Projet Génie Logiciel

Gestion des risques et rendus

Aurélien VILMINOT
Damien CLAUZON
Guilherme KLEIN
Léon ROUSSEL
Pierre ARVY

13 janvier 2022

Résumé

Ce document est un extrait de la documentation de Validation. La gestion des risques et des rendus y est traitée.

Table des matières

1 Gestion des risques

1.1 Motivations

Lors du développement d'un projet logiciel, la gestion des risques est cruciale car ces derniers sont présents à chaque étape, de la conception à l'implémentation. Par exemple, l'outil de versionnage permet de prévenir des risques potentiels de perte de données. Son utilisation doit néanmoins être maîtrisée. La gestion des risques vise à prévoir ces potentielles erreurs, et, en conséquence, s'approprier les outils nécessaires pour les résoudre.

1.2 Date de rendu

Le client exige des dates de rendu. Il faut donc être sûr de pouvoir rendre le travail à temps. Pour cela, nous avons mis en place une liste des tâches qui nous permet de savoir quelles parties nous devons implémenter et tester avant la fin d'un sprint (et donc avant un rendu dans le cadre du projet).

Pour avoir davantage de sécurité, des alarmes sont déclenchées depuis Trello pour nous alerter qu'une tâche est à finir avant une certaine date. Par exemple, avant un suivi ou un rendu, une notification est envoyée à chaque membre de l'équipe pour l'informer de l'événement. L'équipe peut ainsi s'assurer que tous les documents sont rendus à temps.

Nous choisissons de diviser les fonctionnalités du compilateur. Cela est, en effet, un des objectifs de la méthode SCRUM. En développant les fonctionnalités les unes après les autres, par exemple après une rencontre avec le client, nous pouvons lui présenter de nouvelles fonctions à chaque fois. Le client peut ainsi constater les avancements, et mieux se projeter sur la réalisation du produit. Cela diminue aussi les risques concernant les attentes du client. Si une partie du projet ne convient pas, nous pouvons réagir en conséquence dans le sprint suivant.

Ce qui est décrit précédemment est un des principaux avantages de la méthode SCRUM. En travaillant avec une méthode plus classique, le risque est de passer beaucoup de temps à développer un ensemble de fonctionnalités en parallèle, et de se rendre compte à la fin que le rendu ne correspond pas aux attentes du client.

1.3 Erreurs triviales

Quotidiennement, tout programmeur est exposé à des erreurs triviales. Par exemple, un point-virgule manquant peut empêcher la compilation d'un programme entier. Ces petites fautes, qui ont de grandes conséquences, sont donc à bannir.

Pour cela, nous avons pris plusieurs décisions afin de ne pas faire face à ces erreurs. Tout d'abord, nous utilisons tous un environnement de développement intelligent (Intellij par Jetbrains). L'utilisation d'un IDE a certes des inconvénients, mais sur ce point, c'est un atout de taille. En effet, notre IDE nous préviens lors d'une erreur de syntaxe évidente. Il permet aussi de faciliter l'écriture du code. Pour un projet de cette envergure, l'IDE nous permet de développer avec plus de sécurité.

Nous avons également mis en place des revues de code. À l'aide de Git, nous pouvons assigner des revues de codes aux différents membres de l'équipe. Cela a pour effet de corriger les erreurs évidentes, et par la même occasion, les différents membres prennent connaissance du code des autres. Sur les quelques revues de code que nous avons fait, des erreurs ont été retrouvées, la mise en place de ce fonctionnement est donc un succès.

Afin de garantir un minimum de fonctionnalités, des tests de bases sont développés durant le projet pour assurer que la version courante valide des fonctionnalités de base. Ces tests portent sur la base de chacune des parties des différents sous-langages de Deca. Des scripts comme `common-test.sh` entrent dans ce principe. En cas de succès de ces derniers, nous nous assurons que les fonctionnalités minimales sont couvertes.

1.4 Git

Deux types de risques peuvent advenir sur Git. Le premier est le non fonctionnement de GitLab, la plateforme sur laquelle est stockée tout notre travail. Si GitLab cesse de fonctionner, nous pouvons toujours reprendre une version sauvegarder dans les dernières 24 heures. En effet, tous les jours, nous faisons une copie de l'entièreté du dépôt GitLab. Pour continuer le projet, nous possédons tous un compte GitHub. Il nous est donc possible de continuer le

projet sur cette plateforme.

Un autre problème est la gestion du Git. Toutes les personnes du groupe n'ayant pas un niveau équivalent, il se peut que certaines personne en viennent à endommager le dépôt Git. Pour palier à cela, un responsable Git est désigné. Il aide donc les autres membres du groupe pour effectuer les opérations de `commit`, `push`, `merge`, `rebase`, si ces derniers ne sont pas familiers avec l'environnement.

1.5 Tests

La gestion des risques existe également au sein-même de la gestion des tests. Comme le précise cette documentation liée à la validation, il y a plusieurs types de tests. Les tests dits *boîte-noire* sont les plus sûr. Nous vérifions le contenu exact de la réponse. Les tests oracles sont moins sûrs. Nous vérifions uniquement si l'exécution s'est déroulée avec une erreur ou non. Il faut donc essayer de cibler le but de chaque test. Voulons-nous tester qu'un programme s'exécute correctement ? Voulons-nous vérifier qu'un programme donne une bonne réponse ? Voulons-nous vérifier l'erreur d'un programme invalide ?

Prenons l'exemple du test du parser. Nous réalisons des tests sur l'arbre produit. Sur ces tests, nous répliquons exactement les arbres attendus, et nous les comparons avec les résultats. Cette méthode de test est très précise mais elle demande beaucoup de temps pour produire les tests. C'est pourquoi les tests oracles permettent de créer une base de tests beaucoup plus grande, et plus exhaustive. Il faut donc juger du nombre de test à effectuer dans chaque catégorie.

1.6 Méthodes générales

Pour limiter les biais de compréhension et d'interprétation, les testeurs ne doivent pas être en contact avec le développement. Ainsi, les testeurs n'ont pas de supposition quant au code source, notamment sur ses faiblesses et ses forces.

Pour l'extension, notre groupe utilise une méthode de développement dirigé par les tests. Avec cette technique de travail, les tests sont créés avant même le début du développement. Ainsi, les développeurs doivent passer tous les tests créés pour affirmer qu'une fonctionnalité est implémentée. Cette méthode de travail permet d'augmenter la fiabilité du code produit.

Pour ce qui est des documentations, nous les rédigeons au fur et à mesure. Cela nous permet d'être tous au courant des avancements du projet. Les clients peuvent également comprendre à tout moment ce qui est en train d'être développer, et ils peuvent aussi comprendre les méthodes de travail de l'équipe.

2 Gestion des rendus

2.1 Motivations

En génie logiciel, la gestion des mises en production d'un rendu est une technique utilisée pour planifier, gérer et contrôler un déploiement ou une mise à jour logicielle. En effectuant les étapes adéquates, la gestion des rendus permet à l'équipe d'obtenir des informations importantes au bon moment afin de favoriser le succès du déploiement. Dans la vie d'un projet informatique, cette étape est cruciale car c'est à ce moment-là que le client va véritablement découvrir le contenu du produit attendu. La gestion des rendus est un processus de concrétisation du concept original du logiciel et consiste à le placer dans un environnement réel.

2.2 Liste des actions à effectuer

Afin de limiter les risques au moment de la livraison d'une version, nous proposons d'effectuer une liste d'actions avant chaque rendu. À chaque étape, il est nécessaire d'effectuer l'action demandée puis, selon le résultat, nous passons à la suivante ou nous revenons sur une étape précédente.

1. Développement et correction de bugs.

2. Réunion pour discuter de la version à rendre. Vérification de l'implémentation des fonctionnalités. Effectuer ou modifier la liste des points forts/faibles de la version.
3. Obtenir l'accord des testeurs pour passer à l'action suivante. Il est nécessaire d'avoir assez de tests pour effectuer le rendu. Un calcul de la couverture de tests est effectué avec l'outil Jacoco pour vérifier cela. *Tant que la couverture n'est pas assez haute, ou que les testeurs pensent que les tests ne sont pas exhaustifs, rester à l'étape 2). Si un test ne passe pas, revenir à l'étape 0).*
4. Sur une machine personnelle, après s'être assuré qu'il n'y a plus de fichier à `commit`, vérifier que l'exécution de la commande `mvn test` s'effectue correctement. Si des fichiers ne sont pas encore `commit`, vérifier leur contenu, comprendre leurs impacts, et `push` ces fichiers. *Si l'ensemble des tests passent, continuer à l'étape suivante, sinon, revenir à l'étape 0).*
5. Effectuer un `merge` de la branche courante sur la branche `master`. Les tests et le code source doivent se retrouver sur la branche `master` à l'issue de cette opération.
6. Sur une machine candide de l'école, s'assurer que le fichier `.bashrc` ajoute bien les chemins requis pour utiliser les différents outils (Maven, IMA...). Faire un `git clone` du projet à partie du dernier commit de la branche `master`. Puis, effectuer la commande `mvn verify`. S'assurer que tous les tests passent et que la couverture Jacoco reste la même que celle de l'étape 2). *Si certains tests échouent, retourner à l'étape 3), il se peut que certains fichiers n'aient pas été commit sur cette version.*
7. Supprimer le projet localement.
8. Livrer la version.