UFR Sciences et techniques de Franche-Comté

Rapport de stage de
2ème année de Master Informatique
Option Systèmes distribués et réseaux

Maître de stage : Olivier Albiez
Tuteur universitaire : Fabien Peureux

Introduction d'un modeleur open source dans la solution Smartesting

Cyril MOUTENET

Besançon, le 15 juin 2009

Remerciements

Je tiens à remercier toute l'équipe de R&D de Smartesting, chacun de ses membres m'ayant beaucoup apporté. C'est une équipe qui m'a parfaitement accueilli et qui m'a énormément appris sur la cohésion de groupe et le travail dans la bonne humeur.

Merci à toute l'équipe R&D : AMA, CBO, DAD, FLE, JLH, OAL, SBL, SCO, SGR, SMI, RDE. Cela m'a fait très plaisir de partager un binômage avec chacun d'entre eux.

Merci à Fabien Peureux pour son soutien durant le stage.

Et un grand merci à Isabelle et notre petit bout pour leur soutien.

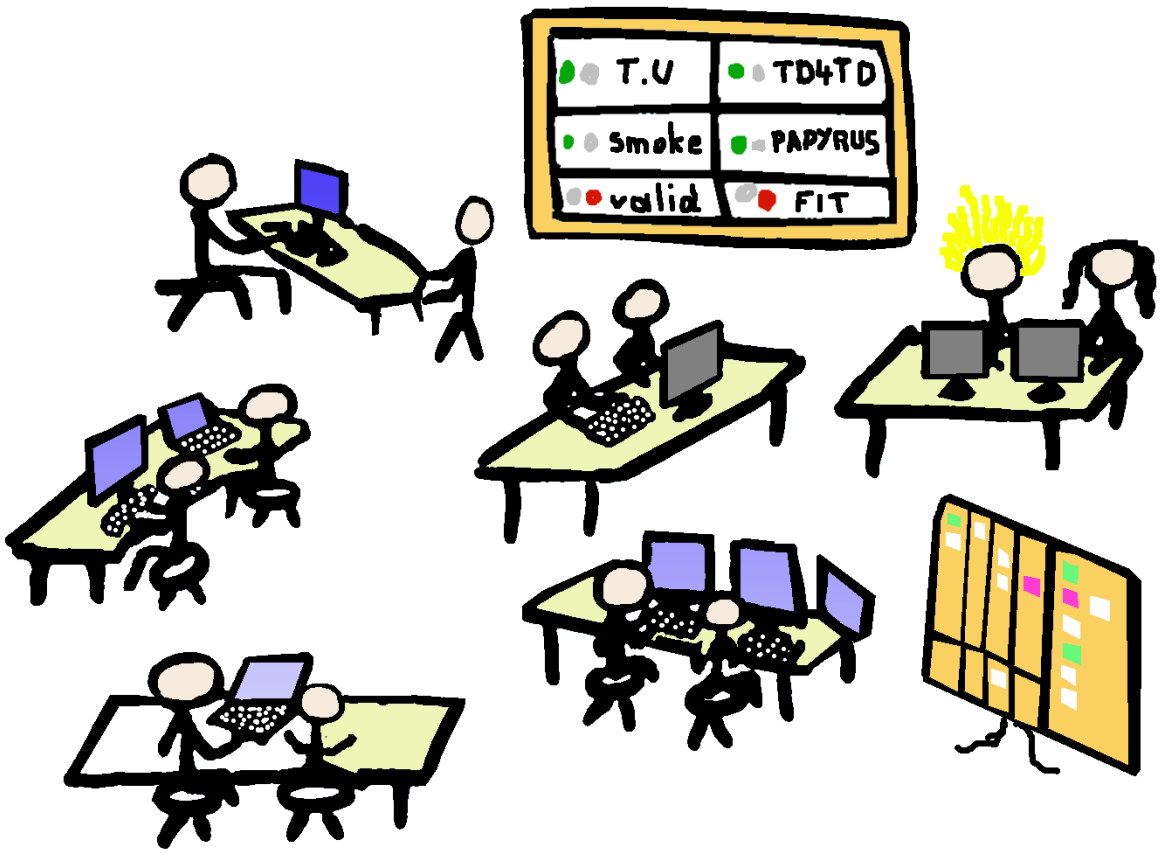


Table des matières

Chapitre 1

Introduction

Il s'agit dans ce chapitre, de présenter la société Smartesting, son produit développé et, le contexte de travail dans lequel le stage de Master Professionnel Informatique option Systèmes Distribués et Réseaux de l'université de franche-comté a eu lieu.

1.1 Présentation de l'entreprise

Le stage s'est déroulé dans la société Smartesting située à Temis (figure 1.1) de Besançon. Il s'agit d'une société née il y a six ans à partir de travaux de recherches menés au LIFC [1] sur la génération de tests. C'est Laurent Py et Bruno Legeard qui, en 2003, ont ainsi créé la société Leirios qui deviendra Smartesting en juin 2008. Celle-ci emploie plus d'une trentaine de personnes dont dix en R&D [2] où mon stage a eu lieu.

Figure 1.1 – *Temis innovation*

Smartesting est un éditeur de logiciel de génération automatique des cas de test à partir d'une modélisation UML des exigences. Smartesting utilise des outils de modélisation du marché basés sur Eclipse. Deux outils de modélisation sont supportés : IBM Rational Software Modeler [3] et Borland Together. L'interface avec Smartesting se fait par l'intermédiaire d'un «plug-in» Eclipse spécifique à Smartesting (cf. figure 1.2 p. 2).

1. LIFC : Laboratoire d'Informatique de Franche-Comté
2. R&D : Recherche et développement
3. RSM ou équivalent comme IBM Rational Software Architect

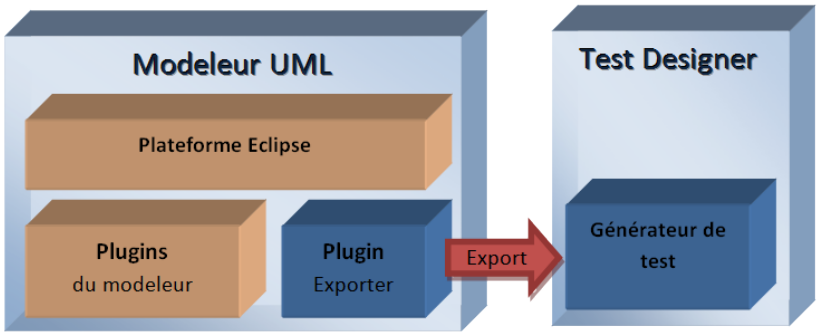


Figure 1.2 – *Solution Smartesting*

La société est organisée en trois services distincts qui sont :
- les commerciaux
- les consultants
- l'équipe R&D

Durant le stage, j'ai rencontré l'ensemble des employés de Smartesting mais j'ai principalement travaillé avec la R&D.

La R&D où s'est déroulé mon stage, était un environnement de développement "Agile". Il s'agit d'une méthode basée sur le développement itératif, où les exigences et les solutions évoluent grâce à la collaboration. La gestion du projet avec les méthodes Agiles encourage les processus d'inspections et de remise en question de l'équipe de façon fréquente. En générale, la philosophie des responsables est l'encouragement de l'équipe, l'auto-organisation et les responsabilités. Ensemble qui pousse l'utilisation de bonnes pratiques de programmation et la mise en place de systèmes de livraison rapide, fiable et en adéquation avec les besoins clients et l'objectif de la société. Ceci étant obtenu grâce à un réseau de communication rapproché et un processus de fabrication sans gaspillage.

1.2 La solution Smartesting

De manière plus précise (cf. 1.3 p.3), à partir des besoins métier et des exigences de test, est réalisé un modèle UML. Ce modèle peut être uniquement réalisé à partir des deux modeleurs RSM et Together. Il s'agit de modeleurs basés sous la plateforme Eclipse. Eclipse est une architecture de plugins qui communiquent les uns avec les autres pour former une application. C'est via ce mécanisme que Smartesting a développé deux plugins d'exportation de modèle pour les deux modeleurs précédemment cités.

Le logiciel "Test Designer" va quant à lui, générer des tests de manière automatique à partir du modèle exporté. Le résultat des tests est stocké dans un référentiel de test de Test Designer. Puis, ces tests vont être publiés dans un environnement de test tiers.

Smartesting peut publier vers plusieurs environnements de gestion de tests. Ces environnements permettent par exemple de suivre l'évolution du test tout au long de son

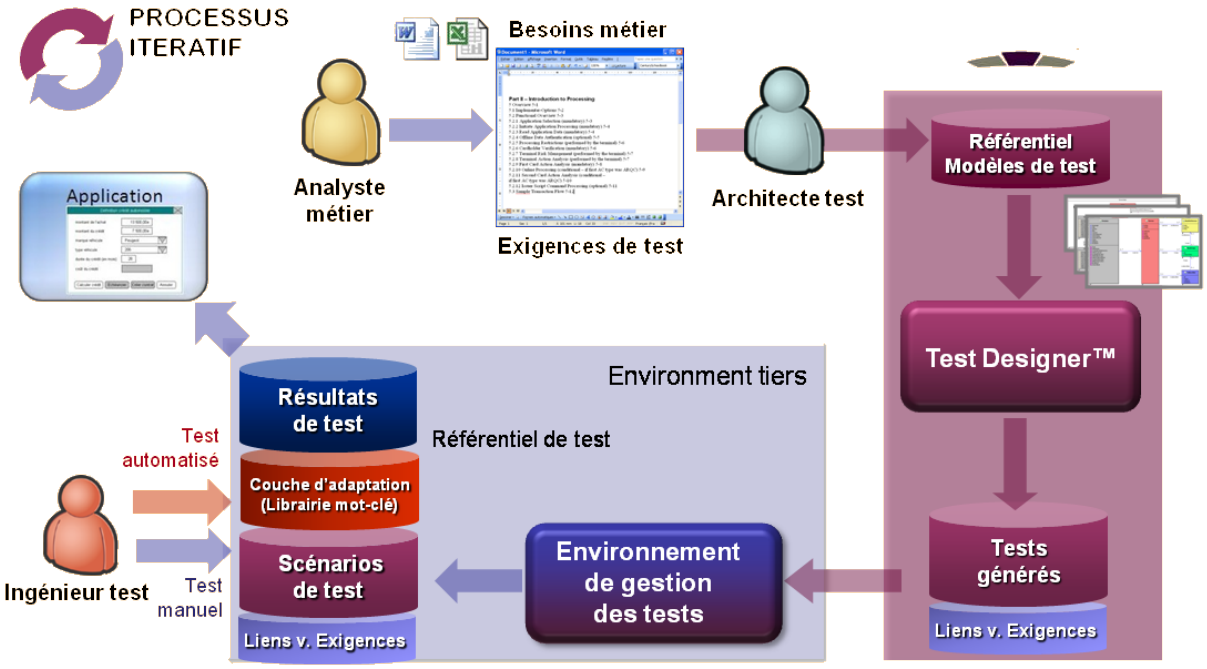


FIGURE 1.3 – *La solution Smartesting complète*

cycle de vie. Ensuite grâce à une couche d'adaptation, l'application modélisée pourra être testée.

Le processus de génération de tests de la solution Smartesting est un processus itératif. C'est à dire que si l'application déjà testée doit évoluée, la prise en compte des nouvelles fonctionnalités, ne générera pas un nouveau coût de génération des tests. Il suffit de modifier le modèle UML de spécifications via le modeleur, puis de générer les tests à nouveau (automatique).

Habituellement la génération de tests est effectuée manuellement par un ingénieur de tests.

1.3 Objectifs

L'objectif de ce stage est d'intégrer un nouveau modeleur, mais cette fois open source, à l'ensemble des modeleurs compatibles avec la solution Smartesting. Et c'est à la suite du projet VETESS [4], dans lequel Smartesting est partenaire, que le choix du modeleur Papyrus a eu lieu.

4. VETESS : Vérification de systèmes embarqués VEhicules par génération automatique de TESts à partir des Spécifications

VETESS : L'objectif stratégique du projet VETESS est de produire des outils conceptuels, méthodologiques et techniques pour la vérification de systèmes mécatroniques embarqués véhicule par génération automatique de tests à partir des spécifications de ces systèmes.

Pour remplir ces objectifs, le projet VETESS s'appuie sur un partenariat fortement complémentaire avec une expertise importante dans l'ingénierie dirigée par les modèles et la génération automatique de tests (figure 1.4 p.4). Il réunit un industriel soucieux de maîtriser la complexité des systèmes grand public (PSA Peugeot Citroën), à une PME innovante (Smartesting) leader dans le domaine du test à partir de modèle et, à un industriel Clemessy spécialisé possédant une offre de premier plan en matière de bancs de test de système électriques et électroniques embarqués. Au niveau académique, les laboratoires impliqués (Université de Haute Alsace – Laboratoire MIPS et Université de Franche-Comté – Equipe LIFC) sont reconnus respectivement dans le domaine de la modélisation logicielle et système, et de la génération de tests.

Figure 1.4 – *Partenaires VETESS*

La solution Smartesting par rapport à VETESS : Il est important que Smartesting puisse proposer sa solution aux entreprises qui souhaitent valider des systèmes embarqués. C'est pour cela qu'intégrer le modeleur choisit dans le projet VETESS est un atout commercial important pour Smartesting. Il s'agit d'offrir la possibilité a ses clients, d'utiliser le modeleur Papyrus comme modeleur de la solution. Sur la figure 1.5 p.4 l'objectif en image, propose trois plugins d'exportation de modèle.

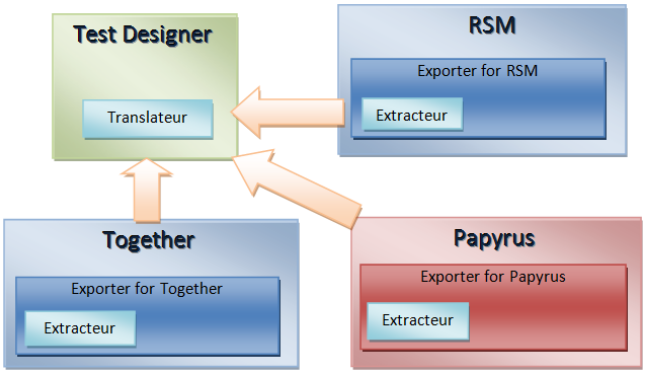


Figure 1.5 – *Objectif : intégration du modeleur Papyrus*

A la différence de la figure 1.5 p.4, il ne s'agit pas de recréer obligatoirement un plugin spécifique à Papyrus. Il faut étudier le modeleur Papyrus afin de constater des éventuels points communs avec les modeleurs existants. C'est à partir de l'étude des points communs avec les autres modeleurs qu'il a été évoqué le besoin de fragmenter l'existant en plugins.

Fragmentation en plugins : La fragmentation en plugins est un objectif secondaire. Cela doit permettre la réorganisation des modules en plugins, afin de pouvoir utiliser du code commun. La "pluginisation" de l'application a posé quelques problèmes techniques. En particulier, la construction du produit appelé le *build* a nécessité un gros chantier d'amélioration.

Le *build* est un terme utilisé pour définir le processus automatisé qui permet de construire et déployer la solution Smartesting sur différentes plateformes telles que Linux et Windows, et sur différents modeleurs comme "Together" , "RSM" et "Papyrus". Pour Smartesting, ce sont des tâches *ant* qui réalisent l'automatisation du processus de construction du produit.

L'amélioration du *build* et la modification de l'infrastructure des modules doivent permettre d'atteindre le genre d'architecture de la figure 1.6 (p.5).

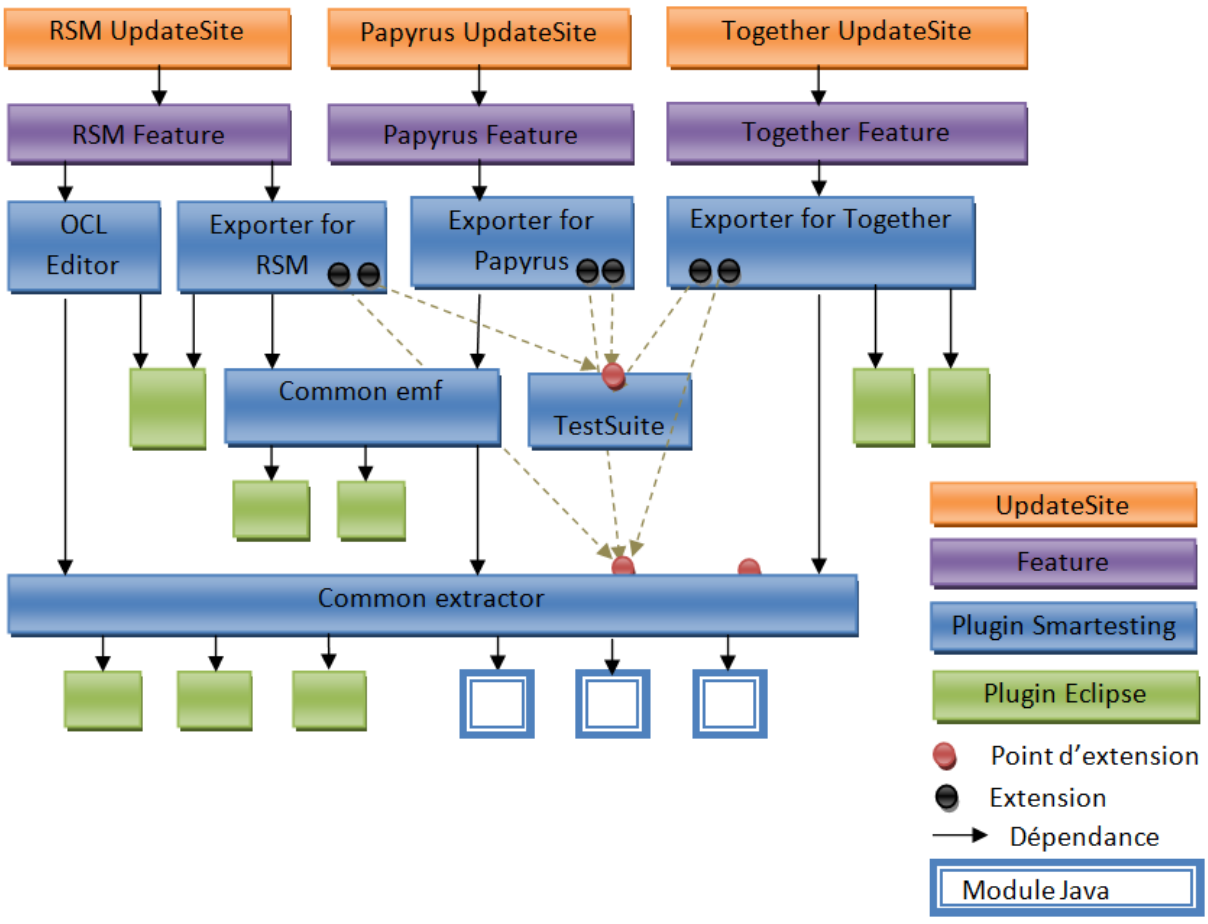


Figure 1.6 – *Réorganisation en plugins*

L'amélioration du *build* doit permettre de créer les plugins mais aussi de les déployer. La figure 1.6 montre les différents modules à déployer avec le *build* :
- Le plugin (en bleu)
- La feature (en violet)
- L'update site (en orange)

Ces trois types de modules permettent de déployer un plugin sur un modeleur type, grâce à un update-site.

La figure 1.6 montre le travail complet de réorganisation de l'architecture de l'application. En outre, l'utilisation des points d'extension a permis de faciliter l'intégration du spécifique dans les modeleurs. Grâce à ce mécanisme, il est possible d'adapter le comportement d'un plugin sans ajouter de dépendances entre le plugin standard et le plugin spécifique. Exemple sur la figure 1.6 p.5 le plugin "Exporter for RSM" utilise le point d'extension du plugin "Common extractor" (point noir). En faisant cela, le plugin "Common extractor" (point rouge) est capable d'interroger le plugin "Exporter for RSM" afin qu'il réalise une extraction spécifique. Cela permet d'éviter de créer une dépendance au plugin spécifique "RSM" . Ceci est aussi appelé une inversion de dépendance.

1.4 Méthode de travail

Le déroulement du stage s'est effectué au sein d'une équipe de développement "Agile" ou eXtrem Programming (XP). Cette méthode valorise la cohésion d'équipe et l'auto-organisation. La fabrication de quelque chose doit toujours être réalisée sans contrainte, dans des conditions dites de "fun", terme extrait d'un livre dont le titre est «Art of Agile development». Il s'agit d'une lecture que j'ai faite durant le stage pour préparer les discussions collectives sur quelques chapitres.

Dès le départ, ce qu'il faut comprendre de la méthode "Agile" est que le processus de développement peut être sans cesse amélioré. Il y a quelques fois un groupe de lecture et chaque semaine une rétrospective qui donne lieu à des discussions. Celles-ci permettent d'améliorer le processus de développement. C'est ainsi que j'ai pu participer à certains changements.

La semaine est organisée en plusieurs événements répartis sur cette dernière et des réunions chaque midi appelées également morning meeting ou standup meeting.

1.4.1 Morning meeting

Chaque midi, l'équipe R&D se réunit avant la pause pour faire le point sur le travail en cours de chacun. Tout le monde peut participer à cette réunion, pas seulement les membres de l'équipe R&D. Cette réunion très courte permet de prendre connaissance du travail de ses collègues. Elle permet également de passer une annonce.

Ensuite, la semaine débute par une rétrospective, genre de bilan d'une semaine de travail, qui représente aussi en méthode XP une itération.

1.4.2 Rétrospective

La rétrospective est une réunion qui dure environ une heure et qui permet de s'exprimer et d'évacuer les frustrations éventuelles ressenties lors de l'itération passée. Chaque semaine un animateur a pour rôle de faire participer tous les employés R&D et de time boxer [5] la réunion. Les stagiaires ni sont pas exempts.

Enfin, le travail de la semaine est planifié par le client XP à la fin de la rétrospective. Éventuellement, celui-ci peut demander une séance de cotation pour estimer le coût de développement d'une ou plusieurs fonctionnalités appelées aussi fiches blanches.

1.4.3 Séance de cotation

Le travail est organisé à partir de fiches représentant le travail à réaliser. Il peut s'agir d'une fonctionnalité à développer, d'un bug à corriger ou d'un travail d'exploration à effectuer.

Une fois les fiches cotées, le client XP peut proposer le travail à faire pour l'itération lors de l'engagement.

1.4.4 L'engagement

Le client XP dispose sur un tableau les fiches à réaliser par l'équipe de développement en fonction de l'effort que peut produire celle-ci. Cet effort est un nombre de points définis par la R&D. Il s'agit de la vélocité, le client XP ne peut dépasser cette valeur.

Chez Smartesting, un point correspond au travail effectué durant une demi-journée sur une fiche sans interruption.

1.4.5 Pair Programming

Le "Pair programming" est une pratique qui consiste à réaliser une tâche par binôme. Ce procédé permet une meilleure qualité du travail accompli, de prendre de meilleures décisions, d'aller plus vite que tout seul et surtout il permet d'éviter de rendre quelqu'un indispensable.

Le travail en paire a déjà été pratiqué à l'Université lors de projets. Mais cela prend une autre dimension en entreprise lorsque l'on travaille avec quelqu'un qui connaît bien son sujet et qui nous l'explique. Très vite, on connaît le principe de fonctionnement de l'application et l'on peut commencer à participer aux cotations de fiches.

5. Time boxé : Expression pour dire par exemple que l'on contrôle le temps de la réunion

1.4.6 Batman et Robin

Le batman est le nom donné à la personne qui joue le rôle de superviseur de l'itération. Il est là pour surveiller le déroulement des fiches. Il discute avec les développeurs afin de vérifier ce qui va être réalisé ou ce qui est en cours de réalisation.

Il peut poser des questions au client XP à la place des développeurs. Il peut aussi demander au client XP d'aller voir les développeurs pour ré-expliquer ce qu'il attend. Avec le client XP, il peut préparer la démonstration pour la rétrospective de l'itération.

Il est associé au veilleur (Robin) qui aide certain binôme dans la réalisation de leur fiche. Le veilleur peut être amené à faire rencontrer des développeurs, car il a jugé que cela pourrait les aider à mieux avancer dans leur travail.

Au delà du côté humain de la méthode de travail "Agile", il y a toute l'infrastructure de développement du logiciel. Celle-ci permet de garantir la fiabilité et la non régression du logiciel développé.

1.4.7 Intégrateur continu

L'intégration continue est un élément essentiel gage de qualité fonctionnelle du programme développé. L'intégrateur est chargé d'exécuter en continu plusieurs tâches :

- Les tests unitaires
- Les tests fonctionnels (Test fit)
- Les tests haut niveau (High level)
- Les tests smokes

La durée de détection d'un bug varie suivant la tâche exécutée. Les tests sont répartis sur plusieurs ordinateurs pour être exécutés en parallèle.

Les tests unitaires sont les plus rapides à détecter les erreurs de bas niveau. Pour arriver jusqu'aux tests dit smoke qui permettent de vérifier que l'application déployée fonctionne correctement. Exemple, ce test peut détecter un problème de déploiement de librairie ou l'utilisation d'une librairie non compatible avec tel ou tel modeleur.

Pour avoir une plus grande réactivité, un panneau lumineux représentant les grandes catégories de test définies plus haut, a été installé pour visualiser plus rapidement un test en échec. Ensuite, c'est à chacun de s'autogèrer pour réparer au plus tôt le problème.

1.5 Synthèse

L'objectif du stage dans la société Smartesting est principalement l'intégration du modeleur open source papyrus dans la solution Smartesting. Il y a cependant un autre objectif secondaire, qui est la fragmentation de l'existant en plugins. Ces deux objectifs sont étroitement liés, puisque sans la fragmentation en plugin, il n'est pas possible

de créer un nouveau plugin pour Papyrus sans duplication de code. On remarque aussi que la fragmentation en plugin a généré des problèmes sur la construction du produit Smartesting.

En fait, l'objectif principal et secondaire de ce stage, sont aussi des objectifs pour Smartesting. Grâce à la fragmentation par exemple, Smartesting se rapproche d'avantage d'une intégration complète dans Eclipse et donne la possibilité de développer d'autres plugins plus aisément.

Voyons maintenant dans le second chapitre, comment le modeleur Papyrus fonctionne et quelles améliorations ont permis la fragmentation en plugins.

Chapitre 2

Le modeleur open source

L'intégration d'un plugin Test Designer pour Papyrus a nécessité une phase d'exploration. Celle-ci a permis de mettre en évidence des correspondances entre les librairies RSM et Papyrus. A la suite de cette exploration, un certain nombre de décisions concernant la réorganisation de module en plugin a été planifié.

L'amélioration du *build* a fourni une aide précieuse dans le déploiement d'autres plugins. Cela a contribué à la fragmentation en plugins.

Malgré les ressemblances au modeleur RSM, certaines spécificités ont du être développées pour le modeleur Papyrus. A la suite de quoi, j'ai étudié le mécanisme de point d'extension d'Eclipse.

2.1 Étude du modeleur Papyrus

Cette partie traite de l'exploration menée sur le modeleur Papyrus. Elle permet de mettre en évidence les actions à mener, pour créer un plugin exporter pour Papyrus. Aux vues des premières constatations sur l'ensemble des plugins utilisés par le modeleur. La ressemblance avec le modeleur RSM fut constatée.

Papyrus est un modeleur en cours de développement par le CEA [1]. Il est actuellement livré dans sa version 1.11 mais la version 2 est en cours de développement. Avec ce modeleur certaines fonctionnalités n'ont tout simplement pas été développées.
Pour étudier le modeleur Papyrus, j'ai téléchargé le code source du modeleur [2].

Fonctionnalités non fonctionnelles :
- Impossibilité de réaliser des transitions internes dans un diagramme d'état.
- Initialisation automatique des liens entres deux instances d'objet.
- Impossible de définir des stéréotypes manuellement.

1. Commissariat à l'Energie Atomique
2. Disponible sur https ://speedy.supelec.fr/Papyrus/svn/Papyrus/core/releases/1.11.0/

CEA : C'est un acteur majeur en matière de recherche, de développement et d'innovation, le CEA intervient dans trois grands domaines :
- l'énergie
- les technologies pour l'information et la santé
- la défense et la sécurité

2.1.1 Points communs

Les premières constatations du modeleur ont mis en évidence des points communs avec le modeleur RSM. Papyrus utilise le même framework de stockage du modèle : Emf [3].

Lors de l'exploration, j'ai utilisé le plugin d'exportation d'RSM. Cela m'a permis de montrer une grande compatibilité avec ce modeleur et la possibilité de réutiliser le code d'extraction d'un modèle RSM dans Papyrus. Pour réutiliser ce code, j'ai envisagé de fragmenter le plugin "RSM Exporter", et créer un autre plugin "Emf Exporter". Cela a pour but de garder uniquement le code spécifique à chaque modeleur dans leur plugin respectif.

Il n'y a toute fois pas que des points communs.

2.1.2 Différences

Manque d'utilitaires : Comme Papyrus est un modeleur incomplet, il manque d'utilitaires qui permettent de faciliter l'exportation du modèle UML. Dans RSM, il y a par exemple, une classe qui retourne la liste des modèles ouverts. Ainsi à partir de celle-ci et du projet en cours de sélection, le modèle du projet sélectionné est récupéré.

Dans l'interface graphique quelque soit la version d'RSM, il faut ouvrir au moins un élément pour pouvoir accéder au modèle. Comme on peut le voir sur la figure 2.1 p.11, il y a une différence entre projet ouvert et fermé. Tant que le modèle n'a pas été chargé en mémoire, il est impossible d'en extraire les données pour les exporter vers "Test Designer". C'est une particularité du modeleur que n'a pas "Together".

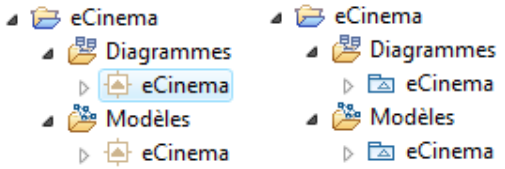


Figure 2.1 – *Projet non ouvert / Projet ouvert*

3. Editing Modeling Framework

En fait les données du projet RSM sont stockées dans un fichier “emx”. Ce fichier est un document XMI [4] utilisé par le framework Emf pour y stocker les données du modèle. Il s'agit d'un framework Java et un générateur de code pour les applications basées sur les modèles structurés. Il permet de prendre en compte toutes les informations contenues dans les diagrammes UML [6].

Edition du modèle : Quelque soit le modeleur, il existe une fonctionnalité dans l'exporter de modèle qui permet de définir un filtre sur des suites du modèle. La particularité de cette propriété, est qu'elle est enregistrée dans le modèle UML. Ainsi dans RSM, on utilise le framework de stockage des éléments UML pour y stocker une propriété supplémentaire sur l'élément suite.

Le principe utilisé dans RSM pour modifier le modèle, ne peut être utilisé de façon identique dans Papyrus. Le manque d'utilitaire empêche la réalisation simple du même procédé. C'est à la suite de la découverte de cette différence que j'ai étudié la procédure d'édition du modèle d'RSM (paragraphe 2.1.4 ”Edition du modèle UML d'RSM” p.13).

Parmi les fonctionnalités du plugin d'exportation seul deux fonctionnalités modifient l'état du modèle :

- Le filtre sur les suites. Une suite est un package UML, qui spécifie l'état initial du modèle de génération de tests.
- L'éditeur OCL [7]. Il s'agit d'un plugin développé pour RSM qui permet d'éditer plus facilement une garde et un effet sur une transition dans un diagramme d'état. Il offre notamment la complétion [8] et la coloration syntaxique [9].

2.1.3 Structure d'un projet Papyrus

L'étude de la struture du projet Papyrus a aidé à la récupration du modèle.

Comme on peut l'observer sur la figure 2.2 p.13, un projet Papyrus est composé de deux fichiers. Le fichier `.project` appartient à la plateforme Eclipse.

- Fichier `.di2` : Contient les données graphiques des diagrammes de celui-ci. Il s'agit d'un format Xmi comme RSM mais avec un élément `<di2>` spécifique.
- Fichier `.uml` : Celui-ci contient les données du modèle UML enregistrées au format Emf (XMI).

Pour récupérer les données du modèle Emf, il faut utiliser les utilitaires d'Emf pour charger le fichier `.uml` afin d'en extraire les données UML. C'est à partir de ces données que l'exportation du modèle Emf vers “Test Designer” est effectuée.

4. XML [5] Metata Interchange
6. Unified Modeling Language
7. *O*bject *C*onstraint *L*anguage
8. Complément automatique
9. Repésentation des mots-clés sous une police différente pour les mettre en évidence

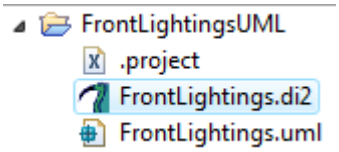


Figure 2.2 – *Fichiers d'un projet Papyrus*

A partir de ces données est-il possible de les modifier ?

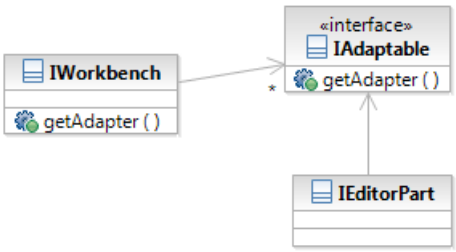
2.1.4 Edition du modèle UML

En l'état, un modèle Emf peut être modifié par des méthodes de modification du modèle. Hors ce procédé ne peut fonctionner correctement. Sur un modèle Emf, il est donc obligatoire de passer par un mécanisme centralisé d'édition du modèle. Ce mécanisme a pour but de modifier le modèle de façon transactionnel. Toute modification de la propriété d'un élément est réalisée de façon atomique. Si jamais la modification devait échouée en cours de transaction, le modèle reviendra à l'état précédant.

Dans Emf, il est appelé "EditingDomain". Les modeleurs sont libres de l'utiliser comme ils le souhaitent. Il faut de toute façon pouvoir récupérer l'"editingDomain" des modèles ouverts. C'est ce que permet de faire RSM via un utilitaire.

2.1.5 Récupération d'un modèle

Papyrus utilise le mécanisme d'adaptation d'Eclipse. Ce mécanisme permet de retourner un objet instancié par un éditeur ou un objet `IAdaptable`. Dans Eclipse, les objets adaptables sont par exemple, les éditeurs et les ressources du projet.



Tous les éditeurs implémentent `IEditorPart`. Ainsi tous les éditeurs qui surchargent la méthode `getAdapter(Class class)` de l'interface `IAdaptable` permettent de retourner un objet de la classe passée en paramètre.

Figure 2.3 – *Diagramme de classe d'un éditeur*

C'est ce mécanisme qui a été utilisé dans Papyrus pour récupérer le modèle ouvert. Exemple, à partir du "Workbench" qui est l'espace de travail d'Eclipse. Il est possible de demander le modèle UML en faisant :

```
Model theModel = IWorkbench.getAdapter(org.eclipse.uml2.uml.Model.class)
```

Autre exemple si l'on souhaite récupérer un "editingDomain", il faudra utiliser :

```
EditingDomain editingDomain = IWorkbench.getAdapter(EditingDomain.class)
```

Il y a toute fois un inconvénient à cette solution. Si aucun éditeur n'est ouvert, il n'y a pas de modèle récupérable.

2.1.6 Les stéréotypes

Les stéréotypes sont une fonctionnalité qui a posé problème. Cette partie a pour but d'expliquer, à quoi ils servent.

Les stéréotypes sont des propriétés des objets UML, ils permettent de spécifier le comportement de ceux-ci. Dans notre cas, d'autres ont été ajoutés pour permettre de spécifier par exemple le comportement d'une opération (figure 2.4 p.14). En spécifiant "observation" sur l'opération "myOperation" cela signifie que la post condition qui détermine l'état final après exécution de l'opération, ne modifie pas l'état du modèle. Et celle-ci doit retourner une valeur. Concernant la vérification des stéréotypes, cela est effectué via un contrôle d'erreur au moment de l'exportation du modèle. La post-condition est une propriété de l'opération écrit en OCL.

Dans RSM les stéréotypes sont gérés par une extension spécifique au modeleur, qui permet d'ajouter d'autres stéréotypes que ceux standards (interface, abstract, enumeration, ...), par exemple :

- observation
- setup
- teardown

Une extension est un mécanisme qui permet d'étendre les fonctions du modeleur à condition que celui-ci l'accepte.

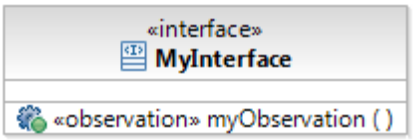


Figure 2.4 – *Exemple de stéréotype dans RSM*

Quel bilan peut on faire à partir des constations suivantes ?

2.1.7 Résultat

Le bilan de cette étude montre en effet que Papyrus est proche d'RSM. Toute fois, il est nécessaire que la fragmentation soit effectuée avant de factoriser le code commun pour les deux modeleurs.

Maintenant, concernant les différences et les problèmes constatés en réalité, ceux-ci ne posent aucun problème. Il y a en effet dans les objectifs de jalon suivant, une modification sur la gestion des suites qui doit supprimer les modifications du modèle. Et Smartesting souhaite d'autre part ne plus rien enregistrer dans le modèle.

Pour l'éditeur OCL, il s'agit d'un plugin qui n'est destiné qu'à RSM. En sachant que Papyrus n'a pas besoin d'éditer plus facilement une "garde" et un "effet" car cela l'est suffisamment [10].

Pour la récupération du modèle, il a été fait le choix de récupérer le modèle à partir du fichier `.uml`. La manière qui utilise le mécanisme d'adaptation d'Eclipse ne correspond pas à l'attente fonctionnelle du client XP.

2.2 Problèmes rencontrés

Durant l'exploration, un certain nombre de problèmes et de questions ont été soulevés. Pour certains, il s'agit tout simplement de l'incapacité du modeleur à modéliser pour le test. Dans d'autres cas, il s'agit d'un fonctionnement qui engendre des erreurs dans le modèle.

Liste des problèmes :
- Problème sur les instances de lien dans le modèle
- Ajout de nouveaux stéréotypes
- Gestion des ranges sur les entiers
- Problème d'extraction des commentaires

2.2.1 Problème sur les instances de lien dans le modèle

Lorsque l'on souhaite créer un lien entre deux instances dans Papyrus, une erreur non explicite est détectée au moment de l'exportation du modèle. Après une recherche de la cause possible de cette erreur, nous avons fini par en déduire la cause d'une mauvaise construction du lien dans Papyrus. Nous avons constaté que les slots du lien en question, ne sont pas créés automatiquement contrairement à ceux du modeleur RSM.

La figure 2.5 p.16 est une capture du modeleur Papyrus qui met en évidence le problème lié à la création de lien d'instance (1). On peut voir entouré en rouge (2) que les slots ne sont pas initialisés. Pour cela, il faut les initialiser manuellement pour arriver au résultat encadré en orange (3).

C'est via ce problème que nous avons mis en évidence un manque de clarté de certains messages d'erreurs pas assez explicites du genre `NullPointerException` ou `IndexOutOfBoundException` qui ont été corrigés pour donner le résultat dans l'encadré en pointillés rouges (4) figure 2.5 p.16.

2.2.2 Gestion des ranges sur les entiers

La fonctionnalité de range d'entier est réalisée à partir d'un point d'extension spécifique à RSM. Cette fonctionnalité n'existe pas dans le modeleur Together. Cette fonctionnalité offre la possibilité de paramétrer graphiquement le range représentant un entier. Lors de la

10. Trois cliques de souris avec Papyrus contre une dizaine avec RSM

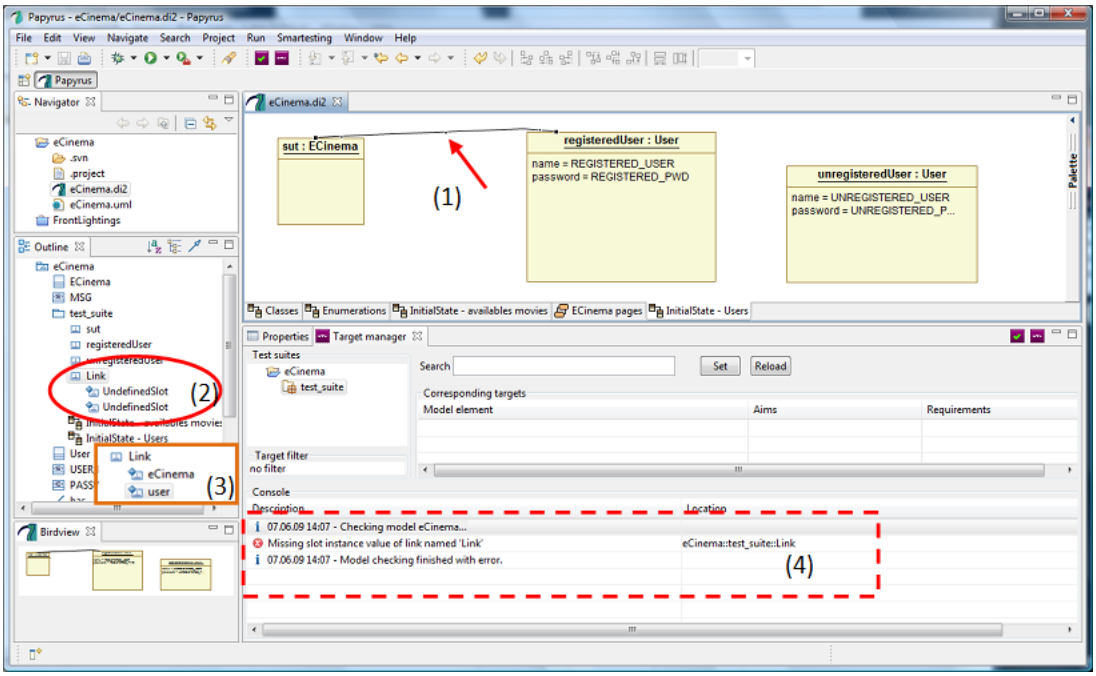


FIGURE 2.5 – *Modeleur Papyrus - Problème sur le lien entre instances*

modélisation pour les tests, les valeurs des paramètres d'entrée d'une opération génèrent autant de cas de tests que de combinaisons de valeurs possibles de ces paramètres. Pour diminuer le nombre de tests, on utilise des ranges sur les entiers. A noter qu'il est possible de limiter les valeurs prises par le paramètre entier, via la pré-condition de l'opération. Ce qui ne constitue donc pas un problème pour Papyrus.

2.2.3 Problème d'extraction des commentaires

La gestion des commentaires dans RSM et Papyrus est différente. Il faut donc gérer cette fonctionnalité comme spécifique dans les deux modeleurs. La récupération est bien effectuée sur le même objet EMF, mais un filtre pour RSM est appliqué pour récupérer la documentation texte ou HTML saisie par l'utilisateur ne correspond pas pour Papyrus. Il est donc nécessaire de développer spécifiquement deux extracteurs de commentaires, un pour RSM et un pour Papyrus.

2.2.4 Comment est géré le spécifique de chaque modeleur ?

L'adaptation spécifique à chaque modeleur est réalisée au moment du clique sur le bouton "check" et "export". C'est l'endroit de plus haut niveau où il est permis de spécifier les objets spécifiques au modeleur. Exemple :

```
EclipseTranslatorUtils.translateAndCheckModel(
                selection.project,
                selection.umlModel,
                new RsmModelExtractor(),
                new RsmTestSuiteTranslator(),\ldots);
```

Sur l'exemple ci-dessus, la méthode "translateAndCheckModel" réalise l'extraction du modèle. Celle-ci prend en paramètre le projet sélectionné et le modèle Emf parce qu'il s'agit d'RSM. Pour "Together" , c'est un objet `Emfapi`. Ensuite la méthode d'extraction est réalisée spécifiquement par passage de l'utilitaire d'extraction "RsmModelExtraction" en paramètre.

Pour pouvoir réutiliser le code d'RSM pour l'adapter à Papyrus nous avons généralisé l'extracteur d'RSM en "EmfModelExtractor" avec spécification de la méthode de récupération des descriptions dans un driver. Ce qui donne :

```
return EclipseTranslatorUtils.translateAndCheckModel(
                selection.project,
                selection.umlModel,
                new EmfModelExtractor(new RsmExtractorDriver()),
                new EmfTestSuiteTranslator(new RsmExtractorDriver()),
```

En réalisant cette généralisation, il est possible d'ajouter des traitements spécifiques dans le driver d'extraction.

2.3 La plateforme Eclipse

Avant de parler de fragmentation en plugins, il est important de définir le fonctionnement d'Eclipse.

2.3.1 Historique

Créée en 2001, le but du projet Eclipse était de fournir un socle pour la création d'environnements de développement.

En 2004, lors du lancement d'Eclipse RCP [11], l'objectif du projet a été étendu pour prendre en compte l'utilisation du framework Eclipse par des applications clientes. Au départ, Eclipse était conçu pour créer un environnement de développement Java autour duquel aurait été construit d'autres applications. Finalement, la base est conçue comme un framework utilisable pour des développements d'applications clientes dîtes riches.

2.3.2 Principe

Eclipse est une plateforme composée de plugins qui interagissent les uns avec les autres (figure 2.6 p.18). La plateforme Eclipse est fournie avec un framework minimaliste appelé "Runtime". Contrairement à une application extensible qui est composée d'une gros framework commun. Autour de ce framework d'autres plugins sont déployés afin de permettre la création d'une application riche. Exemple :

- SWT : création de composants graphiques
- JFace : création de composants plus riches basés sur SWT

11. Rich Client Platform

– Workbench : gestion de l'environnement de travail (menu, action, ...)

Cette plateforme basée sur l'extensibilité est réalisée grâce au mécanisme de points d'extension.

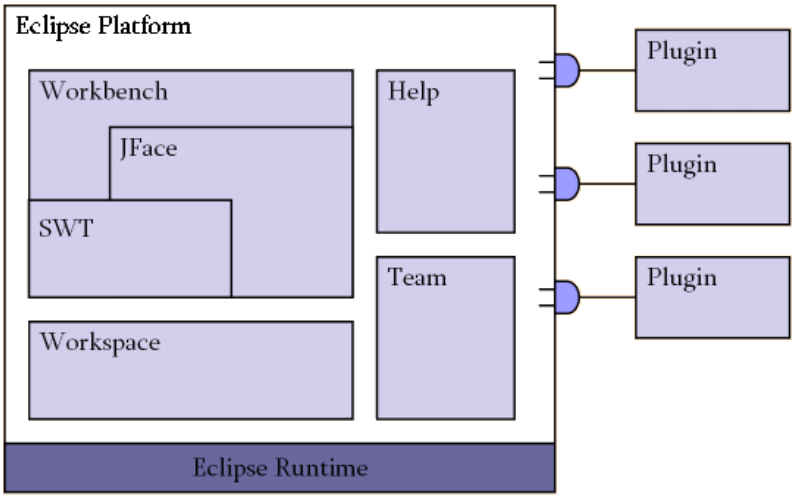


Figure 2.6 – *Diagramme représentant une plateforme Eclipse*

Le framework de base sert de conteneur pour les extensions. Toutes les fonctionnalités sont développées dans des «plug-in» (Bundles). Cela offre l'avantage d'être ouvert et transparent. Il est par exemple plus facile de remplacer une fonctionnalité par une autre.

Une application basée sur Eclipse utilise le registre d'extension et les services OSGI [12]. Ce standard permet de gérer les dépendances de plugin et l'extensibilité de l'application. Il évite aussi le couplage des modules Java. Pour modulariser l'application, il suffit d'utiliser les "features" Eclipse.

Une feature Eclipse permet de déployer des plugins différemment suivant le client ou suivant la plateforme cible.

2.3.3 Déploiement d'un plugin sur une plateforme Eclipse

La plateforme Eclipse implémente le mécanisme de mise à jour d'une application que l'on retrouve sur quasi tous les logiciels. Dans Eclipse un plugin peut être déployé par un "update-site".

En réalité, un "update-site" déploie des "features" composées de "plugins" (figure 2.7 p.19). Le mécanisme d'installation, vérifie en fonction des plugins requis par la feature à installer si la plateforme Eclipse les possède. En cas d'échec les plugins ne sont pas installés.

12. Open Services Gateway initiative

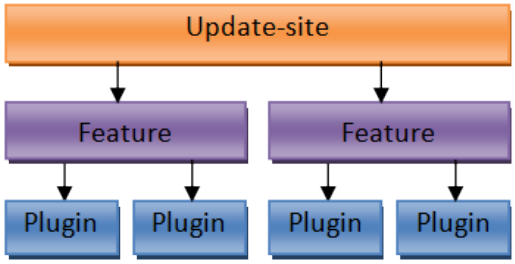


Figure 2.7 – *Architecture de déploiement d'un plugin sur la plateforme Eclipse*

2.3.4 Structure

Cette partie définie la composition des différents éléments Eclipse : Plugin et feature.

Feature

Une feature est gérée par un fichier `feature.xml` qui contient au format XML, la liste des plugins composants la fonctionnalité.

Plugin

L'administration des plugins est gérée par OSGI, qui utilise le fichier `MANIFEST.MF` contenu dans la structure du plugin. C'est ce fichier qui permettra de gérer les dépendances de plugins.

Quant à Eclipse, la plateforme utilise entre autre le fichier `plugin.xml`. Ce fichier contient les données XML de construction de l'interface utilisateur. On peut y déclarer des actions, des menus, des vues, des éditeurs, ...

OSGI

Eclipse utilise le framework OSGI, qui permet de gérer le chargement des classes, l'administration des plugins et les dépendances entre eux. Pour généraliser, ce framework Java gère le cycle de vie d'une application, les services, un environnement d'exécution et des modules.

2.4 Le build

La tâche de *build* concerne un processus automatisé qui permet de réaliser des opérations de compilation et de déploiement pour construire par exemple les versions distribuées par Smartesting à ses clients. Ce processus a été modifié afin de pouvoir fragmenter les plugins existants.

Au début du stage, le *build* permettait de déployer deux plugins pour les deux modeleurs RSM et Together. L'objectif fut donc d'entreprendre la modification de celui-ci pour

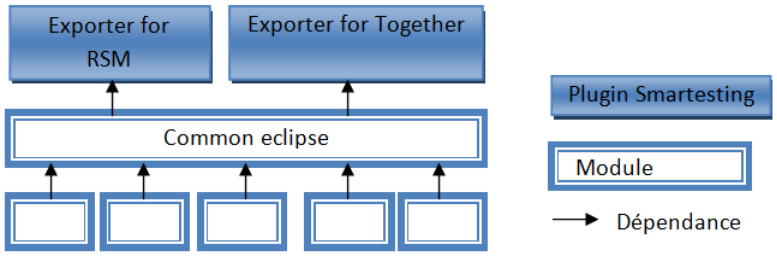


FIGURE 2.8 – *Projet avant amélioration du build*

faciliter la création et le déploiement d'autres plugins, en vue de progressivement passer des modules Java en plugins pour Eclipse.

Sur la figure 2.8 p.20, les modules de type plugin sont gérés spécifiquement pour être déployés via un "update-site".

2.4.1 Objectifs

L'un des objectifs de l'amélioration du *build* est de pouvoir créer des plugins Eclipse et les déployer. La stratégie doit permettre de différencier plusieurs types de modules :

- Update-site
- Feature
- Plugin (Smartesting)
- Module Java
- Librairie Java
- Plugin Eclipse

La figure 2.9 p.21 montre l'amélioration apportée au *build* . On peut remarquer la diversité des types de modules et des relations de dépendance. La gestion des dépendances qui est aussi un enjeu puisqu'il faut que le *build* construise les fichiers de configuration des plugins.

2.4.2 Fonctionnement du *build*

Le développement de la solution Smartesting est réalisé grâce à l'éditeur Java, IntelliJ [13] de JBrain. Il s'agit d'un éditeur payant dont le fonctionnement est proche de l'éditeur Java fonctionnant sous Eclipse. Cet éditeur offre de nombreuses fonctionnalités qui rendent le développement plus facile.

Le *build* mis en place par Smartesting, utilise les fichiers d'IntelliJ pour gérer les dépendances entre les modules. Il y a donc une tâche *ant* qui a été développée par Smartesting pour réaliser cela.

13. IntelliJ ou IDEA

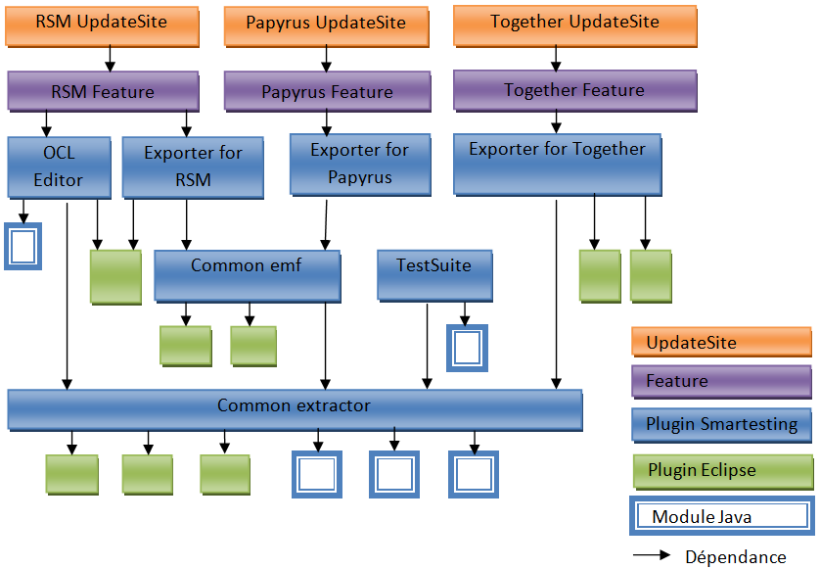


Figure 2.9 – *Projet après amélioration du build*

Génération du manifest :

La génération du manifest du plugin a été effectuée à partir d'un fichier manifest maintenu par le programmeur. Il est situé dans les sources du module plugin d'IntelliJ. On y trouve la description du plugin et certains paramètres. A l'intérieur de ce fichier, la version, le nom et les librairies Java sont générés automatiquement.

Exemple :

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Exporter Plug-in
Bundle-SymbolicName: @plugin.identifier@;singleton:=true
Bundle-Version: @plugin.version@
Bundle-Vendor: SMARTESTING
Bundle-RequiredExecutionEnvironment: J2SE-1.5,
 JavaSE-1.6
Bundle-Classpath: .@plugin.runtime.manifest@
Require-Bundle: com.ibm.xtools.modeler;visibility:=reexport,
 com.ibm.xtools.modeler.ui;visibility:=reexport,
 \ldots
 com.smartesting.eclipse.emf;visibility:=reexport,
 com.smartesting.testsuite
Bundle-Activator: com.smartesting.ltd.eclipse.common.plugin.SmartestingPlugin
Bundle-Localization: plugin
Bundle-ActivationPolicy: lazy
Export-Package: com.smartesting.ltd.eclipse.rsm7.testsuite,
 com.smartesting.ltd.eclipse.rsm7.translator.extractor
```

Dans cette exemple on y retrouve encadré par les @ les informations générées automatiquement. Les autres informations sont spécifiques à chaque plugin.

Résultat de la génération :

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Exporter Plug-in
Bundle-SymbolicName: com.smartesting.rsm.exporter;singleton:=true
Bundle-Version: 1.0.0.44657
Bundle-Vendor: SMARTESTING
Bundle-RequiredExecutionEnvironment: J2SE-1.5,
 JavaSE-1.6
Bundle-Classpath: .,
 lib/commons-lang-2.3.jar
Require-Bundle: com.ibm.xtools.modeler;visibility:=reexport,
 com.ibm.xtools.modeler.ui;visibility:=reexport,
 \ldots
```

```
 com.smartesting.eclipse.emf;visibility:=reexport,
 com.smartesting.testsuite
Bundle-Activator: com.smartesting.ltd.eclipse.common.plugin.SmartestingPlugin
Bundle-Localization: plugin
Bundle-ActivationPolicy: lazy
Export-Package: com.smartesting.ltd.eclipse.rsm7.testsuite,
 com.smartesting.ltd.eclipse.rsm7.translator.extractor
```

2.5 Création d'un point d'extension Eclipse

Pour introduire des fonctionnalités spécifiques à un modeleur, j'ai étudié la possibilité d'utiliser un point d'extension d'Eclipse pour gérer les parties spécifiques du modeleur.

2.5.1 Principe

Le principe d'un point d'extension est le suivant. Eclipse est une grosse boîte sur laquelle on peut se brancher, mais pas seulement (cf. figure 2.10). Il s'agit aussi de plusieurs centaines de petites boîtes reliées les unes aux autres qui peuvent être connectées par un plugin pour le modeleur Papyrus. Les plug-ins ou (bundles) fournissent des points d'extensions.

Il faut imaginer qu'il s'agit d'une multiprise de courant où l'on peut brancher autant de connexions (extensions) que ne peut supporter la multiprise (un point d'extension).

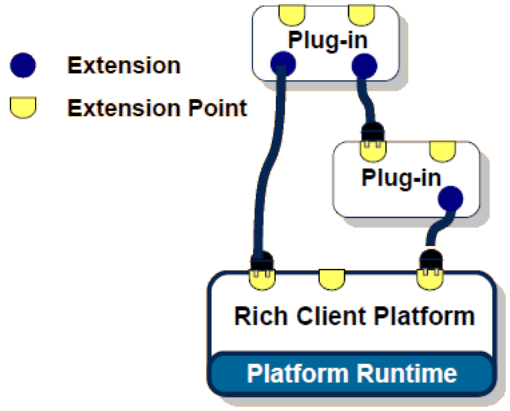


Figure 2.10 – *Fonctionnement d'un point d'extension*

La première chose effectuée au moment du chargement d'un plugin, c'est les scans des metadata de tous les plugins, soit le fichier `plugin.xml`. Ceci n'étant effectué uniquement si les informations dans le cache d'Eclipse ne sont plus à jour, corrompues ou forcées (option -clean).

Si on exploite correctement les points d'extensions, il est possible de construire une interface graphique sans charger le plugin avec juste les metadata. Charger un plugin prend évidement de la mémoire et allonge le temps de démarrage d'Eclipse.

Que se passe t'il lorsqu'un point d'extension est chargé ? Tout d'abord le fournisseur d'un point d'extension déclare celui-ci au registre d'extension. Ensuite chaque plugin utilisant une extension s'identifie auprès du registre d'extension. Ces deux tâches sont automatiques, elles sont réalisées à partir du contenu des metadata (`MANIFEST.MF` et `plugin.xml`).

A partir de là, le fournisseur du point d'extension peut demander à instancier la classe déclarée par l'utilisateur du point d'extension (cf. diagramme de 2.11 p. 23).

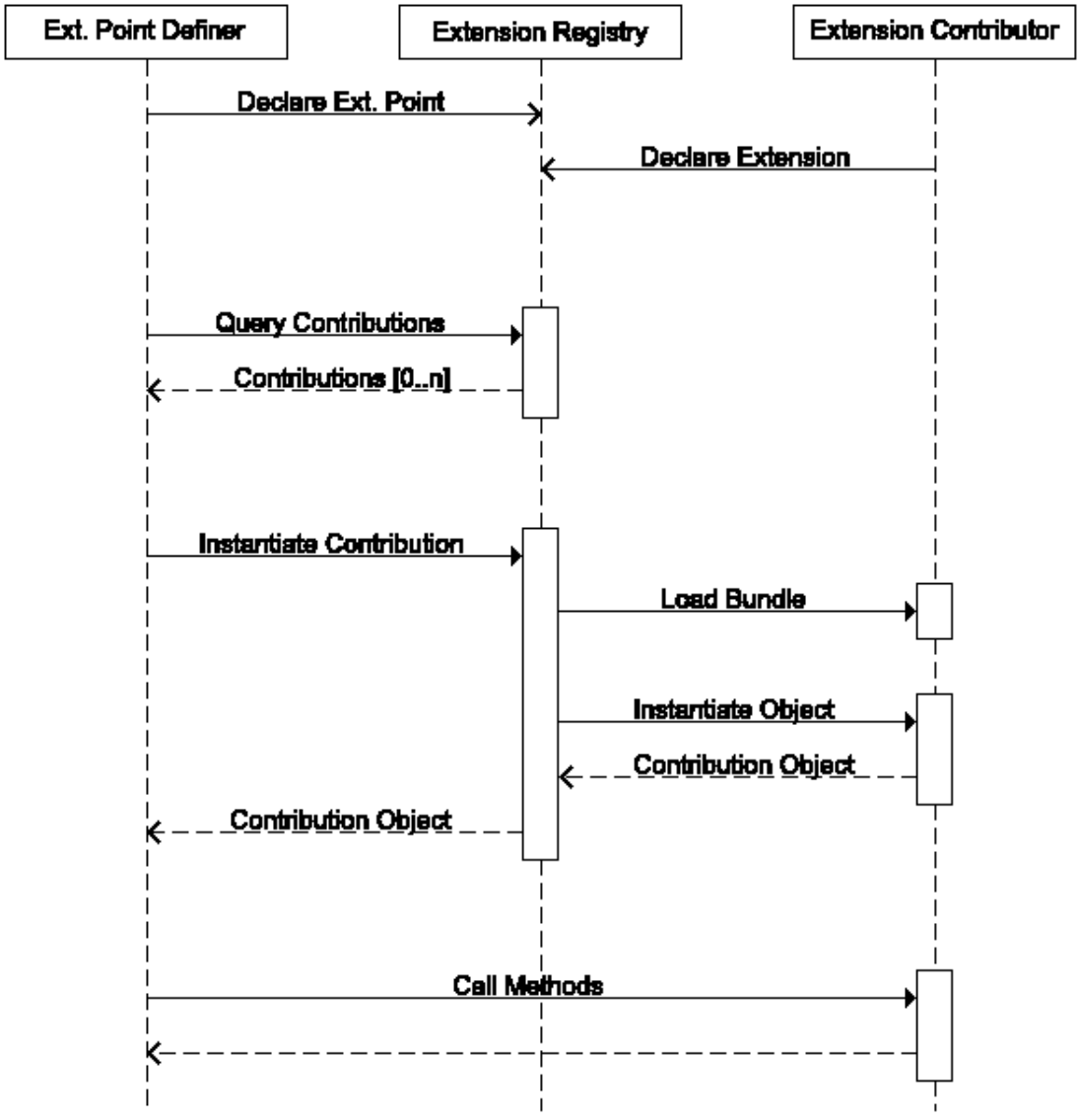


FIGURE 2.11 – *Diagramme de séquence de l'enregistrement d'un point d'extension*

Précautions : Si un point d'extension est déclaré pour fournir une classe implémentant une interface I dans le plugin A. L'interface I doit être fournie par le plugin A déclarant le point d'extension. Chaque plugin possède son propre environnement d'exécu-

tion [14] (cf .2.3.4 p.19). Ce qui signifie que la classe doit implémenter l'interface I du même environnement d'exécution (plugin A). Même chose pour le type de retour des fonctions et ses paramètres, eux aussi doivent appartenir au même environnement d'exécution.

C'est a cette occasion que l'on utilise l'exportation de package dans le `MANIFEST.MF` via la propriété "Export-Package". Cette propriété permet de rendre visible par un autre plugin les classes contenues dans un package cible.

Avec ce mécanisme, il faut faire attention aux différents espaces de chargement des classes [15]. Exemple : le plugin A propose le point d'extension "com.smartesting.core.extractor" qui permet de définir une classe qui implémente l'interface "com.smartesting.extractor.Extractor" via l'élément "class". Le plugin B utilise le point d'extension "com.smartesting.core.extractor" avec comme classe "mon.plugin.extractor.MyExtractor".

2.5.2 Mise en œuvre

Ce paragraphe explique le moyen de mise en œuvre d'un point d'extension pour gérer un extracteur de modèle spécifique au modeleur Papyrus. (A noter que les . . . des exemples correspondent à un chemin de package).

Etape 1 : Créer un point d'extension

Il faut tout d'abord déclarer un point d'extension dans le plugin `Common Extractor` puisque tous les plugins d'export en dépendent.

Ensuite le point d'extension est définie dans le fichier `ModelExtractor.exsd`. Puis on déclare dans le fichier `plugin.xml` le point d'extension :

```
<plugin>
        <extension-point id="ModelExtractor"
                name="com.smartesting.eclipse.core"
        schema="schema/ModelExtractor.exsd"/>
</plugin>
```

Figure 2.12 – *Extrait du fichier `ModelExtractor.exsd` dans le plugin "Common extractor"*

Ensuite le fichier `Modelextractor.exsd` est configuré afin de permettre la déclaration de l'utilisation de l'extension comme ceci :

```
<extension point="com.smartesting.eclipse.core.ModelExtractor">
  <ModelExtractor
          class="...PapyrusModelExtractor"/>
</extension>
```

Figure 2.13 – *Extrait du fichier `plugin.xml` du plugin "Papyrus extractor"*

14. ClassLoader pour les programmeurs Java
15. ClassLoader

Dans l'exemple précédent, l'attribut "class" permet de définir la classe qui permettra de rendre le service. Il a été choisi que la classe `PapyrusModelExtractor` devrait implémenter l'interface "ModelExtractor".

Etape 2 : Utiliser le point d'extension

Pour utiliser les données du point d'extension, il faut passer par la plateforme Eclipse et interroger l'`ExtensionRegistry` [16]. Grâce à ce service, nous pouvons récupérer l'instance de l'interface `ModelExtractor` qui sera `PapyrusModelExtractor`.

Etape 3 : Exporter les classes dépendantes

La dernière étape mais non des moindres, doit permettre de rendre visible depuis un autre plugin l'interface `ModelExtractor` dont `PapyrusModelExtractor`. Pour cela, on ajoute la ligne suivante dans le fichier `MANIFEST.MF` du plugin "Common extractor" :

```
Export-Package: . . .ModelExtractor
```

Figure 2.14 – *Fichier `MANIFEST.MF` du plugin "Common extractor"*

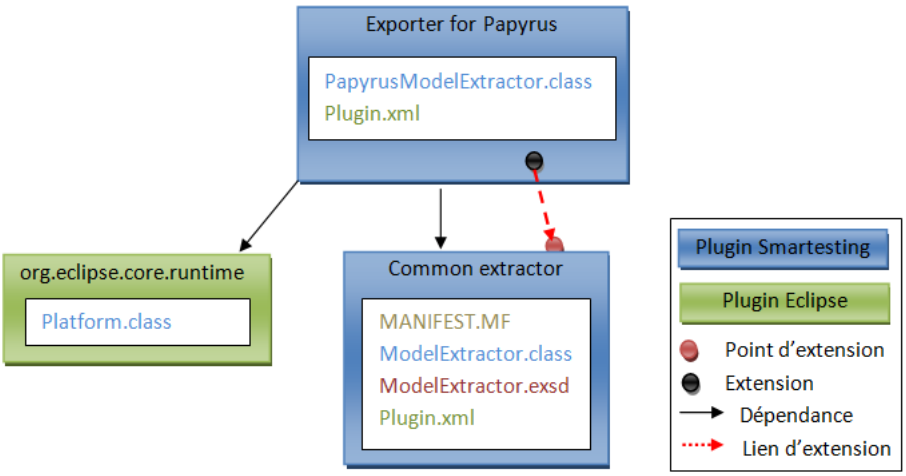


Figure 2.15 – *Mise en œvre d'un point d'extension pour extraire le modèle UML*

La figure 2.15 p.25 montre l'infrastructure des plugins pour la mise en place du mécanisme de point d'extension.

Le plugin "Common extractor" contient le fichier manifest dans lequel est déclaré visible la classe `ModelExtractor`. Le fichier `ModelExtractor.exsd` contient la déclaration du point d'extension, son nom, sa structure XML. `Plugin.xml` contient la déclaration du fichier `ModelExtractor.exsd`.

Le plugin "Papyrus extractor" contient un manifest contenant la dépendance avec "Common extractor", la classe spécifique d'extraction du modèle appelé `PapyrusModelExtractor` et le fichier `plugin.xml` pour déclarer l'utilisation du point d'extension.

16. Platform.getExtensionRegistry()

2.5.3 Avantages /inconvénients

La mise en place du mécanisme d'un point d'extension est simple à mettre en place et offre des résultats rapides. Il faut éviter le piège sur le problème d'environnement d'exécution différent des plugins.

Il y a cependant, un problème lié à l'environnement d'Eclipse, qui utilise des variables globales comme "Platform" qui empêchent la réalisation de tests unitaires simples. La solution est d'abstraire la plateforme Eclipse pour pouvoir tester le mécanisme.

2.6 Synthèse

Le cheminement logique du travail réalisé fut, l'exploration du modeleur Papyrus. A la suite de quoi, la fragmentation du plugin RSM pour extraire le code commun avec Papyrus. La fragmentation a été freinée par l'amélioration du *build* nécessaire à la mise en œuvre d'une architecture en plugin. Et pour finir, la création d'un point d'extension a permis de régler un problème de dépendance pour faire du spécifique.

Quoi qu'il en soit, toutes ces actions ont été menées avec succès. Et il ne reste plus qu'à en tirer les conclusions du stage qui s'imposent.

Chapitre 3

Conclusion

Ce chapitre est la conclusion de tout le travail réalisé dans l'équipe R&D de Smartesting. C'est aussi, la conclusion d'un challenge personnel de reprise d'étude après cinq années passées dans la vie active. Et pour finir, c'est l'occasion de dire, la chance que j'ai d'entrer dans l'équipe Smartesting.

3.1 Professionnelle

Le stage s'est déroulé dans de très bonnes conditions. J'ai appris à travailler d'une façon très différente. Et cela m'a très fortement intéressé tout au long de celui-ci.

3.1.1 Objectifs

Les missions confiées durant le stage ont toutes été remplies. La modification du *build* a permis de fragmenter l'application en plugins réutilisables. Cela apporte aussi plus de possibilités pour le déploiement de l'application.

La création d'un plugin d'exportation de modèle pour Papyrus a été réalisée avec succès. Il est désormais possible d'exporter un modèle Papyrus vers Test Designer. Quant à la réorganisation des modules en plugins et l'utilisation des points d'extensions, cela devrait permettre le développement plus rapide et plus simple de fonctionnalités spécifiques à chaque modeleur.

3.1.2 Échange de connaissance

J'ai reçu de la part des collègues de Smartesting bien plus que des conseils ou une méthode de programmation particulière, mais une culture de programmation commune. Par exemple la programmation par intention : il s'agit d'écrire ce que fait quelque chose en langage de programmation Java sans avoir besoin d'insérer de commentaires dans tout le code. L'intérêt est de ne pas avoir à maintenir la mise à jour des commentaires lorsque l'on procède à un redécoupage fonctionnel.

La transmission de connaissances n'a pas marché que dans un sens. J'ai en effet partagé mes connaissances de la plateforme Eclipse avec l'équipe. J'ai même réussi le challenge d'améliorer le temps de test des plugins pour les modeleurs.

Initialement, la technique employée consistait à installer de manière conventionnelle le plugin d'exportation pour le tester. En proposant, une technique différente le temps de test est passé d'un maximum de quinze minutes à six minutes. C'est en fait le temps personnel que j'ai passé à étudier la plateforme Eclipse qui m'a permis de réaliser tout ça.

3.1.3 L'agilité

Avant d'arriver chez Smartesting et même avant le Master, j'avais travaillé cinq ans en entreprise. La méthode de développement était très classique et pesante. Je pense aujourd'hui que le développement aurait pu être plus "fun" avec la pratique de l'agilité et tout aussi productif. Chez Smartesting, je suis heureux d'avoir pu participer à cette expérience.

J'ai énormément appris des méthodes "Agile", je pense être capable de proposer cette méthode de travail auprès de mes prochains collègues de travail. J'ai trouvé chez Smartesting une très bonne cohésion de groupe qui, je pense, est due aux discussions et aux conditions de travail dans la bonne humeur.

3.2 Personnelle

3.2.1 Enjeu de carrière

Ce stage fut pour moi, un enjeu de carrière. Je suis arrivé au terme des trois années de reprise d'étude que je m'étais fixée avec raison. Grâce à l'université de franche-comté et de la formation continue, j'ai atteint le niveau de compétence que je m'étais juré d'obtenir.

L'enseignement que j'ai reçu m'a totalement convaincu. Je pense que c'est grâce à mon bagage en entreprise, que chaque difficulté me semblait nécessaire et juste. De tout mon cursus, c'est bizarrement les matières sur les tests fonctionnels (le B) qui m'ont toujours posées le plus de difficulté. Et pourtant, c'est le domaine dans lequel j'ai travaillé.

3.2.2 le futur

Pour finir, je retire une grande satisfaction personnelle d'avoir accompli autant de choses aussi intéressantes durant une période de stage aussi courte. Et je suis content que Smartesting puisse me garder au sein de l'équipe R&D pour une durée de sept mois.

Dorénavant, je suis certain d'avoir les compétences requises pour n'importe quel poste en informatique. Suffit de s'en donner les moyens.

Bibliographie / Netographie

Mark Melvin : Creating Your Own Extension Points : It's Easier Than You Think !
March 20, 2008
http://www.eclipsecon.org/2008/?page=sub/&id=388

Apache Ant 1.7.1 Manual
http://ant.apache.org/manual/index.html

OSGi Bundle Manifest Headers
Version 3.1 - Last revised June 20, 2005
http ://help.eclipse.org/help32/index.jsp (topic : bundle_manifest)

Eclipse Ganymede documentation
http://help.eclipse.org/ganymede/index.jsp

Smartesting
Dernière visite le 1er juin 2009
http://www.smartesting.com

Papyrus
Dernière visite le 2 mars 2009
http://www.papyrusuml.org

VETESS
Dernière visite le 22 avril 2009
http://lifc.univ-fcomte.fr/vetess

Eclipse Plug-ins (3rd Edition)
by Eric Clayberg and Dan Rubel, Additions Wesley, 2009
http://www.qualityeclipse.com

The Art of Agile Development
By James Shore, Shane Warden, O'RAILLY, October 2007
Pragmatic guide to agile software development

Résumé

Ce rapport résume les quatre mois de stage passés dans la société Smartesting, au sein d'une équipe R&D "Agile". Smartesting est un éditeur de génération automatique de tests à partir d'une modélisation UML des exigences. C'est à partir de modeleurs non open source basé sur une plateforme d'Eclipse que la modélisation est effectuée jusqu'alors.

Le but de ce stage fut l'introduction du modeleur open source Papyrus dans la solution Smartesting. En créant un nouveau plugin pour celui-ci, Smartesting a souhaité modifier le processus de construction de l'application afin de permettre de s'implanter d'avantage sur la plateforme Eclipse.

En remplissant tous les objectifs fixés du stage cela m'a permis de découvrir, le framework de modélisation Emf, la plateforme Eclipse, les mécanismes d'extension de plugin et la méthode "Agile".

Mots-clés

Méthode "Agile", Open source, Eclipse, modeleur, Rsm, Together, Plugin, feature, update-site, OSGI, EMF

Abstract

This report summarizes about four months of training spent in the company Smartesting, with an "Agile" R&D team. Smartesting is a software editor which automates test case generation from the functional model of a UML specification, with closed source modelers based on the Eclipse platform.

The goal of the training period was the introduction of an open source modeler in the Smartesting solution. With the new plugin creation, the company wanted to modify the build process to take advantage of the Eclipse platform.

Achieving these objectives allowed we to discover the Emf framework to study the Eclipse platform, and to take part in "Agile" development.

Key words

Agile method, Open source, Eclipse, modeler, Rsm, Together, Plugin, feature, update-site, OSGI, EMF