



Les tests logiciels : Étude de cas avec Visual Studio 2010

Auteurs :

Stéphane Goudeau (Microsoft France - MTC Paris)
Florent Santin (Access it)
Etienne Margraff (Access it)



Microsoft | Technology Center

Sommaire

Avertissement	4
Introduction	5
1. L'omniprésence du logiciel...et de ses dysfonctionnements.....	5
2. Le coût des dysfonctionnements logiciels	5
3. Gouvernance pour la qualité logicielle	6
4. Le test logiciel	7
5. Les impératifs du test logiciel	7
6. Le rôle de chaque membre de l'équipe de production logicielle.....	7
A. Qui doit tester ?.....	8
B. Quoi tester ?.....	8
C. Quand tester ?	8
L'offre Visual Studio	9
1. Les défis du développement logiciel.....	9
2. Une réponse à ces défis : l'ALM	10
3. L'offre ALM bâtie autour de Microsoft Visual Studio 2010.....	11
4. Les trois éditions Visual Studio 2010.....	12
5. Team Foundation Server 2010.....	12
6. Visual Studio Test Elements 2010	13
7. Visual Studio Team Lab Management 2010.....	13
8. Le portail d'équipe.....	14
9. L'architecture de Team Foundation Server.....	16
L'environnement Visual Studio pour les tests.....	17
1. Outilage	17
2. Automatisation des tests	21
3. Notion de tests impactés.....	21
Les différents tests Visual Studio	23
1. Les différents types de tests.....	23
2. Les tests unitaires.....	24
A. Introduction	24
B. MSTest	34

C. La couverture de code.....	26
D. Les tests unitaires d'interface utilisateur.....	26
3. Les tests de base de données.....	27
4. Les tests génériques.....	29
5. Les tests d'interface utilisateur	30
A. Introduction	30
B. Tests automatisés d'interface utilisateur	31
6. Les tests d'interface Web.....	35
7. Profiling.....	38
8. Les tests de charge.....	38
A. Objectifs.....	38
B. Cible	39
C. Démarche.....	40
D. Préparation du Test	40
E. Infrastructure de Test.....	42
9. Les tests manuels	44
Stratégies de tests avec Visual Studio 2010	46
1. Introduction.....	46
2. Planification des tests.....	47
3. Communication, échanges et traçabilité des tests	50
A. Exécution.....	50
B. Analyse de résultats.....	53
C. Communication avec l'équipe de développement	54
4. Reporting sur les tests	56
5. Gestion d'environnements virtuels de déploiement et de test	58
Synthèse	63
Annexe : Etendre le Framework de test.....	64
Annexe : Tests automatisés d'interface utilisateur :	
Technologies supportées	66
Annexe : Tests automatisés d'interface utilisateur :	
Systèmes supportés	67
Annexe : Références techniques.....	69

Avertissement

Ce document s'adresse aux testeurs, aux architectes, aux concepteurs et dévelopeurs, ainsi qu'aux chefs de projet qui veulent pouvoir contrôler les composants logiciels qu'ils produisent d'une manière plus systématique et mieux outillée que celle généralement pratiquée dans l'industrie aujourd'hui.

Ce document est fourni uniquement à titre indicatif. MICROSOFT N'APPORTE AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, À CE DOCUMENT. Les informations figurant dans ce document, notamment les URL et les références aux sites Internet, peuvent être modifiées sans préavis. Les risques d'utiliser ce document ou ses résultats sont entièrement à la charge de l'utilisateur. Sauf indication contraire, les sociétés, les entreprises, les produits, les noms de domaine, les adresses électroniques, les logos, les personnes, les lieux et les événements utilisés dans ce document sont fictifs. Toute ressemblance avec des entreprises, noms d'entreprise, produits, noms de domaine, adresses électroniques, logos, personnes ou événements réels serait purement fortuite et involontaire.

Stéphane GOUDEAU est Architecte au Microsoft Technology Center

Il conseille les clients et partenaires de Microsoft dans leurs choix d'architecture liés à l'évolution de leur système d'information.

Il possède une connaissance générale ainsi qu'une longue expérience terrain de l'ensemble des produits et solutions de la plateforme Microsoft, acquise lors de ses activités au sein du consulting Microsoft.

Florent SANTIN et Etienne MARGRAFF sont tous les deux consultants formateurs dans la société de services Access it, spécialisée dans les technologies Microsoft.

Ils sont reconnus Microsoft MVP (Most Valuable Professional) sur Visual Studio.

Pour les contacter :
fsantin@access-it.fr
emargraff@access-it.fr

Introduction

1. L'omniprésence du logiciel...et de ses dysfonctionnements

L'importance du logiciel, qu'il s'exécute sur le poste de travail de l'utilisateur, sur son « smartphone », sur sa console de salon, dans sa voiture, sur les serveurs de l'entreprise, ou dans le « Cloud », ne cesse de s'étendre. La garantie de son bon fonctionnement est donc une nécessité absolue pour tout un chacun.

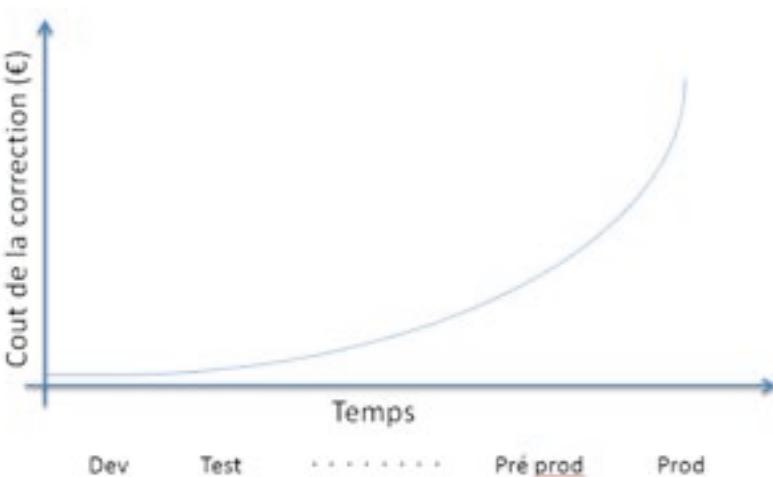
Pourtant, force est de constater qu'au sein des unités de production logicielle, la mise en place systématique de processus permettant de réduire le risque de dysfonctionnement est loin d'être une pratique généralisée.

2. Le coût des dysfonctionnements logiciels

Le développement logiciel est intrinsèquement sujet à erreurs qui tôt ou tard finiront par se manifester. Hors, plus un bogue apparaît tard dans les différentes phases de projet, plus celui-ci va prendre du temps à être corrigé et coûter cher, parfois dans des proportions exponentielles.

- Un bogue trouvé en phase de développement au moment de son apparition va pouvoir être corrigé par le développeur en quelques minutes.
- Un bogue qui apparaît en environnement de test peut être réglé en quelques heures : il va devoir être identifié, remonté au développeur chargé de le corriger qui doit le reproduire, puis livrer une nouvelle version corrigée de l'application en test.
- Un bogue trouvé en environnement de production va suivre le même cheminement qu'en environnement de test, sauf que la livraison de celui-ci sous forme de patch ou de nouvelle version est souvent très contraignante et onéreuse (avec parfois l'accroissement du coût de correction par un facteur de 1 à 100). De plus, il arrive fréquemment que les développeurs de l'application ne soient pas les mêmes personnes que ceux assurant la maintenance (et un bogue peut apparaître très tard, voire trop tard...), ce qui demande un temps de reproduction et d'identification du problème beaucoup plus long.

L'impact de l'introduction précoce d'erreurs avant même l'intervention du développeur est encore plus désastreuse. Ainsi, il est préférable de découvrir une erreur dans la définition d'une exigence fonctionnelle pendant la phase de spécification, car ce type d'erreur affecte beaucoup plus de lignes de code que les erreurs apparaissant dans les phases de conception ou de codage. Les découvrir une fois le logiciel déployé en production entraîne de graves conséquences : nouveaux développements et remise en cause l'existant.



3. Gouvernance pour la qualité logicielle

En outre, les conséquences de la défaillance d'un composant logiciel sont d'autant plus visibles lorsqu'il devient le maillon réutilisable d'une chaîne logicielle du système d'information de l'entreprise. En effet, parmi les briques logicielles du système d'information, certaines sont critiques et nécessitent une totale couverture de tests assortie d'une vraie assurance qualité.

Pour limiter les risques d'erreurs logicielles et les conséquences des dysfonctionnements qui en résultent, il est donc primordial de mettre en place une véritable gouvernance pour la qualité logicielle. Cette gouvernance suppose une amélioration des processus logiciel par le respect de meilleures pratiques et la mise en place d'une gestion de cycle de vie des applications pour les équipes projet à l'aide d'un jeu de plusieurs disciplines, parmi lesquelles les tests logiciels sont l'élément pouvant avoir le meilleur impact sur la qualité logicielle.

Mais qu'est-ce que le test logiciel ?

Le test logiciel

Certaines organisations assimilent test logiciel avec la simple capacité offerte au développeur de déboguer le composant logiciel dont il est l'auteur ou à la mise en place d'un système d'assurance qualité reposant sur des tests fonctionnels ou de performances. Ces éléments sont importants mais ne suffisent pas à garantir la qualité d'un système logiciel.

Tester un logiciel va d'abord consister à lancer les composants logiciels ciblés dans un contexte d'exécution permettant de mesurer les résultats obtenus afin de déterminer si les objectifs de ces composants sont atteints. Mais détecter les erreurs du logiciel et les exposer pour correction n'est pas suffisant car les erreurs ne se limitent pas aux erreurs de codage de logiciels ; elles peuvent se produire à différentes étapes du cycle logiciel (y compris dans les étapes préliminaires de spécification fonctionnelle et de conception).

C'est pourquoi, au-delà de l'examen des résultats de l'exécution du composant ciblé, le test logiciel englobe tout le processus associé, ce qui inclut la définition de la nature et la portée de ces tests en correspondance avec les exigences fonctionnelles (parfois avant même le développement des composants logiciels concernés (« Test Driven Développement » ou « Test First Programming »), ainsi que l'attribution de rôle de responsable des tests au sein de l'organisation de l'équipe de production logicielle...

4. Les impératifs du test logiciel

Une démarche de test logiciel se doit de respecter plusieurs impératifs : la définition des politiques définissant les modalités d'applications de ces tests (planification dans les différentes phases du cycle de vie de l'application, fréquence et conditions d'exécution, rôle de chaque membre de l'équipe), le déploiement de l'infrastructure (matérielle et logicielle) permettant la mise en œuvre de ces politiques ainsi que les processus assurant le contrôle du respect des politiques.

Les environnements matériels sont nécessaires pour l'intégration fonctionnelle et la mise en œuvre de tests de performances pertinents. Un logiciel spécialisé est nécessaire pour la gestion du plan de tests, la création de données fonctionnelles et de tests de performances, et les tests unitaires.

5. Le rôle de chaque membre de l'équipe de production logicielle

Les tests participent à garantir la qualité d'une application en ayant un fort impact sur les bogues. Ainsi une équipe projet appliquant une politique de tests complète pourra :

- En phase de développement, valider en continu la bonne intégration des briques développées par chaque développeur ou sous-équipe en évitant la régression de fonctionnalités déjà implémentées et validées (par des tests).

- En phase de production, limiter l'apparition des bogues en validant au préalable via les tests tous les scénarios utilisateurs possibles, ceux considérés comme bons mais aussi les mauvais (tests aux limites, tests sur des jeux de données complets...).

A. Qui doit tester ?

Chaque membre de l'équipe projet doit être impliqué dans les phases de test. Cependant, chacun doit tester ce qu'il connaît le mieux et avec les outils les plus adaptés : il n'est pas approprié qu'un testeur fonctionnel ait à valider le fonctionnement technique d'objets et qu'un développeur ait à tester des scénarios fonctionnels de navigation. Le contraire est par contre fortement conseillé.

B. Quoi tester ?

De la même façon que chacun des intervenants du projet doit participer aux tests, le maximum des éléments d'une application doit être testé. Plus les tests sont présents, plus les fonctionnalités de l'application seront couvertes.

Cependant, tester deux fois la même fonctionnalité d'une application au travers de deux points d'entrées différents peut être considéré comme une perte de temps.

Par exemple, il est inutile de tester une couche d'accès aux données auto générée. Il est bien plus opportun de se consacrer à la validation de la couche métier l'exploitant. Au contraire, il sera plus utile de tester une base de données plutôt que la couche d'accès l'exploitant si cette première possède beaucoup de procédures stockées représentant une logique métier.

L'interface graphique d'une application console n'aura aucun sens à être testée, par contre l'interface d'une application riche ayant beaucoup d'interactions avec l'utilisation de validateurs de saisies devra être couverte au maximum.

C. Quand tester ?

L'exécution des tests doit être omniprésente au sein du cycle de développement d'une application.

Certains types de tests rapides, comme les tests unitaires ou de base de données doivent être exécutés le plus fréquemment possible : après chaque compilation dans l'environnement de développement, avant chaque archivage des modifications dans le contrôle de code source, après chaque génération de livrable par le service de build. Les tests d'interface (Web ou Windows), un peu plus lent à l'exécution, doivent au minimum être exécutés une fois par jour par le service de build afin de garantir constamment une non régression de fonctionnalités et offrir un niveau de réactivité élevé en cas de bug. Enfin, les tests de charges peuvent être classés dans la catégorie des tests à n'exécuter que ponctuellement, à la demande, sur des périodes données.

L'offre Visual Studio

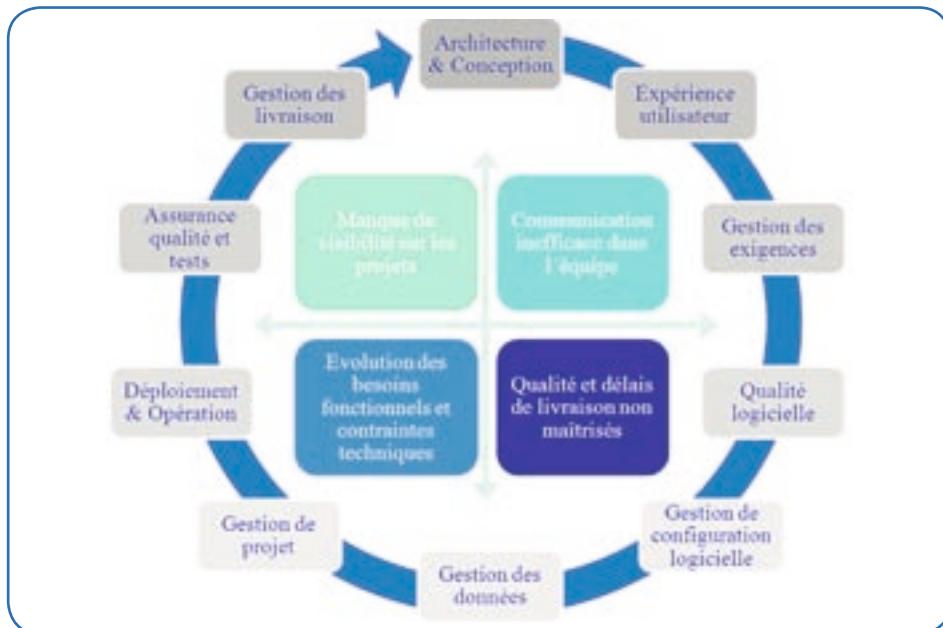
1. Les défis du développement logiciel

Aujourd'hui, très peu de projets logiciels s'achèvent en respectant délais et budget. En outre, les projets menés à terme sont rarement en totale adéquation avec les spécifications initiales.

Pourquoi ? Parce que le développement d'applications doit faire face à de nombreux défis :

- La gestion des exigences métier fluctuant tout au long du projet.
- Les lacunes de communication entre les utilisateurs finaux et l'équipe de développement.
- La difficulté de tester la qualité et la fiabilité des applications de façon adéquate.
- Le suivi du projet afin de pouvoir prendre les bonnes décisions au bon moment et assurer la prédictibilité de la livraison du projet.

Il ne faut pas perdre de vue que le cycle de vie des applications couvre toute la période durant laquelle une organisation engage des moyens (budget, ressources) sur son patrimoine applicatif de l'idée de départ jusqu'à la fin de vie de la solution.

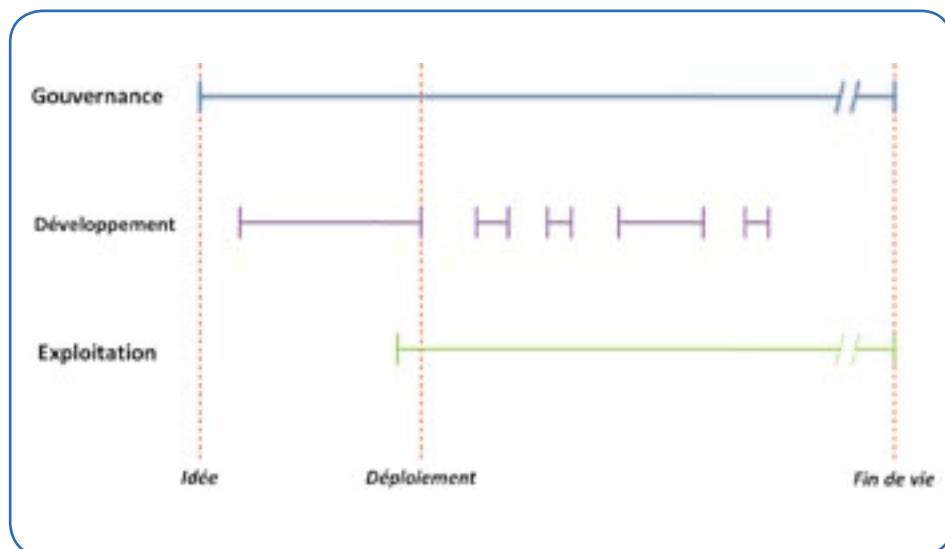


2. Une réponse à ces défis : l'ALM

Comment surmonter ces défis ? La gestion de cycle de vie des applications (ALM : Application LifeCycle Management) pour les équipes projet contribue à réaliser des projets logiciels de façon rentable et prévisible à l'aide d'un jeu de plusieurs disciplines : gestion des exigences, gestion de portefeuille de projet, « Test Driven Development », suivi de projet, fonctions de collaboration, compilations incrémentales, gestion de configuration, méthodes agiles) liées pour influer sur le processus de développement.

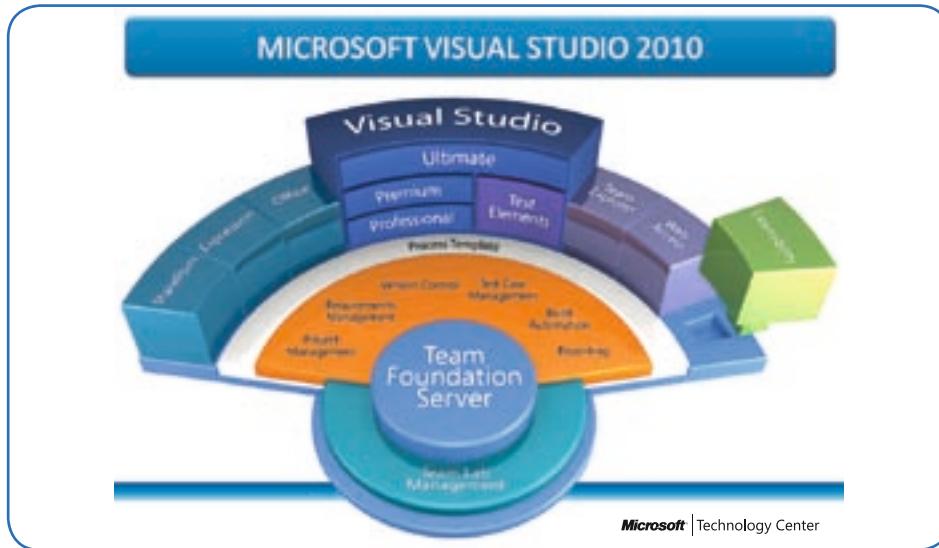
Les trois composantes associées à l'ALM (Application LifeCycle Management) sont :

- La gouvernance : la prise de décision et la gestion de projets.
- Le développement : les processus qui permettent de créer (et mettre à jour) les applications.
- L'exploitation : l'ensemble des tâches requises pour exploiter et maintenir les applications.



3. L'offre ALM bâtie autour Microsoft Visual Studio 2010

Avec Visual Studio 2010, Microsoft propose une solution ALM intégrée comprenant outils, processus et meilleures pratiques pour aider chaque membre de l'équipe projet à améliorer ses compétences et à collaborer plus efficacement.



L'offre ALM Visual Studio propose une gamme de multiples catégories de produits dont le rôle est de rendre des services pour organiser les équipes de développement et centraliser toutes les données liées à un projet. Celles-ci sont accessibles par chacun au travers de l'outil le plus adapté tel que Visual Studio pour les développeurs, MS Project pour le chef de projet, Microsoft Test and Lab Manager pour les testeurs, une interface Web pour les fonctionnels...

Visual Studio couvre tout le cycle de développement des applications (modélisation graphique des applications, des bases de données et des processus, configuration de code source, industrialisation, tests et suivi de projet).

Visual Studio permet d'améliorer la collaboration et la communication au sein de l'équipe projet avec la gestion de l'ensemble des données des projets dans un module de gestion de configuration fiable, et de piloter les projets beaucoup plus efficacement grâce aux nombreux rapports et à l'intégration avec MS Excel et MS Project.

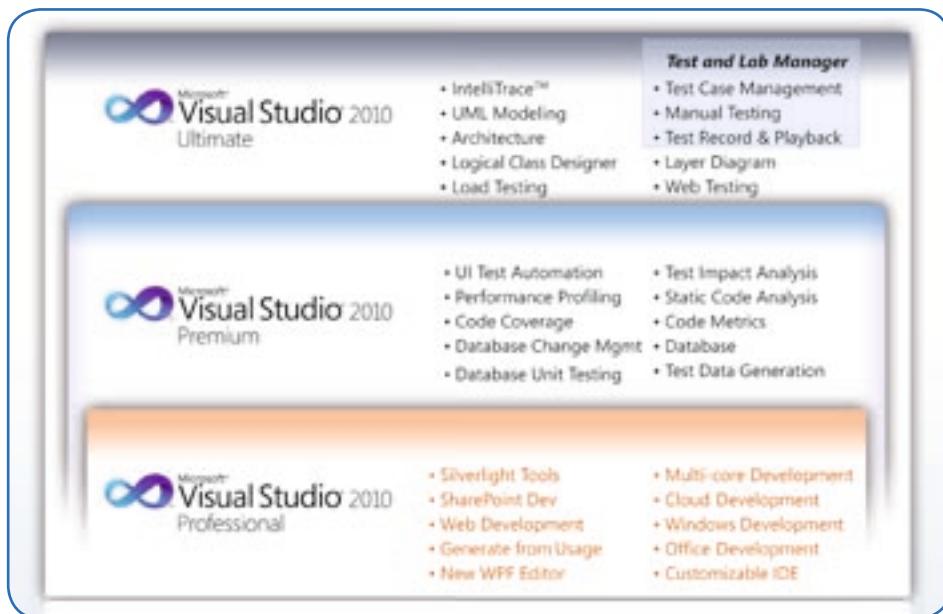
La version 2010 de Visual Studio offre de nombreuses nouvelles fonctions :

- Diagrammes de séquences pour application SOA, Construction d'applications WCF, Création et visualisation des interactions entre classes...

- Ensemble de règles d'analyse statique, Débugger historique (IntelliTrace™), Visualisation des impacts sur les tests des modifications de code, Débugger «isolé» pour traquer les anomalies sur les postes clients...
- Planification de campagnes de tests, Exécution des cas de test manuels...
- Hiérarchie de Work Items dans Excel, Création de répertoire pour gérer ses requêtes, Rollback de checkin, Destruction, suppression et renommage des Work Items...

4. Les trois éditions Visual Studio 2010

Les outils clients sont déclinés selon trois éditions, chacune pouvant être utilisée indépendamment du serveur mais collaborent très bien pour stocker les informations et fluidifier la communication :



5. Team Foundation Server 2010

Team Foundation Server est la partie serveur de l'offre ALM Visual Studio : automatisation des définitions de builds ou des tests, gestion de configuration/des versions, gestion du changement, portail d'équipe, reporting sur l'état d'avancement d'un projet... Toutes ces fonctionnalités sont exposées sous forme de services Web, et donc accessibles depuis les éditions Visual Studio Team

Edition, mais aussi depuis des applications comme Excel ou Project ou des applications Web. Il est également possible de se connecter à Team Foundation à partir de Team Explorer, un client riche prévu à cet effet.

Ce référentiel offre la centralisation des éléments suivants :

- Code source (gestion de source, développement parallèle).
- Eléments de travail (bogues, tâche, demandes d'évolution ou tout type de fiches).
- Données organisationnelles (documents, calendrier partagé, annonces, wiki, blogs...).
- Build (génération et gestion des livrables).
- Rapport (état de santé du projet, rapports de qualité, état d'avancement...).

6. Visual Studio Test Elements 2010

Cet environnement intègre Microsoft Test and Lab Manager qui permet de définir une stratégie de tests structurée selon un plan de test associé au projet Team System. Elle offre également les agents «Lab Management», permettant à Team Foundation Server de piloter une machine virtuelle pour y réaliser des tests manuels ou automatisés.



7. Visual Studio Team Lab Management 2010

Visual Studio Team Lab Management 2010 permet de mettre en place des laboratoires de tests. Cette fonctionnalité offre la possibilité de créer des modèles d'environnements qui pourront être instanciés à la demande, ou de manière automatique pendant une génération de build serveur.



8. Le portail d'équipe

Un portail (figure suivante), fondé sur Windows Sharepoint, complète l'intégration des différents outils collaboratifs (Visual Studio, Excel, Project,...) en offrant l'affichage des rapports d'avancement générés à partir des « workitems » publiés.

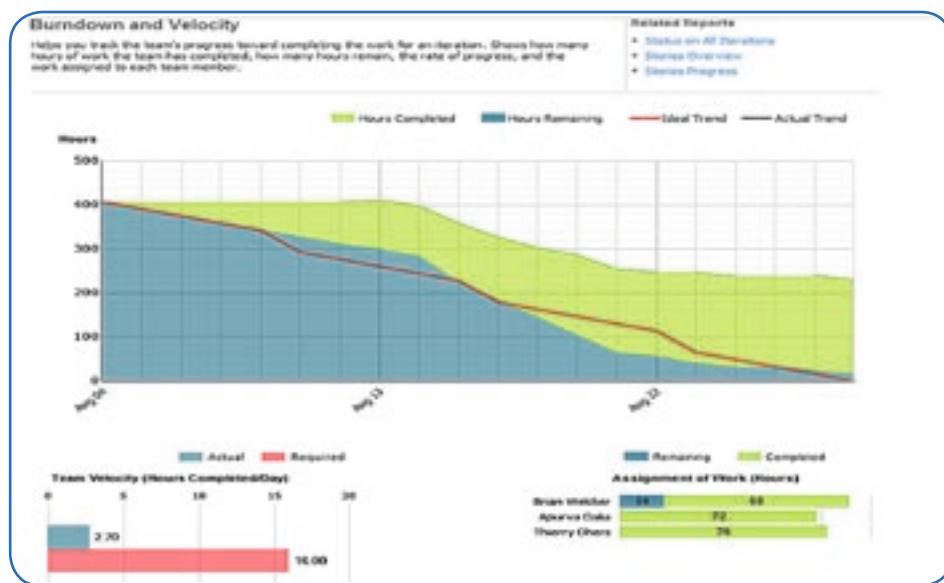
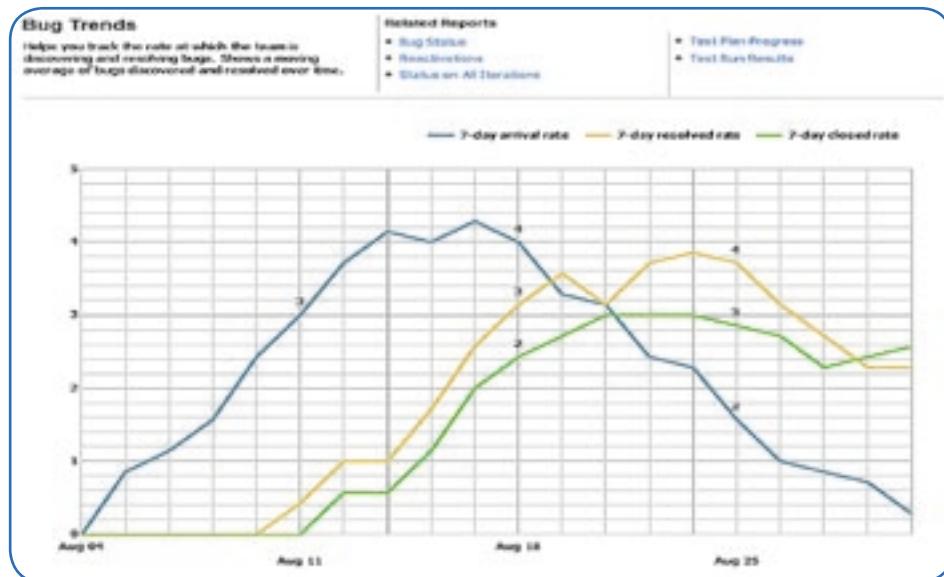
The screenshot shows the Microsoft Visual Studio Team System Web Access interface. On the left, there's a navigation menu with sections like Favorites, My Queries, Team Queries, Reports, and My Team. The main content area has two main sections: 'Critical Alerts (None)' and 'My Work'. The 'Critical Alerts' section displays quality metrics such as Requirements Test Coverage (70% active requirements covered by test cases), Requirements Pass Rate (100% P1 and 85% P2 Requirements Passing), Active and Severity of Bug Count (<10 P1 and <20 P2 bugs), Test Execution Rate (=> 1 execution per active test case), Test Pass Rate (70% pass rate on last build), and Automation Rate (20% test cases automated). The 'My Work' section shows a list of tasks with columns for Title, Reason, and Status, along with a 'Reason' dropdown menu. To the right, there's a 'Team Progress' chart showing a downward trend over time.

Ce portail peut être personnalisé et reflète le choix de la méthodologie de développement associée au projet.

The screenshot shows the 'Requirements Progress and Quality' section of the Microsoft Visual Studio Team System Web Access portal. It includes an overview of system quality for the given build, a breakdown of requirements progress, and a detailed list of requirements with their status and progress. The 'Overview' section shows the total number of active requirements (30), total number of tests run (400), and a summary of test results: Tests Passed (210, 52%), Tests Failed (120, 30%), Tests not executed (48, 12%), and Total Bugs (50). The 'Breakdown per Requirement' table lists 20 requirements with their IDs, descriptions, and progress details. For example, requirement #2001 is 'Application look and feel should reflect new corporate branding' with a status of 'In Progress' and a progress of 90/90. Requirement #2008 is 'Application enables secure customer account access' with a status of 'Not Started' and a progress of 0/10.

ID	Title	Task Progress	Bugs	Tests Passed/Failed/Not Run	Build
2001	Application look and feel should reflect new corporate branding	90/90	3	11/10	IPS_31200.00
2009	Application enables secure customer account access	20/30	1	9/10	IPS_31200.00
2008	Application shall provide online services between GlobalBank...	30/30	1	8/10	IPS_31200.00
2001	Application shall provide linking to external banking services	30/30	3	4/10	IPS_31200.00
2002	Application shall support online C2 purchase	30/30	10/10	0/0	IPS_31200.00
2003	Application shall support online purchase for treasury funds	30/30	1	10/10	IPS_31200.00
2004	Application shall provide access to special investment funds	20/30	9	4/10	IPS_31200.00
2005	Application shall provide specialized research reports	20/30	2	10/10	IPS_31200.00
2006	Application shall provide check monitor service	20/30	0/0	10/10	IPS_31200.00
2007	Application shall allow user to apply and get approval for a loan	0/30	3	10/10	IPS_31200.00
2008	Application will analyze reports and detect potentially suspicious...	0/30	12/10	0/0	IPS_31200.00

Le portail permet également d'afficher les multiples types de rapports : suivi des bugs, vélocité du projet...



9. L'architecture de Team Foundation Server

La figure suivante illustre l'architecture de Team Foundation Server :



L'environnement Visual Studio pour les tests

1. Outilage

Depuis sa version 2005, l'environnement de développement Visual Studio dispose d'une édition enrichie dédiée à l'écriture et l'exécution de tests. Intitulée Visual Studio Team Test, celle-ci est enrichie au fil des versions par des nouveaux types de tests. Dans l'édition 2010 de Visual Studio, la dénomination « Team » disparaît pour laisser place à 3 nouvelles éditions. Les outils concernant les tests se retrouvent désormais dans les éditions Professional, Premium et Ultimate pour les aspects test web et test de charge.

Visual Studio 2010 met à disposition l'outillage complet permettant à chacun de s'impliquer dans les phases de test :

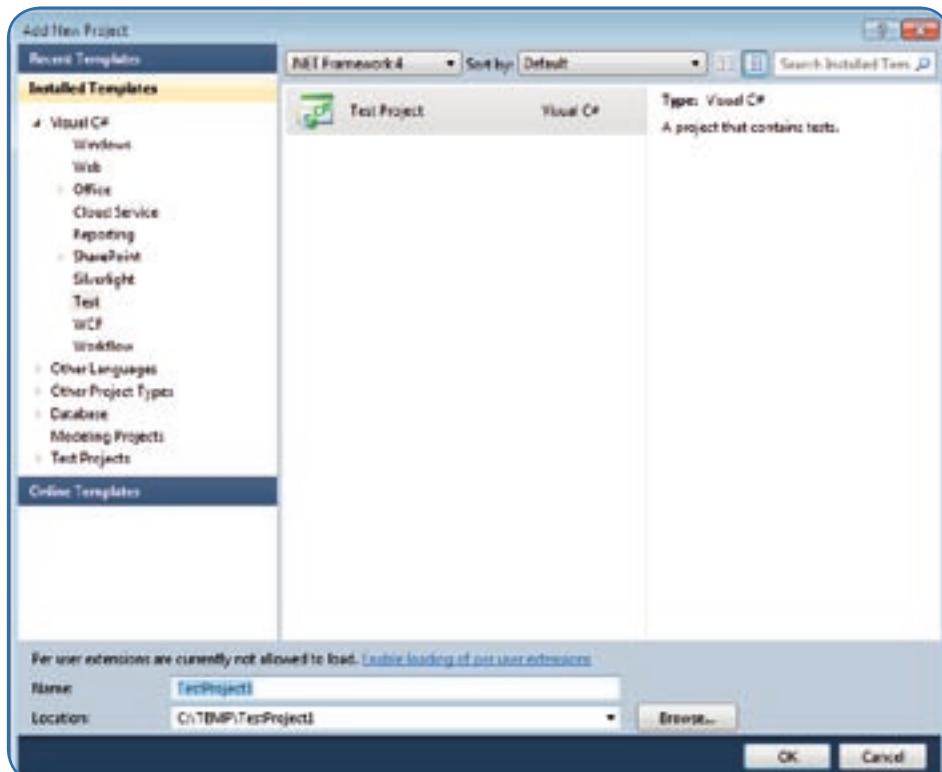
- Le développeur au travers de tests techniques (test unitaire, test générique).
- Le développeur senior ou l'architecte grâce à des tests de validation de qualité et d'anticipation des problèmes (test de charge, test WCF, test codé, test d'interface).
- Le spécialiste de bases de données en utilisant des tests écrits en langage SQL pour tester des cas SQL (test de base de données).
- Le fonctionnel ou le testeur pour valider des scénarios de navigation et des cas de test (Test Web, manuel).

Pour démocratiser leur utilisation, les tests unitaires sont depuis 2008 mis à disposition des développeurs dans la version Professionnel de Visual Studio, sans coût supplémentaire.

La création et la gestion des tests s'effectuent directement depuis l'environnement de développement.

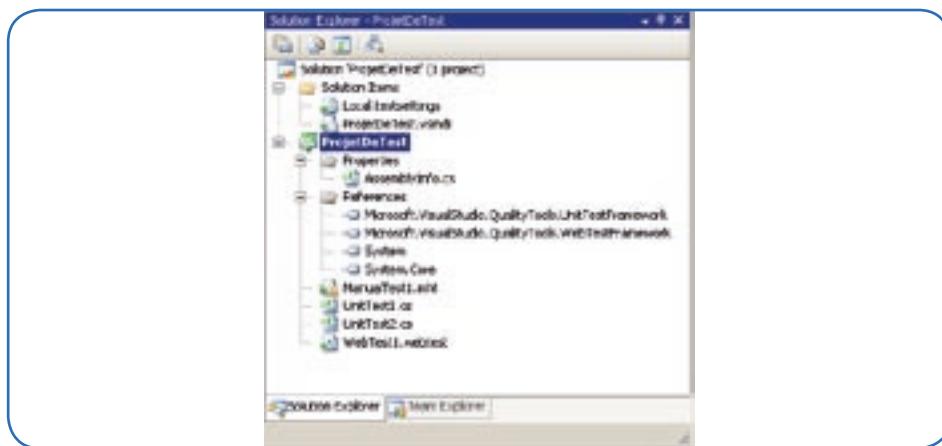
Les tests sont physiquement organisés par projets de tests, projets .Net pouvant faire partie d'une solution .Net classique.

La création d'un projet de test peut donc être réalisée par n'importe quel membre de l'équipe technique, au travers de l'assistant de création de projet de Visual Studio.



Création d'un projet de test

Une fois créé, le projet de test ressemble à tout autre projet de développement .Net : il possède des références vers le Framework de test de Microsoft et peut contenir des types de fichiers spécifiques représentant des tests.

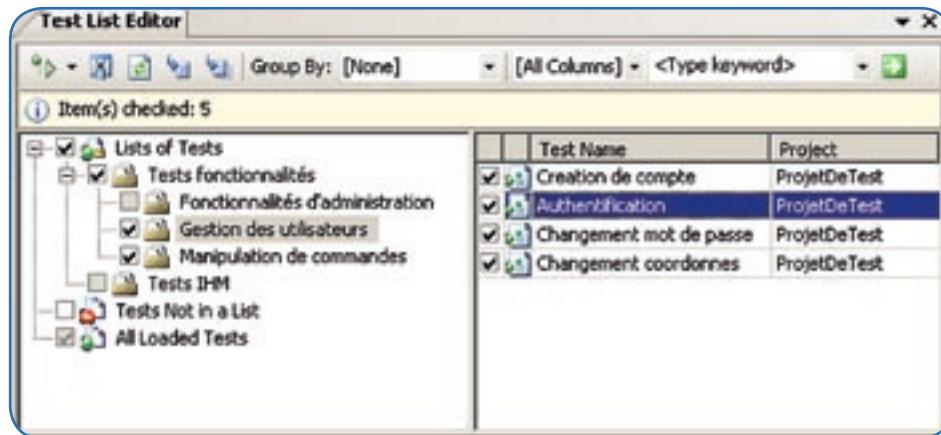


Contenu d'un projet de test

Les tests peuvent être organisés logiquement par groupe pour être exécutés ensemble. Toute cette gestion s'effectue au travers de la fenêtre « Test List Editor ». Celle-ci offre la possibilité de :

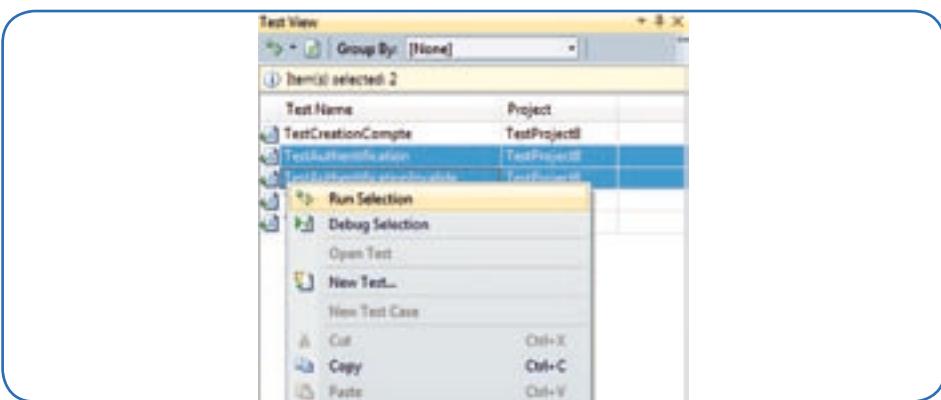
- Créer une arborescence d'organisation des tests.
- Classer et regrouper les tests à plusieurs niveaux de l'arborescence.
- Demander l'exécution de l'intégralité des tests disponibles dans un ou plusieurs nœuds d'une arborescence.

Tous les types de tests peuvent être pris en compte dans cette interface.



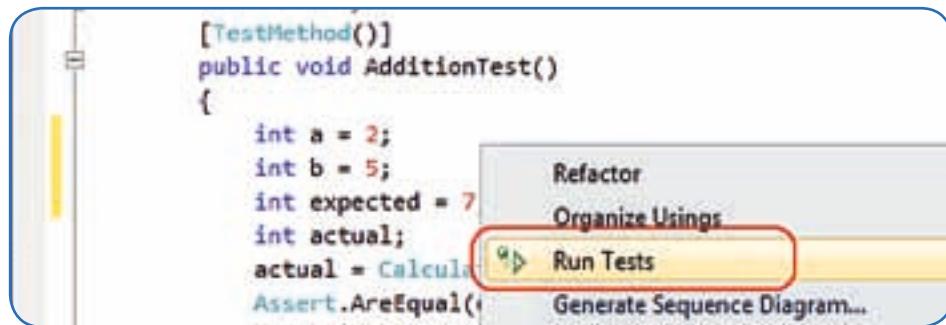
Fenêtre d'organisation des tests dans Visual Studio

L'éditeur de listes de tests offre la possibilité de démarrer l'exécution d'un ensemble de test. Pour être plus rapide et productif dans l'exécution de ceux-ci, Visual Studio dispose d'une autre fenêtre intitulée « Test View ». Celle-ci liste l'ensemble des tests présents pour le projet et propose de les exécuter un à un ou par groupe.



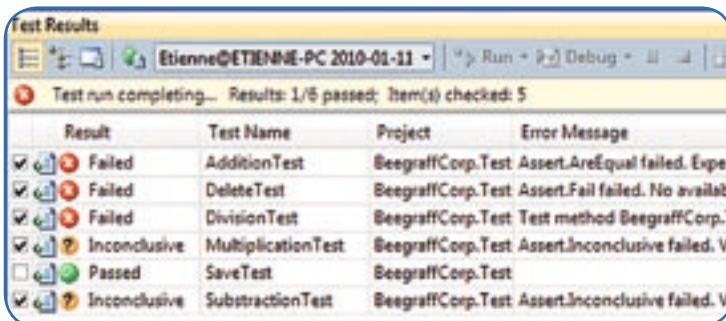
Fenêtre d'exécution des tests

Enfin, il est également possible d'exécuter un test ou un ensemble de tests directement depuis son code grâce au menu contextuel de Visual Studio ou à des raccourcis claviers. Ceci a pour but d'accélérer les phases écriture de test > exécution. Ces facilités sont très importantes lors de la mise en place de pratiques ou les tests unitaires sont cruciaux, tels que le « Test Driven Development ».



Quelques soit le type de test ou le contexte dans lequel il est exécuté, lors de la demande manuelle d'exécution de test, le résultat apparaît toujours dans la fenêtre « Test Results ».

Le résultat d'un test est souvent binaire : réussi ou échoué, représenté par la couleur verte ou rouge. Les tests unitaires possèdent également un état jaune « inconclusive » intermédiaire. Ce résultat apparaît lors de la génération automatique de squelette de test unitaire, si les résultats attendus de ceux-ci n'ont pas été renseignés. En plus d'un état de réussite ou d'échec, la fenêtre de résultat propose en résultat un rapport complet.



Visual Studio 2010 propose les outils de test permettant d'être plus productif tout au long du cycle de vie du développement et de suivre plus facilement la progression. Ces outils de test sont intégrés avec Team Foundation Server, ce qui permet de définir des tests basés directement sur les projets Team System.

2. Automatisation des tests

Dès qu'il y a plus d'une personne qui modifie le code source d'une application, il est vital de valider fréquemment l'intégration du travail de chacun avec celui des autres. Pour cela, une validation manuelle locale par le développeur lui-même ne suffit pas. L'idéal est d'effectuer un ensemble de vérifications sur un environnement dont on maîtrise la configuration et indépendant des postes de l'équipe de développement.

Le processus qui permet d'effectuer cette validation est couramment appelé « Chaîne d'intégration continue ».

Ce processus comprend en général les étapes suivantes :

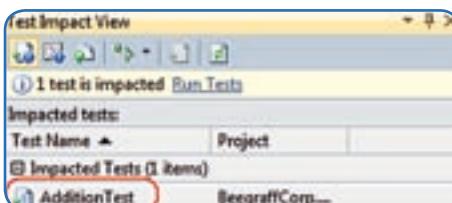
- Compilation de la dernière version disponible de l'application.
- Exécution des tests unitaires.
- Déploiement de l'application sur un environnement de test.
- Exécution de tests supplémentaires (d'interfaces graphiques automatisés, par exemple).

Généralement, ce processus, qui exploite le service de Build de Team Foundation Server, est déclenché le plus fréquemment possible : à chaque mise à jour du référentiel de code source. Il peut cependant être effectué régulièrement sans pour autant le faire à chaque archivage, comme par exemple « Tous les soirs à 2h00 ».

3. Notion de tests impactés

Lors d'une phase de développement, les tests unitaires sont vitaux pour s'assurer que les modifications réalisées sur le code ne donnent pas lieu à un effet de bord sur d'autres éléments de l'application. De ce fait, il est nécessaire d'exécuter les tests le plus souvent possible. Lorsqu'un projet contient un nombre important de tests unitaires, leur exécution peut prendre un certain temps. Pour éviter d'avoir à jouer la totalité de ces tests, Visual Studio 2010 offre une nouvelle fonctionnalité nommée : Analyse de tests impactés.

Cette fonctionnalité permet d'enregistrer à chaque test, en exploitant les informations de couverture de code, la liste des blocs de code utilisés lors de l'exécution du test. Grâce à ces informations Visual Studio est capable de trouver la liste des tests impactés par les modifications que l'on vient de réaliser à chaque compilation de l'application.



Cette fonctionnalité nommée : « Tests impactés » peut être exploitée localement dans Visual Studio, mais également dans le processus de Build ou encore lors de l'exécution d'un test manuel via l'outil Microsoft Test and Lab Manager.

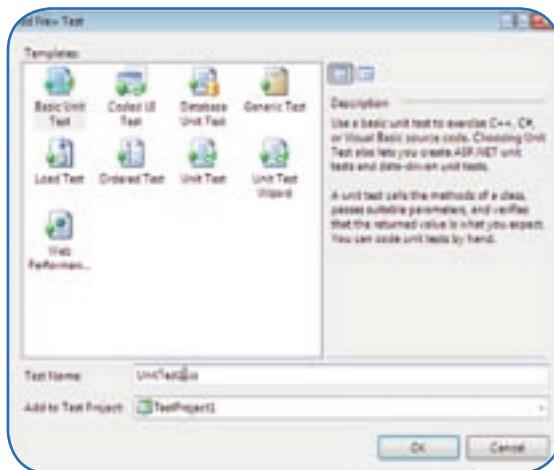
Le plus fréquemment, celle-ci sera exploitée pour les tests unitaires.

Les différents tests Visual Studio

1. Les différents types de tests

À l'aide de Microsoft Visual Studio 2010, vous pouvez créer différents types de tests automatisés ou manuels. Parmi les tests automatisés, on trouve notamment :

- Tests unitaires: le plus classique, qui permet de tester du code en .Net, avec un squelette généré.
- Tests unitaires de base de données : équivalent du test unitaire .Net mais en T-SQL.
- Tests ordonnés : permet d'exécuter un ensemble de tests dans un ordre séquentiel, le test réussit si l'ensemble des sous-tests réussissent.
- Tests Web : capture et rejoue un scénario de navigation Web.
- Tests de charge : permet de rejouer en boucle un ou plusieurs tests en simulant de la charge pour vérifier les performances de votre application sous différents niveaux de stress.
- Tests génériques : test pouvant exécuter n'importe quelle application externe.
- Tests d'interface utilisateur.
- Tests personnalisés : il est possible de créer vos propres types de tests pour simplifier certaines tâches répétitives (par exemple, la génération d'un proxy lorsque l'on teste des services WCF).



Plusieurs types de tests additionnels sont également disponibles en téléchargement sur CodePlex.

2. Les tests unitaires

A. Introduction

Les développeurs mettent en œuvre des tests unitaires afin de vérifier si un module particulier répond aux spécifications fonctionnelles et si les éléments de code de ce module fonctionnent correctement indépendamment du reste de l'application et en toutes circonstances.

L'usage de tests unitaires est fréquent dans les applications critiques et se généralise aujourd'hui à tout type d'applications. En renouvelant leur exécution après la modification du ou des modules concernés, ils permettent de valider la non régression du système.

Certaines méthodologies de développement préconisent leur usage systématique lors de l'implémentation du module, voire en amont de cette implémentation : développement piloté par les tests ou « Test Driven Development ». C'est notamment le cas de la méthodologie XP (« Extreme Programming »).

B. MSTest

MSTest est le Framework de tests unitaires proposé par Microsoft depuis 2005. Les tests unitaires sont présents dans Visual Studio Team System Development Edition et Test Edition depuis la version 2005. En version 2008 et 2010, ils sont présents dans la gamme professionnelle, leur utilisation devenant une commodité. L'intérêt du Framework de test Microsoft est sa très forte intégration dans Visual Studio et dans Team Foundation Server (exécution automatique, stockage et comparaison des résultats sur une échelle de temps...).

Syntaxiquement, son utilisation est très proche des autres Framework de tests unitaires répandus sur le marché tels que NUnit et MbUnit : utilisation d'attributs sur les classes et les méthodes de tests d'assertions pour les validations.

Un test unitaire est composé de code .Net qui est chargé d'utiliser et de valider (« Assert ») du code .Net. A ce titre, il peut être utilisé pour tester tout et n'importe quoi (un accès à une base SQL, OLAP, des services Web...) car il est ouvert aux capacités du Framework .Net qui sont extrêmement étendues. Les tests unitaires peuvent donc être utilisés pour effectuer des tests de charge d'éléments serveur en évitant l'intermédiaire d'une IHM (ex : stresser une base de données Oracle).

```
[TestClass]
public void TestAdd()
{
    int a = 10;
    int b = 20;
    int expected = 30;

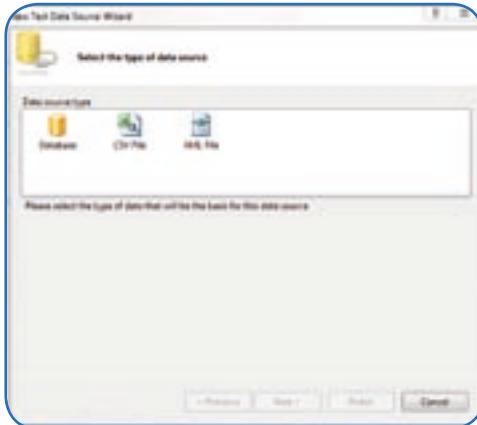
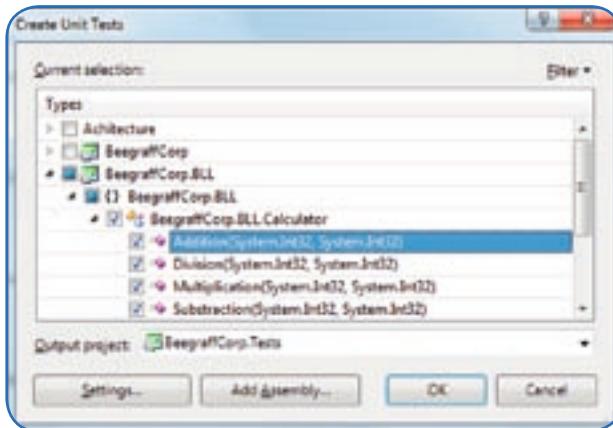
    Calculateur c = new Calculateur();
    int result = c.Add(a, b);

    Assert.AreEqual(expected, result, "L'addition ne fonctionne pas");
}
```

Un test unitaire peut être lié à un jeu de données externe (base de données, fichier CSV, XML) pour être joué à chaque fois avec un ensemble de données en entrées consécutives ou aléatoires.

Assez critiqué dans sa première version pour ses problèmes de performance d'exécution, MSTest rivalise aujourd'hui sans aucun problème avec les Frameworks existants et l'outillage Open Source et dispose de plusieurs avantages pour les développements .Net :

- Une intégration parfaite dans Visual Studio pour l'exécution des tests, la consultation des résultats, ou l'intégration des tests aux builds automatiques.
- Un générateur de squelette de test (instanciation de l'objet à créer, isolation des paramètres, assertion...) à partir de n'importe quelle méthode pour réduire le temps d'écrire des tests unitaires.



C. La couverture de code

Les tests unitaires sont souvent liés à la vérification de la couverture de code, qui permet de s'assurer que les critères de tests permettent de garantir l'exécution d'une partie significative des éléments de code à tester. A quoi bon obtenir 100 % de succès sur les tests si ceux-ci ne couvrent que 20% du code utilisable ? Comme pour les tests unitaires, Microsoft dispose de son propre outillage de couverture. Là également, l'intégration dans l'environnement joue un rôle très important pour faciliter l'adoption et simplifier l'utilisation. Il est ainsi possible d'explorer le code source dans lequel les zones non couvertes par les tests unitaires sont surlignées en rouge.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
# Etienne@ETIENNE-PC 2013-05-13 22:59:36	510	88.29 %	88	13.71 %
# BeagraftCorp.BLL.dll	134	76.14 %	42	23.88 %
# BeagraftCorp.BLL	134	76.14 %	42	23.88 %
# Calculator	1	12.50 %	7	87.50 %
# DepartmentsManager	73	100.00 %	0	0.00 %
# DepartmentsManager->c_Displ...	3	100.00 %	0	0.00 %
# DepartmentsManager->c_Displ...	3	100.00 %	0	0.00 %
# EmployeesManager	48	57.83 %	35	42.17 %

D. Les tests unitaires d'interface utilisateur

Ce type de test repose sur la mise en œuvre de pattern adapté au développement de la couche de présentation. Prenons le cas de WPF (Windows Presentation Foundation). Depuis sa sortie en 2007 avec le .Net Framework 3.0, différents patterns ont été explorés pour faciliter l'utilisation de cette technologie dans un contexte d'applications d'entreprises. Le dernier en date - qui est de loin le plus abouti - est le pattern M-V-VM (Model - View - ViewModel).

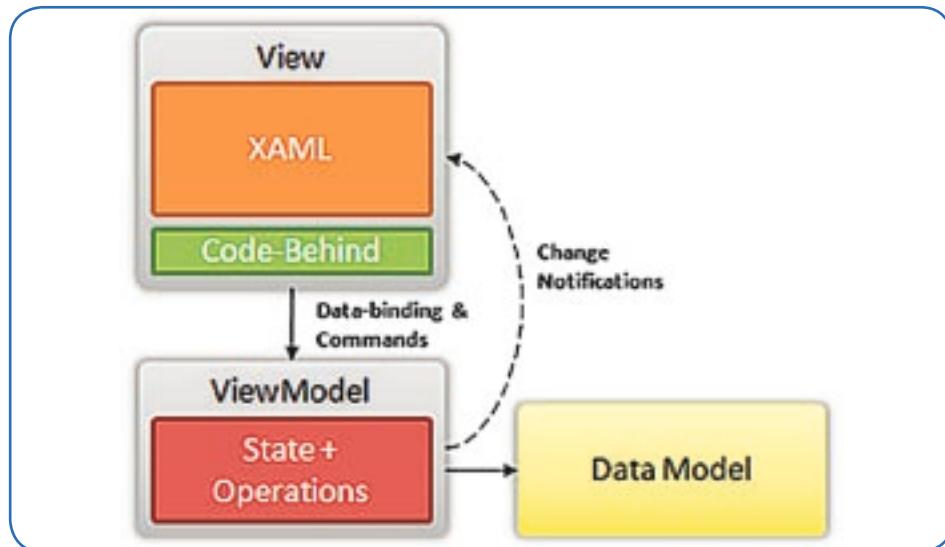
C'est un Pattern dérivé du pattern MVC indépendant de la technologie. Il permet de séparer clairement le métier de l'interface. Il facilite les tests unitaires sur les applications WPF (en aidant à remplacer le test manuel simulant l'interaction utilisateur). Dans le cas des applications Winform, on utilisera plutôt le pattern MVP.

Dans la logique de décomposition proposé par le pattern M-V-VM, le « Model » (ou « DataModel ») est responsable des données métiers, la « View » représente la partie visuelle de l'application tandis que le « ViewModel » expose les données et les comportements (notamment les commandes) pour la « View ».

Ce « ViewModel » est spécifique pour chaque vue mais n'hérite d'aucune classe WPF. Le « ViewModel » est une classe qui implémente le pattern « Observable » : elle implémente l'interface « INotifyPropertyChanged ». De même, les propriétés de la classe « ViewModel » doivent implémenter « INotifyPropertyChanged » (ou « DependencyObject ») ou « ObservableCollection » suivant qu'il s'agisse d'objet ou de collection, afin de prévenir la vue lors de tout changement des données. Ainsi la « View » peut utiliser le « ViewModel » dans son « DataContext » pour le

« binding » afin de transformer les données en éléments visuels avec les « Data Templates ». Le View Model quant à lui ne fait aucune supposition sur la façon dont va s'opérer le binding.

Lorsque ce pattern est mis en œuvre avec WPF, la View ne contient quasiment pas de « code behind » à part les animations et la gestion de « layout ». Le code réside dans le ViewModel qui est facile à tester unitairement, d'où l'intérêt de cette approche dans le cadre de test unitaire d'interface utilisateur.



Source : <http://www.nikhilk.net/Silverlight-ViewModel-Pattern.aspx>

3. Les tests de base de données

Les tests de base de données ont fait leur apparition avec la version Database Professionals de Visual Studio Team System. Leur objectif est de simplifier l'écriture de tests ciblant directement les bases de données SQL Server.

Pour répondre à ce besoin de simplicité, l'écriture des tests ne s'effectue pas en .Net mais directement en T-SQL (Transact SQL). Ceux-ci sont ensuite traduits de manière invisible en code .Net par l'environnement de développement. Le test SQL devient ainsi une surcouche des tests unitaires et permet de générer le même code .Net de validation. Ceci permet d'étendre le produit à des administrateurs de base de données n'ayant pas forcément de compétences en développement .Net.

Un test SQL dispose de validateurs spécifiques qui permettent, une fois la requête ou procédure stockée utilisée, de vérifier un comportement : nombre de résultats, temps d'exécution, validateur personnalisé... Un test SQL peut être utilisé dans un test de charge (test aux limites d'une base SQL Server).

The screenshot shows a 'Database Test' window with the following content:

```

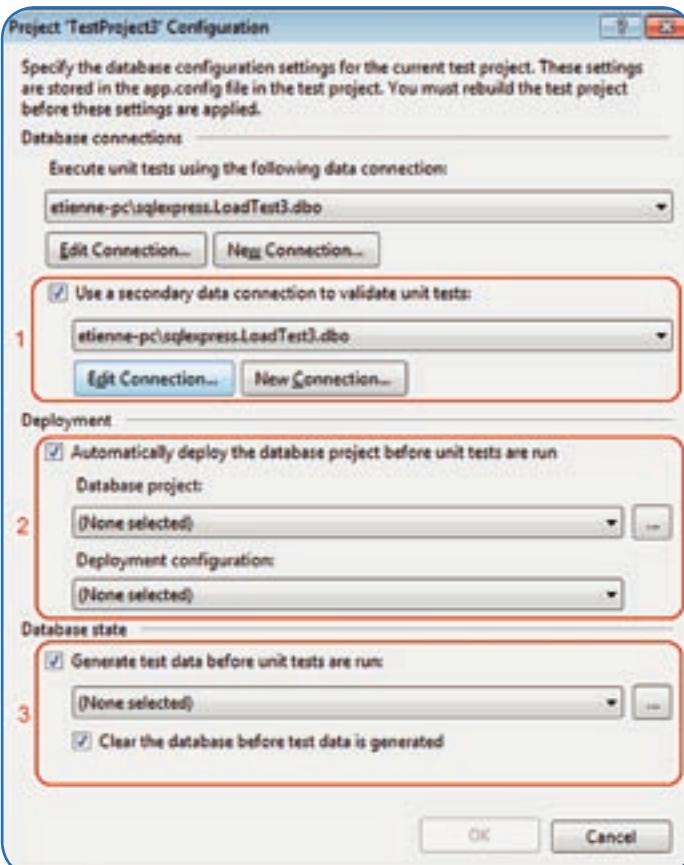
    Exec AddRandomAuteurs;
    Select count(*) from t_auteurs;
  
```

Test Conditions:

Name	Type	Value	Enabled
scalarValueCondition1	Scalar Value	The condition fails if the value in ResultSet2 Row 1 C...	True

Les validateurs sont positionnés par configuration de manière graphique dans l'interface de test.

Lors de l'ajout d'un test unitaire de base de données un assistant nous permet de configurer un certain nombre d'informations :



1. Permet de choisir une source de données contenant les éléments de validation des tests.
2. Permet d'exploiter un projet Visual Studio 2010 de type « Base de données ». Ce type de projet offre la possibilité de définir des scripts de création d'une base de données. Celle-ci sera alors déployée automatiquement pour l'exécution des tests.
3. Permet d'exploiter un fichier de génération automatisé de données de tests. Ce type de fichier peut être créé à partir d'un projet de type « Base de données ».

4. Les tests génériques

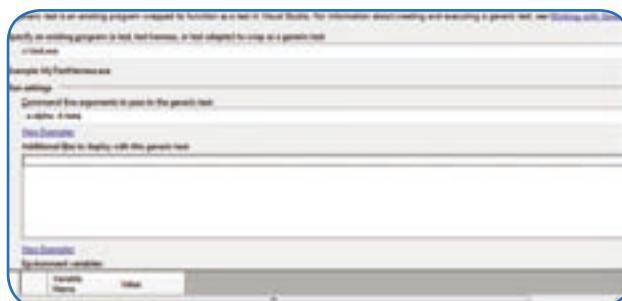
Comme son nom l'indique, un test générique est un test capable d'exécuter n'importe quelle action. Concrètement, un test générique a pour rôle d'exécuter une application tierce, éventuellement avec des paramètres spécifiques afin d'extraire les résultats de sortie et de les intégrer aux côtés des autres résultats de test.

Voici plusieurs exemples pour lesquels l'utilisation de tests génériques s'avère fort utile :

- Utilisation d'un Framework de test tiers (NUnit, MBUnit, JUnit) et intégration des résultats dans l'environnement de développement / rapports de builds aux côtés des autres tests.
- Initialisation de l'application testée afin de la rendre disponible pour la campagne de tests (voir cas des tests ordonnés).
- Exécution d'une application console tierce ou d'un script Windows en charge de l'initialisation de la plateforme de test ou de son nettoyage.

Le test générique permet d'exécuter n'importe quel type d'application tierce n'ayant pas d'interface graphique : .EXE, .BAT, script PowerShell...

Il peut être lié pour initialiser/finaliser une séquence de tests ordonnés, valider un batch ou encore intégrer dans Visual Studio des Frameworks de test tiers (exemple : exécution de NUnit avec un test générique et extraction/transformation des résultats pour les intégrer dans Team Foundation Server).



5. Les tests d'interface utilisateur

A. Introduction

Un grand nombre de types de tests pouvaient déjà être automatisés. La version 2010 de Visual Studio offre un nouveau type de tests nommés « Coded UI tests ». Il s'agit de tests automatisés d'interfaces graphiques permettant de simuler les actions du clavier et de la souris enregistrés lors d'une exécution manuelle d'un scénario d'utilisation d'une application.

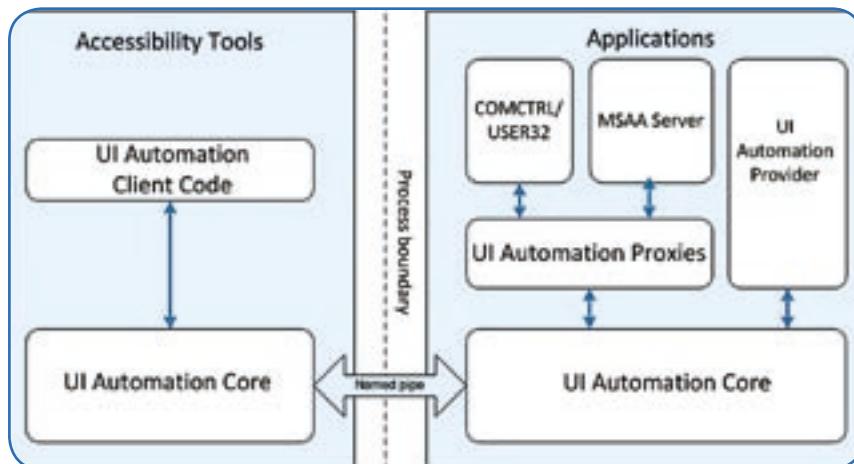
Les tests automatisés d'interface graphique s'appuient sur l'utilisation de Microsoft «UI Automation» (User Interface Automation), le nouveau framework d'accessibilité pour Microsoft Windows fourni avec la version 3.0 du Framework .Net.

Ce Framework évolutif, permet aux développeurs d'accéder par programmation :

- Aux éléments de l'interface.
- Aux informations des éléments de l'IHM.
- D'interagir avec ces éléments.
- De recevoir des notifications liées aux changements de l'interface.
- Et donc de coder des tests automatisés d'interface graphique.

L'UI Automation distingue :

- Les fournisseurs («UI Automation Providers») : les applications existantes ou en développement et qui pourraient-être utilisées par des utilisateurs nécessitant une aide, en terme d'accessibilité.
- Les clients («UI Automation Clients») : les applications qui utilisent ce que propose un fournisseur pour améliorer l'accessibilité.



B. Tests automatisés d'interface utilisateur

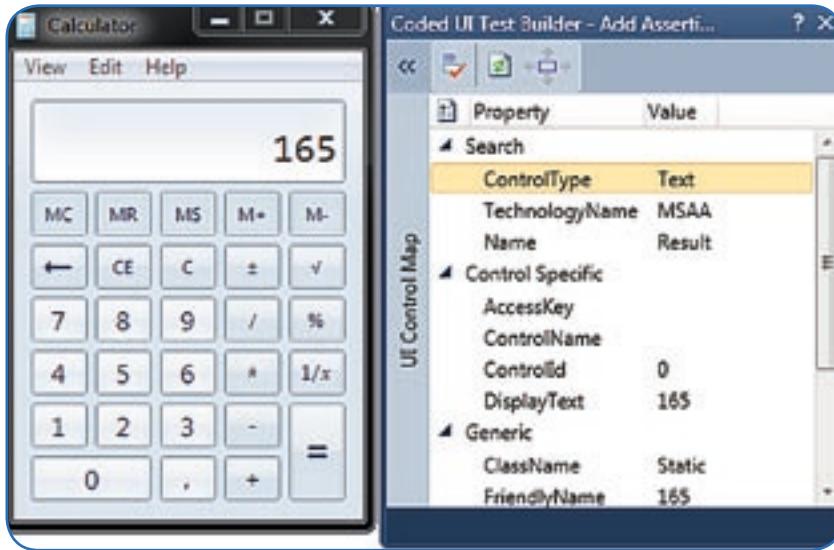
Avec la version Premium et supérieure de Visual Studio 2010, les tests automatisés d'interface graphique, par génération de code, sont intégrés directement dans l'environnement de développement.

Celui-ci dispose ainsi d'un enregistreur capable de capturer une séquence de clicks utilisateurs sur le bureau Windows (indépendamment de la résolution) ou dans un navigateur Web (quel qu'il soit).



L'enregistreur de test d'interface

Dans le premier cas, l'enregistreur s'intéresse aux événements Windows ainsi qu'aux propriétés des objets, permettant de tester toute application que Windows est capable d'afficher ! (nul besoin de posséder son code source).



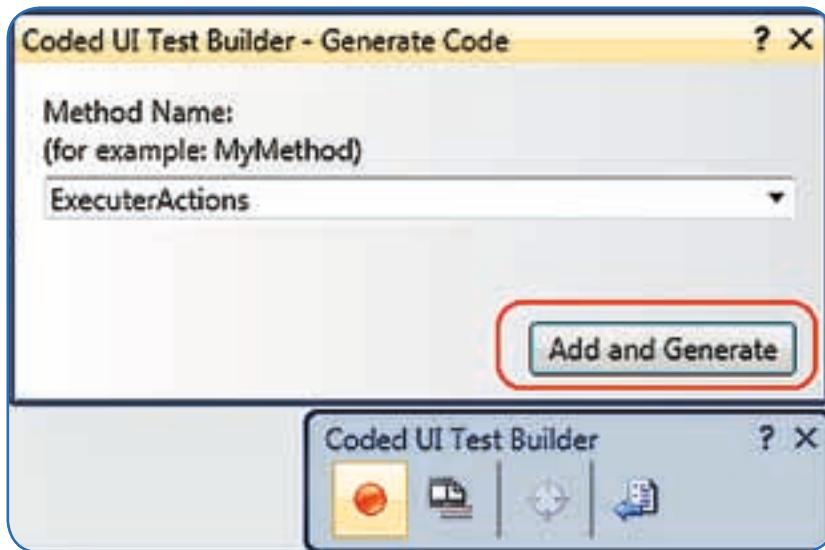
Extraction des propriétés Windows pour une validation

Dans le deuxième cas, l'enregistreur s'intéresse au DOM interprété par le navigateur et est sensible aux événements Javascript et autres applications de feuilles de style CSS. Il est donc bien ici question de test d'interface et de comportement graphique et non de tests de composants techniques.

Lorsque l'on demande la création d'un test d'interface graphique, il nous est

demandé si l'on souhaite créer le test en partant de zéro et en utilisant l'enregistreur ou si l'on souhaite utiliser un enregistrement existant. Il est en effet possible de demander l'enregistrement des actions réalisées par un testeur fonctionnel lors de l'exécution d'un test manuel. Cet enregistrement est alors associé au cas de test (work item) qui pourra être utilisé pour reproduire ces actions dans un test codé d'interface graphique.

Quelques soit le mode de création que l'on utilise, le principe reste le même : des actions ayant un point commun sont regroupées dans des méthodes dont l'exécution peut être enchaînée de façon à reproduire ces actions.



Le plus généralement, on retrouvera des méthodes du type :

- ExecuterApplication() ;
- RealiserActions() ;
- RealiserAutresActions() ;
- FermerApplication() ;

L'avantage de regrouper de manière logique les différentes actions dans de telles méthodes permet de les réutiliser très facilement pour recomposer d'autres tests sans avoir à réenregistrer ces actions.

```

[TestMethod]
public void CodedUITestMethod()
{
    this.UITest.ExecuteApplication();
    this.UITest.RealiserActions();
    //Message verification de l'enregistrement
    this.UITest.RealiserActions();
    this.UITest.FermerApplication();
}

```

Voici un exemple de méthode générée :

```
public void ExecuterActions()
{
    #region Variable Declarations
    Microsoft.Test.Input.ItemButton item1button = this.CalculatorWindow.ItemsWindow.Item1Button;
    Microsoft.Test.Input.ItemButton addbutton = this.CalculatorWindow.ItemsWindow.AddButton;
    Microsoft.Test.Input.ItemButton item2button = this.CalculatorWindow.ItemsWindow.Item2Button;
    Microsoft.Test.Input.ItemButton equalbutton = this.CalculatorWindow.ItemsWindow.EqualsButton;
    #endregion

    // Click '1' button
    Mouse.Click(item1button, new Point(28, 8));

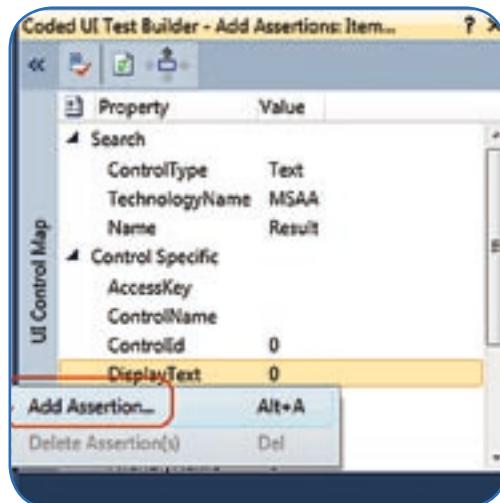
    // Click 'Add' button
    Mouse.Click(addbutton, new Point(15, 8));

    // Click '2' button
    Mouse.Click(item2button, new Point(31, 9));

    // Click "Equal" button
    Mouse.Click(equalbutton, new Point(16, 45));
}
```

Les coordonnées que l'on aperçoit ici ne sont pas les coordonnées absolues du contrôle dans l'écran mais l'endroit où a eu lieu le click en relatif dans le contrôle. Les contrôles sont bien évidemment récupérés via une arborescence de contrôles et suivant des recherches sur des propriétés (si deux « `Button` » se retrouvent au même niveau dans l'arborescence, on les reconnaît grâce à leur propriété, « `Name` »).

Sur le même principe que pour les tests unitaires, il est possible d'ajouter des assertions concernant cette fois ci des propriétés d'un élément de l'application. Pour cela, on sélectionne un élément graphique de l'interface et lui ajoute un test de validation :



Il est alors très simple de choisir le type d'assertion, puis de demander la génération du code de validation correspondant :



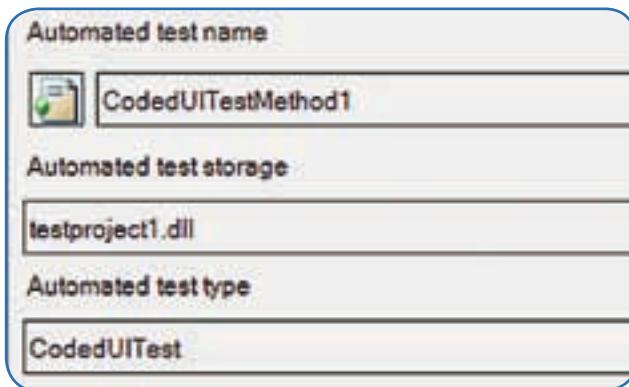
Un certain nombre de fichiers sont générés ou mis à jour lors de la création d'un test automatisé :

- UIMap.uitest : Fichier XML de cartographie de l'application et des actions réalisées.
- Une classe partielle décomposée en :
 - UIMap.Designer.cs : code généré à partir du fichier XML contenant les méthodes enregistrées.
 - UIMap.cs : code personnalisé à créer à partir de duplications de méthodes provenant du fichier UIMap.Designer.cs
- UserControl.cs : Contient des classes permettant de manipuler les contrôles graphiques des applications Windows.
- VotreTest.cs : Contient une classe décorée par l'attribut [CodedUITest] ainsi que des méthodes décorées par l'attribut [TestMethod] contenant les enchaînements d'appels aux méthodes que vous avez générées.

Tout comme les tests unitaires, l'exécution d'un test d'interface graphique peut se faire directement via le code à partir du menu contextuel (puis Run Test), ou à partir de la fenêtre « Test View ». De plus, l'utilisation des sources de données est également possible. On pourra donc utiliser en entrée de tests d'interface

graphique des données provenant d'une base de données SQL Server, d'un fichier CSV ou XML, voir même d'un nouveau type de source de données disponible pour ce type de test : les « Test Cases DataSources ». Grâce à cela, un test peut récupérer des paramètres décrits dans un cas de test manuel défini par un testeur fonctionnel dans Microsoft Test and Lab Manager. L'avantage est de n'avoir besoin de définir les valeurs pour ces paramètres qu'à un seul endroit que ce soit pour une exécution manuelle ou automatisée.

Une fois qu'un test d'interface graphique est automatisé il est possible de l'utiliser pour enrichir un cas de test manuel existant. De cette façon, le testeur généraliste aura le choix d'exécuter ledit test manuellement ou de façon automatique. Pour cela, il suffit d'associer le test automatisé au work item correspondant au cas de test :

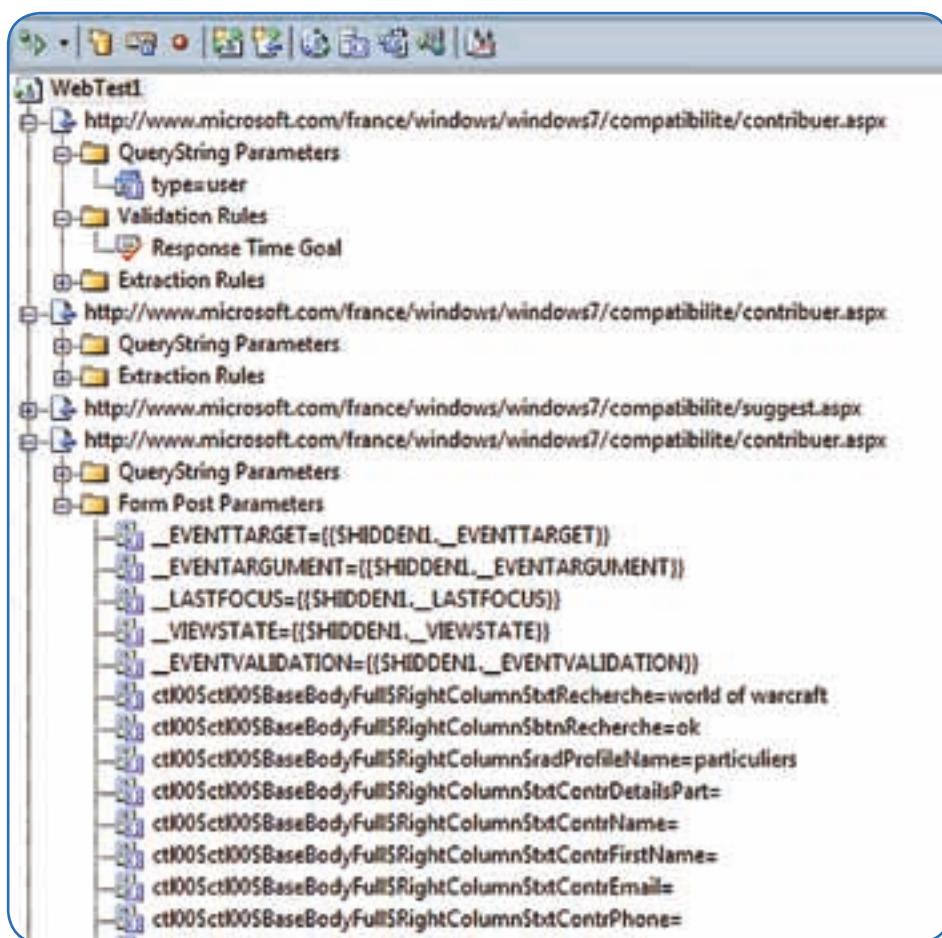


NB : Du fait que ce test automatisé n'est pas connu de l'outil Test and Lab Manager utilisé pour démarrer l'exécution d'un cas de test fonctionnel, il est nécessaire d'associer le plan de test (voir « Stratégie de tests avec Visual Studio 2010 ») à une définition de build contenant les binaires du test.

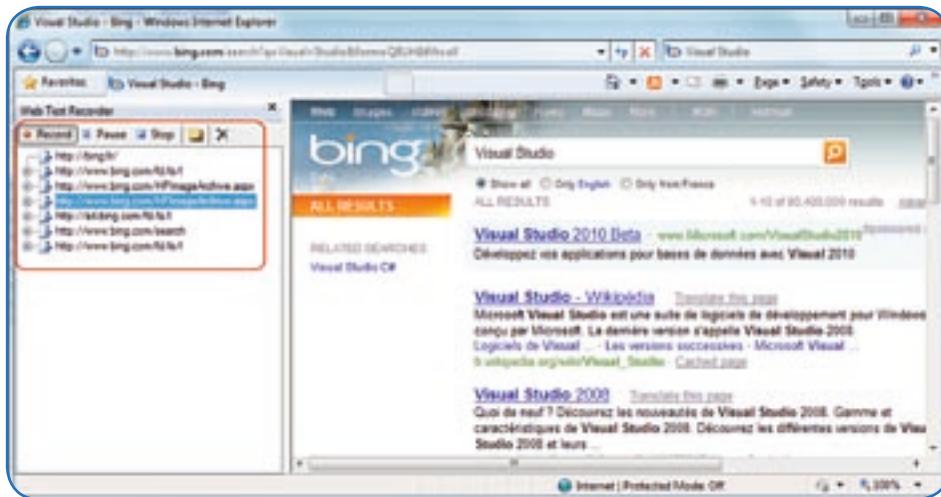
6. Les tests d'interface Web

Un test Web a pour rôle de reproduire un scénario de navigation sur un ou plusieurs sites Web. Concrètement, le test Web permet de décrire un ensemble de requêtes HTTP (GET / POST) et se charge de rejouer celles-ci en assurant que les requêtes et le contenu HTML renvoyé par le serveur sont ceux attendus.

Il ne s'agit donc pas d'un test d'interface graphique (pas de validation ou d'interprétation JavaScript ou CSS) mais bel et bien d'un test d'enchaînement de requête (de navigation).



Le test Web permet d'automatiser un scénario de navigation Web. Il se compose d'un ensemble de requêtes HTTP (POST / GET) pouvant être rejoué en boucle. Pour capturer un test Web, le plus simple est d'utiliser un enregistreur (proxy) HTTP qui stocke les échanges client/serveur et les transcrit dans un fichier. Deux outils sont très bien adaptés pour ceci, le Web test Recorder, proposé par Visual Studio et qui s'intègre à Internet Explorer ou Fiddler2, outil gratuit proposé par Microsoft qui enregistre l'ensemble des trames HTTP quelque soit le navigateur Web utilisé et est capable de générer un fichier .webtest : un format lu par Visual Studio 2010. A noter que Fiddler est très intéressant lors d'un test Web sur une application client/serveur basée sur des services web, car il est également capable d'enregistrer les échanges SOAP et de les reproduire.



Enregistrement à partir de l'outil Web Test Recorder

Un test web peut contenir des règles d'extraction (capture de variables dans une requête pour les réinjecter dans une autre), de validation (code d'erreur http, texte présent dans la page, temps de réponse) et peut être lié à une source de données externe (par exemple : test d'un formulaire d'authentification avec plusieurs identifiants).

Un test Web est décrit dans un fichier XML mais peut, dans un contexte de personnalisation avancé, être transformé en code .Net.

Le Test Web peut servir à valider d'un point de vue fonctionnel :

- « La page renvoyée doit afficher « vous êtes bien connecté après transfert de tels identifiants / mots de passe ».
- « Pas d'erreur 404 (page non trouvée) ou autre lors d'une navigation sur toute l'application » ... mais également d'un point de vue qualitatif.
- « Les pages doivent s'exécuter en moins de 2 secondes ».
- « Les pages doivent avoir un poids inférieur à 2 ko ».

Une dizaine de règles sont proposées par défaut (« Find Test », « Maximum Request Time », « Required Tag », etc.), mais la liste peut très facilement être enrichie au travers de règles personnalisées.

Depuis la version 2010 de Visual Studio il est possible d'utiliser des boucles et des conditions dans les requêtes. Ceci peut être extrêmement pratique lorsque le nombre d'exécutions d'une requête dépend d'un paramètre dynamique, récupéré lors de l'exécution du test web.



7. Profiling

Autre sujet qui sort un peu du contexte des tests, Visual Studio dispose depuis sa version 2005 d'un outil pour analyser les performances des applications et mettre en avant les goulets d'étranglement. L'intérêt pour les analyses des performances est que cet outil peut être exécuté directement depuis un test.

Quelques exemples :

- Analyse des performances d'une fonctionnalité applicative après isolation dans un test unitaire
- Analyse des performances d'une page Web et de ses sous-contrôles utilisateurs à partir d'un test Web

8. Les tests de charge

A. Objectifs

Deux cas fréquents d'utilisation des tests de charge existent :

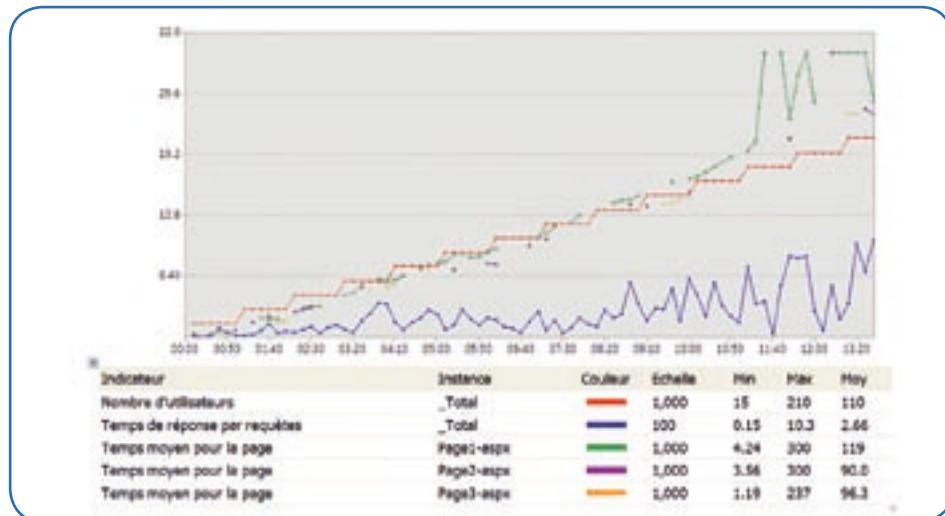
- Pour valider une infrastructure cible avant une mise en production : s'assurer que l'infrastructure et l'application vont supporter la charge utilisateur visée.
- Pour assurer une qualité en continue de l'application : s'assurer quotidiennement qu'il n'y ait pas de régression en termes de performances de l'application afin de pouvoir réagir immédiatement en cas de souci.

Les tests de charge permettent donc de s'assurer que la solution fonctionne correctement avec un grand nombre d'utilisateurs. Au final, le périmètre couvert par ces tests regroupe :

- Les tests caractéristiques de performances : comment la réactivité de l'application est affectée en augmentant la charge.
- Les tests de stress : comment la solution gère les niveaux extrêmes de charge.
- Les tests d'évolutivité : comment la solution réagit en fonction du modèle de « scaling » (« scale-up » ou « scale out »).
- Les tests de durée : comment se comporte la solution si une charge est appliquée sur une période prolongée de test.
- Test de la continuité d'activité : comment la solution gère une mise en échec partielle de son infrastructure logicielle ou matérielle.

Le test de charge est le seul test dont le résultat n'est pas binaire. Tous les résultats de test sont stockés dans la base de données « TFS Data Warehouse » ce qui facilite ensuite la création de rapports de synthèse de test de charge détaillés

pour l'analyse de résultats de test performance. Le rapport généré peut être consulté au fur et à mesure ou a posteriori, car stocké dans une base de données relationnelle. Si Team Foundation Server est utilisé, les résultats peuvent également y être stockés. Il est possible d'industrialiser l'exécution des tests de charge et la mise à disposition des rapports en utilisant conjointement Visual Studio Team Test et Team Foundation Server (service de build). Des modèles de rapports dédiés aux tests de charge sont disponibles dans ce dernier. L'intégration VS-TFS aide donc à la génération de rapports personnalisés qui peuvent être utilisés pour l'analyse des causes des échecs observés lors des tests de charge.



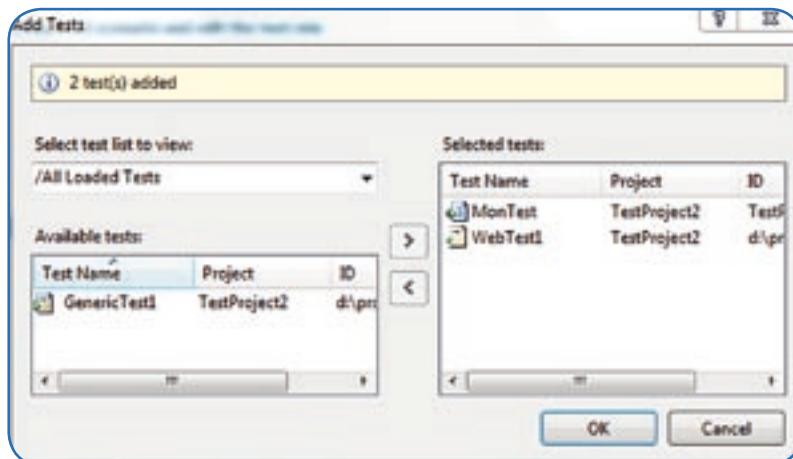
B. Cible

L'outil Microsoft permet de tester en charge les applications spécifiques Web, Windows (client/serveur) en environnement .Net, Java, Php aussi bien que les logiciels (SharePoint, SQL Server ou autre SGDB, SQL Server Reporting Services...).

Seules contraintes :

- Etre capable d'écrire un type de test simulant l'utilisation.
- Etre capable de récolter les compteurs de performance.

Un test de charge n'est rien d'autre qu'un test capable d'exécuter en boucle plusieurs autres types de test. Le test de charge est souvent utilisé avec des tests Web pour valider des applications intra/extranet mais peut très bien être utilisé avec des tests unitaires ou SQL pour valider la solidité de tout autre type d'application.



C. Démarche

La mise en place d'un test de charge nécessite une méthodologie en 4 étapes :

- Définition du périmètre de test : que va-t-on tester ? Comment et avec quels objectifs ?
- Définition de l'architecture matérielle : que va-t-on utiliser comme environnement de test et tester ?
- Création et exécution de la campagne.
- Analyse des résultats : quels sont les problèmes et les axes d'amélioration ? Où est le point de rupture (à partir de quand est-ce que l'on considère que l'application/l'infrastructure a atteint ses limites) ? Et quels sont les facteurs limitant qui permettraient de décaler le point de rupture (processeur trop utilisé, mémoire saturée, goulets d'étranglements dans l'application...)

D. Préparation du Test

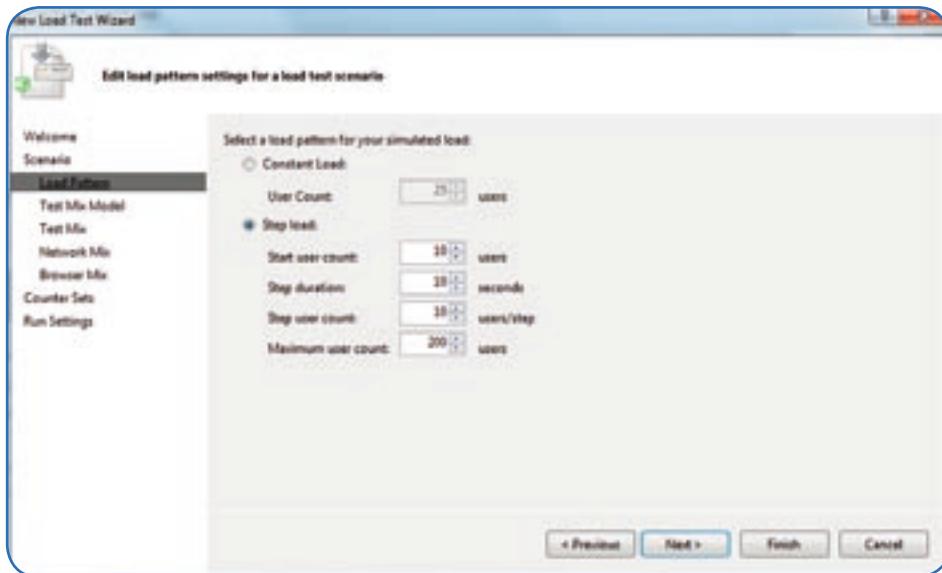
La phase de préparation du test est très importante et passe au travers de plusieurs étapes.

- Définition des scénarios : un test de charge doit se rapprocher au mieux de la réalité, il est donc important lors de la création de scénarios de bien cerner le contexte ciblé. Seront ainsi pris en compte les navigateurs utilisés (une application web peut générer de l'html différent si elle a un Internet Explorer ou un SmartPhone en face), le temps d'attente entre les clics des pages (le but n'est pas de reproduire du clic en continu mais de si-

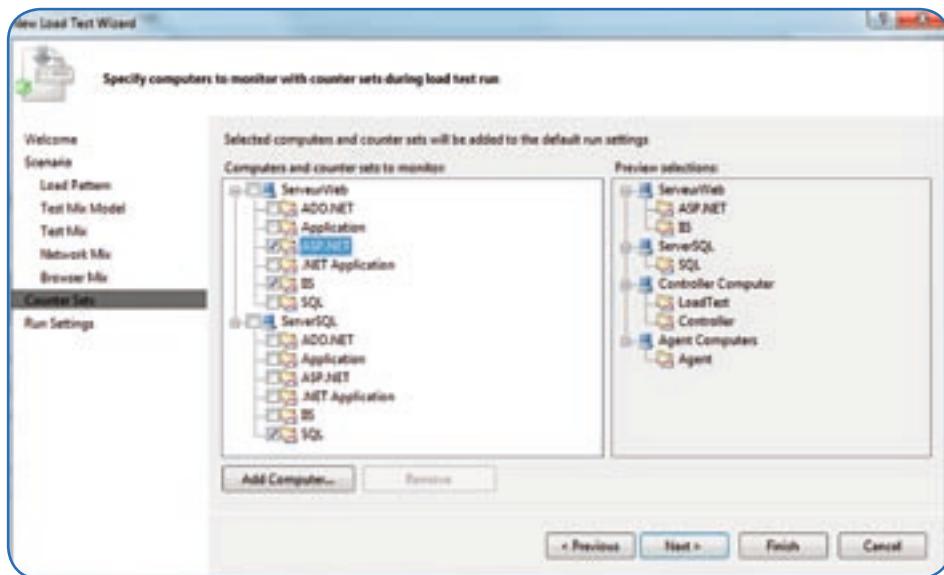
muler des utilisateurs) et la bande passante (l'objectif est d'avoir des temps de réponses acceptables pour tous les utilisateurs, si certains utilisent des modems 56k, ils seront à surveiller en priorité). Plusieurs scénarios peuvent être capturés pour un test de charge, et chacun peut posséder un poids. Par exemple, si 80% des utilisateurs restent sur la page d'accueil de l'application, un scénario doit le simuler, si 10% créent un compte, 10% naviguent dans un catalogue, des scénarios doivent être créés avec un poids similaire dans le test de charge.

- Le type de charge parmi les trois proposées :

- Montante : le nombre d'utilisateurs augmente pendant toute la durée du test, même si la plateforme s'écroule.
- Constante : le nombre d'utilisateurs est stable sur la durée.
- Par Objectif : le nombre d'utilisateurs augmente jusqu'à atteindre un objectif défini (processeur à moins de 80%, temps de réponse à moins de 2 secondes...), lorsque l'objectif est atteint, la charge est lissée pour ne pas écrouler le système.



- Les indicateurs de performance : quels sont les compteurs Windows à prendre en compte dans les résultats de ce test de charge.



E. Infrastructure de Test

Au niveau de l'outillage requis, deux cas existent :

- Pour simuler une charge limitée (en environnement de développement par exemple), Visual Studio seul est souvent suffisant.
- Pour une charge importante (être sûr d'écrouler un environnement de pré-production ou de la répartition de charge) : un contrôleur peut venir se substituer à Visual Studio pour piloter des agents de charge installés sur plusieurs machines. Visual Studio devient dans ce cas le pilote, les agents sont chargés de simuler la charge et de récolter les résultats et le contrôleur de tout centraliser.

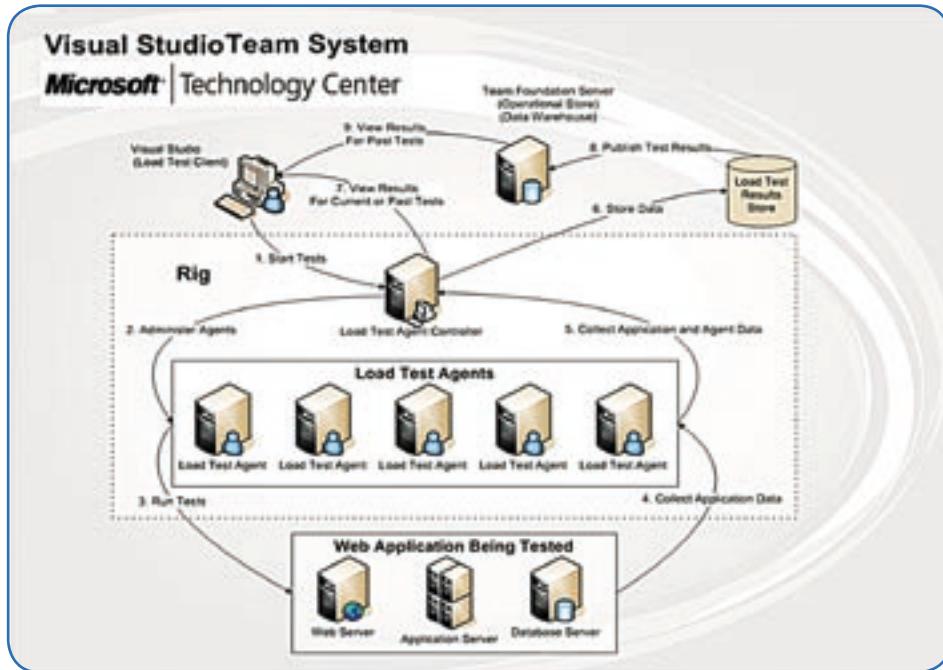
Visual Studio 2010 Test Agent génère des tests de performance des applications Internet. Il permet aux entreprises d'améliorer la qualité de service en testant de manière plus efficace et plus proche de la charge future des serveurs, le comportement de leurs applications Web. Visual Studio 2010 Test Agent simule la charge utilisateur sur les applications Internet et les serveurs pour donner aux équipes "qualité" des informations précises sur le comportement de leurs applications dans un mode de fonctionnement proche de la réalité. Les résultats permettent de connaître les performances de ses applications, de dimensionner les serveurs pour des performances optimales. En un mot d'anticiper. Visual Studio 2010 Test Agent fonctionne avec toutes les technologies d'applications Web.

Les tests web Visual Studio peuvent également être utilisés dans les tests de

performances en les ajoutant dans les tests de charge Visual Studio. Une documentation accompagne le produit et intègre une liste de compteurs, de seuils prédéfinis et leur effet sur les serveurs en charge.

La combinaison de Load Test et des tests Web fournit des règles de validation et d'extraction qui sont essentielles pour rendre les scripts robustes : les règles d'extraction aident à définir comment paramétriser les données générées dynamiquement.

Le schéma ci-dessous illustre le scénario de test de charge tel qu'appliqué pendant la Performance Week au MTC Paris. Le Load agent est installé sur plusieurs machines différentes avec un contrôleur de l'agent de test de charge dans une zone commune d'administration du test de charge. Le test de charge réside sur le client Visual Studio d'édition des tests de charge depuis lequel le testeur va configurer et lancer le test de telle sorte que les injecteurs (les agents de charge) génèrent la charge depuis différents emplacements sur l'application web.

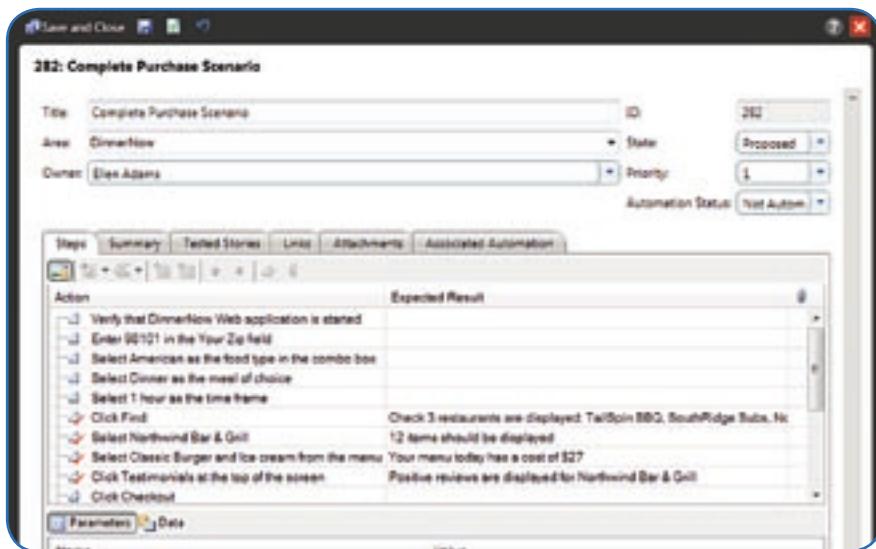


Pour les mesures de performance du code et de la mémoire sur un comportement spécifique d'une portion du code en charge, il est recommandé d'ajouter les tests unitaires dans les tests de charge et d'activer l'analyseur de performance Visual Studio Team System pour pouvoir faire un « profiling » prenant en compte le comportement en charge.

9. Les tests manuels

Avant Visual Studio 2010, les testeurs manuels n'avaient pas réellement d'outils pour définir et exécuter leurs tests fonctionnels. Il était tout de même possible de définir un scénario dans un fichier texte et de l'exécuter à partir de Visual Studio. Cependant, le fait de devoir utiliser Visual Studio pour réaliser ces tests était souvent un frein à l'utilisation et la démocratisation de cet outil.

Avec la version 2010 de Visual Studio arrive un nouveau type de tests : les cas de test. Il s'agit d'éléments de travail (Work Items) au même titre que les tâches ou les bogues. De ce fait, ils bénéficient de toutes les fonctionnalités en termes de partage, communication et traçabilité que tous les autres types de work items. Un cas de test est défini principalement par un ensemble d'étapes à réaliser manuellement par un testeur généraliste pour tester une fonctionnalité de l'application. Chacune de ces étapes peut se voir enrichir d'une indication sur le résultat auquel doit s'attendre le testeur lorsqu'il réalise cette étape ou encore d'un fichier attaché permettant de donner des informations plus complètes sur l'étape en question.



Il est possible de factoriser un ensemble d'étape dans un second type de work items : les Etapes partagées ou Shared Steps. De cette manière, on pourra réutiliser facilement cet ensemble d'étapes dans plusieurs cas de tests distincts.

Très souvent, on veut pouvoir exécuter le même scénario de test avec des paramètres différents. Par exemple si on simule la connexion à un site web, on voudra pouvoir le réaliser avec des informations de connexion différentes.

Pour réaliser cela, un système de paramètres est disponible. On ajoutera alors dans l'intitulé des étapes le nom du paramètre préfixé du caractère « @ » tel que : @ Login. Il ne restera plus qu'à saisir les différentes valeurs pour tous les paramètres dans une grille.

Ceci amènera lors de l'exécution à plusieurs « itérations » de test.

L'exécution d'un cas de test sera abordée dans le chapitre suivant au paragraphe « 3.a : Exécution ».

Stratégies de tests avec Visual Studio 2010

1. Introduction

Visual Studio 2010 introduit un nouvel outil destiné aux testeurs généralistes et fonctionnels, non obligatoirement techniques : Microsoft Test and Lab Manager. Indépendant de Visual Studio cet outil reste connecté à Team Foundation Server pour permettre de faciliter et fluidifier la communication principalement entre testeurs et développeurs.

Cet outil offre deux centres de fonctionnalités :

- Le « Testing Center ».
- Le « Lab Center ».

Le **Testing Center** permet de mettre en place des plans de test (campagnes) qui définissent la manière dont vos équipes de test vont tester fonctionnellement l'application en cours de développement. C'est au sein de ces plans de test que seront gérés des cas de test (Test Cases) qui définissent un scénario de test de l'application. Ces cas de tests sont des work items stockés et partagés dans le serveur Team Foundation. Ce centre permet également d'exécuter des scénarios de tests, d'en suivre les résultats, de reporter des anomalies au reste de l'équipe et de valider des corrections réalisées par l'équipe de développement.

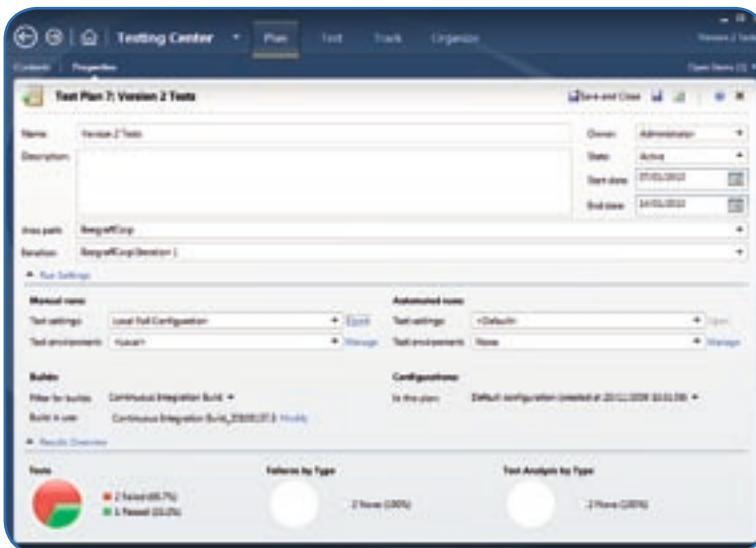
L'exécution des tests donne lieu à la production de rapports sur la progression d'un plan de test et permet de répondre aux questions suivantes :

- Combien de tests ont réussi ou échoué ?
- Combien de tests sont encore à exécuter pour cette itération ?
- Quand les tests seront-ils achevés ?
- Quelles zones ont des taux d'échec élevé de tests ?
- Quelles configurations de test ont des taux d'échec élevés de tests ?
- Qui a encore le plus de tests à exécuter ?
- Comment réaffecter les ressources pour mieux équilibrer les tests restants ?

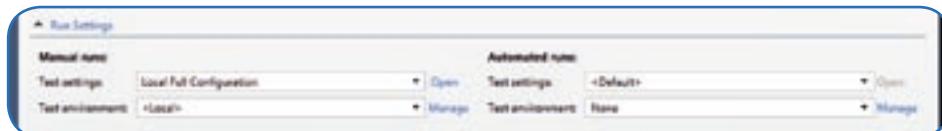
Le Lab Center permet de gérer un laboratoire de tests. C'est dans ce centre que l'on pourra définir des environnements virtuels de tests et les instancier pour pouvoir mettre en place rapidement et simplement des machines virtuelles de tests.

2. Planification des tests

La planification de l'effort de tests à réaliser passe par le Testing Center. Dans cet outil il est possible de créer un ou plusieurs plans de tests pour chaque projet d'équipe du serveur Team Foundation. Il faut voir chaque plan de test comme une vue sur les différents cas de tests disponibles pour le projet d'équipe. Un plan est défini principalement par son titre, sa classification (zone et itération), la personne qui le gère, et les paramètres d'exécution :



Un plan de tests peut contenir et permettre d'exécuter à la fois des tests manuels et automatisés. C'est pour cette raison que l'on retrouve deux types de paramètres d'exécution :

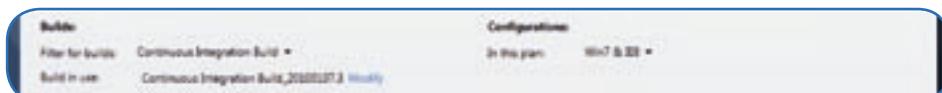


Les paramètres d'exécution permettent de définir les informations qui seront collectées automatiquement lors de l'exécution d'un test. On retrouve plusieurs collecteurs par défaut, tels que :

- **Vidéo** : enregistre ce qu'il se passe à l'écran pendant que le testeur exécute le test.
- **IntelliTrace** : collecte les informations évènementielles de débug historique permettant au développeur de comprendre rapidement l'origine d'une anomalie si besoin.
- **Action recording et Action Log** : permet d'enregistrer la liste des actions que réalise le testeur dans le but de pouvoir générer un test d'interface graphique codé automatiquement.
- **Test Impact** : permet d'activer l'analyse de test impacté de façon à pouvoir automatiquement connaître la liste des tests à rejouer pour valider une nouvelle version et éviter la régression des fonctionnalités déjà testées.

Le but d'un plan de tests est de valider une version cible de l'application, comme par exemple celle produite par l'itération 1. La validation se fait au fur et à mesure du développement, période pendant laquelle les équipes de test vont fréquemment valider les nouvelles versions intermédiaires produites par les équipes de développement à travers le service de Build. De manière à garantir la traçabilité, il est possible et conseillé d'indiquer quel est le contexte courant de test en précisant la Build à utiliser. De cette manière, quand une anomalie sera remontée, elle sera automatiquement marquée comme « découverte dans la définition de build : XXX ».

De même, il est possible de définir un ensemble de configurations au sein desquelles on doit exécuter la campagne de test. On pourra par exemple imaginer devoir valider une application web avec un système d'exploitation Windows 7 doté d'un navigateur Internet Explorer 8 et dans un Windows Vista exploitant Internet Explorer 7.

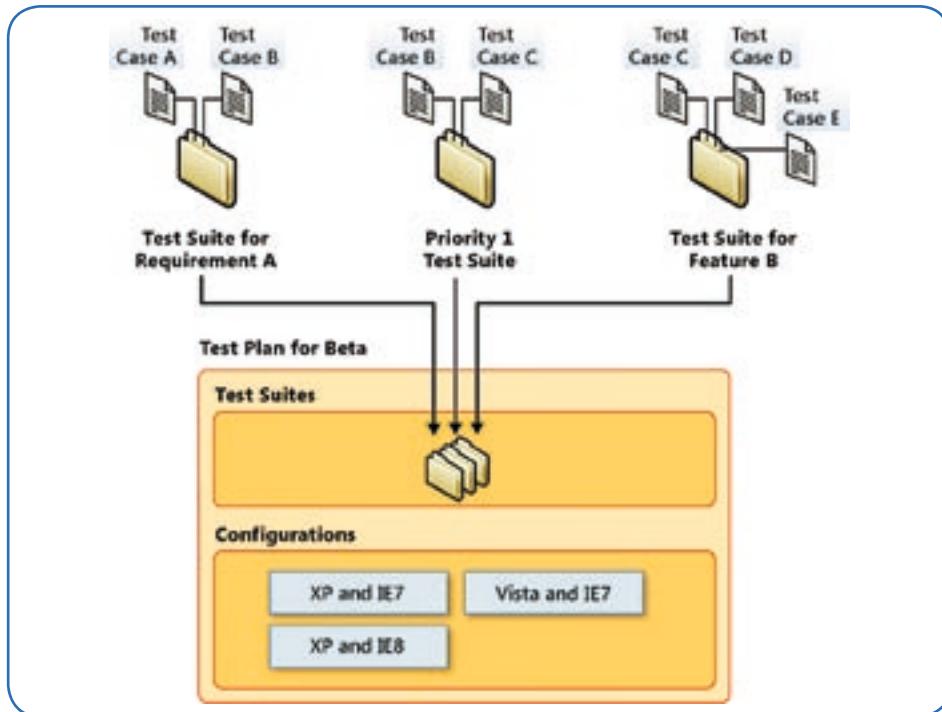


Au-delà d'une simple fiche de définition, le plan de test permet de gérer les cas de tests qui permettront de valider fonctionnellement l'application. Ces cas de tests proviennent du serveur Team Foundation, et peuvent être créés à partir de Microsoft Test and Lab Manager. Ils sont créés dans des listes appelées suites de tests permettant de les organiser pour regrouper ceux qui sont en rapport les uns les autres.

Les suites peuvent être de plusieurs types :

- Suite simple : répertoire permettant d'organiser un ensemble de cas de tests.

- Suite de type « Requirement » : répertoire associé à un work item de type requirement dans Team Foundation Server. Tout test associé à cette suite sera lié au work items requirement.
- Suite de type « Requête » : répertoire dynamique affichant les cas de tests renvoyés par une requête de work items.



The screenshot shows the Microsoft Testing Center interface:

- The top navigation bar includes **Testing Center**, **Plan**, **Test**, **Track**, and **Organize**.
- The left sidebar shows **Contents** and a list of test cases under **Version 2 Tests**:

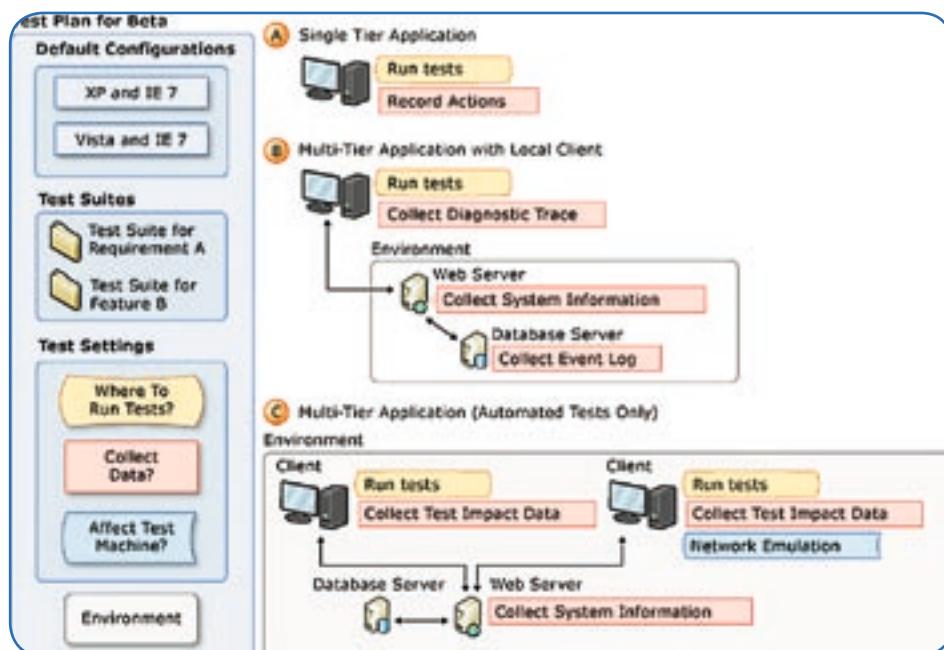
 - ④ 26 The user is able to add an employee
 - ④ 26 The user is able to edit an employee
 - ④ 26 The user is able to add a department

- The main area displays a detailed view of a test case titled "Test suite: 30. The user is able to edit a department (Requirement 18)".
- The details pane shows the following information:
 - Default configurations (1): Default configuration created at 2011/09/09 10:51 - Status: In progress
 - Links: [Edit] [Add] [Rep] [Delete] [Assign] [Configurations]
 - Table of test cases:

ID	Title	Priority	Conf.	Testers	Area Path
78	Editing one department	3	1	Administrator	SampleCorp
79	Editing two departments	3	1	Administrator	SampleCorp

C'est dans ces suites que les cas de tests seront créés ou ajoutés s'ils existent déjà dans le serveur.

Le schéma suivant récapitule les informations stockées dans un plan de test Microsoft Test and Lab Manager.

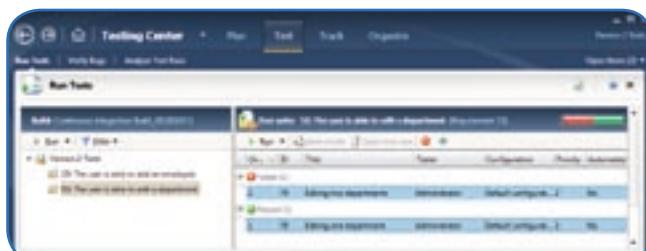


3. Communication, échanges et traçabilité des tests

A. Exécution

Au-delà d'un simple outil de définition de cas de tests et de campagnes les organisant, Microsoft Test and Lab Manager permet de les exécuter et de reporter des anomalies vers les équipes de développement et de suivre l'évolution des corrections.

Les outils nécessaires à l'exécution de cas de test sont compris dans l'onglet Test du testing center de Microsoft Test and Lab Manager :



Les outils disponibles permettent :

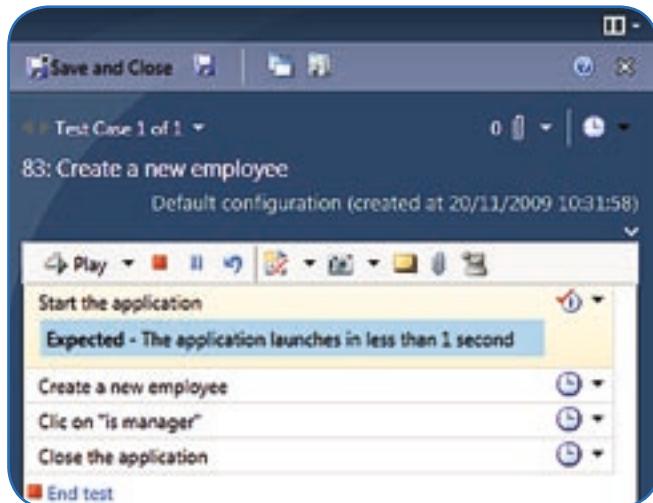
- D'exécuter un ensemble de cas de tests.
- De suivre l'évolution des anomalies (bogues) reportées à l'équipe de développement.
- D'analyser les résultats d'exécutions de tests passés.

On y retrouve les différentes suites de tests exécutables pour ce plan de test ainsi que les cas de tests les comprenant.

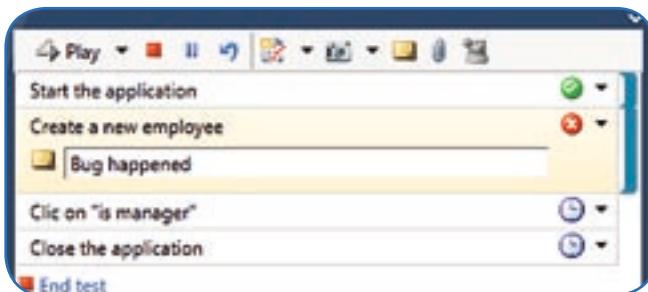
Chaque cas de test est identifié par un état indiquant la dernière information connue pour ce test. On retrouvera notamment les états Failed et Passed indiquant le résultat de la dernière exécution, mais également les états Active et Running permettant de savoir respectivement si le test doit être exécuté, ou s'il est en cours d'exécution par un autre testeur.

Chaque fois que l'on démarre une exécution de test, l'outil se place dans une logique de « Test Run ». Un test run inclut potentiellement l'exécution de plusieurs cas de tests et un rapport sera disponible pour l'ensemble du test run et enfin pour chacun des tests contenus dans ce run.

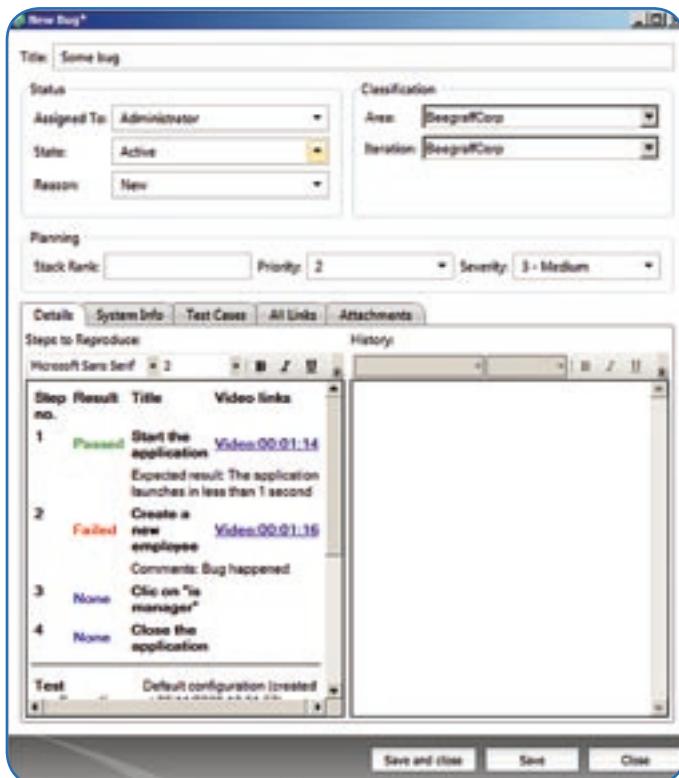
L'exécution d'un ou plusieurs tests transforme l'outil Test and Lab Manager en un autre outil destiné à guider le testeur dans son travail : le Test Runner. Il permet de connaître la liste des tests à exécuter ainsi que les étapes à réaliser pour chacun de ces tests :



Chaque étape est alors à réaliser, et à marquer comme réussie ou échouée :



La force de cet outil est de faciliter la communication avec le reste de l'équipe. De ce fait, lorsqu'une anomalie est découverte, il est très simple de reporter un bogue à l'équipe de développement. Ce bogue sera automatiquement rempli avec les informations récupérées par les différents collecteurs configurés pour le plan de test :



On retrouvera des informations telles que :

- La vidéo et des « marques pages » dans celle-ci pour chacune des étapes.
- La liste des étapes et leur état échoué ou réussi.
- Les commentaires associés à ces étapes.
- Les fichiers attachés à ces étapes, s'ils existent.

Mais également des informations relatives à l'environnement d'exécution (utilisation de la mémoire, architecture du processeur, version de Windows, etc.).

De plus, un lien est automatiquement créé entre ce nouveau Bogue Team Foundation Server et le cas de test utilisé par le testeur pour trouver cette anomalie.

De cette manière, le développeur en charge de l'analyse de ce problème identifié sur l'application aura toutes les informations nécessaires à la reproduction du bogue. Ceci est identifié sous le terme : « Plus jamais de non-reproductibilité ! ». L'avantage de cette solution est que le testeur n'a plus qu'à se concentrer sur ce qui fait sa valeur ajoutée : l'exécution et l'analyse de tests, la collection des informations étant automatisée.

B. Analyse de résultats

Chaque « test run » amène à un rapport exploitable directement à partir de Microsoft Test and Lab Manager. Ce rapport contient les informations telles que :

- Quand a été exécuté le run ?
- Par qui ?
- Quelle était la définition de build associée au plan lors de cette exécution ?
- Combien de tests étaient exécutés dans ce run ?
- Etc.

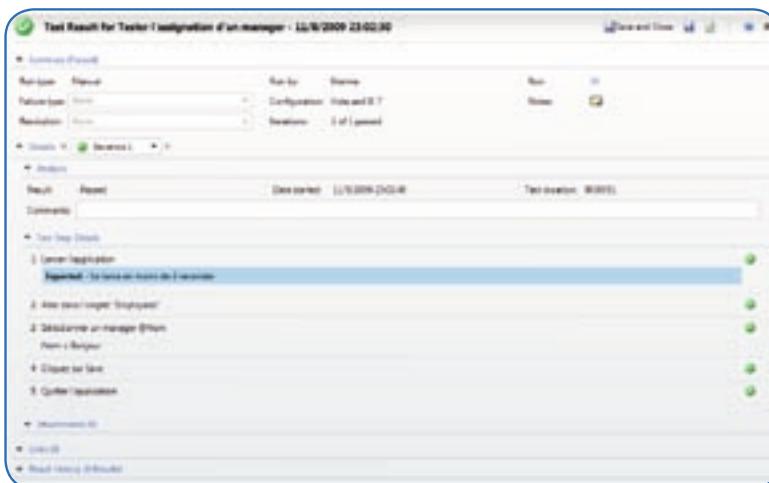
On y retrouve également un accès au rapport de chacun de tests exécutés :

The screenshot shows the Microsoft Test and Lab Manager interface. At the top, there's a summary card for 'Test Run ID: Manual testing'. It includes details like 'Type: Manual testing', 'Design: Scenario', 'Run created: 22/8/2018 (10:00:00)', 'Run completed: 22/8/2018 (10:00:00)', 'Run status: Manual', and 'Build: Team Foundation 1.2018 (10:00:00)'. Below this, there are two main sections: 'Test' and 'Test Results by Type'. The 'Test' section has a green circular icon with a checkmark and the text '1 passed (100%)'. The 'Test Results by Type' section shows two rows: '1 Errors (0%)' and '0 Warnings (0%)'. At the bottom, there's a table titled 'Select the failure type for the associated results' with columns: 'Name', 'Type', 'Configured', 'Run Date', 'Description', and 'Status'. One row is visible: '1 Error - Test execution failed - 22/8/2018 - Failed - Test execution failed - 22/8/2018'.

Pour chacun des tests, un rapport complet et détaillé est disponible dans lequel on retrouve notamment :

- Sa durée.
- Les détails de chacune des étapes réalisées.
 - Réussie ou échouée.
 - Liste des fichiers attachés.
 - Commentaires du testeur.

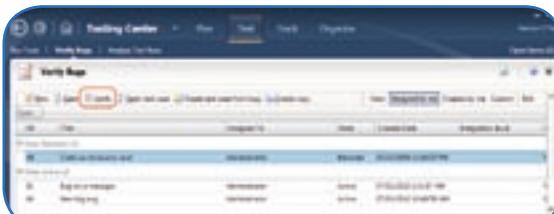
On y retrouve également la liste des anomalies associées, potentiellement reportées lors de l'exécution de ce test.



C. Communication avec l'équipe de développement

Lorsque les développeurs mettent à disposition une nouvelle version de l'application, celle-ci peut potentiellement corriger des anomalies, ou contenir des évolutions de fonctionnalités existantes.

Il est important pour l'équipe de test de pouvoir facilement suivre ces modifications et corrections. Pour cela, chaque testeur pourra visionner à n'importe quel moment la liste des anomalies qu'il a reportées



A partir de la vue **Verify Bugs**, il peut très simplement connaître la liste des bogues résolus et demander de valider la correction associée.

Pour cela il lui suffit de cliquer sur le bouton Verify et le cas de test ayant été utilisé à l'origine pour détecter l'anomalie sera automatiquement ré-exécuté.

Si le test est un succès, et que le testeur estime que la correction est valide, il sera invité à marquer le bogue comme résolu. Dans le cas inverse, il pourra le marquer comme actif à nouveau pour le « renvoyer » vers l'équipe de développement.

Nous avons vu précédemment qu'un plan de test est associé à une définition de build. L'onglet Track permet notamment de visionner la liste des builds disponibles, et d'en assigner une nouvelle (par exemple : la plus récente) au plan courant. De ce fait, une vue lui est proposée mettant en avant :

- La liste des tests recommandés.
- La liste des éléments de travail (work item) associés à cette définition de build.

Les tests recommandés sont liés à la notion d'analyse de tests impactés. Quand un testeur exécute un test manuel, un collecteur analyse le code exécuté et enregistre cette information. Cette analyse est alors couplée au code qui a été modifié dans la définition de build courante, et tous les tests qui ont été marqués comme ayant utilisé ce code sont alors proposés comme « Tests recommandés ». Il s'agit ici d'un premier niveau de non régression. Cela n'évite bien entendu pas de jouer les cas de tests pour les nouvelles fonctionnalités.

ID	Name	Status	Priority	Last run date	Suite	Test configuration
1	Tester l'application d'un manager	Failed	2	11/02/2013 22:54:15	Nombreux tests	Valeurs E/T

Chaque fois qu'un développeur archive (envoie) du code dans le contrôle de code source, il a la possibilité d'y associer un work item. Celui-ci peut être un bogue, une tâche ou encore un scénario d'utilisation ou requirement. Cette information est vitale pour les testeurs car elle permet de savoir si de nouvelles fonctionnalités ont été implémentées et par là-même, quels tests il faudra jouer voir créer.

4. Reporting sur les tests

Toutes les données de Team Foundation Server sont stockées dans une base de données. Celles-ci alimentent un entrepôt de données permettant de générer des rapports donnant une vue macro sur l'évolution du projet dans son ensemble. Ceci est également vrai pour la partie test, d'autant que chaque rapport de Microsoft Test and Lab Manager est stocké dans cet entrepôt.

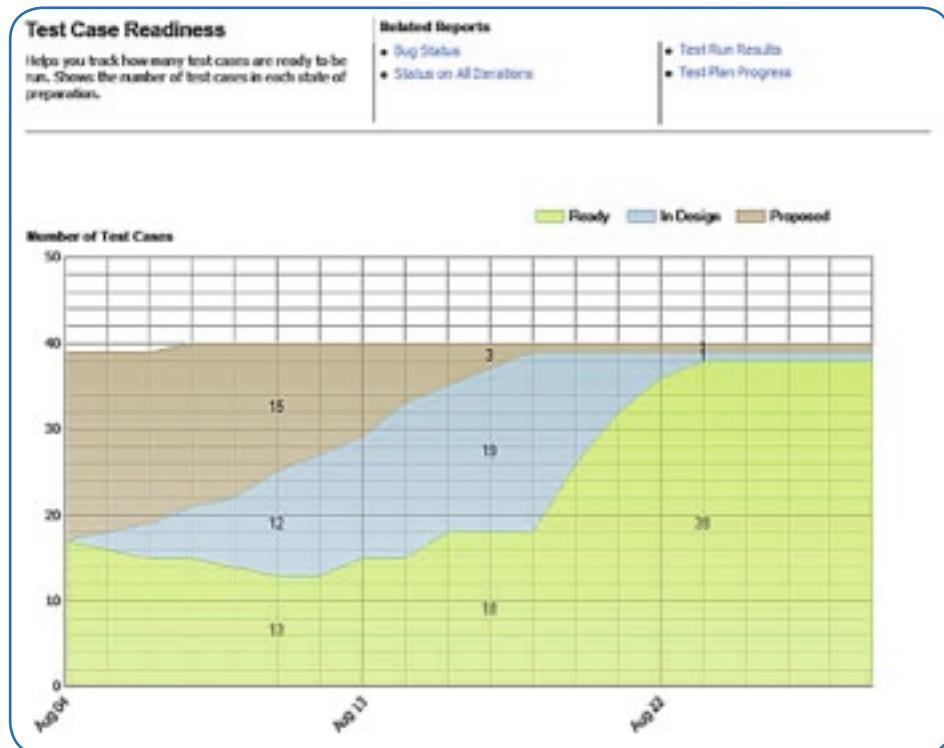
Il existe par défaut un certain nombre de rapports disponibles suivant la méthode que vous avez sélectionnée lors de la création de votre projet d'équipe.

Ils permettent de mettre en avant suivant différentes vues l'évolution de l'avancement de l'exécution ou la création des plans de tests par votre équipe de test.

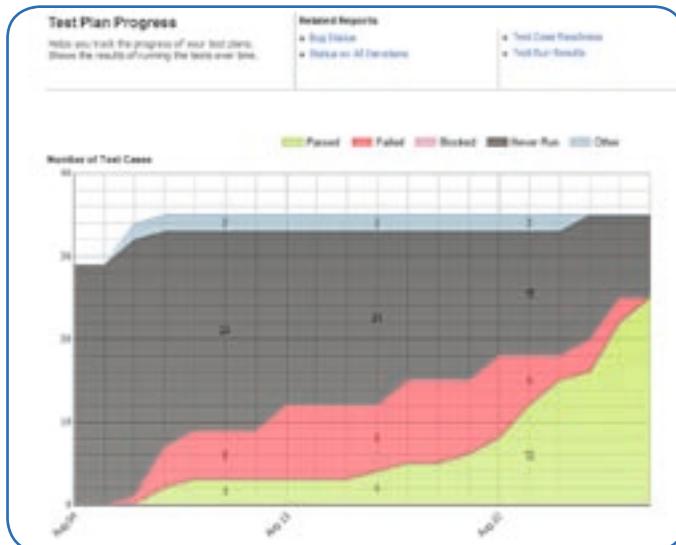
Un premier exemple pertinent, est le rapport de « Test Case Readiness » permettant d'avoir une vue globale de l'avancement de la création des cas de tests, tous plans confondus. En effet, un cas de test peut avoir trois états (suivant la méthode) :

- Proposed (proposé).
- In Design (en cours de réalisation).
- Ready (prêt à être exécuté).

Voici un exemple de rapport généré :



Le rapport « Test Plan Progress » quant à lui permet de mettre en avant l'avancement de l'aspect exécution des plans de tests du projet d'équipe. Vous pouvez voir par exemple sur le graphique suivant que 25 tests sur 40 sont actuellement dans un état « Réussi ». Ceci permet d'avoir une excellente vision sur la qualité de l'application et l'effort réalisé par l'équipe de test pour la valider fonctionnellement.



Le dernier rapport disponible permettra d'avoir une vue différente de la progression des tests, plus tabulaire et organisée par plan de test, puis par requirement.

Test Plan	Requirement	Status	# of Test Cases Passed/Retest/Not Run/Not Yet Run/Other				
			Passed	Retest	Blocked	Never Run	Other
Test Plan For Service 1							
TS_001	TS_001_001	In Progress	10	0	0	20	0
TS_001	TS_001_002	In Progress	8	0	0	12	0
TS_001	TS_001_003	In Progress	10	0	0	10	0
TS_001	TS_001_004	In Progress	8	0	0	12	0
TS_001	TS_001_005	In Progress	8	0	0	12	0
TS_001	TS_001_006	In Progress	10	0	0	10	0
TS_001	TS_001_007	In Progress	8	0	0	12	0
TS_001	TS_001_008	In Progress	8	0	0	12	0
TS_001	TS_001_009	In Progress	8	0	0	12	0
TS_001	TS_001_010	In Progress	8	0	0	12	0
TS_001	TS_001_011	In Progress	8	0	0	12	0
TS_001	TS_001_012	In Progress	8	0	0	12	0
TS_001	TS_001_013	In Progress	8	0	0	12	0
TS_001	TS_001_014	In Progress	8	0	0	12	0
TS_001	TS_001_015	In Progress	8	0	0	12	0
TS_001	TS_001_016	In Progress	8	0	0	12	0
TS_001	TS_001_017	In Progress	8	0	0	12	0
TS_001	TS_001_018	In Progress	8	0	0	12	0
TS_001	TS_001_019	In Progress	8	0	0	12	0
TS_001	TS_001_020	In Progress	8	0	0	12	0
TS_001	TS_001_021	In Progress	8	0	0	12	0
TS_001	TS_001_022	In Progress	8	0	0	12	0
TS_001	TS_001_023	In Progress	8	0	0	12	0
TS_001	TS_001_024	In Progress	8	0	0	12	0
TS_001	TS_001_025	In Progress	8	0	0	12	0
TS_001	TS_001_026	In Progress	8	0	0	12	0
TS_001	TS_001_027	In Progress	8	0	0	12	0
TS_001	TS_001_028	In Progress	8	0	0	12	0
TS_001	TS_001_029	In Progress	8	0	0	12	0
TS_001	TS_001_030	In Progress	8	0	0	12	0
TS_001	TS_001_031	In Progress	8	0	0	12	0
TS_001	TS_001_032	In Progress	8	0	0	12	0
TS_001	TS_001_033	In Progress	8	0	0	12	0
TS_001	TS_001_034	In Progress	8	0	0	12	0
TS_001	TS_001_035	In Progress	8	0	0	12	0
TS_001	TS_001_036	In Progress	8	0	0	12	0
TS_001	TS_001_037	In Progress	8	0	0	12	0
TS_001	TS_001_038	In Progress	8	0	0	12	0
TS_001	TS_001_039	In Progress	8	0	0	12	0
TS_001	TS_001_040	In Progress	8	0	0	12	0
Test Plan For Service 2							
TS_002	TS_002_001	In Progress	8	0	0	12	0
TS_002	TS_002_002	In Progress	8	0	0	12	0
TS_002	TS_002_003	In Progress	8	0	0	12	0
TS_002	TS_002_004	In Progress	8	0	0	12	0
TS_002	TS_002_005	In Progress	8	0	0	12	0
TS_002	TS_002_006	In Progress	8	0	0	12	0
TS_002	TS_002_007	In Progress	8	0	0	12	0
TS_002	TS_002_008	In Progress	8	0	0	12	0
TS_002	TS_002_009	In Progress	8	0	0	12	0
TS_002	TS_002_010	In Progress	8	0	0	12	0
TS_002	TS_002_011	In Progress	8	0	0	12	0
TS_002	TS_002_012	In Progress	8	0	0	12	0
TS_002	TS_002_013	In Progress	8	0	0	12	0
TS_002	TS_002_014	In Progress	8	0	0	12	0
TS_002	TS_002_015	In Progress	8	0	0	12	0
TS_002	TS_002_016	In Progress	8	0	0	12	0
TS_002	TS_002_017	In Progress	8	0	0	12	0
TS_002	TS_002_018	In Progress	8	0	0	12	0
TS_002	TS_002_019	In Progress	8	0	0	12	0
TS_002	TS_002_020	In Progress	8	0	0	12	0
TS_002	TS_002_021	In Progress	8	0	0	12	0
TS_002	TS_002_022	In Progress	8	0	0	12	0
TS_002	TS_002_023	In Progress	8	0	0	12	0
TS_002	TS_002_024	In Progress	8	0	0	12	0
TS_002	TS_002_025	In Progress	8	0	0	12	0
TS_002	TS_002_026	In Progress	8	0	0	12	0
TS_002	TS_002_027	In Progress	8	0	0	12	0
TS_002	TS_002_028	In Progress	8	0	0	12	0
TS_002	TS_002_029	In Progress	8	0	0	12	0
TS_002	TS_002_030	In Progress	8	0	0	12	0
TS_002	TS_002_031	In Progress	8	0	0	12	0
TS_002	TS_002_032	In Progress	8	0	0	12	0
TS_002	TS_002_033	In Progress	8	0	0	12	0
TS_002	TS_002_034	In Progress	8	0	0	12	0
TS_002	TS_002_035	In Progress	8	0	0	12	0
TS_002	TS_002_036	In Progress	8	0	0	12	0
TS_002	TS_002_037	In Progress	8	0	0	12	0
TS_002	TS_002_038	In Progress	8	0	0	12	0
TS_002	TS_002_039	In Progress	8	0	0	12	0
TS_002	TS_002_040	In Progress	8	0	0	12	0

Les rapports sont vitaux pour assurer le bon déroulement d'un projet, et d'autant plus pour l'aspect qualité de celui-ci. C'est pour cette raison qu'il est possible de personnaliser les rapports par défaut et d'en créer de nouveaux suivant les besoins des projets. Ceci peut être réalisé très simplement du fait de l'utilisation de Reporting Service 2008.

5. Gestion d'environnements virtuels de déploiement et de test

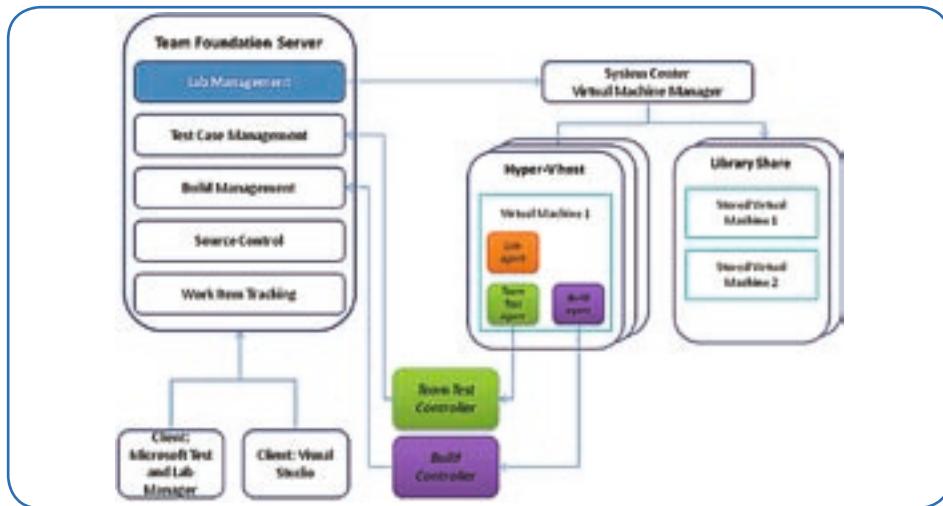
De nos jours les applications sont de plus en plus complexes, destinées à être exécutées sur de nombreux types d'environnements différents ou sur des infrastructures contenant un nombre élevé de machines.

Mettre en place de tels environnements dans le cadre des processus d'assurance qualité est long, complexe et répétitif. Depuis quelques années, la virtualisation a déjà permis de faire un grand pas en avant dans la simplification de la mise en place d'environnement de test, en permettant de dépasser la limite des machines physiques et de dupliquer aisément une ou plusieurs machines virtuelles. Cela étant, cette opération reste manuelle et gourmande en temps.

Visual Studio 2010 Team Lab Management permet d'automatiser la création d'environnement, offrant aux testeurs la possibilité de créer et contrôler eux-mêmes la génération de ces environnements.

Team Foundation Server est ainsi connecté à l'application System Center Virtual Machine Manager 2008. Cette application permet de contrôler un serveur de virtualisation Hyper-V et de stocker des modèles de machines virtuelles dans une bibliothèque.

Vous pouvez voir dans le schéma ci-dessous la nouvelle infrastructure des tests induite par cette nouvelle fonctionnalité :



Le **Lab Center** de l'outil Microsoft Test and Lab Manager permet de gérer tous les aspects du Lab Management.

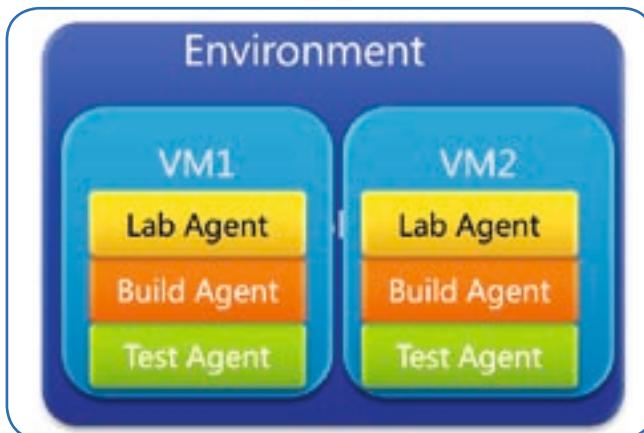
Le principe est simple : le testeur ajoute au plan de tests des modèles de machines virtuelles préalablement créés dans l'outil System Center Virtual Machine Manager.

Il peut alors créer des modèles d'environnements. Un environnement contient potentiellement plusieurs machines virtuelles, comme par exemple : Un serveur SQL, un serveur Web et un client contenant un navigateur web.

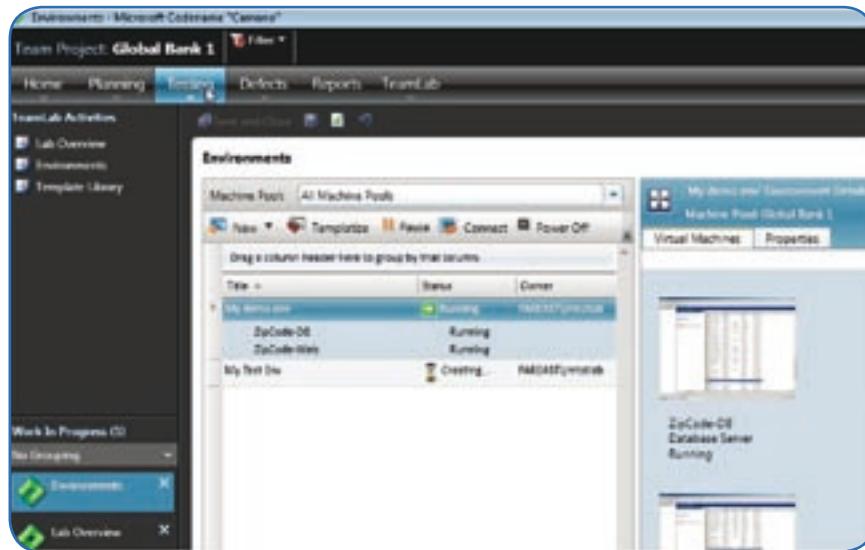
Chaque machine virtuelle se verra dotée de 3 services préalablement installés :

- Le **Lab Agent** : permettant au Lab Management de gérer cette machine à distance.
- Le **Test Agent** : permettant à un testeur de lancer l'exécution d'un test automatisé sur cette machine, et de récolter les différentes informations provenant des collecteurs sélectionnés dans le plan de test.
- Le **Build Agent** : permettant d'exécuter un workflow. Ceci est principalement utilisé dans le cadre d'une définition de build, lors de la phase de déploiement de l'application sur la machine. Un workflow est alors exécuté sur cette machine pour exécuter un certain nombre d'instruction comme par exemple la copie de fichiers, ou l'exécution d'un installeur (MSI).

Un exemple d'environnement pourrait ressembler au schéma suivant :

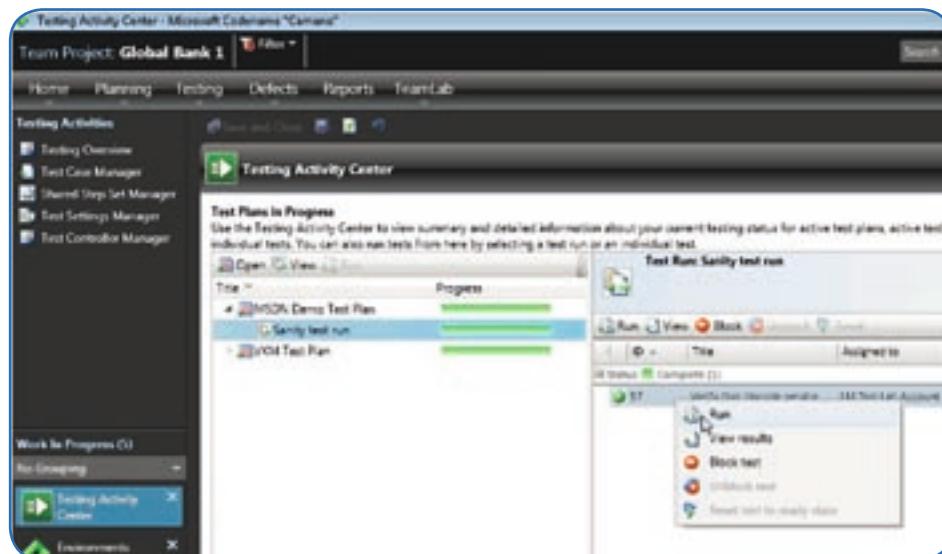


Une fois cet environnement créé, il est possible de l'instancier autant de fois que nécessaire. On pourra alors en un simple clic générer une nouvelle version de ce dernier, dans le but d'y exécuter des tests.



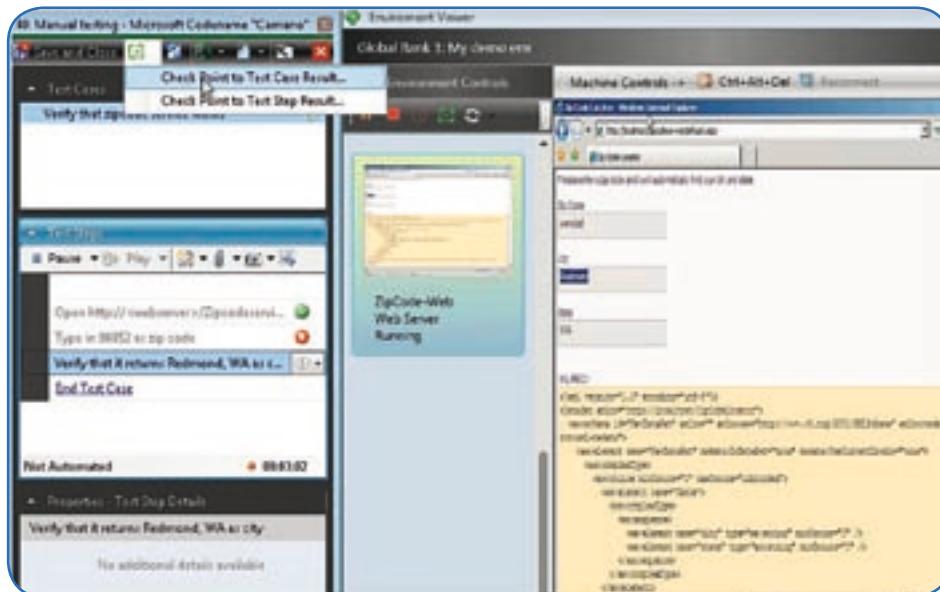
Au-delà d'un simple outil de création d'environnements de tests, le Lab Management peut être exploité lors de l'exécution d'un test manuel ou automatisé.

En effet, il est possible de choisir un environnement virtuel lors de l'exécution d'un test. Toute l'exécution se produira alors dans cet environnement. Un des avantages de cette solution est d'éviter de demander au testeur d'avoir l'application installée sur son poste pour pouvoir la tester. Un autre avantage évident est de pouvoir facilement tester l'application dans plusieurs environnements différents, comme par exemple : sur Windows Vista, Windows 7, etc. sans avoir à changer de poste.



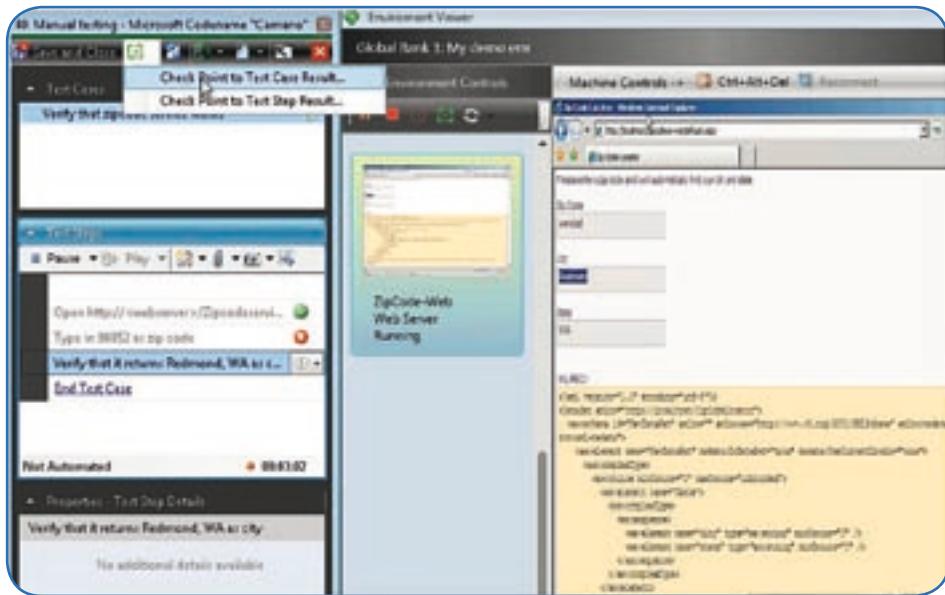
Lorsqu'un testeur exécute un test dans un environnement virtualisé, il a à sa disposition une nouvelle fonctionnalité nommée : « Snapshot ». Le principe est simple :

- Le testeur découvre une anomalie et il veut conserver l'environnement tel qu'il est au moment où cette anomalie est mise en évidence.
- Il sauvegarde un snapshot de tout l'environnement. Celui-ci est alors attaché à l'étape courante.



Lorsque le développeur ouvre le bogue créé par le testeur, il a la possibilité de cliquer sur un lien correspondant à ce snapshot et par là même d'ouvrir l'environnement tel qu'il était lorsque le snapshot a été pris.

Grâce à cette fonctionnalité, on peut revenir au moment même où l'erreur a eu lieu : pas uniquement pour la machine cliente, mais pour l'ensemble des machines de l'environnement, incluant le serveur de base de données, le serveur web, etc.



Pour exploiter pleinement tous les aspects d'automatisation du Lab Management, des activités de workflow de build existent permettant d'avoir un contrôle total sur les environnements gérés par ce service. On pourra donc, lors d'un processus d'intégration continue, en plus de la compilation de l'application, l'exécution des tests unitaires automatiques, la génération de la couverture de code : générer ou ré-exploiter un environnement du lab management pour déployer l'application, prête à être testée !

Synthèse

Les différentes analyses montrent que les tests prennent effectivement de plus en plus d'importance en entreprise, et que ceux-ci ont de beaux jours devant eux. Cependant, la mise en œuvre de ces tests suppose deux pré-requis :

- D'un point de vue technique, s'assurer que les développeurs ont facilement accès aux différents types de tests et que leur écriture ne se traduit pas par une contrainte mais par un réel gain de productivité.
- D'un point de vue organisationnel, rapprocher le monde des testeurs fonctionnels de celui des développeurs responsables de la correction de bogues, en simplifiant et fluidifiant leur communication.

Et c'est exactement sur ces deux points que Visual Studio 2010 innove, en outillant la création de tests d'une manière parfaitement intégrée dans l'environnement de développement et en offrant un référentiel commun aux testeurs fonctionnels et aux développeurs sur l'ensemble du cycle de vie des applications tout en leur permettant d'utiliser des outils qui leur sont propres.

Ce livre blanc nous aura permis de faire un tour d'horizon de la problématique des tests et de la réponse que Microsoft y apporte avec Visual Studio 2010.

Annexe : Etendre le Framework de test

Lors de la mise en œuvre de tests de composants logiciels, il est fréquent d'avoir recours au pattern « Stub de Service » (Martin Fowler) pour s'assurer que le code sous test a obtenu tous les éléments requis pour mettre à l'épreuve sa logique, indépendamment de toute autre logique dont il dépende. La mise en œuvre de ce pattern est réalisée grâce à l'usage d'un système de « bouchonnage » souvent appelé « Mock ». Un objet « Mock » partage la même interface (au sens type .Net) que les objets utilisés par la logique à tester et est conçu pour en permettre la validation.

L'écriture « manuelle » de ces objets « Mock » est souvent répétitive et fastidieuse. Fort heureusement, il existe des frameworks offrant un jeu d'API permettant l'automatisation de la création de ces « Mocks ». Roy Osherove, auteur de l'ouvrage « Art of Unit Testing » propose une liste quasi exhaustive des outils que vous pouvez utiliser lors de l'écriture de vos tests unitaires (voire vos tests d'intégration) sur son site : http://www.artofunittesting.com/Chapters/Tools_and_frameworks

Voici un aperçu de quelques-uns de ces outils :

- « Pex » (<http://research.microsoft.com/projects/pex>) est un assistant pour le programmeur. Avec Pex, il est possible d'associer des attributs à des tests paramétrés. Dans un test unitaire paramétré, le moteur de Pex génère automatiquement une suite de tests unitaires avec un taux de couverture de code élevé. Pex permet ainsi d'identifier les cas aux limites mal gérés dans le code.
- « Microsoft Chess » (<http://research.microsoft.com/projects/CHESS>) est un outil automatisé pour rechercher des erreurs dans les logiciels multithreads par exploration systématique. Il détecte les erreurs, telles que les « deadlocks », les suspensions et les violations d'accès induites par corruption de données, qui sont extrêmement difficiles à trouver avec les outils de test actuels. Une fois l'erreur localisée, il fournit une exécution entièrement reproduitible du programme conduisant à l'erreur, et facilite donc grandement le processus de débogage. En outre, il propose une couverture de test adaptée pour les programmes multithreads.
- « Project White » (<http://www.codeplex.com/white>) est un framework de tests d'intégration proposant une API pour tester les applications riches (WPF, WinForm, Win32 et Java SWT).
- « WCFMock » (<http://wcfmock.codeplex.com/>) est un framework facilitant le développement de tests unitaires ciblant les services WCF. Les services WCF s'appuient souvent sur la notion de contexte WCF (OperationContext et WebOperationContext) pour effectuer différentes actions (en particulier les services REST pour lesquels le contexte est nécessaire pour les paramètres et l'obtention de codes d'en-têtes http). « WCFMock » fournit un ensemble de classes pour supprimer toutes les dépendances explicites avec le contexte de l'opération et propose une façon adaptée de simuler le comportement de ce contexte.
- « Moq » (<http://code.google.com/p/moq/>) propose une API de Mocking très simple d'usage et exploite les fonctions du .Net Framework 3.5 comme les Lambda Expressions et les méthodes d'extension.
- « MockingBird » (<http://mockingbird.codeplex.com/>) est un ensemble d'outils facilitant le « bouchonnage » de services web. Le moteur de « MockingBird » est un intercepteur de messages configurable pour n'importe quel type de service web. « MockingBird » facilite le développement de services Web en proposant une approche « contract first » ainsi que l'isolation des serveurs de développement et de build vis-à-vis des dépendances externes.

Annexe : Tests automatisés d'interface utilisateur : Technologies supportées

Le tableau suivant détaille l'ensemble des technologies supportées :

Windows Internet Explorer 7.0	Fully supported
Windows Internet Explorer 8.0 (Including HTML and AJAX)	Fully supported
Windows Internet Explorer 6.0	Not supported
Chrome	Not supported
Opera	Not supported
Safari	Not supported
Silverlight 3.0	Not supported
Flash/Java	Not supported
Windows Forms 2.0 and greater	Fully supported
WPF 3.5 and greater	Fully supported
Windows Win32	May work with some known issues, but not officially supported
MFC	May work with some known issues, but not officially supported
SharePoint	Partially supported
Office Client Applications	Not supported
Dynamics (Ax)	Partially supported
SAP	Not supported
Citrix/Terminal Services	Partially supported

Annexe : Tests automatisés d'interface utilisateur : Systèmes supportés

Le tableau suivant détaille l'ensemble des systèmes supportés.

Operating Systems	Windows 2000 with Service Pack 4 Windows XP with Service Pack 3 Windows Server 2003 with Service Pack 1 Windows Server 2003 with Service Pack 2 Windows Server 2008 and Windows Server 2008 R2 Note Windows Internet Explorer Enhanced Security must be turned off for Windows Server 2003 and Windows Server 2008. Windows 7
32-bit / 64-bit Support	32-bit Windows that is running 32-bit Microsoft Test and Lab Manager can test 32-bit applications. 64-bit Windows that is running 32-bit Microsoft Test and Lab Manager can test 32-bit WOW Applications that have UI Synchronization. 64-bit Windows that is running 32-bit Microsoft Test and Lab Manager can test 64-bit Windows Forms/WPF Applications that do not have UI Synchronization.
Architecture	x86 and x64 Note Windows Internet Explorer is not supported in 64-bit mode

.Net	<p>.Net 2.0, 3.0, 3.5, and 4</p> <p>Note</p> <p>Microsoft Test and Lab Manager and Microsoft Visual Studio 2010 will both require .Net 4 to operate. However, applications developed by using the listed .Net versions are supported.</p>
Languages	English, Japanese, Korean, Hindi

Annexe : Références techniques

Quelques liens vous permettant d'aller plus loin...

Livres :

<http://www.amazon.fr/Microsoft-Team-Foundation-Server-TFS/dp/274604711X>
http://www.artofunittesting.com/Chapters/The_Basics_Of_Unit_Testing

Blogs :

- Eric Le Loc'h – Spécialiste équipe Team System :
<http://blogs.msdn.com/ericleloch>
- Florent Santin – MVP Team System : <http://blogs.developpeur.org/azra>
- Stephane Goudeau – Architecte MTC : <http://blogs.msdn.com/stephgou>
- Etienne Margraff – MVP Team System : <http://blogs.developpeur.org/etienne>
- Le blog du MTC Paris : <http://blogs.msdn.com/mtcparis/>

Liens:

- Service Stub Pattern
<http://www.martinfowler.com/eaaCatalog/serviceStub.html>
- « Software Testing »
<http://msdn.microsoft.com/en-us/library/cc188960.aspx>
- « MSDN Test Edition »
ms-help://MS.VSCC.v90/MS.MSDNQTR.v90.en/dv_vsetetesttoc/html/796b7d6d-ad45-4772-9719-55eaf5490dac.htm
- Page officielle Team System: <http://msdn.microsoft.com/teamsystem>
- Ensemble de ressources Françaises: <http://www.team-system.fr>
- Microsoft® Visual Studio® Team System 2008 Team Foundation Server SP1 and Team Suite SP1 VPC Image (Trial)
<http://www.microsoft.com/downloads/details.aspx?FamilyID=c7a809d8-8c9f-439f-8147-948bc6957812&displaylang=en>
- Visual Studio® Team System 2008 Team Foundation Server SP1 VPC Image (Trial)
<http://www.microsoft.com/downloads/details.aspx?FamilyID=72262ead-e49d-43d4-aa45-1da2a27d9a65&displaylang=en>

- Team Foundation Server Administrator's Guide
<http://www.microsoft.com/downloads/details.aspx?familyid=2AED0ECC-1552-49F1-ABE7-4905155E210A&displaylang=en>
- Guide d'installation
<http://www.microsoft.com/downloads/details.aspx?familyid=ff12844f-398c-4fe9-8b0d-9e84181d9923&displaylang=en>
- Conversion depuis VSS
<http://www.codeplex.com/Wiki/View.aspx?ProjectName=VssConverterGui>
- Reprise Existant
<http://www.codeplex.com/MigrationSyncToolkit>
- TFS 2008 System Recommendations
<http://blogs.msdn.com/bharry/archive/2007/10/18/tfs-2008-system-recommendations.aspx>
- TFS at Microsoft
<http://blogs.msdn.com/bharry/archive/2006/08/22/712746.aspx>
<http://blogs.msdn.com/bharry/archive/2008/07/03/july-08-devdiv-tfs-dogfood-statistics.aspx>
- How do I ? (Videos)
<http://msdn.microsoft.com/en-us/vsts2008/bb507749.aspx?wt.slv=topsectionsee>
- Ressources Francophones
<http://www.team-system.fr/default.aspx>
- Team System Widgets
<http://widgets.accentient.com/>
- Rapport de tests de charge et comparaison (pour 2008)
<http://www.codeplex.com/loadtestreports>
- Licensing
<http://www.microsoft.com/Downloads/details.aspx?FamilyID=ce194742-a6e8-4126-aa30-5c4e969af2a3&displaylang=en>
- TFS Power Tools
<http://www.microsoft.com/downloads/details.aspx?FamilyId=FBD14EEA-781F-45A1-8C46-9F6BA2F68BF0&displaylang=en>
- Blog de l'équipe de test Microsoft
<http://blogs.msdn.com/vstsqualitytools/>

- Blog de l'équipe de Team Lab de Microsoft
http://blogs.msdn.com/lab_management/
- Test Scribe (outil de génération de documentation de plan de test)
<http://visualstudiogallery.msdn.microsoft.com/en-us/d18873c7-909d-4788-a56e-0c496a1d8bb9>
- Outil pour générer une machine virtuelle prête pour le Lab Management
<http://vslabmgmt.codeplex.com/documentation>

Les tests Logiciels : Une étude de cas avec Visual Studio 2010

Les tests prennent de plus en plus d'importance dans le monde de l'entreprise. Fort de ce constat, Microsoft étend sa suite logicielle Visual Studio en couvrant la totalité du spectre de la technique au fonctionnel. Ce Livre Blanc a pour objectif de faire un état des lieux de l'utilisation des tests en entreprise en détaillant les possibilités qu'offre aujourd'hui cet environnement.

