R4.02 – Qualité de développement
Notes de cours 3

Semestre 4
2022/2023

1 Le modèle RIPR

La **conception de tests** est l'activité qui consiste à créer une suite de tests avec les meilleures chances de révéler les défauts logiciels potentiels. On sait qu'il n'y pas de règle qui permette à coup sûr de trouver tous les défauts, néanmoins il existe des principes à suivre pour faciliter cette tâche. Le modèle RIPR est un de ces principes. Il est nommé d'après les quatre actions qu'un test doit effectuer pour révéler un défaut, à savoir :

- **Reach** : pour révéler un défaut logiciel, un test doit avant tout atteindre et exécuter ce défaut.
- **Infect** : suivant l'état d'exécution, exécuter un défaut logiciel n'est pas toujours suffisant pour provoquer une erreur (un état d'exécution incorrect). Pour révéler le défaut, un test devra non seulement exécuter ce défaut mais aussi le faire dans des conditions qui mènent à une erreur.
- **Propagate** : une fois l'erreur survenue, celle-ci doit se propager jusqu'à l'état final et se manifester par une défaillance (un résultat incorrect du programme).
- **Reveal** : enfin pour qu'un test révèle la défaillance, il faut contrôler une propriété de l'état final qui mette en lumière la défaillance.

Les trois premières actions correspondent respectivement aux différents stades d'un problème logiciel : défaut, erreur, et défaillance. La dernière action est une incarnation de la spécification, elle fait la différence ce qui relève du comportement attendu et ce qui constitue une défaillance.

1.1 Les éléments d'un cas de test

Lors du premier cours, nous avons défini un cas de test comme un scénario comprenant des entrées (une entrée peut être une valeur passée en argument d'une méthode, mais aussi une action effectuée sur le système) et une propriété à vérifier. En suivant le modèle RIPR, on peut préciser cette définition et décomposer un cas de test de la manière suivante :

— **valeurs préfixes** : les entrées nécessaires pour amener le système dans un état propice au cas de test
— **valeurs du cas de test** : les entrées nécessaires pour exécuter la fonctionnalité à tester
— **valeurs postfixes** : les entrées nécessaires pour amener le système dans un état propice à être analysé (valeurs postfixes de vérification), ou bien pour remettre le système dans un état stable (valeurs postfixes de sortie)
— **propriété à vérifier** : la condition qui détermine si le test passe ou échoue.

Pour illustrer ces éléments, imaginons que l'on souhaite tester une application de scolarité. Parmi les différents cas de test, concentrons-nous sur celui qui vérifie qu'un étudiant ayant validé tous les modules valide son année, et que cette validation est inscrite dans la base de données. Pour tester un tel cas, il faudra effectuer des actions préalables : peupler une base de données avec diverses informations (étudiant, modules, notes...) puis ouvrir une connexion avec cette base de données. Ce sont les valeurs préfixes du cas de test, des actions à effectuer avant de pouvoir exécuter le test proprement dit. Dans un deuxième temps, on appellera la fonction de validation avec pour paramètre l'identifiant de l'étudiant en question. Ce sont les valeurs du cas de test. Enfin il faudra lancer une requête pour obtenir le contenu de la base de données après le test, puis fermer la connexion à la base de données. Ce sont les valeurs postfixes (de vérification et de sortie, respectivement). À un moment donné, le cas de test devra aussi déterminer si le contenu de la base de données indique bien que l'étudiant a validé son année : c'est la propriété à vérifier.

Ces différents éléments correspondent à ceux du modèle RIPR. Les valeurs préfixes ont pour but d'atteindre (*reach*) un état où le défaut potentiel est accessible. Les valeurs de cas de test ont pour but d'exécuter le code et déclencher une erreur (*infect*). Les valeurs postfixes de vérification ont pour but d'amener le système dans un état final où la défaillance est visible (*propagate*). La propriété à vérifier indique (*reveal*) que le comportement observé est une défaillance.

Pour certains tests, les valeurs préfixes ou postfixes peuvent être vides. Cela signifie simplement qu'il n'y a rien de particulier à faire avant ou après le test. C'est assez fréquent pour les tests unitaires, où les défauts sont facilement accessibles et les défaillances facilement visibles. Au contraire, les tests qui couvrent une partie plus large d'un système (tests d'intégration, tests système...) peuvent demander plus de travail pour atteindre les défauts ou propager les erreurs.

Il est possible que plusieurs cas de test dans une suite partagent les même valeurs préfixes ou postfixes (par exemple l'ouverture et fermeture d'une connexion à une base de donnée). Les outils de test proposent en général de séparer le code correspondant à ces valeurs pour pouvoir les organiser. Par exemple, JUnit 5 utilise les étiquettes `@BeforeEach` et `@BeforeAll` pour déclarer des fonctions exécutées avant les tests, et `@AfterEach` et `@AfterAll` pour des fonctions exécutées après.

2 La testabilité

La testabilité d'un logiciel est la facilité avec laquelle on peut le tester. Pour un logiciel avec une bonne testabilité, les tests seront simples à mettre en place et auront une forte probabilité de découvrir les défauts potentiels. Au contraire, si la testabilité est mauvaise, il sera difficile de mettre en place des tests efficaces, et par conséquent certains défauts pourront échapper à la détection malgré un grand nombre de tests. La testabilité requiert deux qualités :

- la **contrôlabilité** est la facilité avec laquelle on peut agir sur le comportement du système, en lui fournissant des entrées (valeurs ou actions).
- l'**observabilité** est la facilité avec laquelle on peut observer le comportement du système : les données qu'il calcule, les effets qu'il a sur son environnement ou sur d'autres systèmes logiciels et matériels, etc.

Des défis de contrôlabilité et d'observalité peuvent survenir lorsque les systèmes interagissent avec l'extérieur, en particulier avec des composants non-logiciels. Par exemple, si le système réagit à des entrées fournies par des capteurs physiques, il faudra généralement simuler ces données pour pouvoir lancer des tests de manière fiable et sans coûts excessifs. Inversement, si le système produit des effets externes (envoyer des mails, activer un actionneur...), il faudra utiliser un environnement de test capable d'observer les sorties du système à tester sans effectuer réellement ces actions. Pour garantir la contrôlabilité et l'observabilité, il est donc important que l'architecture favorise le découplage entre composants du système (par exemple en utilisant l'inversion de contrôle). Plus généralement, une bonne conception aide à garantir la testabilité.

Inversement, la testabilité peut être un indicateur de la qualité de la conception du système. Par exemple, une conception trop monolithique, avec des composants de trop grande taille et qui prennent en charge trop de fonctionnalités, va donner lieu à des problèmes aussi bien pour contrôler que pour observer ces composants. Il faudra donc revoir la conception et modulariser le système, ce qui sera bénéfique pour les tests, mais aussi pour d'autres aspects du logiciel (maintenance, utilisation, etc.). Le développement piloté par les tests permet d'éviter ces écueils : en donnant au logiciel ses premiers utilisateurs (les testeurs) dès que possible, il permet de s'apercevoir très rapidement des défauts de conception.

Attention toutefois à ne pas faire passer la testabilité avant les bonnes pratiques de génie logiciel, et en particulier le masquage de l'information et l'encapsulation des données. Par exemple, un testeur trop zélé pourrait insister pour que chaque méthode soit couvert par un test unitaire, et donc rendre toutes les méthodes publiques. C'est évidemment une très mauvaise architecture, car elle expose à l'utilisateur tous les détails de l'implémentation. On se contentera donc de tester les méthodes publiques, et plus généralement on se basera sur l'interface publique d'un composant logiciel pour concevoir les tests de ce composant.

3 Automatisation de tests

3.1 Tests pilotés par les données

Pour tester un composant logiciel, et en particulier pour les tests unitaires d'une méthode, il est généralement nécessaire d'exécuter plusieurs cas de tests très similaires, mais avec différentes valeurs d'entrées. Si ils sont implémentés de manière naïve, ces tests mèneront à de la duplication de code. Or les bonnes pratiques de développement s'appliquent aussi aux tests : comme le reste du code, les tests doivent pouvoir être lus, modifiés, maintenus, etc.

Afin d'éviter la duplication de code, il est possible d'utiliser les tests pilotés par les données (*data-driven testing*). Au lieu d'écrire n méthodes de tests correspondant à n cas de tests similaires, on écrit une seule méthode de tests qui prend un ou plusieurs arguments, et on fournit une table de valeurs à n lignes. La méthode de test est ensuite exécutée une fois pour chaque ligne de la table.

Considérons par exemple la méthode `max` qui retourne le maximum de deux arguments. On détermine qu'il faut au moins trois cas de tests : un cas où le premier argument est supérieur au second, un cas où il est inférieur, et un cas où les deux arguments sont égaux. Au lien d'écrire trois méthodes de tests, il est possible d'écrire un seul test paramétrique, avec trois arguments x, y et z, qui vérifie

$$\texttt{max}(x, y) = z$$

et de l'exécuter avec la table d'arguments suivante, qui comprend trois tuples de valeurs (x, y, z) correspondant à trois cas de tests :

x	y	z
2	1	2
1	3	3
1	1	1

Dans cette table, nous avons fourni à la fois des valeurs d'entrées x et y mais aussi la valeur de sortie attendue z. Une autre façon de procéder est d'écrire une propriété *universellement quantifiée*, c'est-à-dire une propriété doive être vraie pour toute valeur d'entrée. Écrire une telle propriété revient typiquement à formaliser la spécification. Par exemple, pour la méthode `max`, on peut tester que la méthode retourne une des deux valeurs qui lui sont passées en argument, et que la valeur de retour est supérieure ou égale aux deux arguments :

$$(\texttt{max}(x, y) = x \vee \texttt{max}(x, y) = y) \wedge \texttt{max}(x, y) \geq x \wedge \texttt{max}(x, y) \geq y$$

Cette propriété doit être vérifiée pour toutes valeurs d'entrées x et y, il n'est donc pas nécessaire de fournir une valeur de sortie pour exécuter le test.

Dans JUnit 5, les tests pilotés par les données sont indiqués par l'étiquette `@ParameterizedTest`. Les données d'entrées peuvent être fournies par un tableau (si la méthode de test ne prend qu'un argument), ou par une méthode dédiée.

Dans la suite du cours, nous verrons une méthode pour choisir les valeurs d'entrées de manière à maximiser les chances de découvrir des défauts sans avoir à recourir à un trop grand nombre de cas de test.

3.2 Intégration continue

L'intégration continue (souvent abrégée CI, d'après l'anglais) est un ensemble de pratiques de développement logiciel visant à intégrer très régulièrement les modifications faîtes par les développeurs afin de maintenir une copie de travail commune du logiciel. Cette vision s'oppose au *feature branch workflow*, où le développement de nouvelles fonctionnalités se fait à chaque fois sur une nouvelle branche, qui ne sera intégrée au tronc (branche principale) du projet qu'une fois la fonctionnalité complète. Des développeurs travaillant en intégration continue doivent au contraire partager leurs contributions (*commit*) sans attendre, au moins une fois par jour mais souvent plus fréquemment. Le but de l'intégration continue est d'éviter la divergence entre branches et les problèmes qui surviennent lorsqu'il faut réconcilier ces branches.

L'intégration continue fait une part importante aux tests : à intervalle régulier (par exemple après chaque journée de travail), le code est compilé (*nightly builds*) et la suite de tests est exécutée afin de vérifier que des régressions n'ont pas été introduites par les modifications. Le fait d'exécuter les tests aussi régulièrement impose des contraintes fortes. D'une part, les tests doivent être aussi automatisés que possible, depuis la compilation du code jusqu'à la collecte et l'affichage des résultats, sans quoi il est impossible de les exécuter quotidiennement. D'autre part, les tests doivent pouvoir s'exécuter dans une fenêtre de temps raisonnable, afin que les résultats soient accessibles rapidement et que les problèmes détectés puisse être corrigés. Optimiser le temps de d'exécution demander de limiter le nombre de cas de tests (et donc de donner la priorité aux cas qui sont les plus susceptibles de détecter des défauts graves) mais aussi d'optimiser leur exécution. Ce dernier aspect est particulièrement important pour les tests de haut niveau (tests d'intégration, test systèmes), lesquels peuvent s'avérer longs à exécuter.

4 Ressources supplémentaires

— Une liste (en français) d'anti-patrons, des choses à ne pas faire dans les tests :
`https://bruno-orsier.developpez.com/tutoriels/java/antipatrons-tests-unitaires/`

— Un article de blog (en français) sur la mise en place des tests dans le processus d'intégration continue :
`https://latavernedutesteur.fr/2018/04/11/integration-continue-vers-le-continuous-testing/`