DS de MEDEV Janvier 2016

Qu'est-ce que la complexité cyclomatique ?

https://fr.wikipedia.org/wiki/Nombre_cyclomatique

La complexité cyclomatique mesure la complexité d'un programme informatique. Cette mesure reflète le nombre de décisions d'un algorithme en comptabilisant le nombre de « chemins » linéairement indépendants au travers d'un programme représenté sous la forme d'un graphe.

La complexité cyclomatique d'un programme structuré définie par :

M = E – N +2P
où :
M = complexité cyclomatique ;
E = le nombre d'arêtes du graphe ;
N = le nombre de nœuds du graphe ;
P = le nombre de composantes connexes du graphe.

Un code simple, au faible nombre cyclomatique, est théoriquement plus facile à lire, à tester et à entretenir :

- un effort de compréhension plus soutenu doit être effectué durant la lecture d'un code complexe, de façon à retenir et à prendre en compte chaque embranchement ;
- les tests unitaires doivent tester toutes les possibilités d'embranchement dans la fonction, de façon à suivre chacun des « chemins » ; or, les tests unitaires ont primordialement été conçus pour tester des fonctions en isolation par un cas simple ;
- de même, la correction d'un bug ou l'amélioration d'une fonction simple sera facilitée par l'absence d'obligation de prise en compte de ces embranchements et possibilités.

Que font les tests ?

- Prévenir les anomalies
- Détecter des anomalies
- Détecter toutes les anomalies

Git est-il un logiciel de versioning centralisé ou décentralisé ? Décentralisé

Quels sont parmi ces logiciels les logiciels de versioning ?

https://fr.wikipedia.org/wiki/Logiciel_de_gestion_de_versions

1ère génération (mono fichier, utilisation locale, exclusion mutuelle)

- 1972 : SCCS
- 1982 : RCS
- 1985 : PVCS

2ème génération (multi-fichiers, client-serveur, fusion après commit)

- 1986 : CVS
- 1992 : Rational ClearCase
- 1994 : Visual SourceSafe

3ème génération (2ème + atomicité)

- 1995 : Perforce
- 2000 : subversion

les prémisses au début des années 2000

- Bitkeeper
- GNU Arch

les outils actuels (2005)

- Bazaar
- Mercurial
- git

Que pensez-vous de cette phrase « les tests ne permettent pas de dire qu'il n' y a pas d'erreurs » ?

https://www.agixis.com/test-et-validation/

Il est important de faire la distinction entre une erreur, un défaut (ou bug) et une défaillance. Cette distinction permet de comprendre d'où vient le problème pour pouvoir le corriger.

Les tests préviennent et garantissent le niveau de qualité du produit sur une période déterminée. Ils donnent aussi des indications de fiabilité. Tout cela, permet de vérifier l'adéquation du produit aux besoins exprimés par le client et donc de satisfaire la demande du client.

Le second enjeu consiste à pallier les carences du génie logiciel. Le test relève d'un processus de développement non industrialisé et de contrôle non déterministe. Face à l'impossibilité de fournir des preuves, il permet la visibilité sur la qualité du produit et sur son bon ou mauvais fonctionnement.

La capacité des tests à trouver des défaillances, à prévenir les défauts et à élever le niveau de qualité du produit permettent donc de créer une relation de confiance entre le client et le fournisseur.

Principe 1 : les tests montrent la présence de défauts

Les tests ne peuvent pas prouver l'absence de défauts mais ils peuvent en prouver la présence ! Lorsque qu'il y a des erreurs, on parle d'objectifs atteints. En revanche, si aucune erreur de ce type n'est trouvée mais que rien n'informe sur les erreurs d'un autre type, on parle d'objectifs non atteints. Les tests sont faits pour démontrer qu'il y a des défauts.

Principe 7 : l'illusion d'absence d'erreurs

La notion d'erreur est typique des objectifs et non du savoir-faire. L'absence d'erreur peut être expliquée par plusieurs éléments : des tests non pertinents, des données de tests non

pertinentes, beaucoup d'erreurs de même types, etc. L'absence d'erreur ne signifie donc pas, réellement, qu'il n'y a aucune erreur.

Quels sont parmi ces tests les tests qui existent vraiment ?

http://www-igm.univ-mlv.fr/~dr/XPOSE2000/TesTs/SiteWeb/typestests.htm

Les tests unitaires

Les tests unitaires consistent à tester individuellement les composants de l'application. On pourra ainsi valider la qualité du code et les performances d'un module.

Les tests d'intégration

Ces tests sont exécutées pour valider l'intégration des différents modules entre eux et dans leur environnement exploitation définitif.
Ils permettront de mettre en évidence des problèmes d'interfaces entre différents programmes.

Les tests fonctionnels, d'acceptation

Ces tests ont pour but de vérifier la conformité de l'application développée avec le cahier des charges initial. Ils sont donc basés sur les spécifications fonctionnelles et techniques.

Les tests de non-régression

Les tests de non-régression permettent de vérifier que des modifications n'ont pas altérées le fonctionnent de l'application.
L'utilisation d'outils de tests, dans ce dommaine, permet de facilité la mise en place de ce type de tests.

Les tests IHM

Les tests IHM ont pour but de vérifier que la charte graphique a été respectée tout au long du développement.
Cela consiste à contrôler :

- la présentation visuelle : les menus, les paramètres d'affichages, les propriétés des fenêtres, les barres d'icônes, la résolution des écrans, les effets de bord,…
- la navigation : les moyens de navigations, les raccourcis, le résultat d'un déplacement dans un écran,…

Les tests de configuration

Une application doit pouvoir s'adapter au renouvellement de plus en plus fréquent des ordinateurs. Il s'avère donc indispensable d'étudier l'impact des environnements d'exploitation sur son fonctionnement.
Voici quelques sources de problèmes qui peuvent surgir lorsque l'on migre une application vers un environnement différent :

- l'application développée en 16 bits migre sur un environnement 32 bits,
- les DLL sont incompatibles,
- les formats de fichiers sont différents,
- les drivers de périphériques changent,
- les interfaces ne sont pas gérées de la même manière…

Ainsi, pour faire des tests efficaces dans ce contexte, il est nécessaire de fixer certains paramètres comme par exemple :

- la même résolution graphique,
- le même nombre de couleurs à l'écran,
- une imprimante identique,
- les mêmes paramètres pour le réseau...

Les tests de performance

Le but principal des tests de performance est de valider la capacité qu'ont les serveurs et les réseaux à supporter des charges d'accès importantes.

On doit notamment vérifier que les temps de réponse restent raisonnable lorsqu'un nombre important d'utilisateurs sont simultanément connectés à la base de données de l'application. Pour cela, il faut d'abord relever les temps de réponse en utilisation normale, puis les comparer aux résultats obtenus dans des conditions extrêmes d' utilisation.

Une solution est de simuler un nombre important d'utilisateur en exécutant l'application à partir d'un même poste et en analysant le trafic généré.

Le deuxième objectif de ces tests est de valider le comportement de l'application, toujours dans des conditions extrêmes. Ces tests doivent permettrent de définir un environnement matériel minimum pour que l'application fonctionnement correctement.

Les tests d'installation

Une fois l'application validée, il est nécessaire de contrôler les aspects liés à la documentation et à l'installation.

Les procédures d'installation doivent être testées intégralement car elles garantissent la fiabilité de l'application dans la phase de démarrage.

Bien sûr, il faudra aussi vérifier que les supports d'installation ne contiennent pas de virus.

Qui teste ? (les développeurs, les clients, aucun des deux)

Les développeurs et les clients

Quelle est la commande pour envoyer des fichiers en ligne sur git ? (git push, git pull, git fetch, git merge)

https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf
https://gist.github.com/aquelito/8596717

Différence entre push et pull :

- push : Cela envoie le fichier en ligne sur git
- pull : Cela met à jour le dépôt local

Git push

Qu’est-ce qu’une couverture de test ?

https://fr.wikipedia.org/wiki/Couverture_de_code

La couverture de test (parfois appelée couverture de code, code coverage en anglais) est la mesure du taux de code source testé dans un système informatique.
Un programme avec une haute couverture de code, mesurée en pourcentage, a davantage de code exécuté durant les tests ce qui laisse à penser qu'il a moins de chance de contenir de bugs logiciels non détectés, comparativement à un programme avec une faible couverture de code.

Il y a de nombreuses méthodes pour mesurer la couverture de code. Les principales sont :

- **Couverture des fonctions** (Function Coverage) - Chaque fonction dans le programme a-t-elle été appelée ?
- **Couverture des instructions** (Statement Coverage) - Chaque ligne du code a-t-elle été exécutée et vérifiée ?
- **Couverture des points de tests** (Condition Coverage) - Toutes les conditions (tel que le test d'une variable) sont-elles exécutées et vérifiées ? (Le point de test teste-t-il ce qu'il faut ?)
- **Couverture des chemins d'exécution** (Path Coverage) - Chaque parcours possible (par exemple les 2 cas vrai et faux d'un test) a-t-il été exécuté et vérifié ?

Quand je code qu’est-ce que je dois tester ? (mon code, celui du voisin...)

Mon code, et comment mon code intéragit avec celui des autres.