Testeur Certifié

Syllabus Niveau Avancé
Analyste Technique de Test

Version 2012

International Software Testing Qualifications Board

Groupe de travail Analyste Technique de Test avancé: Graham Bath (Responsable), Paul Jorgensen, Jamie Mitchell; 2010-2012.

Traduction Française :

- Réalisée par le Comité Français des Tests Logiciels
- Les remarques et corrections à apporter sur la traduction sont à transmettre par mail à traductions@cftl.fr

Historique des modifications

Version	Date	Remarques
ISEB v1.1	04SEP01	ISEB Practitioner Syllabus
ISTQB 1.2E	SEP03	ISTQB Advanced Level Syllabus from EOQ-SG
V2007	12OCT07	Certified Tester Advanced Level syllabus version 2007
V2007FR	31DEC08	Testeur Certifié Niveau Avancé, version Française
D100626	26JUN10	Incorporation des changements acceptés en 2009, division des chapitres pour les différents modules
D101227	27DEC10	Acceptation des changements de format et des corrections n'ayant pas d'impact sur la signification des phrases.
Draft V1	17SEP11	Première version du nouveau syllabus séparé TTA à partir du document spécifiant le périmètre de chaque syllabus. Revue du groupe de travail niveau avancé.
Draft V2	20NOV11	Version revue par les comités nationaux
Alpha 2012	09MAR12	Incorporation de tous les commentaires reçus des comités nationaux sur la livraison d'Octobre.
Beta 2012	07APR12	Version beta soumise à l'AG
Beta 2012	08JUN12	Livraison d'une version éditée pour les comités nationaux
Beta 2012	27JUN12	Incorporation des commentaires issus des groupes de travail Examen et Glossaire
RC 2012	15AUG12	Version Candidate pour livraison – incorporation des derniers commentaires des comités nationaux
RC 2012	02SEP12	Commentaires du BNLTB et de Stuart Reid incorporés Vérification par Paul Jorgensen
GA 2012	19OCT12	Edition finale et nettoyage pour livraison à l'AG
FR 2012	15JAN13	Traduction française par le CFTL

Table des matières

Remerciements

Ce document a été produit par une équipe issue du groupe de travail Niveau Avancé de l'International Software Testing Qualifications Board – Analyste Technique de Test Avancé: Graham Bath (Responsable), Paul Jorgensen, Jamie Mitchell.

L'équipe remercie l'équipe de revue ainsi que tous les Comités Nationaux pour leurs suggestions et entrées.

Au moment de la finalisation du Syllabus Niveau Avancé le groupe de travail Niveau Avancé était constitué des membres suivants (par ordre alphabétique):

Graham Bath, Rex Black, Maria Clara Choucair, Debra Friedenberg, Bernard Homès (Vice Chair), Paul Jorgensen, Judy McKay, Jamie Mitchell, Thomas Mueller, Klaus Olsen, Kenji Onishi, Meile Posthuma, Eric Riou du Cosquer, Jan Sabak, Hans Schaefer, Mike Smith (Responsable), Geoff Thompson, Erik van Veenendaal, Tsuyoshi Yumoto.

Les personnes suivantes ont participé aux relectures, revues et choix pour ce document:

Dani Almog, Graham Bath, Franz Dijkman, Erwin Engelsma, Mats Grindal, Dr. Suhaimi Ibrahim, Skule Johansen, Paul Jorgensen, Kari Kakkonen, Eli Margolin, Rik Marselis, Judy McKay, Jamie Mitchell, Reto Mueller, Thomas Müller, Ingvar Nordstrom, Raluca Popescu, Meile Posthuma, Michael Stahl, Chris van Bael, Erik van Veenendaal, Rahul Verma, Paul Weymouth, Hans Weiberg, Wenqiang Zheng, Shaomin Zhu.

Ce document a été officiellement livré par l'assemblée générale de l'ISTQB® le 19 octobre 2012.

0. Introduction à ce Syllabus

0.1 Objectifs de ce document

Ce syllabus forme la base de la Qualification Internationale en Test de Logiciels au Niveau Avancé pour l'Analyste Technique de Test. L'ISTQB® fournit ce syllabus comme suit :

1. Aux Comités Nationaux, pour traduction dans leur langue et pour l'accréditation des fournisseurs de formation. Les Comités Nationaux peuvent adapter le syllabus aux particularités de leur langage et modifier les références pour les adapter à leurs publications locales.
2. Aux Comités d'Examens, pour en dériver des questions d'examen adaptées aux objectifs d'apprentissage de chaque module dans la langue locale.
3. Aux fournisseurs de formation, pour concevoir les cours et déterminer les méthodes de formation appropriées.
4. Aux candidats à la certification, pour préparer l'examen (dans le cadre d'une formation ou indépendamment).
5. A la communauté internationale du logiciel et de l'ingénierie des systèmes, pour faire progresser la profession de testeur de logiciels et de systèmes, et comme base pour des livres et articles.

L'ISTQB® peut autoriser d'autres entités à utiliser ce syllabus pour d'autres objectifs à condition qu'elles demandent et obtiennent une autorisation écrite préalable.

0.2 Vue générale

Le Niveau Avancé est composé de trois syllabi séparés:

- Test Manager
- Analyste de Test
- Analyste Technique de Test

Le document Vue Générale du Niveau Avancé [ISTQB_AL_OVIEW] contient les informations suivantes:

- Gains Métier pour chaque syllabus
- Résumé pour chaque syllabus
- Relations entre les syllabi
- Description des niveaux cognitifs (Niveaux K)
- Annexes

0.3 Objectifs d'apprentissage examinables

Les Objectifs d'Apprentissage correspondent aux Gains Métier et sont utilisés pour créer l'examen de passage de la certification Analyste Technique de Test Avancé. Toutes les parties de ce syllabus sont examinables au niveau K1, c.à.d. le candidat devra reconnaître, se souvenir et mémoriser un terme ou un concept. Les Objectifs d'Apprentissage significatifs aux niveaux K2, K3 et K4 sont fournis au début de chaque chapitre.

0.4 Attentes

Certains des objectifs d'apprentissage pour l'Analyste Technique de Test supposent qu'une expérience de base est disponible dans les domaines suivants :

- Concepts généraux de programmation
- Concepts généraux des architectures logicielles

1. Les Tâches de l'Analyste Technique de Test dans le Test Basé sur les Risques - 30 mn.

Mots clé

risque produit, analyse de risque, évaluation des risques, identification des risques, niveau de risque, mitigation des risques, test basé sur les risques

Objectifs d'apprentissage pour Les Tâches de l'Analyste Technique de Test dans le Test Basé sur les Risques

1.3 Evaluation des Risques

TTA-1.3.1 (K2) Résumer les facteurs de risque génériques que l'Analyste Technique de Test a généralement besoin de considérer

Objectifs d'apprentissage communs

Les objectifs d'apprentissage suivants se rapportent à du contenu couvert par plus d'une section de ce chapitre.

TTA-1.x.1 (K2) Résumer les activités de l'Analyste Technique de Test dans une approche basée sur les risques pour la planification et l'exécution du test

1.1 Introduction

Le Test Manager a la responsabilité globale pour établir et gérer une stratégie de test basée sur les risques. Le Test Manager va généralement demander l'implication de l'Analyste Technique de Test pour assurer une mise en œuvre correcte de l'approche basée sur les risques.

Compte-tenu de leur expertise technique spécifique, les Analystes Techniques de Test sont activement impliqués dans les tâches suivantes du test basé sur les risques:

- Identification des risques
- Evaluation des risques
- Mitigation des risques

Ces tâches sont réalisées de façon itérative tout au long du projet pour traiter les risques produit émergeants et changer les priorités, et pour régulièrement évaluer et communiquer le statut des risques.

Les Analystes Techniques de Test travaillent dans le cadre du test basé sur les risques établi pour le projet par le Test Manager. Ils apportent leur connaissance des risques techniques inhérents au projet, tels que les risques liés à la sécurité, à la fiabilité du système et à la performance.

1.2 Identification des Risques

C'est en faisant appel à l'échantillon de parties prenantes le plus large possible que le processus d'identification des risques aura le plus de chances de détecter le plus grand nombre de risques significatifs.
Comme les Analystes Techniques de Test possèdent des compétences techniques uniques, ils sont particulièrement adaptés à la conduite d'interviews d'experts, de brainstorming avec des collègues et également à l'analyse de l'expérience passée et actuelle pour déterminer où se trouvent les domaines de risques les plus probables. Les Analystes Techniques de Tests travaillent de façon rapprochée de leurs pairs techniques (par ex., développeurs, architectes, ingénieurs opérationnels) pour déterminer les domaines de risque technique.

On peut citer parmi les exemples de risques pouvant être identifiés

- Des risques de performance (par ex., incapacité à satisfaire des temps de réponse sous des conditions de forte charge)
- Des risques de fiabilité (par ex., application incapable de satisfaire la disponibilité spécifiée dans le « Service Level Agreement » (Accord sur le niveau de service)

Des domaines de risques associés à des caractéristiques de qualité logicielle spécifiques sont couverts dans les chapitres correspondant de ce syllabus.

1.3 Evaluation des risques

Alors que l'identification des risques consiste à identifier autant de risques pertinents que possible, l'évaluation des risques est l'étude de ces risques identifiés afin de catégoriser chacun et de déterminer la probabilité et l'impact associées à chaque risque.

Déterminer le niveau de risque implique en général d'évaluer pour chaque risque, la probabilité d'occurrence et l'impact par rapport à une occurrence. La probabilité d'occurrence est en général interprétée comme la probabilité qu'a le problème potentiel d'exister dans le système sous test.

L'Analyste Technique de Test contribue trouver et comprendre le risque technique potentiel pour chaque élément de risque alors que l'Analyste de Test contribue à comprendre l'impact métier potentiel du problème s'il survenait.

Parmi les facteurs génériques devant être considérés, on peut citer:

- La complexité de la technologie
- La complexité de la structure du code
- Le conflit entre les parties prenantes au sujet des exigences techniques
- Des problèmes de communication résultant de la répartition géographique de l'organisation du développement
- Les outils et la technologie
- La pression relative au temps, aux ressources et à la gestion
- Le manque d'assurance qualité au début
- Un taux important de changement des exigences techniques
- Un grand nombre de défauts trouvés relatifs à des caractéristiques techniques de la qualité
- Des problèmes d'interface technique et d'intégration

Compte tenu de l'information relative au risque disponible, l'Analyste Technique de Test définit les niveaux de risque selon les recommandations établies par le Test Manager. Par exemple, le Test Manager peut déterminer que les risques devraient être catégorisés avec une valeur allant de 1 à 10, avec A étant le risque le plus élevé.

1.4 Mitigation des Risques

Durant le projet, les Analystes Techniques de Test influencent la façon avec laquelle le test répond aux risques identifiés. Cela implique généralement ce qui suit:

- Réduire le risque en exécutant les tests les plus importants et en mettant en œuvre les activités de mitigation et de contingence appropriées, comme défini dans la stratégie de test et le plan de test
- Evaluer les risques sur la base d'informations additionnelles collectées lorsque le projet se développe, et utiliser ces informations pour mettre en œuvre des actions de mitigation destinées à réduire la probabilité ou l'impact des risques précédemment identifiés et analysés

2. Tests Basés sur la Structure - 225 mn.

Mots clé

condition atomique, test des conditions , test de flot de contrôle, test des décisions et conditions, test des conditions multiples, test des chemins, court-circuiter (short-circuiting), test des instructions, technique basée sur la structure

Objectifs d'apprentissage pour Tests Basés sur la Structure

2.2 Test des Conditions

TTA-2.2.1 (K2) Comprendre comment réaliser une couverture des conditions et pourquoi cela peut être un test moins rigoureux que la couverture des décisions

2.3 Test des Décisions et Conditions

TTA-2.3.1 (K3) Ecrire des cas de test en appliquant les techniques de conception de test des décisions et conditions pour atteindre un niveau de couverture défini

2.4 Test de la couverture des Conditions/Décisions Modifiées

TTA-2.4.1 (K3) Ecrire des cas de test en appliquant la technique de conception de test de la couverture des conditions/décisions modifiées pour atteindre un niveau de couverture défini

2.5 Test des Conditions Multiples

TTA-2.5.1 (K3) Ecrire des cas de test en appliquant la technique de conception de test des conditions multiples pour atteindre un niveau de couverture défini

2.6 Test des Chemins

TTA-2.6.1 (K3) Ecrire des cas de test en appliquant la technique de conception de test des chemins

2.7 Test des API

TTA-2.7.1 (K2) Comprendre l'applicabilité du test des APIs et les types de défauts qu'il trouve

2.8 Sélectionner une Technique Basée sur la Structure

TTA-2.8.1 (K4) Sélectionner une technique basée sur la structure appropriée pour une situation de projet donnée

2.1 Introduction

Ce chapitre décrit principalement les techniques de conception de test basées sur la structure, qui sont aussi connues sous le nom de techniques de test boîte blanche ou basées sur le code. Ces techniques utilisent le code, les données et l'architecture et/ou les flux système comme base pour la conception des tests.
Chaque technique spécifique permet d'obtenir des cas de test de façon systématique et se concentre sur un aspect particulier de la structure à considérer. Les techniques fournissent des critères de couverture qui doivent être mesurés et associés à un objectif défini par chaque projet ou organisation. Atteindre une couverture totale ne signifie pas que l'ensemble des tests est complet, mais plutôt que la technique utilisée ne suggère plus de tests utiles pour la structure considérée.

A l'exception de la couverture des conditions, les techniques de conception des tests considérées dans ce syllabus sont plus rigoureuses que les techniques de couverture des instructions et des décisions couvertes dans le Syllabus Fondation [ISTQB_FL_SYL].

Dans ce syllabus, les techniques suivantes sont considérées:

- Test des conditions
- Test des décisions et conditions
- Test de la couverture des conditions/décisions modifiées
- Test des conditions multiples
- Test des chemins
- Test des API

Les quatre premières techniques listées ci-dessus sont basées sur des prédicats de décisions et trouvent largement le même type de défauts. Peu importe la complexité du prédicat de décision, il donnera VRAI ou FAUX, avec un chemin parcouru dans le code et l'autre non. Un défaut est détecté quand le chemin prévu n'est pas pris à cause d'un prédicat de décision complexe qui n'est pas valorisé comme prévu.

En général les quatre premières techniques sont successivement plus rigoureuses ; elles nécessitent la définition de plus de tests pour atteindre la couverture ciblée et pour trouver des instances plus subtiles de ce type de défaut.

Se référer à [Bath08], [Beizer90], [Beizer95], [Copeland03] et [Koomen06].

2.2 Test des Conditions

Comparé au test des décisions (branches), qui considère la décision comme un tout et évalue les sorties VRAI et FAUX dans des cas de test séparés ; le test des conditions considère « comment » une décision est prise. Chaque prédicat de décision est composé d'une ou plusieurs condition(s) "atomique(s)", chacune évaluée en une valeur booléenne discrète. Elles sont combinées pour déterminer la sortie finale de la décision. Chaque condition atomique doit être évaluée des deux façons, VRAI et FAUX, par les cas de test pour atteindre ce niveau de couverture.

Applicabilité
Le test des conditions n'est probablement intéressant qu'en théorie à cause des difficultés notées ci-dessous. Le comprendre, cependant, est nécessaire pour atteindre de meilleurs niveaux de couverture qui s'appuient dessus.

Limitations/Difficultés

Quand il y a deux ou plus conditions atomiques dans une décision, un choix imprudent dans la sélection des données de test peut aboutir à l'atteinte de la couverture des conditions tout en n'atteignant pas la couverture des décisions. Par exemple, considérons le prédicat de décision, "A et B".

	A	B	A et B
Test 1	FAUX	VRAI	FAUX
Test 2	VRAI	FAUX	FAUX

Pour atteindre 100% de couverture des conditions, il suffit d'exécuter les deux tests présents dans le tableau ci-dessus. Bien que ces deux tests atteignent 100% de couverture des conditions, ils ne permettent pas d'obtenir la couverture des décisions, puisque dans les deux cas le prédicat est valorisé à FAUX.

Quand une décision consiste en une unique condition atomique, le test des conditions est identique au test des décisions .

2.3 Test des Décisions et Conditions

Le test des décisions et conditions spécifie que le test doit atteindre la couverture des conditions (voir ci-dessus), et nécessite également que la couverture des décisions (voir Syllabus Fondation [ISTQB_FL_SYL]) soit aussi satisfaite. Un choix réfléchi des valeurs des données de test pour les conditions atomiques peut résulter en l'atteinte de ce niveau de couverture sans ajouter de cas de test supplémentaires en plus de ce qui est nécessaire pour atteindre la couverture des conditions.

L'exemple ci-dessous teste le même prédicat de décision que ci-dessus, "A et B". La couverture des décisions et conditions peut être atteinte avec le même nombre de tests en sélectionnant différentes valeurs de test.

	A	B	A et B
Test 1	VRAI	VRAI	VRAI
Test 2	FAUX	FAUX	FAUX

Cette technique peut donc avoir un avantage en efficacité.

Applicabilité
Ce niveau de couverture devrait être considéré quand le code testé est important mais non critique.

Limitations/Difficultés
Comme cela peut nécessiter davantage de cas de test qu'au niveau des décisions, cela peut être problématique quand le temps est un problème.

2.4 Test de la couverture des Conditions/Décisions Modifiées

Cette technique apporte un plus haut niveau de contrôle de la couverture des flux. En supposant que l'on a N conditions atomiques uniques, la couverture des conditions/décisions modifiées peut généralement être atteinte avec N+1 cas de test uniques. La couverture des conditions/décisions

modifiées atteint la couverture des décisions et conditions, mais demande ensuite à ce que ce qui suit soit satisfait :

1. Au moins un test pour lequel le résultat de la décision changerait si la condition atomique X était à VRAI
2. Au moins un test pour lequel le résultat de la décision changerait si la condition atomique X était à FAUX
3. Chacune des différentes conditions atomiques a des tests qui satisfont les exigences 1 et 2

	A	B	C	(A ou B) et C
Test 1	VRAI	FAUX	VRAI	VRAI
Test 2	FAUX	VRAI	VRAI	VRAI
Test 3	FAUX	FAUX	VRAI	FAUX
Test 4	VRAI	FAUX	FAUX	FAUX

Dans l'exemple ci-dessus, la couverture des décisions est atteinte (le résultat du prédicat de décision est VRAI et FAUX), et la couverture des conditions est atteinte (A, B, et C prennent tous les deux valeurs VRAI et FAUX).

Dans Test 1, A est VRAI et la sortie est VRAI. Si A est changé à FAUX (comme dans Test 3, sans changer les autres valeurs) le résultat devient FAUX
Dans Test 2, B est VRAI et la sortie est VRAI. Si B est changé à FAUX (comme dans Test 3, sans changer les autres valeurs), le résultat devient FAUX
Dans Test 1, C est VRAI et la sortie est VRAI. Si C est changé à FAUX (comme dans Test 4, sans changer les autres valeurs) la sortie devient FAUX

Applicabilité
Cette technique est largement utilisée dans l'industrie logicielle aérospatiale et beaucoup d'autres systèmes à sécurité critique. Elle devrait être utilisée lorsqu'il s'agit de logiciel à sécurité critique où toute défaillance peut causer une catastrophe.

Limitations/Difficultés
Atteindre la couverture des Conditions/Décisions modifiées peut être compliqué quand il y a de nombreuses occurrences d'un terme spécifique dans une expression ; quand cela se produit, le terme est dit « couplé ». Selon l'instruction de décision dans le code, il peut être impossible de faire varier la valeur du terme couplé pour qu'il fasse à lui seul changer le résultat de la décision. Une approche pour traiter ce problème est de préciser que seules les conditions atomiques non couplées doivent être testées pour le niveau de couverture des Conditions/Décisions. L'autre approche est d'analyser au cas par cas chaque décision dans laquelle on trouve des couples.

Certains langages de programmation et/ou des interpréteurs sont conçus de façon à mettre en évidence des comportements de court-circuit lors de l'évaluation d'instructions de décision complexe dans le code. En fait, le code exécuté n'évalue pas forcément une expression entière si le résultat final de l'évaluation peut être déterminé après l'évaluation d'une partie seulement de l'expression. Par exemple, lors de l'évaluation de la décision "A et B", il n'y a pas de raison d'évaluer B si A est FAUX. Aucune valeur de B ne peut changer la valeur finale et donc, le code peut économiser du temps d'exécution en n'évaluant pas B. Le court-circuit peut affecter la capacité à atteindre la couverture des Conditions/Décisions modifiées, car certains des tests requis peuvent ne pas être réalisables.

2.5 Test des Conditions Multiples

Dans de rares cas, il peut être nécessaire de tester toutes les combinaisons de valeurs possibles qu'une décision peut contenir. Ce niveau de test exhaustif est appelé couverture des conditions multiples. Le nombre de tests requis dépend du nombre de conditions atomiques dans l'instruction de décision et peut être déterminé en calculant 2^n avec n le nombre de conditions atomiques non couplées. En utilisant le même exemple que précédemment, les tests suivants sont nécessaires pour atteindre la couverture des conditions multiples :

	A	B	C	(A ou B) et C
Test 1	VRAI	VRAI	VRAI	VRAI
Test 2	VRAI	VRAI	FAUX	FAUX
Test 3	VRAI	FAUX	VRAI	VRAI
Test 4	VRAI	FAUX	FAUX	FAUX
Test 5	FAUX	VRAI	VRAI	VRAI
Test 6	FAUX	VRAI	FALSE	FAUX
Test 7	FAUX	FAUX	VRAI	FAUX
Test 8	FAUX	FAUX	FAUX	FAUX

Si le langage utilise des court-circuits, le nombre effectif de cas de test sera souvent réduit, selon l'ordre et le regroupement des opérations logiques qui sont réalisées sur les conditions atomiques.

Applicabilité
Traditionnellement, cette technique était utilisée pour tester du logiciel embarqué qui devait fonctionner de façon fiable sans planter pendant de longues périodes de temps (par ex., des commutateurs téléphoniques qui devaient durer 30 ans). Ce type de test est en passe d'être remplacé par le test des conditions/décisions modifiées dans les applications les plus critiques.

Limitations/Difficultés
Comme le plus grand nombre des cas de test peuvent être dérivés directement d'une table de vérité contenant toutes les conditions atomiques, ce niveau de couverture peut facilement être déterminé. Cependant, le nombre absolu de cas de test requis, rend la couverture des conditions/décisions plus applicable à la plupart des situations.

2.6 Test des Chemins

Le test des chemins consiste à identifier les chemins au travers du code et à créer ensuite les tests les couvrant. Conceptuellement, il serait utile de tester chaque chemin unique dans le système. Cependant, dans tout système non-trivial le nombre de cas de test pourrait devenir excessivement important à cause de la nature des structures à boucles.

En laissant de côté le problème des boucles indéfinies, il est réaliste de réaliser du test des chemins. Pour appliquer cette technique, [Beizer90] recommande que les tests soient créés de façon à suivre plusieurs chemins au travers du module, de l'entrée à la sortie. Pour simplifier ce qui pourrait être une

tâche complexe, il recommande que cela soit fait systématiquement, en utilisant la procédure suivante:

1. Choisissez comme premier chemin le chemin le plus simple pour une fonctionnalité sensible, de l'entrée à la sortie.
2. Choisissez chaque chemin supplémentaire comme une petite variation du chemin précédent. Essayez de changer seulement une branche dans le chemin qui est différent pour chaque test successif. Lorsque cela est possible, privilégiez les chemins courts plutôt que les longs. Privilégiez les chemins qui donnent un sens fonctionnel à ceux qui n'en donnent pas
3. Ne choisissez de chemins ne donnant pas de sens fonctionnel que lorsque cela est requis pour la couverture. Beizer note dans cette règle que de tels chemins peuvent être hors sujet et devraient être questionnés.
4. Utilisez l'intuition lors du choix des chemins (c.à.d., quels chemins ont le plus de chances d'être exécutés)

Noter que certains segments de chemins seront probablement exécutés plus d'une fois en utilisant cette stratégie. Le point clé de cette stratégie est de tester toutes les branches possibles à travers le code au moins une fois et éventuellement plusieurs.

Applicabilité
Le test des chemins partiels—tel que défini ci-dessus—est souvent réalisé dans le logiciel à sécurité critique. Il constitue un bon complément à d'autres méthodes couvertes dans ce chapitre car il observe les chemins dans le logiciel plutôt que seulement la façon avec laquelle les décisions sont prises.

Limitations/Difficultés
Bien qu'il soit possible d'utiliser un graphe de contrôle de flux pour déterminer les chemins, dans la réalité un outil est nécessaire pour les calculer pour des modules complexes.

Couverture
Créer suffisamment de tests pour couvrir tous les chemins (à l'exception des boucles) garantit l'atteinte de la couverture des instructions et des branches. Le test des chemins apporte un test plus approfondi que la couverture des branches avec une augmentation relativement faible du nombre de tests. [NIST 96]

2.7 Test des API

Une interface de programmation de l'application (API: Application Programming Interface) est du code permettant la communication entre différents processus, programmes et/ou systèmes. Les APIs sont souvent utilisées dans les relations client/serveur où un processus apporte une certaine fonctionnalité à d'autres processus.

A certains égards, le test d'API est très similaire au test d'interface graphique (GUI: graphical user interface). Il porte sur l'évaluation de valeurs d'entrée et de données retournées.

Le test négatif est souvent crucial lorsque l'on a affaire à des APIs. Les programmeurs qui utilisent les APIs pour accéder à des services externes à leur propre code peuvent essayer d'utiliser les interfaces API de façons pour lesquelles elles n'ont pas été prévues. Cela signifie qu'une gestion des erreurs solide est essentielle pour éviter des opérations incorrectes. Du test combinatoire de différentes interfaces peut être nécessaire parce que les APIs sont souvent utilisées en conjonction avec d'autres APIs et parce que une simple interface peut contenir plusieurs paramètres dont les valeurs peuvent être combinées de nombreuses façons.

Les APIs sont fréquemment pauvrement couplées, ce qui résulte en la possibilité bien réelle de transactions perdues ou de problèmes de temps. Cela nécessite le test minutieux des mécanismes de récupération ou répétition. Une organisation qui fournit une interface API doit assurer que tous les services ont une très haute disponibilité, ce qui requiert souvent le test strict de la fiabilité par l'éditeur de l'API de même que par le support pour l'infrastructure.

Applicabilité
Le test d'API devient de plus en plus important à cause du nombre croissant de systèmes distribués et de l'utilisation de processus distants. On peut citer comme exemple des appels au système d'exploitation, des architectures orientées services (SOA), des appels de procédures distantes (RPC: remote procedure calls), des web services, et pratiquement toute autre application distribuée.

Limitations/Difficultés
Tester directement une API demande généralement à l'Analyste Technique de Test d'utiliser des outils spécialisés. Comme il n'y a pas d'interface graphique directement associée à une API, des outils peuvent être requis pour mettre en place l'environnement initial, rassembler les données, invoquer l'API, et déterminer les résultats.

Couverture
Le test des APIs est une description d'un type de test; il ne dénote pas un niveau de couverture particulier. Au minimum, le test des APIs devrait inclure l'exercice de tous les appels à l'API de même que toutes les valeurs valides ou invalides.

Types de Défauts
Les types de défauts pouvant être trouvés par le test des APIs sont très disparates. Des problèmes d'interface sont communs, comme des problèmes de traitement de données, des problèmes de temps, des pertes ou duplications de transactions.

2.8 Sélectionner une Technique Basée sur la Structure

Le contexte du système sous test va déterminer le niveau de couverture du test basé sur la structure. Plus le système est critique, plus le niveau de couverture requis est haut. En général, plus le niveau de couverture requis est haut, plus on aura besoin de temps et de ressources pour atteindre ce niveau.

Parfois le niveau de couverture requis peut être dérivé de standards applicables qui s'appliquent au système logiciel. Par exemple, si le système devait être utilisé dans un environnement aéronautique, il pourrait être nécessaire de se conformer au standard DO-178B (en Europe, ED-12B). Ce standard, contient les cinq conditions de défaillance suivantes :

A. Catastrophique: la défaillance peut causer la perte d'une fonction critique requise pour voler en sécurité ou poser l'avion
B. Dangereux: la défaillance peut avoir un impact très négatif sur la sécurité ou la performance
C. Majeur: la défaillance est significative, mais moins sérieuse que A ou B
D. Mineur: la défaillance est visible, mais avec moins d'impact que C
E. Pas d'effet: la défaillance n'a pas d'impact sur la sécurité

Si le système logiciel est catégorisé au niveau A, il doit être testé pour la couverture des conditions et décisions multiples. Au niveau B, il doit être testé pour le niveau de couverture des décisions bien que la couverture des conditions et décisions multiples soit optionnelle. Le niveau C requiert au minimum la couverture des instructions.

De même, IEC-61508 est un standard international pour la sureté fonctionnelle des systèmes programmables, électroniques, reliés à la sureté. Ce standard a été adapté dans beaucoup de

domaines différents, incluant l'automobile, le ferroviaire, les processus industriels, les centrales nucléaires, et les machines. La criticité est définie en utilisant une échelle graduée pour les niveaux d'intégrité de sureté (SIL : Safety Integrity Level) (1 étant le moins critique, 4 le plus critique) et la couverture est recommandée comme suit:

1. Couverture des instructions et des branches recommandée
2. Couverture des instructions hautement recommandée, couverture des branches recommandée
3. Couverture des instructions et des branches hautement recommandée
4. Couverture des conditions et décisions multiples hautement recommandée

Dans les systèmes modernes, il est rare que tous les traitements soient faits dans le même système. Le test des APIs devrait être institué à chaque fois que l'un des traitements est à faire à distance. La criticité du système devrait déterminer quelle quantité d'effort devrait être investie dans le test des APIs.

Comme toujours, le contexte du système logiciel sous test devrait guider l'Analyste Technique de Test sur les méthodes utilisées dans le test.

3. Techniques Analytiques - 255 mn.

Mots clé

analyse du flux de contrôle , complexité cyclomatique, analyse du flux de données, paires définition-usage, analyse dynamique, fuite mémoire, test d'intégration par paires, test d'intégration par voisinage, analyse statique, pointeur sauvage

Objectifs d'apprentissage pour Techniques Analytiques

3.2 Analyse Statique

TTA-3.2.1 (K3) Utiliser l'analyse du flux de contrôle pour détecter si le code a des anomalies de flux de contrôle

TTA-3.2.2 (K3) Utiliser l'analyse du flux de données pour détecter si le code a des anomalies de flux de données

TTA-3.2.3 (K3) Proposer des façons d'améliorer la maintenabilité du code en appliquant l'analyse statique

TTA-3.2.4 (K2) Expliquer l'utilisation des graphes d'appel pour établir des stratégies de test d'intégration

3.3 Analyse Dynamique

TTA-3.3.1 (K3) Spécifier les buts à atteindre par l'utilisation de l'analyse dynamique

3.1 Introduction

Il y a deux types d'analyse: analyse statique et analyse dynamique.

L'analyse statique (Section 3.2) comprend le test analytique qui peut se produire sans exécuter le logiciel. Comme le logiciel n'est pas exécuté, il est examiné soit par un outil, soit par une personne pour déterminer s'il agira correctement lors de son exécution. Cette vue statique du logiciel permet une analyse détaillée sans avoir à créer les données et pré conditions qui permettraient aux scénarios d'être exercés.

Noter que les différentes formes de revues qui sont pertinentes pour l'Analyste Technique de Test sont couvertes au chapitre 5.

L'analyse dynamique (Section 3.3) requiert l'exécution effective du code et est utilisée pour trouver des fautes de codage qui sont plus faciles à détecter quand le code est exécuté (par ex., fuite mémoire). L'analyse dynamique, comme avec l'analyse statique, peut s'appuyer sur des outils ou sur un individu contrôlant le système en exécution en surveillant des indicateurs comme une augmentation rapide de la mémoire.

3.2 Analyse Statique

L'objectif de l'analyse statique est de détecter des fautes réelles ou potentielles dans le code et l'architecture du système et d'améliorer leur maintenabilité. L'analyse statique est en général assistée par des outils.

3.2.1 Analyse du Flux de Contrôle

L'analyse du flux de contrôle est la technique statique où le flux de contrôle à travers un programme est analysé, soit par l'utilisation d'un graphe de flux de contrôle, soit par un outil. Beaucoup d'anomalies peuvent être trouvées dans un système en utilisant cette technique, comme des boucles qui sont mal conçues (par ex., en ayant plusieurs points d'entrée), des cibles ambiguës d'appels de fonctions dans certains langages (par ex., Scheme), un séquencement incorrect des opérations, etc.

Un des usages courants de l'analyse du flux de contrôle est de déterminer la complexité cyclomatique. La valeur de la complexité cyclomatique est un entier positif qui représente le nombre de chemins indépendants dans un graphe fortement connecté avec des boucles et des itérations ignorées dès qu'elles ont été traversées une fois. Chaque chemin indépendant, de l'entrée à la sortie, représente un chemin unique au travers du module. Chaque chemin unique doit être testé.

La valeur de la complexité cyclomatique est en général utilisée pour comprendre la complexité globale d'un module. La théorie de Thomas McCabe [McCabe 76] était que plus le système est complexe, plus il serait difficile de le maintenir et plus il contiendrait de défauts. Au fil des ans, beaucoup d'études ont remarqué cette corrélation entre la complexité et le nombre de défauts contenus. Le NIST (Institut National des Standards et de la Technologie) recommande une valeur de complexité maximum de 10. Tout module qui est mesuré avec une complexité supérieure peut avoir besoin d'être divisé en plusieurs modules.

3.2.2 Analyse du Flux de Données

L'analyse du flux de données couvre une variété de techniques qui collectent des informations sur l'utilisation des variables dans un système. Un examen minutieux est apporté au cycle de vie des

variables, (c.à.d, où elles sont déclarées, définies, lues, évaluées et détruites), car des anomalies peuvent se produire durant toutes ces opérations.

Une technique commune est appelée la notation définition-usage où le cycle de vie de chaque variable est divisé en trois actions atomiques différentes:

- d: quand la variable est déclarée, définie ou initialisée (d pour « declared »)
- u: quand la variable est utilisée ou lue soit dans un calcul, soit dans un prédicat de décision (u pour « used »)
- k: quand la variable est tuée, détruite ou sort du périmètre (k pour “killed”)

Ces trois actions atomiques sont combinées en paires (“paires définition-usage”) pour illustrer le flux de données. Par exemple, un « chemin du » représente un fragment de code où une donnée variable est définie et ensuite utilisée.

On peut citer comme anomalies possibles la réalisation de la bonne action sur une variable au mauvais moment ou l’exécution d’une action incorrecte sur la donnée dans une variable. Ces anomalies incluent:

- L’affectation d’une valeur invalide à une variable
- L’échec dans l’affectation d’une valeur à une variable avant de l’utiliser
- La prise d’un chemin incorrect à cause d’une valeur incorrecte dans un prédicat de contrôle
- Essayer d’utiliser une variable après qu’elle ait été détruite
- Référencer une variable quand elle est hors périmètre
- Déclarer et détruire une variable sans l’utiliser
- Redéfinir une variable avant qu’elle ait été utilisée
- Ne pas parvenir à tuer une variable allouée dynamiquement (pouvant causer une possible fuite mémoire)
- Modifier une variable, en provoquant des effets de bords inattendus (par ex., des effets d’ondulation en changeant une variable globale sans considérer toutes les utilisations de cette variable)

Le langage de développement utilisé peut orienter les règles utilisées dans l’analyse du flux de données. Les langages de programmation peuvent permettre au programmeur de réaliser certaines actions avec des variables qui ne sont pas illégales mais peuvent causer un comportement du système différent de celui attendu par le programmeur dans certaines circonstances. Par exemple, une variable peut être définie deux fois sans être réellement utilisée si un certain chemin est suivi. L’analyse du flux de données marquera souvent ces utilisations comme “suspicieux”. Même si cela peut être un usage légal de la capacité d’affectation de la variable, cela peut mener à de futurs problèmes de maintenabilité dans le code.

Le test des flux de données “utilise le graphe de contrôle de flux pour explorer les choses peu raisonnables pouvant arriver aux données" [Beizer90] et trouve par conséquent des défauts différents de ceux du test des flux de contrôle. Un Analyste Technique de Test devrait inclure cette technique lors de la planification du test car beaucoup de ces défauts peuvent causer des défaillances intermittentes qui sont difficiles à trouver lors de la réalisation de tests dynamiques.

Cependant, l’analyse du flux de données est une technique statique : elle peut passer à côté de certains problèmes qui se produisent sur les données lors de l’exécution du système. Par exemple, la variable de donnée statique peut contenir un pointeur vers un tableau créé dynamiquement qui n’existe même pas avant l’exécution. L’utilisation de multiprocesseurs et le multitâches préemptif peuvent créer des conditions qui ne seront par trouvées par l’analyse du flux de données ou du flux de contrôle.

3.2.3 Utiliser l'Analyse Statique pour Améliorer la Maintenabilité

L'analyse statique peut être appliquée de nombreuses façons pour améliorer la maintenabilité du code, de l'architecture et des sites web.

Du code pauvrement écrit, non documenté et non structuré tend à être plus difficile à maintenir. Il peut nécessiter plus d'effort de la part des développeurs pour localiser et analyser les défauts dans le code et la modification du code pour corriger un défaut ou ajouter une nouvelle fonctionnalité peut aboutir à l'introduction de défauts supplémentaires.

L'analyse statique est utilisée avec le support d'outils pour améliorer la maintenabilité du code en vérifiant le respect de standards et directives de codage. Ces standards et directives décrivent les pratiques de codage requises comme des conventions de nommage, les commentaires, l'indentation et la modularisation du code. Noter que les outils d'analyse statique émettent généralement des warnings plutôt que des erreurs même si le code peut être correct d'un point de vue syntaxique.

Des conceptions modulaires aboutissent en général à du code plus maintenable. Les outils d'analyse statique supportent le développement de code modulaire des façons suivantes:

- Ils recherchent le code répété. Ces sections de code peuvent être des candidats à la refactorisation en modules (même si le temps d'exécution supplémentaire imposé par les appels de modules peut être un problème pour les systèmes temps réel).
- Ils génèrent des métriques qui sont des indicateurs de valeur sur la modularisation du code. Ils incluent des mesures de couplage et de cohésion. Un système ayant une bonne maintenabilité aura probablement une mesure basse du couplage (le degré auquel les modules reposent les uns sur les autres durant l'exécution) et une mesure haute de la cohésion (le degré auquel un module est autonome et centré sur une tâche unique)
- Ils indiquent, dans du code orienté objet, où les objets dérivés peuvent avoir trop ou trop peu de visibilité sur les classes parent.
- Ils soulignent les domaines dans le code ou l'architecture ayant un haut niveau de complexité structurelle, ce qui est généralement considéré comme un indicateur de mauvaise maintenabilité et un potentiel élevé pour contenir des fautes. Des niveaux acceptables de complexité cyclomatique (voir Section 3.2.1.) peuvent être spécifiés dans des directives pour assurer que le code soit développé de façon modulaire avec en tête la maintenabilité et la prévention des défauts. Un code avec de hauts niveaux de complexité cyclomatique peut être candidat à la modularisation.

La maintenance d'un site web peut aussi être supportée par l'utilisation d'outils d'analyse statique. L'objectif ici est de vérifier si la structure arborescente du site est bien équilibrée ou s'il y a un déséquilibre qui mènera à:

- Davantage de tâches de test difficiles
- Une charge de travail de maintenance augmentée
- Une navigation difficile pour l'utilisateur

3.2.4 Graphes d'Appels

Les graphes d'appels sont une représentation statique de la complexité de communication. Ce sont des graphes dirigés dans lesquels des nœuds représentent des unités de programmes et des arcs représentent la communication entre ces unités.

Les graphes d'appels peuvent être utilisés dans le test unitaire où différentes fonctions ou méthodes s'appellent les unes les autres, ou dans le test d'intégration et le test système où des modules séparés s'appellent les uns les autres, ou dans le test d'intégration de systèmes où des systèmes séparés s'appellent les uns les autres.

Les graphes d'appels peuvent être utilisés pour les buts suivants:

- Concevoir des tests appelant un module ou système spécifique
- Etablir le nombre d'endroits dans le logiciel à partir desquels un module ou système est appelé
- Evaluer la structure du code et de l'architecture système
- Apporter des suggestions pour l'ordre d'intégration (intégration par paire ou par voisinage) Elles sont discutées plus en détail ci-dessous

Dans le syllabus Niveau Fondation [ISTQB_FL_SYL], deux catégories différentes de tests d'intégration ont été discutées: incrémentale (top-down, bottom-up, etc.) et non-incrémentale (big bang). Il a été dit que les méthodes incrémentales étaient préférées parce qu'elles introduisent le code par incréments rendant ainsi l'isolation des fautes plus facile puisque la quantité de code impliquée est limitée.

Dans ce syllabus avancé, trois méthodes non-incrémentales utilisant des graphes d'appels sont introduites. Elles peuvent être préférables aux méthodes incrémentales qui nécessiteront probablement des « builds » supplémentaires pour terminer le test et l'écriture de code non livrable pour supporter le test. Ces trois méthodes sont :

- Le test d'intégration par paires (à ne pas confondre avec la technique de test boîte noire "test par paires") cible des paires de composants qui fonctionnent ensemble, comme cela a été vu dans les graphes d'appels pour le test d'intégration. Bien que cette méthode ne réduise que légèrement le nombre de "builds", elle réduit la quantité de code de harnais de test nécessaire.
- Le test d'intégration par voisinage teste tous les nœuds connectés à un nœud donné comme base pour les tests d'intégration. Tous les nœuds prédécesseurs et successeurs d'un nœud spécifique dans le graphe d'appel sont la base pour le test.
- L'approche du prédicat de conception de McCabe utilise la théorie de la complexité cyclomatique telle qu'appliquée à un graphe d'appels pour modules. Cela nécessite la construction d'un graphe d'appel qui montre les différentes façons pour les modules de s'appeler les uns les autres, incluant :
  - L'appel inconditionnel: l'appel d'un module à un autre se produit toujours
  - L'appel conditionnel: l'appel d'un module à un autre se produit parfois
  - L'appel conditionnel mutuellement exclusif: un module appellera l'un (et uniquement l'un) des modules d'un nombre de modules différents
  - L'appel itératif: un module appelle un autre module au moins une fois mais peut l'appeler plusieurs fois
  - L'appel itératif conditionnel: un module peut en appeler un autre zéro ou plusieurs fois

  Après la création du graphe d'appels, la complexité de l'intégration est calculée, et des tests sont créés pour couvrir le graphe.

Se référer à [Jorgensen07] pour plus d'information sur l'utilisation des graphes d'appels et le test d'intégration par paires.

3.3 Analyse Dynamique

3.3.1 Vue générale

L'analyse dynamique est utilisée pour détecter des défaillances où les symptômes peuvent ne pas être immédiatement visibles.

Par exemple, la possibilité de fuites mémoires peut être détectable par l'analyse statique (trouver du code qui alloue mais ne libère jamais de la mémoire), mais une fuite mémoire est aisément apparente avec l'analyse dynamique.

Des défaillances qui ne sont pas immédiatement reproductibles peuvent avoir des conséquences significatives sur l'effort de test et sur la capacité à livrer ou à utiliser de façon productive le logiciel. De telles défaillances peuvent être causées par des fuites mémoires, par l'usage incorrect de pointeurs et par d'autres corruptions (par ex., sur la pile système) [Kaner02]. A cause de la nature de ces défaillances, qui peuvent inclure la dégradation progressive de la performance du système ou même des plantages du système, les stratégies de test doivent considérer les risques associés à de tels défauts et, là où cela est approprié, réaliser de l'analyse dynamique pour les réduire (typiquement en utilisant des outils). Comme ces défaillances sont souvent les défaillances les plus coûteuses à trouver et à corriger, il est recommandé de réaliser l'analyse dynamique tôt dans le projet.

L'analyse dynamique peut être appliquée pour accomplir ce qui suit:

- Prévenir l'apparition de défaillances en détectant des pointeurs sauvages et des pertes de mémoire système
- Analyser des défaillances du système qui ne peuvent pas être facilement reproduites
- Evaluer le comportement réseau
- Améliorer la performance du système en fournissant des informations sur le comportement du système à l'exécution

L'analyse dynamique peut être réalisée à tout niveau de test et requiert des compétences techniques et système pour faire ce qui suit:

- Spécifier les objectifs de test de l'analyse dynamique
- Déterminer le moment opportun pour commencer et arrêter l'analyse
- Analyser les résultats

Durant le test système, des outils d'analyse dynamique peuvent être utilisés même si les Analystes Techniques de Test ont les compétences techniques minimales, les outils utilisés créent généralement des logs compréhensifs pouvant être analysés par ceux qui ont les compétences techniques nécessaires.

3.3.2 Détecter les Fuites Mémoires

Une fuite mémoire apparait quand les zones de mémoire disponibles pour un programme (RAM) sont allouées pour ce programme mais pas libérées ensuite lorsque que le programme n'en a plus besoin. La zone mémoire est laissée comme allouée et n'est plus disponible pour être réutilisée. Quand cela se produit fréquemment, ou dans des cas avec peu de mémoire, le programme peut être à court de mémoire utilisable. Historiquement, la manipulation de la mémoire était de la responsabilité du programmeur. Toute zone de mémoire allouée devait être libérée par le programme l'ayant allouée dans le bon périmètre pour éviter une fuite mémoire. Beaucoup d'environnements de programmation modernes incluent un "garbage collection" (« ramasse miettes ») automatique ou semi-automatique où la mémoire allouée est libérée sans intervention directe du programmeur. Isoler des fuites mémoires peut être très difficile dans des cas où la mémoire actuellement allouée est libérée par le "garbage collection" automatique.

Des fuites mémoires causent des problèmes qui se développent dans le temps et peuvent ne pas être immédiatement évidents. Cela peut être le cas si, par exemple, le logiciel a été récemment installé ou le système redémarré, ce qui se produit souvent lors du test. Pour ces raisons, les effets négatifs des fuites mémoires peuvent souvent n'être remarqués pour la première fois que lorsque le programme est en production.

Les symptômes d'une fuite mémoire sont une dégradation progressive du temps de réponse du système qui peut aboutir ultimement à une défaillance du système. Bien que de telles défaillances puissent être résolues en redémarrant (« rebootant ») le système, cela n'est pas toujours pratique ou même possible.

Beaucoup d'outils d'analyse dynamique identifient des zones dans le code où se produisent des fuites mémoires afin qu'elles puissent être corrigées. De simples contrôleurs de mémoire peuvent être utilisés pour obtenir une impression générale sur la diminution ou non dans le temps de la mémoire disponible, même si une analyse de suivi sera encore requise pour déterminer la cause exacte de la diminution.

Il y a d'autres sources de fuites qui devraient aussi être considérées. On peut citer comme exemple la gestion des fichiers, les sémaphores et les pools de connexion pour des ressources.

3.3.3 Détecter des Pointeurs Sauvages

Des pointeurs "sauvages" dans un programme sont des pointeurs qui ne doivent pas être utilisés. Par exemple, un pointeur sauvage peut avoir "perdu" l'objet ou la fonction vers laquelle il devrait pointer, ou il ne pointe pas vers la zone mémoire prévue (par ex., il pointe sur une zone qui est au-delà des frontières allouées dans un tableau). Quand un programme utilise des pointeurs sauvages, différentes conséquences peuvent se produire:

- Le programme peut fonctionner comme prévu. Cela peut être le cas lorsque le pointeur sauvage accède à de la mémoire qui n'est pas en cours d'utilisation par le programme et est en théorie « libre » et/ou contient une valeur cohérente.
- Le programme peut se planter. Dans ce cas le pointeur sauvage peut avoir causé l'usage incorrecte d'une partie de la mémoire, ce qui est critique au fonctionnement du programme (par ex., le système d'exploitation).
- Le programme ne fonctionne pas correctement parce que des objets requis par le programme ne peuvent pas être atteints. Dans ces conditions le programme peut continuer à fonctionner, même si un message d'erreur peut être produit.
- Des données dans l'emplacement mémoire peuvent être corrompues par le pointeur et des valeurs incorrectes sont utilisées par la suite.

Notez que tout changement fait sur l'utilisation de la mémoire du programme (par ex., un nouveau "build" consécutif à un changement logiciel) peut provoquer chacune des quatre conséquences listées ci-dessus. Cela est particulièrement critique lorsque le programme se comporte initialement comme prévu malgré l'utilisation de pointeurs sauvages, et ensuite plante de façon inattendue (parfois même en production) à la suite d'un changement logiciel. Il est important de noter que de telles défaillances sont souvent les symptômes d'un défaut sous-jacent (c.à.d, le pointeur sauvage) (Se référer à [Kaner02], "Lesson 74"). Des outils peuvent aider à identifier des pointeurs sauvages quand ils sont utilisés par le programme, indépendamment de leur impact sur l'exécution du programme. Certains systèmes d'exploitation ont des fonctionnalités intégrées pour vérifier les violations d'accès mémoire lors de l'exécution. Par exemple, le système d'exploitation peut générer une exception lorsque l'application essaie d'accéder à un emplacement mémoire qui est en dehors de la zone mémoire allouée à l'application.

3.3.4 Analyse de la Performance

L'analyse dynamique n'est pas utile seulement pour détecter des défaillances. Avec l'analyse dynamique de la performance du programme, des outils aident à identifier des goulots d'étranglement pour la performance et génèrent une large variété de métriques de performance pouvant être utilisées par le développeur pour affiner la performance du système. Par exemple, de l'information peut être fournie sur le nombre de fois qu'un module est appelé lors de l'exécution. Des modules fréquemment appelés seront probablement de bons candidats à une amélioration de la performance.

En fusionnant l'information sur le comportement dynamique du logiciel avec l'information obtenue avec les graphes d'appel lors de l'analyse statique (voir Section 3.2.4), le testeur peut aussi identifier les modules qui pourraient être candidats à du test détaillé et extensif (par ex., des modules qui sont fréquemment appelés et ont de nombreuses interfaces).

L'analyse dynamique de la performance d'un programme est souvent faite lors de la conduite des tests système, même si elle peut aussi être faite lors du test d'un seul sous-système dans les phases amont du test en utilisant des harnais de test.

4. Caractéristiques Qualité pour le Test Technique - 405 mn.

Mots clé

adaptabilité, analysabilité, changeabilité, coexistence, efficacité, installabilité, test de maintenabilité, maturité, test d'acceptation opérationnelle, profil opérationnel, test de performance, test de portabilité, test de récupérabilité, modèle de croissance de fiabilité, test de fiabilité, remplaçabilité, test d'utilisabilité des ressources, robustesse, test de sécurité, stabilité, testabilité

Objectifs d'apprentissage pour Caractéristiques Qualité pour le Test Technique

4.2 Problèmes de Planification Générale

TTA-4.2.1 (K4) Pour un projet et un système sous test en particulier, analyser les exigences non-fonctionnelles et écrire les parties correspondantes du plan de test

4.3 Test de Sécurité

TTA-4.3.1 (K3) Définir l'approche et concevoir des cas de test de haut niveau pour le test de sécurité

4.4 Test de Fiabilité

TTA-4.4.1 (K3) Définir l'approche et concevoir des cas de test de haut niveau pour les caractéristiques qualité de fiabilité et leurs sous-caractéristiques ISO 9126 correspondantes

4.5 Test de Performance

TTA-4.5.1 (K3) Définir l'approche et concevoir des profils opérationnels de haut niveau pour le test de performance

Objectifs d'apprentissage communs

Les objectifs d'apprentissage suivants réfèrent à du contenu couvert dans plus d'une section de ce chapitre.

TTA-4.x.1 (K2) Comprendre et expliquer les raisons d'inclure des tests de maintenabilité, portabilité et d'utilisation des ressources dans une stratégie de test et/ou une approche de test
TTA-4.x.2 (K3) Pour un risque produit particulier, définir le ou les type(s) de test particulier(s) qui sont les plus appropriés
TTA-4.x.3 (K2) Comprendre et expliquer les étapes dans le cycle de vie d'une application où des tests non-fonctionnels devraient être appliqués
TTA-4.x.4 (K3) Pour un scenario donné, définir les types de défauts que vous vous attendriez à trouver en utilisant des types de tests non-fonctionnels

4.1 Introduction

En général, l'Analyste Technique de Test se concentre sur le test de "comment" le produit fonctionne, plutôt que sur les aspects fonctionnels du "quoi" fait par le produit. Ces tests peuvent prendre place à tout niveau. Par exemple, lors du test de composant de systèmes temps réels et embarqués, mener des études de performance et tester l'utilisation des ressources est important. Durant les tests système et les tests d'acceptation opérationnelle, le test des aspects fiabilité, tels que la récupérabilité, est approprié. A ce niveau, les tests sont destinés à tester un système spécifique, c.à.d, des combinaisons de matériel et logiciel. Le système sous test spécifique peut inclure différents serveurs, clients, bases de données, réseaux et autres ressources. Indépendamment du niveau de test, le test devrait être conduit selon les priorités de risques et les ressources disponibles.

La description des caractéristiques qualité de produit fournie dans l'ISO 9126 est utilisée comme un guide pour décrire les caractéristiques. D'autres standards, tels que les séries ISO 25000 (ayant supplanté l'ISO 9126) peuvent aussi être aussi utilisés. Les caractéristiques qualité ISO 9126 sont divisées en caractéristiques, chacune d'elle pouvant avoir des sous-caractéristiques. Celles-ci sont montrées dans le tableau ci-dessous, avec une indication sur la couverture des caractéristiques/sous-caractéristiques par les syllabi Analyste de Test et Analyste Technique de Test.

Caractéristique	**Sous-Caractéristiques**	**Analyste de Test**	**Analyste Technique de Test**
Fonctionnalité	Exactitude, aptitude à l'usage, interopérabilité, conformité	X	
	Sécurité		X
Fiabilité	Maturité (robustesse), tolérance aux fautes, récupérabilité, conformité		X
Utilisabilité	Compréhensibilité, apprentissage, opérabilité, attractivité, conformité	X	
Efficacité	Performance (comportement dans le temps), utilisation des ressources, conformité		X
Maintenabilité	Analysabilité, changeabilité, stabilité, testabilité, conformité		X
Portabilité	Adaptabilité, installabilité, coexistence, remplaçabilité, conformité		X

Même si cette répartition du travail peut varier dans différentes organisations, c'est celle-ci qui est suivie dans ces syllabi ISTQB.

La sous-caractéristique de conformité est montrée pour chacune des caractéristiques qualité. Dans le cas de certains systèmes à sécurité critique ou d'environnements régulés, chaque caractéristique qualité peut avoir à satisfaire des standards ou régulations spécifiques. Comme ces standards peuvent grandement varier selon l'industrie, ils ne seront pas discutés en profondeur ici. Si l'Analyste Technique de Test travaille dans un environnement qui est affecté par des exigences de conformité, il est important de comprendre ces exigences et de s'assurer qu'à la fois le test et la documentation du test satisferont aux exigences de conformité.

Pour toutes les caractéristiques et sous-caractéristiques qualité discutées dans cette section, les risques typiques doivent être reconnus afin qu'une stratégie de test appropriée puisse être construite et documentée. Le test des caractéristiques qualité nécessite une attention particulière au timing du cycle de vie, aux outils requis, au logiciel et à la documentation disponibles, à la disponibilité et à

l'expertise technique. Sans planifier une stratégie pour traiter chaque caractéristique et ses besoins uniques en test, le testeur peut ne pas avoir intégré dans son programme un planning adéquat, un temps de préparation et un temps d'exécution des tests [Bath08]. Une partie de ce test, par ex., le test de performance, nécessite une planification extensive, un équipement dédié, des outils spécifiques, des compétences spécialisées en test et, dans la plupart des cas, une quantité de temps importante. Le test des caractéristiques et sous-caractéristiques qualité doit être intégré à la programmation globale du test, avec des ressources adéquates allouées à l'effort. Chacun de ces domaines a des besoins spécifiques, cible des problèmes spécifiques et peut se produire à différents moments durant le cycle de vie logiciel, comme discuté dans les sections ci-dessous.

Alors que le Test Manager devra compiler et communiquer l'information synthétique des métriques relatives aux caractéristiques et sous-caractéristiques qualité, l'Analyste de Test ou l'Analyste Technique de Test (selon le tableau ci-dessus) collecte l'information pour chaque métrique.

Des mesures des caractéristiques qualité collectées durant les tests en pré-production par l'Analyste Technique de Test peuvent constituer la base pour les SLAs (« Service Level Agreements » ou Accords sur le Niveau de Service) entre le fournisseur et les parties prenantes (par ex., clients, opérateurs) du système logiciel. Dans certains cas, les tests peuvent continuer à être exécutés après que le logiciel soit entré en production, souvent par une équipe ou une organisation séparée. Cela est généralement vu pour des tests d'efficacité et de fiabilité qui peuvent montrer dans l'environnement de production des résultats différents de ceux obtenus dans l'environnement de test.

4.2 Problèmes de Planification Générale

L'échec dans la planification des tests non-fonctionnels peut mettre le succès d'une application fortement en risque. L'Analyste Technique de Test peut se voir demander par le Test Manager d'identifier les principaux risques pour les caractéristiques qualité pertinentes (voir le tableau dans la Section 4.1) et de traiter tout problème de planification associé aux tests proposés. Ceux-ci peuvent être utilisés dans la création du Plan de Test Maître. Les facteurs généraux suivants sont pris en compte lors de la réalisation de ces tâches :

- Exigences des parties prenantes
- Acquisition des outils requis et formation à ces outils
- Exigences d'environnements de test
- Considérations organisationnelles
- Considérations relatives à la sécurité des données

4.2.1 Exigences des Parties Prenantes

Les exigences non-fonctionnelles sont souvent pauvrement spécifiées ou même inexistantes. A l'étape de planification, l'Analyste Technique de Tests doit être capable d'obtenir de la part des parties prenantes concernées les niveaux d'attente relatifs aux caractéristiques qualité techniques et d'évaluer les risques qu'ils représentent.

Une approche commune est de considérer que si le client est satisfait avec la version existante du système, il continuera à être satisfait par les nouvelles versions, aussi longtemps que les niveaux de qualité atteints seront maintenus. Cela permet à la version existante du système d'être utilisée comme comparaison. Cela peut être une approche particulièrement utile à adopter pour certaines des caractéristiques qualité non-fonctionnelles comme la performance, où les parties prenantes peuvent trouver difficile de spécifier leurs exigences.

Elle peut être conseillée pour obtenir des points de vue multiples lors de la collecte des exigences non fonctionnelles. Elles doivent être élicitées des parties prenantes telles que les clients, les

utilisateurs, les équipes opérationnelles et les équipes de maintenance ; sinon certaines exigences seront probablement omises.

4.2.2 Acquisition des Outils Requis et Formation à ces Outils

Les outils commerciaux ou simulateurs sont particulièrement pertinents pour la performance et certains tests de sécurité. Les Analystes Techniques de Test devraient estimer les coûts et les délais impliqués pour acquérir, apprendre et mettre en œuvre les outils. Là où des outils spécialisés doivent être utilisés, la planification devrait prendre en compte les courbes d'apprentissage pour les nouveaux outils et/ou le coût d'emploi des spécialistes outils externes.

Le développement d'un simulateur complexe peut représenter un projet de développement à part entière et devrait être planifié comme tel. En particulier, le test et la documentation de l'outil développé doivent être pris en compte dans l'organisation du planning et des ressources. Un budget et un temps suffisants devraient être planifiés pour mettre à jour et re-tester le simulateur lorsque le produit simulé change. La planification pour des simulateurs à utiliser dans des applications à sécurité critique doit prendre en compte le test d'acceptation et la possible certification du simulateur par en organisme indépendant.

4.2.3 Exigences d'Environnements de Test

Beaucoup de tests techniques (par ex., tests de sécurité, tests de performance) nécessitent un environnement de test représentatif de la production afin de fournir des mesures réalistes. Selon la taille et la complexité du système sous test, cela peut avoir un impact significatif sur la planification et le fondement des tests. Comme le coût de tels environnements peut être élevé, les alternatives suivantes peuvent être considérées :

- Utiliser l'environnement de production
- Utiliser une version précédente du système. Une attention doit être portée pour que les résultats obtenus soient suffisamment représentatifs du système en production.

Le timing de telles exécutions de test doit être planifié avec précaution et il est fort probable que de tels tests puissent seulement être exécutés à des moments spécifiques (par ex., à des moments de faible utilisation).

4.2.4 Considérations Organisationnelles

Des tests techniques peuvent impliquer la mesure du comportement de plusieurs composants dans un système complet (par ex., serveurs, bases de données, réseaux). Si ces composants sont distribués sur un certain nombre de sites et organisations différents, l'effort requis pour planifier et coordonner les tests peut être significatif. Par exemple, certains composants logiciels peuvent être disponibles pour le test système seulement à certains moments du jour ou de l'année, ou des organisations peuvent offrir du support au test pour seulement un nombre limité de jours. Echouer à confirmer que les composants système et l'équipe provenant d'autres organisations (c.à.d, l'expertise "empruntée") sont disponibles "sur appel" pour des buts de test peut aboutir à une sérieuse perturbation dans les tests programmés.

4.2.5 Considérations relatives à la Sécurité des Données

Des mesures spécifiques de sécurité implémentées pour un système devraient être prises en compte à l'étape de la planification des tests pour assurer que toutes les activités de test sont possibles. Par exemple, l'utilisation de données cryptées peut rendre difficile la création des données de test et la vérification des résultats.

Des politiques de protection de données et des lois peuvent empêcher la génération de toute donnée de test requise sur la base de la protection des données. Rendre anonyme les données de test est une tâche non triviale qui doit être planifiée comme faisant partie de l'implémentation des tests.

4.3 Test de Sécurité

4.3.1 Introduction

Le test de sécurité diffère des autres formes de test fonctionnel dans deux domaines significatifs :

1. Des techniques standard pour la sélection les données de test en entrée peuvent manquer d'importants problèmes de sécurité
2. Les symptômes des défauts de sécurité sont très différents de ceux trouvés avec d'autres types de test fonctionnel

Le test de sécurité évalue la vulnérabilité aux menaces d'un système en essayant de trouver un compromis à la politique de sécurité du système. Ce qui suit est une liste de menaces potentielles qui devraient être explorées durant le test de sécurité:

- Copie non autorisée de l'application ou des données
- Un contrôle d'accès non autorisé (par ex., la possibilité de réaliser des tâches pour lesquelles l'utilisateur n'a pas les droits). Les droits, les accès et les privilèges des utilisateurs sont la cible de ce test. Ces informations devraient être disponibles dans les spécifications du système
- Du logiciel qui montre des effets de bords non escomptés lors de la réalisation de la fonction prévue. Par exemple, un lecteur media qui joue correctement le son mais le fait en écrivant des fichiers dans un stockage temporaire non crypté provoque un effet de bord qui peut être exploité par des pirates logiciels
- Du code inséré dans une page web qui peut être exercée par des utilisateurs subséquents (scripting entre sites ou XSS). Ce code peut être malveillant.
- Un débordement de pile qui peut être causé par la saisie dans le champ de saisie d'une interface utilisateur de chaînes qui sont plus longues que ce que le code peut correctement manipuler. Une vulnérabilité de débordement de pile représente une opportunité pour exécuter des instructions de code malveillantes.
- Un déni de service, qui empêche des utilisateurs d'interagir avec une application (par ex., en surchargeant un serveur web avec des requêtes « polluantes »).
- L'interception, l'imitation et/ou l'altération et le relai consécutif de communications (par ex., transactions de cartes de crédit) par un tiers de façon à ce que l'utilisateur reste ignorant de la présence du tiers (attaque "Man in the Middle")
- Casser les codes de cryptage utilisés pour protéger des données sensibles
- Des bombes logiques (parfois appelées Œufs de Pâques), qui peuvent être insérées de façon malveillante dans le code et qui s'activent seulement sous certaines conditions (par ex., à une date spécifique). Lorsque les bombes logiques s'activent, elles peuvent effectuer des actes malveillants comme l'effacement de fichiers ou le formatage de disques.

4.3.2 Planification du Test de Sécurité

En général les aspects suivants sont d'une pertinence particulière lors de la planification des tests de sécurité :

- Comme des problèmes de sécurité peuvent être introduits lors de l'architecture, de la conception et de l'implémentation du système, le test de sécurité peut être programmé pour les niveaux de test unitaire, intégration et système. A cause de la nature changeante des menaces de sécurité, les tests de sécurité peuvent aussi être programmés régulièrement après que le système soit entré en production.

- Les stratégies de test proposées par l'Analyste Technique de Test peuvent inclure des revues de code et de l'analyse statique avec des outils de sécurité. Elles peuvent être efficaces pour trouver dans l'architecture, les documents de conception et le code des problèmes de sécurité qui sont facilement manqués durant le test dynamique.
- L'Analyste Technique de Test peut se voir demander de concevoir et exécuter certaines "attaques" de sécurité (voir ci-dessous) qui requièrent une planification attentive et une coordination avec les parties prenantes. D'autres tests de sécurité peuvent être effectués en coopération avec les développeurs ou avec des Analystes de Test (par ex., test des droits, accès et privilèges des utilisateurs). La planification des tests de sécurité doit inclure une considération attentive des problèmes organisationnels tels que ceux-ci.
- Un aspect essentiel de la planification des tests de sécurité est d'obtenir des approbations. Pour l'Analyste Technique de Test, cela signifie obtenir la permission explicite du Test Manager de réaliser les tests de sécurité prévus. Tous tests additionnels, non planifiés, réalisés, pourraient apparaitre comme étant des attaques réelles et la personne réalisant ces tests pourrait encourir le risque d'une action en justice. Sans aucun écrit montrant l'intention et l'autorisation, l'excuse "Nous exécutions un test de sécurité" pourrait être difficile à expliquer de façon convaincante.
- Il devrait être noté que des améliorations pouvant être apportées à la sécurité d'un système peuvent affecter sa performance. Après avoir fait des améliorations de sécurité, il est conseillé de considérer le besoin de conduire des tests de performance (voir Section 4.5 ci-dessous).

4.3.3 Spécification du Test de Sécurité

Des tests de sécurité particuliers peuvent être groupés [Whittaker04] selon l'origine du risque de sécurité:

- Liés à l'interface utilisateur – accès non-autorisés et entrées malveillantes
- Liés au système de fichiers – accès à des données sensibles enregistrées dans des fichiers ou des répertoires
- Liés au système d'exploitation – sauvegarde d'informations sensibles telles que des mots de passe sous forme non cryptée qui pourraient être dévoilés si le système est planté par le biais d'entrée malveillantes
- Liés à du logiciel externe – des interactions qui peuvent se produire parmi des composants externes utilisés par le système. Celles-ci peuvent être au niveau réseau (par ex., transmission de paquets ou données incorrectes) ou au niveau composant logiciel (par ex., défaillance d'un composant logiciel sur lequel le logiciel repose).

L'approche suivante [Whittaker04] peut être utilisée pour développer des tests de sécurité:

- Collecter des informations pouvant être utiles à la spécification des tests comme des noms d'employés, des adresses physiques, des détails relatifs aux réseaux internes, des adresses IP, l'identification du logiciel ou matériel utilisé et la version du système d'exploitation.
- Réaliser un "scan" de vulnérabilité en utilisant des outils largement disponibles. De tels outils ne sont pas utilisés directement pour compromettre le(s) système(s), mais pour identifier des vulnérabilités qui sont, ou peuvent résulter en, une violation de la politique de sécurité. Des vulnérabilités spécifiques peuvent aussi être identifiées en utilisant des checklists telles que celles fournies par le NIST (National Institute of Standards and Technology) [Web-2].
- Développer des "plans d'attaques" (c.à.d, un plan d'actions de test destinées à compromettre la politique de sécurité d'un système particulier) utilisant l'information collectée. Plusieurs entrées via différentes interfaces (par ex., interface utilisateur, système de fichiers) sont à spécifier dans les plans d'attaques pour détecter les fautes de sécurité les plus graves. Les différentes "attaques" décrites dans [Whittaker04] sont une source valable de techniques développées spécifiquement pour le test de sécurité.

Des problèmes de sécurité peuvent aussi être mis en évidence par des revues (voir chapitre 5) et/ou l'utilisation d'outils d'analyse statique (voir Section 3.2). Les outils d'analyse statique contiennent un ensemble important de règles qui sont spécifiques aux menaces de sécurité et par rapport auxquelles le code est vérifié. Par exemple, des problèmes de débordement de pile, causés par des défaillances dans la vérification de la taille de la pile avant l'affectation des données, peuvent être trouvés par l'outil.

Des outils d'analyse statique peuvent être utilisés pour le code web pour vérifier l'exposition possible à des vulnérabilités de sécurité telles que de l'injection de code, la sécurité des cookies, le « scripting » transverse au site, la manipulation de ressources et l'injection de code SQL.

4.4 Test de Fiabilité

La classification ISO 9126 des caractéristiques qualité produit définit les sous-caractéristiques de fiabilité suivantes:

- Maturité
- Tolérance aux fautes
- Récupérabilité

4.4.1 Mesurer la Maturité du Logiciel

Un objectif du test de fiabilité est de suivre une mesure statistique de la maturité du logiciel dans le temps et de la comparer à un objectif de fiabilité désiré qui peut être exprimé comme un « Service Level Agreement » ou «Accord de Niveau de Service ». Les mesures peuvent prendre la forme d'un Temps Moyen Entre les Défaillances (MTBF: "Mean Time Between Failures"), Temps Moyen avant Réparation (MTTR: Mean Time To Repair) ou toute autre forme de mesure de l'intensité des défaillances (par ex., nombre de défaillances d'une certaine sévérité se produisant par semaine). Celles-ci peuvent être utilisées comme critère de sortie (par ex., pour une livraison en production).

4.4.2 Tests pour la Tolérance aux Fautes

En plus du test fonctionnel qui évalue la tolérance aux fautes du logiciel en termes de gestion de valeurs d'entrée inattendues (aussi appelé test négatif), du test additionnel est nécessaire pour évaluer la tolérance d'un système à des fautes qui se produisent à l'extérieure de l'application sous test. De telles fautes sont typiquement rapportées par le système d'exploitation (par ex., disque plein, processus ou service non disponible, fichier non trouvé, mémoire non disponible). Les tests de tolérance aux fautes au niveau système peuvent être assistés par des outils spécifiques.

Noter que les termes "robustesse" et "tolérance aux erreurs" sont aussi communément utilisés lorsque l'on parle de la tolérance aux fautes (voir [ISTQB_GLOSSARY] pour des détails).

4.4.3 Test de Récupérabilité

D'autres formes de tests de fiabilité évaluent la capacité du système logiciel à se remettre de défaillances matérielles ou logicielles d'une façon prédéterminée qui permet ensuite à des opérations normales de reprendre. Les tests de récupérabilité incluent des tests de Réplication de Backup et de Restauration.

Les tests de réplication sont réalisés où les conséquences d'une défaillance logicielle sont si négatives que des mesures matérielles et/ou logicielles spécifiques ont été implémentées pour assurer le fonctionnement du système même dans le cas d'une défaillance. Des tests de réplication peuvent être applicables, par exemple, où le risque de pertes financières est extrême oubien où des problèmes de sécurité critiques existent. Là où les défaillances peuvent aboutir à des évènements

catastrophiques, cette forme de test de récupérabilité peut aussi être appelée test de « récupération de désastre ».

Des mesures préventives typiques pour des défaillances matérielles peuvent inclure de la répartition de charge entre plusieurs processeurs et la réplication de serveurs ou disques afin de pouvoir immédiatement s'appuyer sur un autre en cas de défaillance (systèmes redondants). Une mesure logicielle typique peut être l'implémentation de plus d'une instance indépendante d'un système logiciel (par exemple, le système de contrôle de vol d'un avion) ayant ainsi des systèmes redondants différents. Des systèmes redondants sont typiquement une combinaison de mesures logicielles et matérielles et peuvent être appelés des systèmes « duplex », « triplex » ou « quadruplex », selon le nombre d'instances indépendantes (respectivement deux, trois ou quatre). Un aspect différent pour le logiciel est atteint quand les mêmes exigences logicielles sont fournies à deux (ou plus) équipes de développement indépendantes et non connectées, avec l'objectif d'avoir les mêmes services fournis par des logiciels différents. Cela protège les systèmes redondants différents dans le sens où une même entrée défectueuse aura moins de chance de provoquer le même résultat. Ces mesures prises pour améliorer la récupérabilité d'un système peuvent influencer directement sa fiabilité aussi et devraient aussi être prise en compte lors de la réalisation des tests de fiabilité.

Le test de réplication est conçu pour tester explicitement des systèmes en simulant des modes de défaillance ou en causant réellement des défaillances dans un environnement contrôlé. A la suite d'une défaillance le mécanisme de réplication est testé pour assurer que les données ne sont pas corrompues ou perdues et que tous les niveaux de services annoncés sont maintenus (par ex., disponibilité des fonctions ou temps de réponse). Pour plus d'information sur le test de réplication, voir [Web-1].

Les tests de Backup et Restauration se concentrent sur les mesures procédurales mises en place pour minimiser les effets d'une défaillance. De tels tests évaluent les procédures (en général documentées dans un manuel) pour prendre différentes formes de backup et pour restaurer ces données en cas de pertes ou corruption de données. Des cas de test sont conçus pour assurer que des chemins critiques au travers de chaque procédure sont couverts. Des revues techniques peuvent être réalisées pour "exécuter à blanc" ces scénarios et valider les manuels par rapport aux procédures actuelles. Des tests d'acceptation opérationnelle (OAT) exercent les scénarios dans un environnement de production ou représentatif de la production pour valider leur utilisation réelle.

Des mesures pour les tests de backup et restauration peuvent inclure les suivantes:

- Temps pris pour réaliser les différents types de backup (par ex., complet, incrémental)
- Temps pris pour restaurer les données
- Niveaux des backups de données garantis (par ex., récupération de toutes les données n'ayant pas plus de 24 heures, récupération de transactions de données spécifiques n'ayant pas plus d'une heure d'ancienneté)

4.4.4 Planification du Test de Fiabilité

En général les aspects suivants sont d'une pertinence particulière lors de la planification des tests de fiabilité:

- La fiabilité peut continuer à être suivie après l'entrée en production du logiciel. L'organisation et l'équipe responsable du fonctionnement du logiciel doivent être consultées lors de la collecte des exigences de fiabilité à des fins de planification des tests.
- L'Analyste Technique de Test peut sélectionner un modèle de croissance de fiabilité qui montre les niveaux de fiabilité attendus dans le temps. Un modèle de croissance de fiabilité peut fournir des informations utiles au Test Manager en permettant la comparaison entre les niveaux de fiabilité attendus et obtenus.

- Les tests de fiabilité devraient être menés dans un environnement représentatif de la production. L'environnement utilisé devrait rester aussi stable que possible pour permettre le suivi dans le temps des tendances de fiabilité.
- Comme les tests de fiabilité requièrent souvent l'utilisation du système entier, les tests de fiabilité sont le plus souvent faits comme une part du test système. Cependant, des composants individuels peuvent faire l'objet de tests de fiabilité de même que des ensembles intégrés de composants. Une architecture détaillée, des revues de conception et de code peuvent aussi être utilisées pour supprimer une partie du risque des problèmes de fiabilité se produisant dans le système implémenté.
- Afin de produire des résultats de test qui soient statistiquement significatifs, les tests de fiabilité requièrent généralement une importante durée d'exécution. Cela peut les rendre difficiles à programmer parmi d'autres tests planifiés.

4.4.5 Spécification du Test de Fiabilité

Le test de fiabilité peut prendre la forme d'un ensemble répété de tests prédéterminés. Il peut s'agir de tests sélectionnés au hasard dans un pool ou de cas de test générés par un module statistique utilisant des méthodes aléatoires ou semi-aléatoires. Les tests peuvent aussi être basés sur des modèles qui sont parfois dénommés "Profils opérationnels" (voir Section 4.5.4).

Certains tests de fiabilité peuvent spécifier que des actions intenses sur la mémoire soient exécutées de façon répétitive afin que de possibles fuites mémoires puissent être détectées.

4.5 Test de Performance

4.5.1 Introduction

La classification ISO 9126 des caractéristiques qualité produit inclue la performance (comportement dans le temps) comme une sous-caractéristique de l'efficacité. Le test de performance se concentre sur la capacité d'un composant ou système à répondre aux entrées de l'utilisateur ou du système dans un temps spécifié et sous des conditions spécifiées.

Les mesures de performance varient selon les objectifs du test. Pour des composants logiciels individuels, la performance peut être mesurée selon les cycles CPU, alors que pour des systèmes basés clients, la performance peut être mesurée en fonction du temps pris pour répondre à une requête utilisateur particulière. Pour les systèmes dont les architectures sont constituées de plusieurs composants (par ex., clients, serveurs, bases de données) des mesures de performance sont prises pour les transactions entre composants individuels afin que des « goulots d'étranglement » de performance puissent être identifiés.

4.5.2 Types de Tests de Performance

4.5.2.1 Test de Charge

Le test de charge se concentre sur la capacité d'un système à prendre en charge des niveaux croissants de charges réalistes anticipées provenant de requêtes de transactions générées par un grand nombre d'utilisateurs ou processus concurrents.
Les temps de réponse moyens pour des utilisateurs dans différents scénarios d'usage typique (profil opérationnels) peuvent être mesurés et analysés. Voir aussi [Splaine01].

4.5.2.2 Test de Stress

Le test de stress se concentre sur la capacité d'un système ou composant à supporter des pics de charges à la limite ou au-delà de la limite des charges qui ont été anticipées ou spécifiées pour lui, ou avec une disponibilité réduite de ressources telles que la capacité disponible de l'ordinateur ou la bande passante disponible. Les niveaux de performance devraient se dégrader lentement et de façon prévisible sans défaillance lors de l'augmentation des niveaux de stress. En particulier, l'intégrité fonctionnelle du système devrait être testée lorsque le système est sous stress afin de trouver de possibles fautes dans le traitement fonctionnel ou des inconsistances de données. Un objectif possible du test de stress est de découvrir les limites auxquelles un système plante réellement afin que le « maillon faible de la chaîne » puisse être déterminé. Le test de stress permet d'ajouter une capacité supplémentaire au système au bon moment (par ex., mémoire, capacité CPU, stockage en base de données).

4.5.2.3 Test d'évolutivité

Le test d'évolutivité se concentre sur la capacité d'un système à satisfaire à de futures exigences d'efficacité, qui peuvent être au-delà de celles actuellement requises. L'objectif des tests est de déterminer la capacité du système à croître (par ex., avec plus d'utilisateurs, de plus grosses quantités de données stockées) sans dépasser les exigences de performance actuellement spécifiées ou planter. Une fois les limites d'évolutivité connues, des valeurs seuil peuvent être fixées et suivies en production pour fournir une alerte de problèmes imminents. De plus, l'environnement de production peut être ajusté avec les quantités nécessaires de matériel.

4.5.3 Planification du Test de Performance

En plus des problèmes généraux de planification décrits dans la Section 4.2, les facteurs suivants peuvent influencer la planification des tests de performance:

- Selon l'environnement de test utilisé et le logiciel testé, (voir Section 4.2.3) les tests de performance peuvent nécessiter une implémentation du système entier avant que du test efficace puisse être fait. Dans ce cas, le test de performance est habituellement programmé pour se produire durant le test système. D'autres tests de performance qui peuvent être conduits efficacement au niveau composant peuvent être programmés pendant le test unitaire.
- En général, il est souhaitable de mener des tests de performance initiaux aussi tôt que possible, même si un environnement représentatif de la production n'est pas disponible. Ces tests menés tôt peuvent trouver des problèmes de performance (par ex. Goulots d'étranglement) et de réduire le risque du projet en évitant des corrections coûteuses en temps dans les phases plus avancées du développement logiciel ou de la production.
- Les revues de code, en particulier celles avec un focus sur les interactions avec la base de données, les interactions avec les composants et la gestion des erreurs, peuvent identifier des problèmes de performance (en particulier par rapport à la logique "attendre et réessayer" et aux requêtes inefficaces) et devraient être planifiées tôt dans le cycle de développement logiciel.
- Le matériel, le logiciel, et la bande passante réseau nécessaires pour exécuter les tests de performance devraient être planifiés et budgétés. Les besoins dépendent principalement de la charge à générer, qui peut être basée sur le nombre d'utilisateurs virtuels à simuler et la quantité de trafic réseau qu'ils sont censés générer. Un échec dans le calcul de ces éléments peut aboutir à la prise de mesures de performance non représentatives. Par exemple, vérifier les exigences d'évolutivité d'un site internet très fréquenté peut nécessiter la simulation de centaines de milliers d'utilisateurs virtuels.
- Générer la charge requise pour des tests de performance peut avoir une influence significative sur les coûts d'acquisition du matériel et des outils. Cela doit être pris en compte

dans la planification des tests de performance pour assurer la disponibilité d'un financement suffisant.

- Les coûts de génération de la charge pour les tests de performance peuvent être minimisés en louant l'infrastructure de test requise. Cela peut impliquer, par exemple, la location de licences additionnelles pour les outils de performance ou l'utilisation des services d'un fournisseur tiers pour répondre à des besoins matériels (par ex., services cloud). Si cette approche est prise, le temps disponible pour mener les tests de performance peut être limité et doit par conséquent être planifié avec précaution.
- Des précautions doivent être prises au moment de la planification pour assurer que l'outil de performance à utiliser apporte la compatibilité requise avec les protocoles de communication utilisés par le système sous test.
- Les défauts relatifs à la performance ont souvent un impact significatif sur le système sous tests. Quand des exigences de performance sont impératives, il est souvent utile de mener des tests de performance sur les composants critiques (par l'intermédiaire de pilotes et bouchons) au lieu d'attendre les tests système.

4.5.4 Spécification du Test de Peformance

La spécification des tests pour différents types de test de performance tels que la charge et le stress est basée sur la définition de profils opérationnels. Ils représentent des formes distinctes de comportements utilisateurs lors de l'interaction avec une application. Il peut y avoir de multiples profils opérationnels pour une application donnée.

Les nombres d'utilisateurs par profil opérationnel peuvent être obtenus en utilisant des outils de supervision (où l'application réelle, ou une application comparable, est déjà disponible) ou en précisant l'utilisation. De telles prédictions peuvent être basées sur des algorithmes ou fournies par l'organisation métier, et sont particulièrement importantes pour spécifier le(s) profil(s) opérationnel(s) à utiliser pour le test d'évolutivité.

Les profils opérationnels sont la base pour la quantité et les types de cas de test à utiliser lors du test de performance. Ces tests sont souvent contrôlés par des outils de test qui créent des utilisateurs "virtuels" ou simulés dans des quantités qui représenteront le profil sous test (voir Section 6.3.2).

4.6 Utilisation des Ressources

La classification ISO 9126 des caractéristiques qualité produit inclue l'utilisation des ressources comme une sous-caractéristique de l'efficacité. Les tests relatifs à l'utilisation des ressources évaluent l'utilisation des ressources du système (par ex., utilisation de la mémoire, capacité du disque, bande passante réseau, connections) par rapport à un benchmark prédéfini. Ils sont comparés à la fois dans des situations de charges normales et dans des situations de stress, telles que des hauts niveaux de transaction et de volumes de données, pour déterminer si une augmentation anormale de l'usage se produit.

Par exemple, pour les systèmes embarqués temps réel, l'utilisation de la mémoire (parfois appelé « empreinte mémoire ») joue un rôle significatif dans le test de performance. Si l'empreinte mémoire dépasse la mesure autorisée, le système peut ne pas avoir la mémoire suffisante pour réaliser ses tâches dans les périodes de temps spécifiées. Cela peut ralentir le système ou même aboutir à un crash système.

L'analyse dynamique peut aussi être appliquée à la tâche d'investigation de l'utilisation des ressources (voir Section 3.3.4) et de détection des goulots d'étranglement de performance.

4.7 Test de Maintenabilité

Le logiciel passe souvent en réalité une plus grande partie de son temps de vie à être maintenu qu'à être développé. Le test de maintenance est réalisé pour tester les changements sur un système opérationnel ou l'impact d'un environnement modifié sur un système opérationnel. Pour assurer que la tâche de conduite de la maintenance est aussi efficace que possible le test de maintenabilité est réalisé pour mesurer l'aisance avec laquelle le code peut être analysé, changé et testé

Des objectifs typiques de maintenabilité des parties prenantes concernées (par ex., le responsable du logiciel ou l'opérateur) incluent:

- Minimiser le coût de possession ou de fonctionnement du logiciel
- Minimiser le temps d'indisponibilité requis pour la maintenance logicielle

Les tests de maintenabilité devraient être inclus dans une stratégie de test et/ou une approche de test où un ou plusieurs des facteurs suivants s'applique:

- Des changements logiciels sont probables après l'entrée du logiciel en production (par ex., pour corriger des défauts ou introduire des mises à jour planifiées)
- Les bénéfices de l'atteinte des objectifs de maintenabilité (voir ci-dessus) au cours du cycle de vie logiciel sont considérés par les parties prenantes concernées pour l'emporter sur les coûts de réalisation des tests de maintenabilité et effectuer tout changement nécessaire
- Les risques d'une mauvaise maintenabilité logicielle (par ex., de longs temps de réponses aux défauts signalés par les utilisateurs et/ou clients) justifient la conduite des tests de maintenabilité

Les techniques appropriées pour le test de maintenabilité incluent l'analyse statique et les revues comme discuté dans les sections 3.2 et 5.2. Le test de maintenabilité devrait être commencé dès que les documents de conception sont disponibles et devrait continuer tout au long de l'effort d'implémentation du code. Comme la maintenabilité est intégrée au code et à la documentation pour chaque composant de code individuel, la maintenabilité peut être évaluée tôt dans le cycle de vie sans avoir à attendre un système terminé et exécuté.

Le test de maintenabilité dynamique se concentre sur les procédures documentées développées pour maintenir une application particulière (par ex., pour réaliser des mises à jour logicielles). Des sélections de scénarios de maintenance sont utilisées comme cas de test pour garantir que les niveaux de service requis sont atteignables avec les procédures documentées. Cette forme de test est particulièrement pertinente où l'infrastructure interne est complexe, et où les procédures de support peuvent impliquer de multiples départements/organisations. Cette forme de test peut prendre place dans les tests d'acceptation opérationnelle. [Web-1]

4.7.1 Analysabilité, Changeabilité, Stabilité et Testabilité

La maintenabilité d'un système peut être mesurée en termes d'effort requis pour diagnostiquer les problèmes identifiés dans un système (analysabilité), implémenter les changements dans le code (changeabilité) et tester le système modifié (testabilité). La stabilité relève spécifiquement de la réponse du système aux changements. Les systèmes avec une faible stabilité affichent un grand nombre de problèmes en aval (aussi connu comme « l'effet cascade » à chaque fois qu'un changement est fait. [ISO9126] [Web-1].

L'effort requis pour réaliser des tâches de maintenance est dépendant de nombreux facteurs comme la méthodologie de conception logicielle (par ex., orientée objet) et les standards de codage utilisés.

Noter que la "stabilité" dans ce contexte ne devrait pas être confondue avec les termes "robustesse" et "tolérance aux fautes", qui sont couverts dans la Section 4.4.2.

4.8 Test de Portabilité

Les tests de portabilité en général concernent la facilité avec laquelle un logiciel peut être transféré dans ses environnements prévus, soit initialement, soit à partir d'un environnement existant. Les tests de Portabilité incluent les tests pour l'installabilité, la coexistence/compatibilité, l'adaptabilité et la remplaçabilité. Le test de portabilité peut commencer avec les composants individuels (par ex., remplaçabilité d'un composant particulier tel que passer d'un système de gestion de base de données à un autre) et élargira son périmètre lorsque plus de code sera disponible. L'instabilité peut ne pas être testable avant que tous les composants du produit ne marchent fonctionnellement. La portabilité doit être conçue et implémentée dans le produit et ainsi doit être considérée tôt dans les phases de conception et architecture. Des revues d'architecture et de conception peuvent être particulièrement productives pour identifier de possibles exigences et problèmes de portabilité (par ex., dépendance envers un système d'exploitation particulier).

4.8.1 Test d'Installabilité

Le test d'installabilité est mené sur le logiciel et des procédures écrites utilisées pour installer le logiciel dans son environnement cible. Cela peut inclure, par exemple, le logiciel développé pour installer un système d'exploitation sur un processeur, ou un "wizard" d'installation utilisé pour installer un produit sur un PC client.

Les objectifs typiques de test d'installabilité incluent:

- Valider que le logiciel puisse être installé avec succès en suivant les instructions contenues dans le manuel d'installation (incluant l'exécution de tout script d'installation), ou en utilisant un wizard d'installation. Cela inclue l'essai des options d'installation pour différentes configurations matérielles/logicielles et pour différents degrés d'installation (par ex., initial ou mise à jour).
- Tester si les défaillances qui se produisent durant l'installation (par ex., échec à charger des DLLs particulières) sont traitées correctement par le logiciel d'installation sans laisser le système dans un état indéfini (par ex., logiciel partiellement installé ou configuration système incorrecte)
- Tester si une installation/désinstallation partielle pourra être terminée
- Tester si un wizard d'installation peut identifier avec succès des plateformes matérielles ou des configurations de systèmes d'exploitation invalides
- Mesurer si le processus d'installation peut être terminé dans la limite d'un nombre spécifié de minutes ou d'un nombre spécifié d'étapes
- Valider que le logiciel puisse être remis dans son état précèdent ou désinstallé avec succès

Le test de fonctionnalité est normalement conduit après le test d'installation pour détecter toute faute pouvant avoir été introduite lors de l'installation (par ex., configurations incorrectes, fonctions non disponibles). Le test d'utilisabilité est normalement mené en parallèle du test d'installabilité (par ex., pour valider que l'on fournit aux utilisateurs des instructions compréhensibles et des messages de feedback/erreur durant l'installation).

4.8.2 Test de Coexistence/Compatibilité

Des systèmes informatiques qui ne sont pas liés les uns aux autres sont dit compatibles quand ils peuvent s'exécuter dans le même environnement (par ex., sur le même matériel) sans affecter leurs comportements réciproques (par ex., conflits de ressources). La compatibilité devrait être réalisée quand du logiciel nouveau ou mis à jour sera déployé dans des environnements contenant déjà des applications installées.
Des problèmes de compatibilité peuvent survenir quand l'application est testée dans un environnement où elle est la seule application installée (où des problèmes d'incompatibilité ne sont

pas détectables) et ensuite déployée sur un autre environnement (par ex., production) qui contient aussi d'autres applications en exécution.

Les objectifs typiques des tests de compatibilité incluent :

- Evaluation de l'impact adverse sur la fonctionnalité quand les applications sont chargées dans le même environnement (par ex., conflit d'usage de ressource quand un serveur exécute de multiples applications)
- Evaluation sur toute application de l'impact résultant du déploiement de corrections et mises à jour du système d'exploitation

Les problèmes de compatibilité devraient être analysés lors de la planification des environnements de production cibles mais les tests réels sont normalement exécutés après que les tests système et les tests d'acceptation utilisateurs aient été terminés avec succès.

4.8.3 Test d'Adaptabilité

Le test d'adaptabilité vérifie si une application donnée peut fonctionner correctement dans tous les environnements cibles prévus (matériel, logiciel, middleware, système d'exploitation, etc.). Un système adaptif est par conséquent un système ouvert qui est capable d'adapter son comportement selon les changements dans son environnement ou dans des parties du système lui-même. Spécifier des tests pour l'adaptabilité demande à ce que toutes les combinaisons d'environnements cibles prévus soient identifiées, configurées et disponibles pour l'équipe de test. Ces environnements sont ensuite testés en utilisant une sélection de cas de test fonctionnels qui exercent les différents composants présents dans l'environnement.

L'adaptabilité peut relever de la capacité du logiciel à être porté dans différents environnements spécifiés en suivant une procédure prédéfinie. Des tests peuvent évaluer cette procédure.

Des tests d'adaptabilité peuvent être réalisés en conjonction avec des tests d'installabilité et sont typiquement suivis par des tests fonctionnels pour détecter toute faute pouvant avoir été introduite dans l'adaptation du logiciel à un environnement différent.

4.8.4 Test de Remplaçabilité

Le test de remplaçabilité se concentre sur la capacité des composants logiciels à l'intérieur d'un système à être interchangés avec d'autres. Cela peut être particulièrement pertinent pour les systèmes utilisant des composants logiciels sur étagère pour des composants système spécifiques.

Les tests de remplaçabilité peuvent être réalisés en parallèle des tests d'intégration fonctionnelle où plus d'un composant alternatif est disponible pour l'intégration dans le système complet. La remplaçabilité peut être évaluée par des revues techniques ou des inspections aux niveaux architecture et conception, où l'accent est mis sur la définition claire des interfaces avec des composants potentiellement remplaçables.

5. Revues - 165 mn.

Mots clé
anti-pattern

Objectifs d'apprentissage pour Revues

5.1 Introduction

TTA 5.1.1 (K2) Expliquer pourquoi la préparation des revues est importante pour l'Analyste Technique de Test

5.2 Utiliser des Checklists dans les Revues

TTA 5.2.1 (K4) Analyser une conception d'architecture et identifier des problèmes selon une checklist fournie dans le syllabus

TTA 5.2.2 (K4) Analyser une partie de code ou de pseudocode et identifier des problèmes selon une checklist fournie dans le syllabus

5.1 Introduction

Les Analystes Techniques de Test doivent être des participants actifs au processus de revue, apportant leurs vues uniques. Ils devraient avoir une formation formelle aux revues pour mieux comprendre leurs rôles respectifs dans tout processus de revue technique. Tous les participants à la revue doivent être engagés à contribuer aux bénéfices d'une revue technique bien menée. Pour une description complète des revues, incluant de nombreuses checklists de revue, voir [Wiegers02]. Les Analystes Techniques de Test participent normalement aux revues techniques et aux inspections auxquelles ils apportent un point de vue opérationnel (comportemental) qui peut manquer chez les développeurs. En plus, les Analystes Techniques de Test jouent un rôle important dans la définition, l'application, et la maintenance des checklists de revue et des informations relatives à la sévérité des défauts.

Indépendamment du type de revue mené, l'Analyste Technique de Test doit disposer du temps de préparation adéquate. Cela inclue le temps de revoir le livrable, le temps de vérifier les documents mutuellement référencés pour vérifier la consistance, et le temps de déterminer ce qui peut être manquant dans le livrable. Sans un temps de préparation adéquate, la revue peut devenir un exercice d'édition plutôt qu'une réelle revue. Une bonne revue inclue la compréhension de ce qui est écrit, l'identification de ce qui manque, et la vérification que le produit décrit est consistent avec les autres produits qui sont soit déjà développés, soit en développement. Par exemple, lors de la revue d'un plan de test de niveau intégration, l'Analyste Technique de Test doit aussi considérer les éléments qui sont en cours d'intégration. Sont-ils prêts pour l'intégration? Y-a-t-il des dépendances qui doivent être documentées? Des données sont-elles disponibles pour tester les points d'intégration? Une revue n'est pas isolée du livrable en cours de revue. Elle doit aussi considérer l'interaction de cet élément avec les autres dans le système.

Il n'est pas inhabituel pour les auteurs d'un produit en cours de revue de se sentir critiqués. L'Analyste Technique de Test devrait être sûr d'aborder tout commentaire de revue avec un point de vue de collaboration avec l'auteur pour créer le meilleur produit possible. En utilisant cette approche, les commentaires seront formulés de façon constructive et seront orientés vers le livrable et non vers l'auteur. Par exemple, si une phrase est ambigüe, il est mieux de dire « Je ne comprends pas ce que je devrais tester pour vérifier que cette exigence a été implémentée correctement. Pourriez-vous m'aider à la comprendre ? », plutôt que « Cette exigence est ambigüe et personne ne sera capable de la comprendre ».

Le travail de l'Analyste Technique de Test dans une revue est d'assurer que l'information fournie dans le livrable sera suffisante pour assister l'effort de test. Si l'information n'est pas présente ou n'est pas claire, alors il y a probablement un défaut qui a besoin d'être corrigé par l'auteur. En maintenant une approche positive plutôt qu'une approche critique, les commentaires seront mieux reçus et les réunions plus productives.

5.2 Utiliser des Checklists dans les Revues

Les checklists sont utilisées lors des revues pour rappeler aux participants de vérifier des points spécifiques durant la revue. Les checklists peuvent aussi aider à dépersonnaliser la revue, par ex., « il s'agit de la même checklist que nous utilisons pour toutes les revues, et nous ne ciblons pas seulement votre livrable ». Les checklists peuvent être génériques et utilisées pour toutes les revues ou centrées sur des caractéristiques qualité ou domaines spécifiques. Par exemple, une checklist générique peut vérifier le bon usage des termes "doit" et "devrait", vérifier le bon formatage et des éléments de conformité similaires. Une checklist ciblée peut se concentrer sur des problèmes de sécurité ou des problèmes de performance.

Les checklists les plus utiles sont celles développées progressivement par une organisation individuelle, parce qu'elles reflètent :

- La nature du produit
- L'environnement de développement local
  - Equipe
  - Outils
  - Priorités
- L'historique des succès et défauts précédents
- Des problèmes particuliers (par ex., performance, sécurité)

Les checklists devraient être adaptées à l'organisation et peut-être au projet en particulier. Les checklists fournies dans ce chapitre sont destinées à servir d'exemples.

Certaines organisations étendent la notion usuelle de checklist logicielle pour inclure les "anti-patterns" qui réfèrent aux erreurs communes, aux techniques pauvres, et à d'autres pratiques inefficaces. Le terme est dérivé du concept populaire de "design patterns" (modèles de conception) qui sont des solutions réutilisables pour des problèmes communs qui se sont montrées efficaces dans des situations pratiques [Gamma94]. Un anti-pattern, donc, est une erreur faite fréquemment, souvent mise en œuvre comme un raccourci expédient.

Il est important de se souvenir que si une exigence n'est pas testable, c'est-à-dire qu'elle n'est pas définie d'une façon permettant à l'Analyste Technique de Test de déterminer comment la tester, alors c'est un défaut. Par exemple, une exigence qui affirme "Le logiciel doit être rapide » ne peut pas être testée. Comment l'Analyste Technique de Test peut-il déterminer si le logiciel est rapide ? Si, au lieu de cela, l'exigence disait « Le logiciel doit fournir un temps de réponse maximum de trois secondes sous des conditions de charge spécifiques » alors la testabilité de cette exigence serait substantiellement meilleure, si l'on définit les « conditions de charge spécifiques » (Par ex., .le nombre d'utilisateurs simultanés, ou d'activités réalisées par les utilisateurs). C'est aussi une exigence structurante car elle peut facilement produire de nombreux cas de test individuels dans une application non triviale. La traçabilité de cette exigence aux cas de test est aussi critique car si l'exigence devait changer, il serait nécessaire de revoir tous les cas de test et de les mettre à jour si nécessaire.

5.2.1 Revues d'Architecture

L'architecture logicielle consiste en l'organisation fondamentale d'un système, matérialisé par ses composants, leurs relations entre eux et avec l'environnement, et les principes gouvernant sa conception et son évolution [ANSI/IEEE Std 1471-2000], [Bass03].

Des checklists utilisées pour les revues d'architecture peuvent, par exemple, inclure la vérification de la bonne implémentation des éléments suivants, qui sont extraits de [Web-3]:

- "Connections en pool – réduisant le dépassement du temps d'exécution associé à l'établissement des connections en base de données en établissant un pool de connections partagées
- « Load balancing » - répartition uniforme de la charge entre un ensemble de ressources
- Traitement distribué
- Mise en cache – utilisant une copie locale des données pour réduire le temps d'accès
- Instanciation rationnelle
- Concurrence des transactions
- Isolation des traitements entre le Traitement Transactionnel en Ligne (OLTP : Online Transactional Processing) et le Traitement Analytique en Ligne (OLAP : Online Analytical Processing)

- Réplication de donnée"

Plus de détails (non pertinents pour l'examen de certification) peuvent être trouvés dans [Web-4] qui réfère à un article qui étudie 117 checklists provenant de 24 sources. Différentes catégories d'éléments de checklist sont discutées et des exemples de bons éléments de checklists sont fournis de même que ceux qui devraient être évités.

5.2.2 Revues de Codes

Les checklists pour les revues de code sont nécessairement très détaillées, et, comme avec les checklists pour les revues d'architecture, sont plus utiles lorsqu'elles sont spécifiques à un langage, un projet et une société. L'inclusion d'anti-patterns de niveau code aide, particulièrement pour les développeurs logiciels ayant moins d'expérience.

Les checklists utilisées pour les revues de code peuvent inclure les six éléments suivants (à partir de [Web-5]).

1. Structure
   - Le code implémente-t-il complètement et correctement la conception?
   - Le code se conforme-t-il à des standards de codage pertinents ?
   - Le code est-il bien structuré, consistent en termes de style, et formaté de façon consistante ?
   - Y-a-t-il des procédures non appelées ou inutiles, ou du code inatteignable ?
   - Y-a-t-il des bouchons ou routines de test résiduels dans le code?
   - Du code peut-il être remplacé par des appels à des composants externes réutilisables ou à des fonctions de librairie ?
   - Y-a-t-il des blocs de code répétés qui pourraient être condensés dans une seule procédure ?
   - L'utilisation de la sauvegarde est-elle efficace ?
   - Des symboles sont-ils utilisés plutôt que des constantes « nombres magiques » ou des constantes « chaînes de caractère » ?
   - Y-a-t-il des modules excessivement complexes et qui devraient être restructurés ou répartis dans plusieurs modules ?

2. Documentation
   - Le code est-il documenté clairement et de façon adéquate avec un style de documentation facile à maintenir?
   - Les commentaires sont-ils tous cohérents avec le code ?
   - La documentation satisfait-elle les standards applicables ?

3. Variables
   - Les variables sont-elles toutes définies correctement avec des noms significatifs, consistants et clairs?
   - Y-a-t-il des variables redondantes ou inutilisées ?

4. Opérations Arithmétiques
   - Le code évite-t-il de comparer l'égalité de nombres flottants?
   - Le code prévient-il systématiquement les erreurs d'arrondi ?
   - Les diviseurs sont-ils testés pour la valeur zéro ou du bruit ?

5. Boucles et Branches

- Les boucles, branches, et constructions logiques sont-elles toutes terminées, correctes et intégrées correctement?
- Les cas les plus courants sont-ils tous d'abord testés dans des chaines IF-ELSEIF ?
- Les cas sont-ils tous couverts dans un bloc IF-ELSEIF ou CASE, incluant les clauses ELSE ou DEFAULT
- Chaque instruction « case » a-t-elle un « default » ?
- Les conditions de terminaison de boucle sont-elles évidentes et atteignables de façon invariable ?
- Les indices ou sous-scripts sont-ils correctement initialisés, juste avant la boucle ?
- Les instructions inclues dans des boucles peuvent-elles être placées à l'extérieur des boucles ?
- Le code dans la boucle évite-t-il de manipuler la variable d'index ou de l'utiliser après la sortie de la boucle?

6. Programmation Défensive

- Les indices, pointeurs, et sous-scripts sont-ils testés par rapport aux tableaux, enregistrements ou aux capacités de fichier?
- Les données importées et les arguments en entrée sont-ils testés pour leur validité et leur complétude ?
- Les variables de sortie sont-elles toutes affectées ?
- La bonne donnée est-elle traitée dans chaque instruction ?
- Chaque allocation de mémoire est-elle libérée ?
- Des « timeouts » et les captures d'erreurs sont-ils utilisés pour l'accès à des « devices » extérieurs ?
- L'existence des fichiers est-elle vérifiée avant d'essayer d'y accéder?
- Les fichiers et « devices » sont-ils tous laissés dans le bon état après la fin du programme ?

Pour d'autres exemples de checklists utilisées pour les revues de code à différents niveaux de test voir [Web-6].

6. Outils de Test et Automatisation - 195 min.

Mots clé

test dirigé par les données, outil de débogage, outil d'injection de défauts, outil de test d'hyperliens , test dirigé par les mots-clés, outil de test de performance, outil de capture/rejeu, analyseur statique, outil d'exécution de test, outil de gestion des tests

Objectifs d'apprentissage pour Outils de Test et Automatisation

6.1 Intégration et Echange d'Information Entre Outils

TTA-6.1.1 (K2) Expliquer les aspects techniques à considérer lorsque plusieurs outils sont utilisés ensemble

6.2 Définition du Projet d'Automatisation des Tests

TTA-6.2.1 (K2) Résumer les activités que l'Analyste Technique de Test réalise lors de la mise en place d'un projet d'automatisation des tests
TTA-6.2.2 (K2) Résumer les différences entre automatisation dirigée par les données et l'automatisation dirigée par les mots-clés
TTA-6.2.3 (K2) Résumer les problèmes techniques classiques qui font que les projets d'automatisation échouent à apporter le retour sur investissement prévu
TTA-6.2.4 (K3) Créer une table de mots-clés basée sur un processus métier donné

6.3 Outils de Test Spécifiques

TTA-6.3.1 (K2) Résumer les objectifs des outils d'injection de défauts et d'injection de fautes
TTA-6.3.2 (K2) Résumer les caractéristiques principales et les problèmes d'implémentation pour les outils de test de performance et de surveillance
TTA-6.3.3 (K2) Expliquer l'objectif général des outils utilisés pour le test basé Web
TTA-6.3.4 (K2) Expliquer comment les outils assistent le concept de test basé sur les modèles
TTA-6.3.5 (K2) Souligner l'objectif des outils utilisés pour assister le test de composants et le processus de construction

6.1 Intégration et Echange d'Information Entre Outils

Alors que la responsabilité de sélectionner et intégrer les outils appartient au Test Manager, l' Analyste Technique de Test peut ensuite être appelé pour revoir l'intégration d'un outil ou d'un ensemble d'outils pour assurer le suivi précis des données issues des différents domaines de test tels que l' analyse statique, l'automatisation de l'exécution des tests et la gestion de la configuration. En plus, selon les compétences en programmation de l'Analyste Technique de Test, il peut aussi y avoir une implication dans la création du code qui intègrera ensemble les outils qui ne sont pas intégrés "d'origine".

Un ensemble d'outils idéal devrait éliminer la duplication d'informations entre les outils. Cela demande plus d'effort et est davantage sujet aux erreurs de sauvegarder les scripts d'exécution des tests à la fois dans une base de données de gestion des tests et dans un système de gestion de configuration. Il serait mieux d'avoir un système de gestion des tests qui inclue un système de gestion de configuration ou qu'il est possible d'intégrer à un outil de gestion de configuration déjà en place dans l'organisation. Des outils de suivi des défauts et de gestion des tests bien intégrés permettront à un testeur de lancer un rapport de défaut durant l'exécution d'un cas de test sans avoir à quitter l'outil de gestion des tests. Des outils d'analyse statique bien intégrés devraient permettre de rapporter tout incident ou alerte découvert directement dans le système de gestion des défauts (bien que cela doive être configurable compte-tenu du grand nombre d'alertes pouvant être générées).

Acheter une suite d'outils de test du même éditeur ne signifie pas automatiquement que les outils fonctionneront ensemble adéquatement. Lorsque l'on considère l'approche d'intégration des outils ensemble, il est préférable de se concentrer sur les données. Les données doivent être échangées sans intervention manuelle, au bon moment avec la garantie d'exactitude, incluant la récupération de fautes. Même s'il est utile d'avoir une bonne expérience utilisateur, la saisie, l'enregistrement, la protection et la présentation des données devraient être l'objectif premier de l'intégration d'outils.

Une organisation devrait évaluer le coût d'automatisation des échanges d'information comparé au risque de perdre des informations ou de laisser des données non synchronisées à cause d'une nécessaire intervention manuelle. Comme l'intégration peut être coûteuse ou difficile, elle devrait être une considération première dans la stratégie globale d'outillage.

Certains environnements de développement intégrés peuvent simplifier l'intégration entre des outils qui sont capables de fonctionner dans cet environnement. Cela aide à unifier le « look and feel » des outils qui partagent le même environnement. Cependant, une même interface utilisateur ne garantira pas un échange d'information aisé entre les composants. Du codage peut être nécessaire pour terminer l'intégration.

6.2 Définition du Projet d'Automatisation des Tests

Afin d'être rentables, les outils de test et en particulier les outils d'automatisation des tests, doivent être rigoureusement architecturés et conçus. Mettre en œuvre une stratégie d'automatisation des tests sans une solide architecture aboutit en général à un ensemble d'outils coûteux à maintenir, ne satisfaisant pas l'usage prévu et incapable d'atteindre le retour sur investissement ciblé.

Un projet d'automatisation des tests devrait être considéré comme un projet de développement logiciel. Cela comprend la nécessité d'avoir des documents d'architecture, des documents de conception détaillée, des revues de conception et de code, du test de composant et d'intégration de composants de même que du test système final. Le test peut être retardé inutilement ou complexifié si une automatisation instable ou inadaptée est utilisée. Il y a de nombreuses activités que l'Analyste Technique de Test effectue pour l'automatisation des tests. Celles-ci incluent:

- Déterminer qui sera responsable de l'exécution des tests
- Sélectionner l'outil adapté aux exigences de l'organisation, de planning, de compétences de l'équipe, de maintenance (noter que cela peut signifier de décider de créer un outil à utiliser plutôt que d'en acquérir un)
- Définir les exigences d'interface entre l'outil d'automatisation et d'autres outils comme les outils de gestion des tests et de gestion des défauts
- Sélectionner l'approche d'automatisation, c.à.d, dirigée par les mots-clés ou par les données (voir Section 6.2.1 ci-dessous)
- Travailler avec le Test Manager pour estimer le coût d'implémentation, incluant la formation
- Programmer le projet d'automatisation et allouer le temps pour la maintenance
- Former les Analystes de Test et les Analystes Métier pour utiliser et fournir des données pour l'automatisation
- Déterminer comment les tests automatisés seront exécutés
- Déterminer comment les résultats des tests automatisés seront combinés avec les résultats des tests manuels

Ces activités et les décisions qui en résultent influenceront l'adaptabilité et la maintenabilité de la solution d'automatisation. Un temps suffisant doit être consacré à rechercher les options, à étudier les outils et technologies disponibles et à comprendre les futurs plans de l'organisation. Certaines de ces activités nécessitent plus de considération que les autres, en particulier lors du processus de décision. Celles-ci sont discutées plus en détail dans les sections suivantes.

6.2.1 Sélectionner l'Approche d'Automatisation

L'automatisation des tests ne se limite pas à tester au travers des IHM. Il existe des outils qui aident à tester au niveau API, via une interface de commande en ligne (CLI : Command Line Interface) et d'autres points d'interface dans le logiciel sous test. L'une des premières décisions que l'Analyste Technique de Test doit prendre concerne l'interface la plus efficace à accéder pour automatiser le test.

L'une des difficultés de tester via les IHMs est la tendance pour l'IHM à changer lorsque le logiciel évolue. Selon la façon avec laquelle le code d'automatisation des tests est conçu, cela peut aboutir à une importante surcharge de la maintenance. Par exemple, utiliser la fonctionnalité de capture/rejeu d'un outil d'automatisation de test peut aboutir à des cas de test automatisés (souvent appelé scripts de test) qui ne s'exécutent plus comme prévu si l'IHM évolue. Cela est dû au fait que le script enregistré capture des interactions avec les objets graphiques quand le testeur exécute le logiciel manuellement. Si les objets accédés changent, les scripts enregistrés peuvent avoir besoin d'être mis à jour pour refléter ces changements.

Les outils de capture/rejeu peuvent être utilisés comme un point de départ pratique pour développer des scripts d'automatisation. Le testeur enregistre une session de test et le script enregistré est ensuite modifié pour améliorer la maintenabilité (par ex., en remplaçant des parties dans le script enregistré avec des fonctions réutilisables).

Selon le logiciel testé, les données utilisées pour chaque test peuvent être différentes bien que les étapes de test exécutées soient virtuellement identiques (par ex., tester la gestion d'erreur pour un champ d'entrée en entrant plusieurs valeurs invalides et en vérifiant les erreurs retournées pour chacune). Il est inefficace de développer et maintenir un script de test automatisé pour chacune de ces valeurs à tester. Une solution technique commune à ce problème est de déplacer les données des scripts vers un enregistrement extérieur tel qu'une feuille de calcul ou une base de données. Des fonctions sont écrites pour accéder aux données spécifiques pour chaque exécution du script de test, ce qui permet à un même script de fonctionner avec un ensemble de données de test fournissant les valeurs d'entrée et les valeurs des résultats attendus (par ex., une valeur montrée dans un champ

texte ou un message d'erreur). Cette approche est appelée dirigée par les données. Lors de l'utilisation de cette approche, un script de test qui traitera les données fournies est développé, de même qu'un harnais et l'infrastructure nécessaire pour assister l'exécution du script ou de l'ensemble de scripts.
Les données effectives tenues dans la feuille de calcul ou la base de données sont créées par les Analystes de Test qui sont familiers à la fonction métier du logiciel. Cette division du travail permet aux responsables du développement des scripts de test (par ex., l'Analyste Technique de Test) de se concentrer sur l'implémentation de scripts de test intelligents alors que l'Analyste de test conserve la possession du script réel. Dans la plupart des cas, l'Analyste de Test sera responsable de l'exécution des scripts de test une fois que l'automatisation est implémentée et testée.

Une autre approche, appelée dirigée par les mots-clés ou mots-actions , va un cran plus loin en séparant aussi l'action à réaliser sur la donnée fournie du script de test [Buwalda01]. Afin d'accomplir cette séparation avancée, un méta langage de haut niveau, qui est descriptif plutôt que directement exécutable, est créé par des experts du domaine (par ex., Analystes de Test). Chaque instruction de ce langage décrit entièrement ou partiellement un processus métier du domaine pouvant nécessiter du test. Par exemple les mots-clés de processus métier pourraient inclure "Connexion", "Créer Utilisateur » et « Effacer Utilisateur ». Un mot-clé décrit une action de haut niveau qui sera réalisée dans le domaine de l'application. Des actions de plus bas niveau dénotant des interactions avec l'interface logicielle elle-même, telles que "ClickButton" (cliquer sur le bouton) , "SelectFromList" (sélectionner dans la liste), ou "TraverseTree" (parcourir l'arborescence) peuvent aussi être définies et peuvent être utilisées pour tester les fonctions d'IHM qui ne se traduisent pas concrètement en mots-clés de processus métier.

Une fois que les mots-clés et les données à utiliser ont été définis, l' « automatiseur de test » (par ex., l'Analyste Technique de Test) traduit les mots-clés de processus métier et les actions de plus bas niveau en code d'automatisation de test. Les mots-clés et actions, avec les données à utiliser, peuvent être enregistrés dans des feuilles de calcul ou saisis en utilisant des outils spécifiques qui assistent l'automatisation des tests dirigée par les mots-clés. Le framework d'automatisation des tests implémente les mots-clés comme un ensemble d'une ou plusieurs fonctions ou scripts exécutables. Des outils lisent les cas de test écrits avec des mots clés et appellent les fonctions ou scripts de test appropriés qui les implémentent. Les exécutables sont implémentés de façon fortement modulaire pour permettre une association facile avec les mots clés spécifiques. Des compétences en programmation sont nécessaires pour implémenter ces scripts modulaires.

Cette séparation de la connaissance de la logique métier de la programmation réelle requise pour implémenter les scripts d'automatisation des tests apporte l'utilisation la plus efficace des ressources de test. L'Analyste Technique de Test, dans le rôle de l' « automatiseur de test » peut appliquer de façon efficace des compétences en programmation sans avoir à devenir un expert de domaine couvrant différents aspects du métier.

Séparer le code des données qui peuvent changer aide à isoler l'automatisation des changements, ce qui améliore la maintenabilité globale du code et améliore le retour sur investissement de l'automatisation.

Dans toute conception d'automatisation de test, il est important d'anticiper et de traiter les défaillances logicielles. Si une défaillance se produit, l'automatiseur doit déterminer ce que le logiciel devrait faire. La défaillance devrait-elle être enregistrée et le test continuer ? Les tests devraient-ils s'arrêter ? La défaillance peut-elle être traitée avec une action spécifique (comme cliquer sur un bouton dans une boîte de dialogue) ou peut-être en ajoutant un délai dans le test ? Des défaillances logicielles non traitées peuvent corrompre les résultats des tests suivants de même que causer un problème avec le test qui s'exécutait lorsque la défaillance s'est produite.

Il est aussi important de considérer l'état du système au début et à la fin des tests. Il peut être nécessaire d'assurer que le système revienne dans un état prédéfini une fois que l'exécution du test est terminée. Cela permettra à une suite de tests automatisés d'être exécutée de façon répétée sans intervention manuelle pour réinitialiser le système à un état connu. Pour cela, l'automatisation des tests peut devoir, par exemple, effacer les données qu'elle a créées ou modifier le statut des enregistrements dans une base de données. Le framework d'automatisation devrait garantir qu'une fin adéquate a été mise en œuvre à la fin des tests (c.à.d, se déconnecter après la fin des tests).

6.2.2 Modéliser les Processus Métier pour l'Automatisation

Afin de mettre en œuvre une approche dirigée par les mots clés pour l'automatisation des tests, les processus métier à tester doivent être modélisés dans le langage de mots clés de haut niveau. Il est important que le langage soit intuitif pour ses utilisateurs qui seront probablement des Analystes de Test travaillant sur le projet.

Les mots clés sont généralement utilisés pour représenter des interactions métier de haut niveau avec un système. Par exemple, "Annuler_Commande" peut nécessiter la vérification de l'existence de la commande, la vérification des droits d'accès de la personne demandant l'annulation, l'affichage de la commande à annuler et la demande de confirmation de l'annulation. Des séquences de mots clés (par ex., "Connexion", "Sélectionner_Commande", "Annuler_Commande"), et les données de test correspondantes sont utilisées par l'Analyste de Test pour spécifier les cas de test. Voici une table simple d'entrée basée sur les données qui pourrait être utilisée pour tester la capacité du logiciel à ajouter, vider et supprimer des comptes utilisateurs :

Mot-clé	**Utilisateur**	**Mot de passe**	**Résultat**
Ajouter_Utilisateur	Utilisateur1	Pass1	Message utilisateur ajouté
Ajouter_Utilisateur	@Rec34	@Rec35	Message utilisateur ajouté
Réinitialiser_Mot de passe	Utilisateur1	Bienvenue	Message de confirmation de réinitialisation du mot de passe
Effacer_Utilisateur	Utilisateur1		Message Utilisateur/Mot de passé invalide
Ajouter_Utilisateur	Utilisateur3	Pass3	Message utilisateur ajouté
Effacer_Utilisateur	Utilisateur2		Message utilisateur non trouvé

Le script d'automatisation qui utilise ce tableau devrait chercher les valeurs d'entrée devant être utilisées par le script d'automatisation. Par exemple, lorsqu'il arrive à la ligne avec le mot-clé "Effacer_Utilisateur", seul le nom d'utilisateur est requis. Pour ajouter un nouvel utilisateur à la fois le nom d'utilisateur et le mot de passe sont requis. Des valeurs d'entrée peuvent aussi être référencées à partir d'un enregistrement de données comme cela est montré avec le second mot-clé "Ajouter_Utilisateur" où une référence à la donnée est entrée plutôt que la donnée elle-même, ce qui apporte plus de flexibilité à des données qui peuvent changer quand les tests s'exécutent. Cela permet aux techniques dirigées par les données d'être combinées avec les schémas de mots-clés.

Les problèmes à considérer incluent:

- Plus la granularité des mots-clés est fine, plus les scénarios pouvant être couverts seront spécifiques, mais le langage de haut niveau peut devenir plus complexe à maintenir.
- Permettre aux Analystes de Test de spécifier des actions de bas niveau ("CliquerBouton", "SelectionnerDansListe », etc.) rend les tests par mots-clés plus à même de traiter différentes situations. Cependant, comme ces actions sont directement liées à l'interface graphique, cela peut aussi faire que les tests nécessiteront davantage de maintenance lorsque des changements se produiront.

- L'utilisation de mots-clés agrégés peut simplifier le développement mais compliquer la maintenance. Par exemple, il peut y avoir six mots-clés différents qui ensemble créent un enregistrement. Un mot clé appelant vraiment ces six mots-clés consécutivement devrait-il être créé pour simplifier cette action ?
- Peu importe la quantité d'analyse consacrée au langage de mots-clés, il y aura souvent des moments auxquels des mots-clés nouveaux et différents seront nécessaires. Un mot-clé possède deux domaines séparés (c.à.d, la logique métier qui est derrière et la fonctionnalité d'automatisation pour l'exécuter). Par conséquent, un processus doit être créé pour traiter ces deux domaines.

L'automatisation des tests basée sur les mots-clés peut réduire significativement les coûts de maintenance de l'automatisation des tests, mais elle est plus coûteuse, plus difficile à développer, et demande plus de temps pour une conception correcte afin d'obtenir le retour sur investissement attendu.

6.3 Outils de Test Spécifiques

Cette section contient des informations sur les outils qui sont supposés être utilisés par un Analyste Technique de Test, en plus de ce qui est abordé dans les Syllabi Niveau Avancé Analyste de Test [ISTQB_ALTA_SYL] et Niveau Fondation [ISTQB_FL_SYL].

6.3.1 Outils de Génération/Injections de Fautes

Les outils de génération de fautes sont principalement utilisés au niveau du code pour créer de façon systématique dans le code des fautes de types simple ou limité. Ces outils insèrent délibérément des défauts dans l'objet de test avec le but d'évaluer la qualité des suites de test (c.à.d, leur capacité à détecter des défauts).

L'injection de fautes est centrée sur le test de tout mécanisme de traitement de fautes intégré dans l'objet de test en le soumettant à des conditions anormales. Les outils d'injection de fautes fournissent délibérément des entrées incorrectes au logiciel pour assurer qu'il puisse maîtriser la faute.

Ces deux types d'outils sont généralement utilisés par l'Analyste Technique de Test, mais peuvent aussi être utilisés par le développeur lors du test de nouveau code développé.

6.3.2 Outils de Test de Performance

Les outils de test de performance ont deux fonctions principales:

- Génération de charge
- Mesure et analyse de la réponse du système à une charge donnée

La génération de charge est réalisée en implémentant un profil opérationnel prédéfini (voir Section 4.5.4) en tant que script. Le script peut initialement être capturé pour un seul utilisateur (éventuellement en utilisant un outil de capture/rejeu) et est ensuite implémenté pour le profil opérationnel spécifié en utilisant l'outil de test de performance. L'implémentation doit prendre en compte la variation des données par transaction (ou ensembles de transactions).

Les outils de performance génèrent une charge en simulant un grand nombre d'utilisateurs multiples (utilisateurs "virtuels") en suivant leurs profils opérationnels pour générer des volumes particuliers de données en entrée. En comparaison avec des scripts individuels d'automatisation de l'exécution des tests, beaucoup de scripts de test de performance reproduisent les interactions de l'utilisateur avec le système au niveau du protocole de communication et non en simulant les interactions utilisateur via l'interface graphique utilisateur. Cela réduit généralement le nombre de « sessions » séparées

nécessaires durant le test. Certains outils de génération de charge peuvent aussi piloter l'application en utilisant son interface utilisateur pour mesurer de façon plus précise le temps de réponse pendant que le système est soumis à une charge.

Une large variété de mesures sont prises par un outil de test de performance pour permettre l'analyse pendant ou après l'exécution du test. Les métriques prises et rapports fournis typiques incluent :

- Nombre d'utilisateurs simultanés simulés tout au long du test
- Nombre et type des transactions générées par les utilisateurs simultanés et le taux d'arrivée des transactions
- Temps de réponse à des requêtes de transactions particulières faites par les utilisateurs
- Rapports et graphiques de charge par rapport aux temps de réponse
- Rapports sur l'utilisation des ressources (par ex., utilisation dans le temps avec des valeurs minimum et maximum)

Les facteurs significatifs à considérer dans la mise en œuvre des outils de performance incluent :

- La bande passante matérielle et réseau requise pour générer la charge
- La compatibilité de l'outil avec les protocoles de communication utilisés par le système sous test
- La flexibilité de l'outil pour permettre l'implémentation de différents profils opérationnels
- Les fonctionnalités de pilotage, analyse et reporting requises

Les outils de test de performance sont en général acquis plutôt que développés en interne à cause de l'effort demandé pour les développer. Il peut, cependant, être pertinent de développer un outil de performance spécifique si des restrictions techniques empêchent un produit disponible d'être utilisé, ou si le profil de charge ou les fonctionnalités à fournir sont relativement simples.

6.3.3 Outils pour le Test basé Web

Différents outils spécialisés open source et commerciaux sont disponible pour le test web. La liste suivante montre l'objectif de certains des outils classiques de test basé web :

- Les outils de test d'hyperliens sont utilisés pour scanner et vérifier qu'aucun hyperlien cassé ou manquant n'est présent sur un site web
- Des vérificateurs HTML et XML sont des outils qui vérifient la conformité aux standards HTML et XML des pages qui sont créées par un site web
- Des simulateurs de charge testent comment le serveur va réagir quand un grand nombre d'utilisateurs se connectent
- Des outils d'automatisation légère de l'exécution qui fonctionnent avec différents navigateurs
- Des outils pour scanner dans le serveur pour rechercher des fichiers orphelins (ou non liés)
- Des vérificateurs de syntaxe spécifiques HTML
- Des outils de vérification des CSS (Cascading Style Sheet)
- Des outils pour vérifier des violations de standards, par ex., des standards d'accessibilité Section 508 aux Etats-Unis ou M/376 en Europe
- Des outils qui trouvent différents problèmes de sécurité

Une bonne source d'outils de test web est [Web-7].L'organisation qui se trouve derrière ce site web définit le standards pour Internet et fournit une variété d'outils pour vérifier des erreurs par rapport à ces standards.

Certains outils qui incluent un moteur d'architecture web peuvent aussi fournir des informations sur la taille des pages et sur le temps nécessaire pour les télécharger, et sur la présence ou non de la page (par ex., erreur HTTP 404). Cela apporte de l'information utile au développeur, au webmaster et au testeur.

Les Analystes de Test et Analystes Techniques de Tests utilisent ces outils principalement lors du test système.

6.3.4 Outils de Support au Test Basé sur les Modèles

Le test basé sur les modèle (MBT : Model-Based Testing) est une technique par laquelle un modèle formel tel qu'une machine à états finis est utilisé pour décrire le comportement prévu d'exécution dans le temps d'un système sous contrôle logiciel. Les outils MBT commerciaux (voir [Utting 07]) fournissent souvent un moteur qui permet à un utilisateur d'"exécuter" le modèle. Des parcours d'exécution intéressants peuvent être enregistrés et utilisés comme cas de test. D'autres modèles exécutables, tels que les Réseaux de Pétri et les diagrammes d'états permettent aussi le MBT ; Les modèles (et outils) MBT peuvent être utilisés pour générer des grands ensembles de parcours d'exécution distincts.

Les outils MBT peuvent aider à réduire le nombre très important de chemins possibles qui peuvent être générés dans un modèle.

Tester en utilisant ces outils peut apporter une vue différente du logiciel à tester. Cela peut aboutir à la découverte de défauts qui peuvent avoir été manqués par le test fonctionnel.

6.3.5 Outils de Test de Composants et de Build

Bien que les outils de test de composant et de build soient des outils de développeurs, dans bien des cas, ils sont utilisés et maintenus par des Analystes Techniques de Test, en particulier dans un contexte de développement Agile.
Les outils de test de composants sont souvent spécifiques au langage qui est utilisé pour programmer un module. Par exemple, si Java était utilisé comme langage de programmation, JUnit pourrait être utilisé pour automatiser le test unitaire. Beaucoup d'autres langages ont leurs propres outils de test spéciaux, désignés ensemble comme des « framework » xUnit. Un tel framework génère des objets de test pour chaque classe qui est créée, simplifiant ainsi les tâches que le programmeur a besoin de réaliser lors de l'automatisation du test de composant.

Les outils de débogage facilitent le test manuel de composants à un niveau très bas, permettant aux développeurs et aux Analystes Techniques de Tests de changer des valeurs de variables durant l'exécution et de parcourir le code ligne par ligne lors du test. Les outils de débogage sont aussi utilisés pour aider le développeur à isoler et identifier des problèmes dans le code quand une défaillance est rapportée par l'équipe de test.

Les outils d'automatisation de build permettent souvent à un nouveau build d'être automatiquement déclenché à chaque fois qu'un composant est changé. Après la fin du build, d'autres outils exécutent automatiquement les tests de composants. Ce niveau d'automatisation autour du processus de build est en général vu dans un environnement d'intégration continue.

Mis en place correctement, cet ensemble d'outils peut avoir un effet très positif sur la qualité des builds livrés au test. Dans le cas où un changement fait par un programmeur introduit des défauts de régression dans le build, cela provoquera normalement l'échec de certains tests automatisés, déclenchant une recherche immédiate de la cause des défaillances avant que le build ne soit livré dans l'environnement de test.

7. Références

7.1 Standards

Les standards suivants sont mentionnés dans les chapitres indiqués.

- ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems
  Chapitre 5

- IEC-61508
  Chapitre 2

- [ISO25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE)
  Chapitre 4

- [ISO9126] ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality
  Chapitre 4

- [RTCA DO-178B/ED-12B]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12B.1992.
  Chapitre 2

7.2 Documents ISTQB

- [ISTQB_AL_OVIEW] ISTQB Advanced Level Overview, Version 2012
- [ISTQB_ALTA_SYL] ISTQB Advanced Level Test Analyst Syllabus, Version 2012
- [ISTQB_FL_SYL] ISTQB Foundation Level Syllabus, Version 2011
- [ISTQB_GLOSSARY] ISTQB Glossary of Terms used in Software Testing, Version 2.2, 2012

7.3 Ouvrages

[Bass03] Len Bass, Paul Clements, Rick Kazman “Software Architecture in Practice (2nd edition)”, Addison-Wesley 2003] ISBN 0-321-15495-9
[Bath08] Graham Bath, Judy McKay, “The Software Test Engineer’s Handbook”, Rocky Nook, 2008, ISBN 978-1-933952-24-6
[Beizer90] Boris Beizer, "Software Testing Techniques Second Edition", International Thomson Computer Press, 1990, ISBN 1-8503-2880-3
[Beizer95] Boris Beizer, “Black-box Testing”, John Wiley & Sons, 1995, ISBN 0-471-12094-4
[Buwalda01]: Hans Buwalda, “Integrated Test Design and Automation” Addison-Wesley Longman, 2001, ISBN 0-201-73725-6
[Copeland03]: Lee Copeland, “A Practitioner's Guide to Software Test Design”, Artech House, 2003, ISBN 1-58053-791-X
[Gamma94] Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994, ISBN 0-201-63361-2

[Jorgensen07]: Paul C. Jorgensen, “Software Testing, a Craftsman’s Approach third edition”, CRC press, 2007, ISBN-13:978-0-8493-7475-3
[Kaner02]: Cem Kaner, James Bach, Bret Pettichord; “Lessons Learned in Software Testing”; Wiley, 2002, ISBN: 0-471-08112-4
[Koomen06]: Tim Koomen, Leo van der Aalst, Bart Broekman, Michael Vroon, "TMap Next for result-driven testing"; UTN Publishers, 2006, ISBN: 90-72194-79-9
[McCabe76] Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976. PP 308-320
[NIST96] Arthur H. Watson and Thomas J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", NIST Special Publication 500-235, Prepared under NIST Contract 43NANB517266, September 1996.
[Splaine01]: Steven Splaine, Stefan P. Jaskiel, “The Web-Testing Handbook”, STQE Publishing, 2001, ISBN 0-970-43630-0
[Utting 07] Mark Utting, Bruno Legeard, “Practical Model-Based Testing: A Tools Approach”, Morgan-Kaufmann, 2007, ISBN: 978-0-12-372501-1
[Whittaker04]: James Whittaker and Herbert Thompson, “How to Break Software Security”, Pearson / Addison-Wesley, 2004, ISBN 0-321-19433-0
[Wiegers02] Karl Wiegers, "Peer Reviews in Software: A Practical Guide", Addison-Wesley, 2002, ISBN 0-201-73485-0

7.4 Autres Références

Les références suivantes pointent vers des informations disponibles sur Internet. Même si ces références ont été vérifiées au moment de la publication de ce Syllabus Niveau Avancé, l’ISTQB ne peut pas être tenue responsable si les références ne sont plus disponibles.

[Web-1] www.testingstandards.co.uk
[Web-2] http://www.nist.gov NIST National Institute of Standards and Technology,
[Web-3] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx
[Web-4] http://portal.acm.org/citation.cfm?id=308798
[Web-5] http://www.processimpact.com/pr_goodies.shtml
[Web-6]http://www.ifsq.org
[Web-7] http://www.W3C.org

Chapitre 4: [Web-1], [Web-2]
Chapitre 5: [Web-3], [Web-4], [Web-5], [Web-6]
Chapitre 6: [Web-7]

8. Index