vous ? Y a-t-il des bugs ? Quels commentaires laisseriez-vous ?

Regardez ensuite une revue existante pour le même code. Êtes-vous d'accord avec les commentaires ? En désaccord ? Certains commentaires manquent-ils ?

Le plus gros problème est que la fonctionnalité « recherche » ne comporte pas de tests, ce qui explique que les deux fautes de frappe dans son implémentation n'aient pas été repérées.

L'appel à `sorted` dans le Model est inutile puisque cette fonction retourne une nouvelle valeur, qui ici n'est pas utilisée, au lieu de modifier son entrée.

D'un point de vue architectural, il n'est pas très logique que le présentateur formate les listes en texte, car le format exact est spécifique à la vue. De même, le présentateur ne devrait pas connaître le concept de « Ctrl+C » pour arrêter. Il pourrait peut-être plutôt traiter « exit » comme une commande, et la vue pourrait invoquer cette commande lorsque l'utilisateur exécute une fonction de sortie spécifique à la vue.

Il serait également utile de documenter le format JSON.

De nombreuses questions pourraient être discutées en équipe, telles que le besoin ou non de documentation par classe ou méthode.

Comment collaborer en open source ?

Tout d'abord, que signifie exactement « open source » ? La liberté d'utiliser le code source comme on le souhaite, par exemple pour le lire, l'utiliser, le partager, l'étudier... Il ne s'agit pas d'une définition très précise, et il n'existe d'ailleurs aucune définition objective, mais celle de l'Open Source Initiative est couramment utilisée. Il est important de noter qu'elle exige une distribution gratuite, que cette distribution concerne le code source et non une version compilée ou obscurcie, que les travaux dérivés soient autorisés et qu'il n'y ait aucune discrimination quant à l'attribution des droits. Notez que l'open source ne signifie pas que vous pouvez soumettre des correctifs et les faire fusionner. Un projet peut être open source même s'il n'accepte aucune contribution extérieure.

Les licences sont la version formelle de « qui peut faire quoi et sous quelles conditions ? ». Elles précisent, en termes juridiques, ce que veulent les auteurs du code. Il peut être tentant de penser que vous n'avez pas besoin de licence pour un code simple, mais c'est tout le contraire. Sans licence, dans la plupart des juridictions, votre code sera par défaut protégé par le droit d'auteur, ce qui signifie que vous ne donnez à

personne le droit de faire quoi que ce soit. Les gens ne toucheront généralement pas à un code qui n'a pas de licence, car en théorie, vous pourriez les poursuivre en justice, par exemple pour l'avoir réutilisé.

Alors, comment choisir une licence ? Vous pouvez consulter, par exemple, le comparatif de Wikipédia des licences open source, mais cela vous prendra un certain temps. Il existe des versions plus courtes, telles que choosealicense.com. Mais voici une version très courte et partiale :

- La licence **MIT** est destinée aux cas où vous souhaitez simplement que l'on vous crédite pour toute utilisation
- La licence **GPL** est destinée aux cas où vous souhaitez que toute utilisation soit également open source sous la licence GPL
- Les licences telles que « Unlicense » ou « WTFPL » sont destinées aux cas où vous souhaitez une forme de « domaine public », c'est-à-dire que n'importe qui peut utiliser votre code pour n'importe quoi sans condition

Maintenant que nous avons défini l'open source, à quoi ressemble un projet open source ? En termes de hiérarchie, on trouve généralement :

- Les *mainteneurs*, qui ont un accès en écriture au code et le dernier mot sur les décisions relatives au projet
- Les *contributeurs*, qui soumettent du code et des idées, mais dont les pull requests et les suggestions doivent être approuvées par les mainteneurs
- Les *utilisateurs*, qui ne contribuent pas au code

Les gens maintiennent des projets open source pour le plaisir, pour le travail et, très rarement, pour l'argent. Par exemple, si vous consultez la page des mainteneurs du noyau Linux, vous verrez que la plupart des composants du noyau ont quelqu'un qui s'en occupe, et que certains d'entre eux sont même rémunérés pour le faire. En général, cela signifie qu'une entreprise qui utilise le noyau a décidé d'affecter un ingénieur à cette tâche, au moins à temps partiel. Certains projets largement utilisés, tels que `curl`, ont même des sponsors. Certains projets commerciaux sont, pour diverses raisons, open source, comme les compilateurs C# et Visual Basic .NET.

Comment gravir les échelons et passer du statut d'« utilisateur » à celui de « contributeur » d'un projet qui vous plaît et auquel vous souhaitez apporter votre aide ? En bref :

1. Trouvez une "issue"
2. Lisez les directives de contribution du projet
3. Travaillez sur le problème localement
4. Ouvrez une demande d'extraction

Pour trouver une "issue", recherchez de grands projets qui ont étiqueté des problèmes avec des labels tels que « bonne première issue » ("*good first issue*") ou « aide recherchée » ("*help wanted*"), comme celui-ci, et communiquez votre intention de travailler dessus aux mainteneurs. Vous pouvez également ouvrir votre propre problème, par exemple traduire l'interface utilisateur dans votre langue.

Ces exemples sont tirés du dépôt `microsoft/terminal`, qui contient un guide de contribution expliquant le type de contributions acceptées, par où commencer, etc.

Avant que votre pull request ne soit examinée, vous devrez généralement signer un « Contribution License Agreement », ou « CLA » en abrégé. Il s'agit d'un document qui donne effectivement aux responsables la propriété du code spécifique que vous avez soumis, de sorte que si, à l'avenir, ils ont besoin, par exemple, de modifier la licence du projet, ils n'ont pas à rechercher les centaines de contributeurs passés qui ont soumis du code qui a été intégré dans le projet.

Enfin, n'oubliez pas que si vous contribuez à un projet open source, vous rejoignez une équipe avec des pratiques établies. Ne commencez pas votre pull request en effectuant des refactorisations aléatoires pour l'adapter à votre idée de ce que devrait être la base de code. Ne divisez pas un fichier en plusieurs fichiers simplement parce que vous préférez une certaine structure de code. Ce type de comportement provoque beaucoup de frictions, rend votre code beaucoup plus difficile à réviser et peut entraîner le rejet de votre

pull request sans révision appropriée. Évitez également d'ouvrir de grandes pull requests sans avertissement, car la plupart des mainteneurs ne sont pas intéressés par la révision de milliers de lignes de code pour une fonctionnalité qu'ils n'ont pas explicitement demandée.

Exercice Contribuez à un projet open source !

Vous pouvez par exemple trouver des projets avec le label « good first issue » ici, en filtrant par langage de programmation si nécessaire : https://github.com/topics/good-first-issue

Trouvez un sujet qui vous intéresse, planifiez votre travail et contribuez !

Résumé

Dans ce cours, vous avez appris :

- Travail en équipe : éviter la friction, revues de code
- Méthodes de développement : Waterfall, Scrum, agilité
- Open source : Structure, licences, contributions