CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

CENTRE REGIONAL DE TOULOUSE

MÉMOIRE

présenté en vue d'obtenir
LE DIPLÔME D'INGÉNIEUR CNAM

SPECIALITÉ : Informatique

OPTION : Architecture et Ingénierie des Systèmes et des Logiciels

par

Gautier LEBOURG

Génération automatique de tests d'acceptance par une approche de tests dirigée par les modèles pour une application web

soutenu le : 21 janvier 2021

JURY

PRÉSIDENTE

Mme. LEVY Nicole	*Professeur des Universités - CNAM Paris*

MEMBRES

M. CHAARI Lotfi	*MCF - HDR, INP Toulouse*
M. MILLAN Thierry	*MCF, UPS Toulouse*
M. PETITEAU Ludovic	*Ingénieur logiciel - TALARON Services*
M. THULL Laurent	*Ingénieur logiciel - TALARON Services*

Citation

« Celui qui abandonne une fois, abandonnera toute sa vie. »

Mon Père

Remerciements

Je voudrais remercier en premier lieu M. Talal MASRI, Directeur de l'IPST-CNAM de Toulouse, qui m'a soutenu pour ce mémoire.

Je voudrais remercier M. Lotfi CHAARI qui m'a guidé, accompagné et conseillé, pour ce mémoire, dans un contexte particulier.

Je remercie l'ensemble de mes tuteurs Sarah PRAT, Raoudha AIT-ABDELLAH, Laurent THULL et François RIGAL pour leurs qualités humaines et pédagogiques et qui m'ont permis de monter en compétences tout en profitant de leur expérience techniques et managériales.

Je voudrais remercier M. Ludovic PETITEAU, Fondateur et Directeur Général de l'entreprise Talaron Services, pour ses encouragements ainsi que pour la confiance qu'il m'a accordée.

J'adresse également mes sincères remerciements à Alexandra GAUTIÉ, qui m'a toujours soutenu et sans qui je n'aurais jamais pu présenter ce mémoire.

Je remercie aussi Sonia VARTIAN d'avoir travaillé sur la mise en place d'une politique handicap au sein de l'IPST-CNAM, ainsi que pour son soutien.

J'adresse mes remerciements aux professeurs de l'IPST-CNAM, ceux dont la bienveillance a été ressentie tout au long de ces années.

Je remercie les enseignants qui m'ont suivi depuis le début de mes années universitaires : Fabienne VIALLET, Jean-Paul CARRARA, Thierry MILLAN, Dominique POZZO, Jérome LARTIGAU et Jacques LAFORGUE pour leurs conseils, leurs enseignements et leurs accompagnements.

Je voudrais aussi remercier toute ma famille pour leur soutien tout au long de ces années.

Merci Jonathan pour toutes ces belles années depuis l'IUT et qui vont continuer.

Merci aussi à Mélissa qui m'a gentiment aidé durant ces trois années.

Merci à toute l'équipe du projet Airline 1 : Midge MORNET, Simon-Pierre SAINTE-MARIE, Cyril FOURCADE et Pierre MORENO-MORELL pour la bonne ambiance du début jusqu'à la fin de mon alternance ainsi que pour leur soutien et leurs encouragements.

Et enfin mes remerciements à Miruna, Nathanaël, Mickaël, Paul et Florian, merci à vous pour vos conseils éclairés et tous les bons moments passés avec vous !

Liste des abréviations

— **A/C :** Aircraft

— **ACMS :** Aircraft Centralized Maintenance System

— **ACU :** AirCraft Unique

— **AIDS :** Aircraft Integrated Data System

— **AJAX :** Asynchronous XML and JavaScript

— **ATDD :** Acceptance Test–Driven Development

— **ATL :** ATLAS Transformation Language

— **AUTOSAR :** AUTomotive Open System Architecture

— **BDD :** Behavior-Driven Development

— **BNF :** Backus-Naur Form

— **BPMN :** Business Process Model and Notation

— **CFTL :** Comité Français des Tests Logiciels

— **CIM :** Computer Independent Model

— **CORBA :** Common Object Request Broker Architecture

— **CWM :** Common Warehouse Metamodel

— **DAO :** Data Access Object

— **DDD :** Domain Driven Design

— **DOM :** Document Object Model

— **DSL :** Domain Specific Language

— **DTO :** Data Transfer Object

— **EBNF :** Extendet Backus-Naur Form

— **EFF :** Electronic Flight Folder

— **EMF :** Eclipse Modeling Framework

— **ENS :** Entreprise de Services du Numérique

— **FODA :** Feature-Oriented Domain Analysis

— **FSM :** Finite-State Machine

— **GPL :** General Purpose Language

— **HTML :** Hypertext Markup Language

— **IDM :** Ingénierie Dirigée par les Modèles

— **ISO :** International Organization for Standardization

— **ISTQB :** International Software Testing Qualification Board

— **JPA :** Java Persistence API

— **JSON :** JavaScript Object Notation

— **LHS :** Left-Hand-Side

— **M2M :** Model to Model transformation

— **M2T :** Model to Text

— **MBT :** Model-Based Testing

— **MDA :** Model-Driven Architecture

— **MDD :** Model Driven Development

— **MDE :** Model-Driven Engineering

— **MOF2Text :** MOF Models to Text Transformation

— **MOF :** Meta-Object Facility

— **MTL :** Model Transformation Language

— **MVC :** Architecture en Modèle-Vue-Contrôleur

— **OCL :** Object Constraint Language

— **OMG :** Object Management Group

— **OOOI :** Out Off On In

— **PDM :** Platform Description Model

— **PIM :** Platform Independant Model

— **PSM :** Platform Specific Model

— **QVT :** Query-View Transformation

— **REST :** REpresentational State Transfer

— **RHS :** Right-Hand Side

— **RQP :** ReQuest Parameters

— **SUT :** System Under Test

— **TDD :** Test-Driven Development

— **UC :** Use case

— **UML :** Unified Modeling Language

— **URL :** Uniform Resource Locator

— **XML :** eXtensible Markup Language

— **XQuery :** XML Query

— **XSLT :** eXtensible Stylesheet Language Transformations

Glossaire

— **Airline 1 :** Application web permettant de gérer les données associées aux vols d'essai des avions d'Airbus.

— **Automate de test :** Outil ou logiciel permettant l'exécution ou le support des activités de test en minimisant l'intervention humaine.

— **Behavior-Driven Development :** Programmation pilotée par le comportement.

— **Campagne de tests :** Ensemble de cas de tests à exécuter pour tester une partie ou une version d'une application cible.

— **Common Warehouse Metamodel :** Une spécification de l'OMG qui décrit un langage d'échange de métadonnées à travers un entrepôt de données, un système décisionnel ou un système d'ingénierie des connaissances.

— **Computer Independent Model :** Modèle indépendant de tout calcul.

— **Couverture de test :** Groupe d'éléments couverts par les tests, par exemple les exigences.

— **Domain Specific Modeling Language :** Langages de modélisation spécifique à un domaine de métier précis.

— **Domain-Specific Language :** Langage spécifique à un domaine de métier précis.

— **Edge, arête :** Une transition, une action, qui permet d'aller d'un sommet à un autre.

— **Environnement de test :** Tous les éléments nécessaires pour l'exécution de tests. Cela comprend les logiciels ou les robots d'exécution.

— **Finite-State Machine :** Un modèle qui a un nombre fini d'états et un nombre fini de transitions entre ces états. On peut rendre complet un FSM incomplet en ajoutant une hypothèse telle que toutes les entrées manquantes sont des auto-transitions.

— **Given-When-Then (Donné-Quand-Alors) :** C'est une structure de représentation de test, spécifique à Behavior-Driven Development.

— **Langage ubiquitaire :** Langage compréhensible par toutes les parties prenantes d'un projet.

— **Métamodèle :** Modèle d'un modèle, qui décrit la structure du modèle, qui définit son langage d'expression.

— **Model Transformation Language :** Langage de transformation de modèle en ingénierie des systèmes et en ingénierie logicielle, destiné spécifiquement à la transformation de modèle.

— **Model-Based Testing :** Activité qui permet de concevoir et de dériver (de manière automatique ou non) des cas de tests à partir d'un modèle abstrait et de haut niveau du système sous test (SUT). Le modèle est dit abstrait, car il offre bien souvent une vue partielle et discrète des comportements attendus d'un logiciel ou d'un système.

— **Modèle SUT :** Modèle sous test qui spécifie le comportement d'entrée et de sortie du système sous test.

— **Oracle de test :** Prédiction des valeurs attendues lors de l'exécution d'un test.

— **Platform Description Model :** Modèle de description des plate-forme.

— **Platform Independant Model :** Modèle indépendant de plate-forme.

— **Platform Specific Model :** Modèle spécifique aux plates-formes.

— **Product owner :** Représentant du client au sein de l'équipe SCRUM.

— **SCRUM :** Méthode agile de gestion de projets informatiques privilégiant la communication.

— **Test boîte blanche :** Test basé sur une analyse de la structure interne du code du composant ou du système à tester.

— **Test boîte noire :** Test fonctionnel ou non fonctionnel représentation d'un système sans considérer son fonctionnement interne.

— **Test d'acceptance :** Test client utilisé pour déterminer si un composant ou un système satisfait ou non aux critères d'acceptation.

— **Uplink message :** Type de message envoyé par l'application pour interroger l'avion sur certains paramètres avec un certain nombre d'échantillons et une période donnée.

— **User Story :** Illustration d'un besoin fonctionnel exprimé par les types d'utilisateurs.

— **Vertex, sommet :** Un état attendu que nous souhaitons tester.

Table des matières

Introduction

La qualité reste un enjeu important pour les concepteurs, il en va de même pour le domaine du logiciel. En effet, notre société est de plus en plus connectée. Cette "inter-connexion" s'accompagne d'une évolution toujours plus importante des systèmes mis en oeuvre. Cette évolutivité s'accompagne nécessairement de la montée en nombre des exigences afin de répondre au plus près aux différentes attentes sans pour autant négliger la qualité qui demeure la principale préoccupation des différents acteurs du marché. Le niveau de qualité d'un logiciel est évalué lors de l'étape de tests. La compression de la phase de test peut avoir un impact insignifiant s'il s'agit de simples affichages pour la finalité du logiciel. Mais elle peut tout aussi avoir un impact beaucoup plus important en terme de vies humaines et sur le plan économique. Ce fut le cas, par exemple, lors des deux crashs des Boeing 737 MAX à cause d'une défaillance du système de contrôle des appareils [Dut19].

À cet effet, ce mémoire propose deux différentes approches possibles et a pour but de montrer la pertinence de la génération automatique de tests d'acceptance pour une application web par une approche de tests dirigée par les modèles, dans un contexte industriel, tout en privilégiant la qualité.

Dans un premier temps le premier chapitre, celui du contexte industriel, s'attarde sur les activités de tests, les problèmes observés et les différentes analyses pour aborder une solution possible et ainsi énoncer la problématique de ce mémoire. Le deuxième chapitre de génération automatique de test dirigée par les modèles aborde les différents principes généraux puis les différentes méthodes qui seront évaluées. Dans la mise en œuvre du chapitre trois on présente le cas d'étude, l'outillage disponible et la conception de la solution. Pour finir, le quatrième chapitre évaluera la pertinence de la solution avec un bilan réalisé à l'aide de différents tests.

Chapitre I

Contexte industriel

1.1 L'entreprise et le client

1.1.1 Talaron Services

Talaron Services est une Entreprise de Services du Numérique (ENS) sensibilisée à l'intégration des personnes en situation de handicap et à l'accompagnement personnalisée. Cette société a été créée en juillet 2010 par un ancien collaborateur du groupe CAPGEMINI Monsieur PETITEAU.

Cette entreprise propose deux types de contrats dans le cadre des prestations de service :

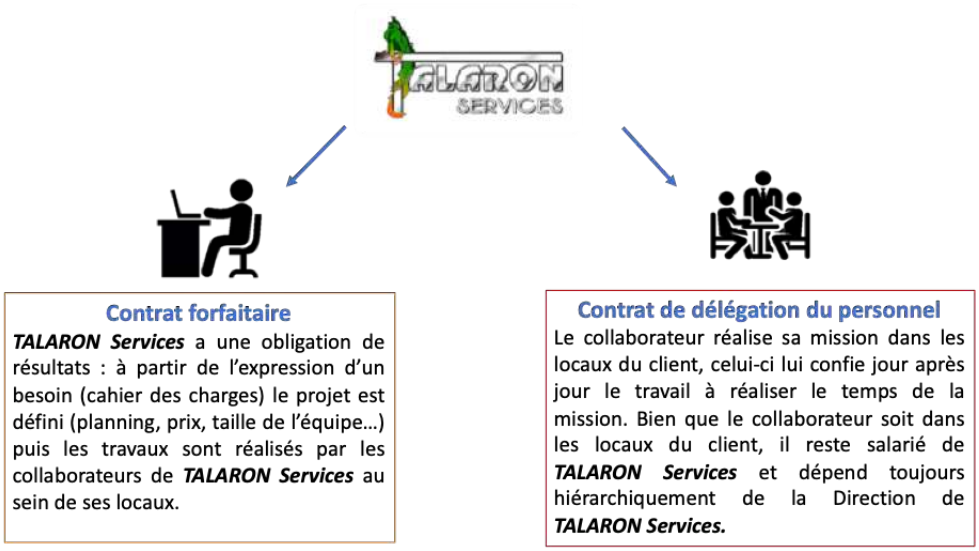

FIGURE 1.1 – Types de contrat de prestation de services.
(Source : Talaron Services)

Talaron Services est présent sur plusieurs sites en France :

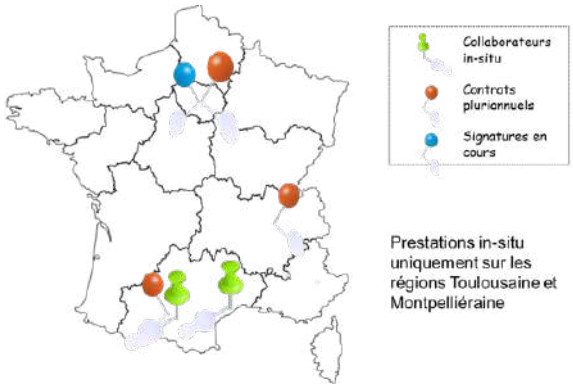

FIGURE 1.2 – Couverture géographique Talaron Services.
(Source : Talaron Services)

1.1.2 Sopra Steria

Sopra Group a fusioné avec Steria en 2014. Ces deux entreprises sont devenues Sopra Steria qui est actuellement un leader européen du marché de la transformation numérique.

Sopra Steria possède une offre de service très complète dans différents domaines :

- Consulting
- Intégration de systèmes
- Édition de solution métier
- Infrastructure Management
- Busines Process Services

FIGURE 1.3 – Couverture géographique de Sopra Steria.
(Source : Sopra Steria)

1.2 Activité de tests

Sopra Steria a fait appel à la société Talaron Service, à travers le contrat de délégation du personnel, pour une activité de tests sur les outils d'essai en vol d'Airbus. Les avions d'essai servent à la certification d'un nouveau modèle ou à des vérifications des évolutions ou des nouvelles options. Il y a également des avions de série qui, avant livraison, doivent aussi être testés. Dans le cycle de vie des avions d'Airbus, l'application *Airline 1* intervient surtout au niveau de la phase "avions en service et maintenance".

FIGURE 1.4 – Cycle de vie des avions Airbus.
(Source : Airbus)

L'application à fournir doit être compatible avec différents supports pour les ingénieurs d'Airbus. Le but du projet *Airline 1* est d'améliorer la fiabilité des avions. Elle permet d'apporter une aide technique aux équipes de maintenance des appareils lors de toutes les phases de mise en service et d'exploitation. Cet outil permettant de suivre des vols en temps réel, il est primordial à chaque livraison de s'assurer qu'il n'y ait pas de dégradation des fonctionnalités pré-existantes.

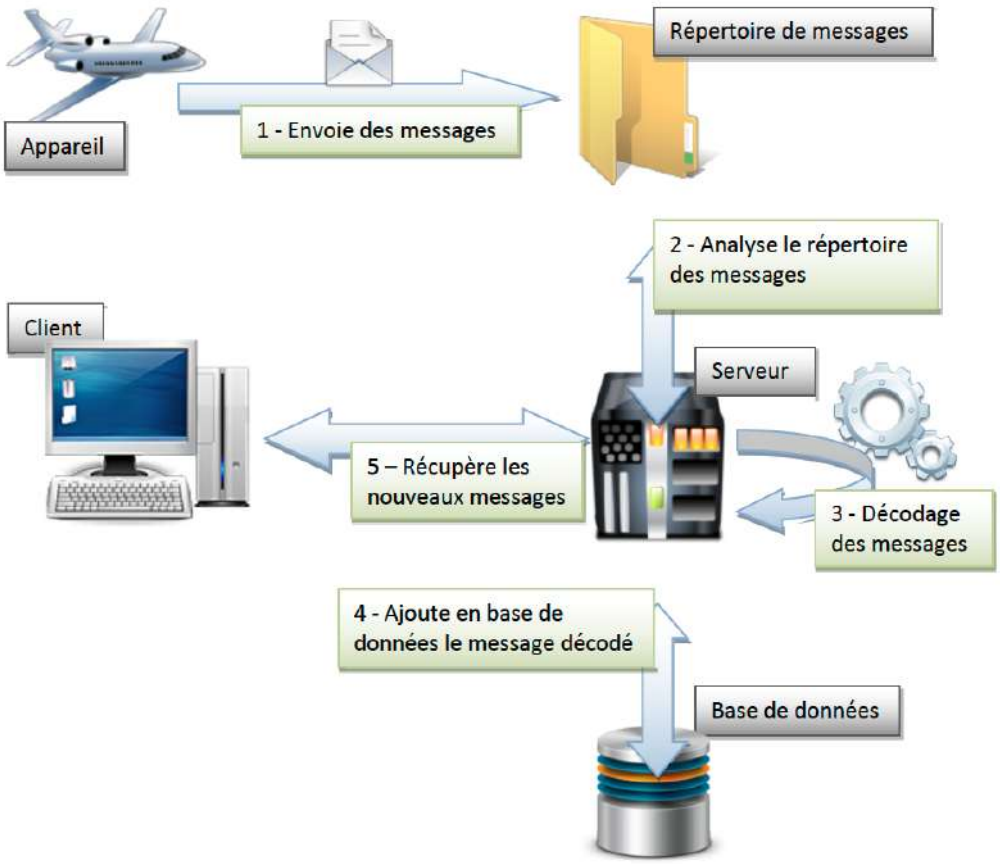

FIGURE 1.5 – Schéma explicatif de l'objectif *d'Airline 1*.

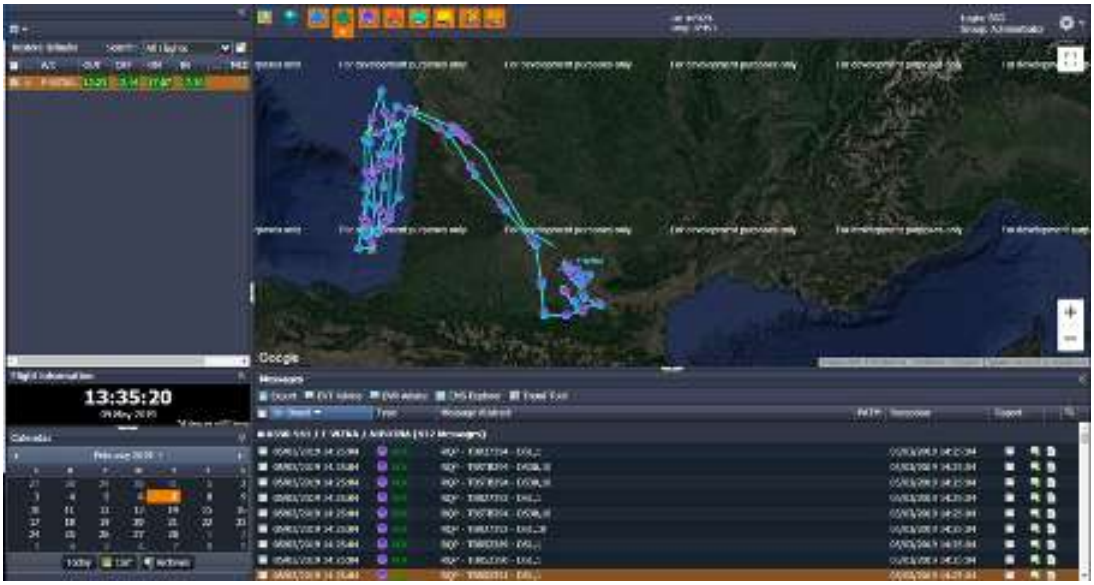

FIGURE 1.6 – Extrait de l'interface principale *d'Airline 1*.
(Source : Sopra Steria)

L'architecture est conçue pour permettre aux utilisateurs d'accéder à la solution *Airline 1* à partir de leur propre navigateur Web. Les fonctionnalités des applications sont accessibles en HTTP grâce à l'architecture REST.
La solution *Airline 1* est constituée de deux applications logicielles principales :

- Une application «web», qui gère les demandes des utilisateurs.
- Une application «batch», en charge des activités batch.

Un serveur Batch permet des traitements par lots, pouvant éventuellement être réalisés en temps différé, sans que l'intervention d'un opérateur soit nécessaire.
Les données de l'application quand à elles sont stockées dans une base de données SQL Server 2008.

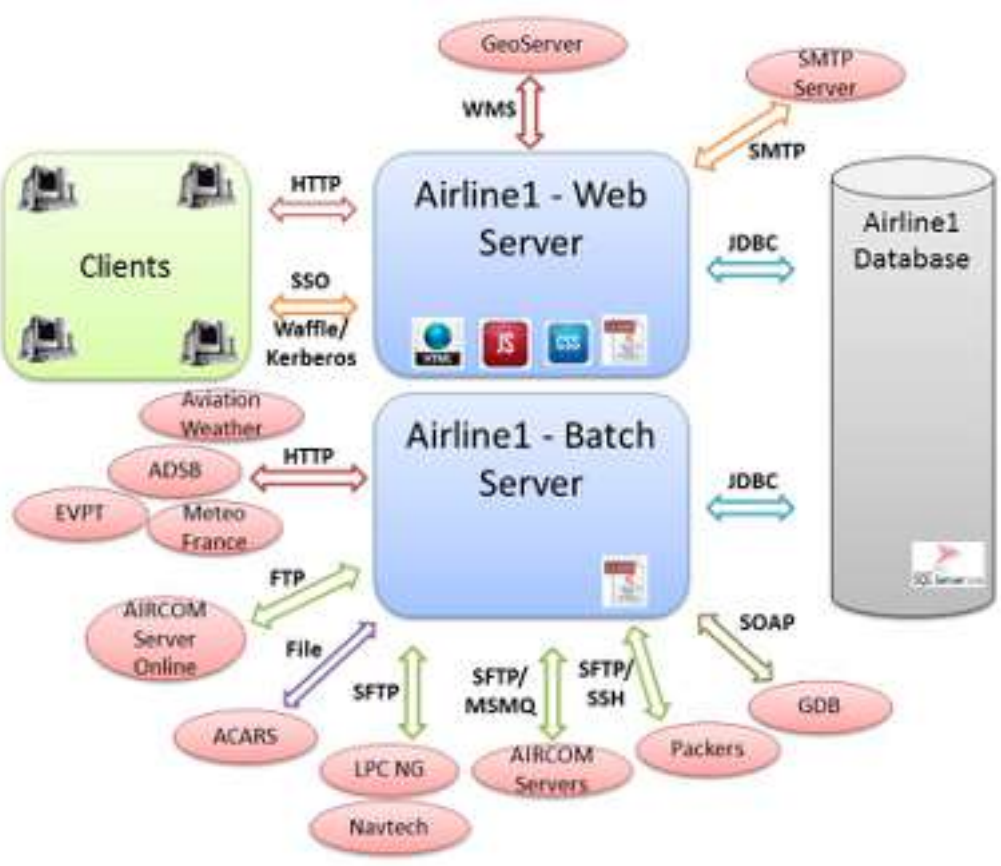

FIGURE 1.7 – Schéma de l'ensemble de l'architecture système *d'Airline 1*.
(Source : Airbus)

Pour plus de détails concernant l'application *Airline 1* en elle-même, voici en suivant son schéma d'architecture, montrant bien qu'elle est composée de deux parties : client et serveur. Le serveur est divisé en packages de haut niveau, les packages les plus importants sont :

- Package Restlet : ce package contient toutes les classes nécessaires à l'infrastructure RESTlet pour exposer les services d'application dans REST aux applications clientes. Les ressources RESTlet obtiennent l'objet DTO envoyé par le côté client, appellent des classes de service et envoient une réponse au client.
- Service package : ce package permet d'exécuter les traitements métier demandés par le client. Il gère la persistance grâce à la couche de persistance JPA. Les services sont appelés par les ressources RESTlet du package restlet.
- Package DAO : ce package permet la persistance de l'entité de domaine dans la base de données, en utilisant l'API JPA implémentée par Hibernate. Les DAO sont appelés par la couche Service.
- Package DTO : il contient tous les objets de transfert de données, qui est un modèle de conception utilisé pour transférer des données entre des sous-systèmes d'application logicielle. Ces classes sont une représentation des classes de domaine qui contiennent uniquement les informations nécessaires du côté client. La traduction entre les objets Domain et DTO est gérée par les classes Traductors.
- Package de domaine : il contient toutes les classes métier qui sont une représentation objet de la table de base de données.

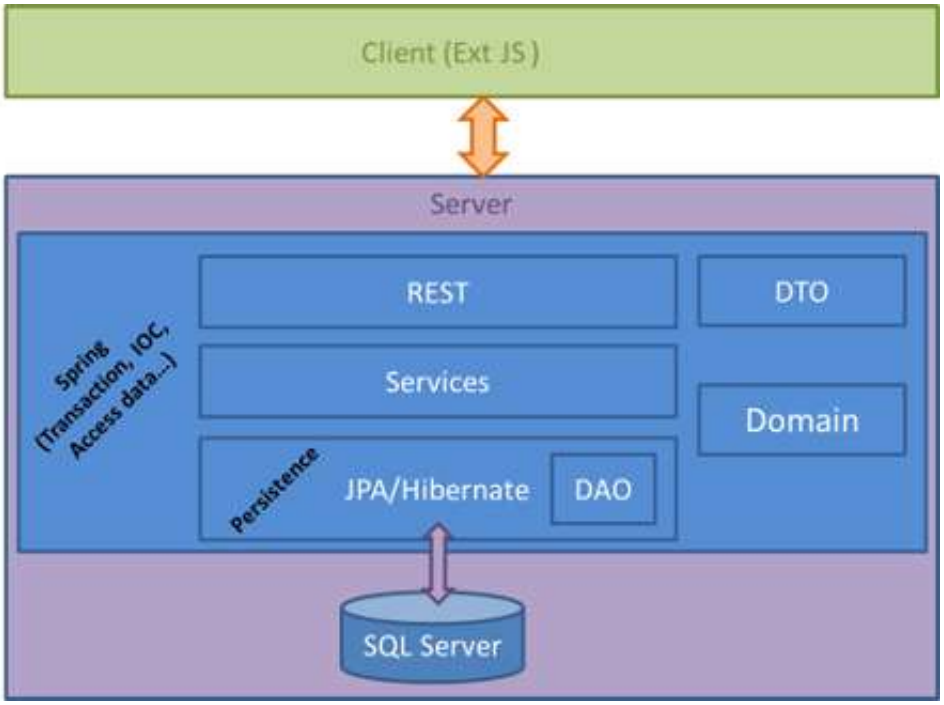

FIGURE 1.8 – Schéma de l'architecture du serveur de l'application *d'Airline 1*.
(Source : Airbus)

Pour veiller à ce qu'il n'y ait pas de dégradation entres les différentes versions de l'application l'ensemble des exigences de tests est compilé dans un fichier Excel de telle manière à regrouper ainsi plusieurs types d'informations :

- Le nom de la spécification de test.
- La date d'émission du test.
- La partie de l'application qui est testée.
- Les différentes étapes à vérifier lors du test avec le résultat attendu et son statut.

1.03 Split/merge flight			
Date	23/01/2019		
Tester	MRO		
Initial conditions:			
Step	Actions	Expected result	Status
1	Split flight	The selected item to split a Flight is the first one on the created flight	OK
2	Merge 2 flights		OK
			OK

TABLE I.1 – Tableau illustrant quelques étapes de tests.
(Source : Sopra Steria)

1.3 Problèmes observés

Arrivant sur le projet *Airline 1* déjà en phase de développement, il a été constaté que l'ensemble des tests fonctionnels était effectué manuellement, notamment les tests de non-régressions. Ce qui à pour conséquence qu'un développeur est monopolisé, souvent des jours entiers, par le test de fonctionnalités qu'il avait déjà implémenté mais il doit vérifier si la nouvelle version de l'application est toujours en adéquation avec les fonctionnalités déjà établies. Cela engendre deux problèmes majeurs :

- **La qualité** : La solution doit permettre d'avoir une meilleure couverture des tests afin de pouvoir garantir un certain niveau de qualité.
- **Les délais** : La solution a pour but de réduire le délai engendré par les différents tests de non-régressions.

Il en résulte que pour pouvoir garantir un niveau de réponse optimal à l'ensemble des problèmes posés, il est nécessaire de faire une analyse du besoin, ceci sera détaillée dans la section suivante.

1.4 Analyse du besoin

L'analyse du besoin est l'ensemble des activités à exécuter au préalable pour bien définir la finalité du futur système. Dans cette partie, cela reviendra donc à se focaliser sur :

- La spécification de ce que la solution doit être amenée à résoudre.
- Le niveau d'exigence que l'on sera amené à adopter pour aboutir à une solution pérenne.
- Les différentes contraintes à respecter.

Pour cela nous aborderons les différentes analyses :

- Analyse de l'environnement du système existant.
- Analyse de la méthode de gestion de projet.
- Analyse conceptuelle du système existant.
- Analyse dynamique du système existant.
- Analyse de la source du problème.

1.4.1 Analyse de l'environnement du système existant

Il est important de rappeler que le système interagit avec plusieurs éléments externes :

- L'aéronef, par lequel le système reçoit des données de vols, peut envoyer différents types de messages, de différentes natures, à l'équipe de maintenance.
- L'équipe de maintenance est tenue informée, via une interface web, du statut des différents aéronefs. Elle peut également, exporter les données de vols, au besoin d'autres opérations.
- Le manager, via une interface web, dirige les différentes équipes et peut également exporter des données de vols.
- L'utilisateur "Uplink"a la possibilité d'envoyer des messages à l'aéronef.
- Le simple utilisateur ne peut que consulter l'application.

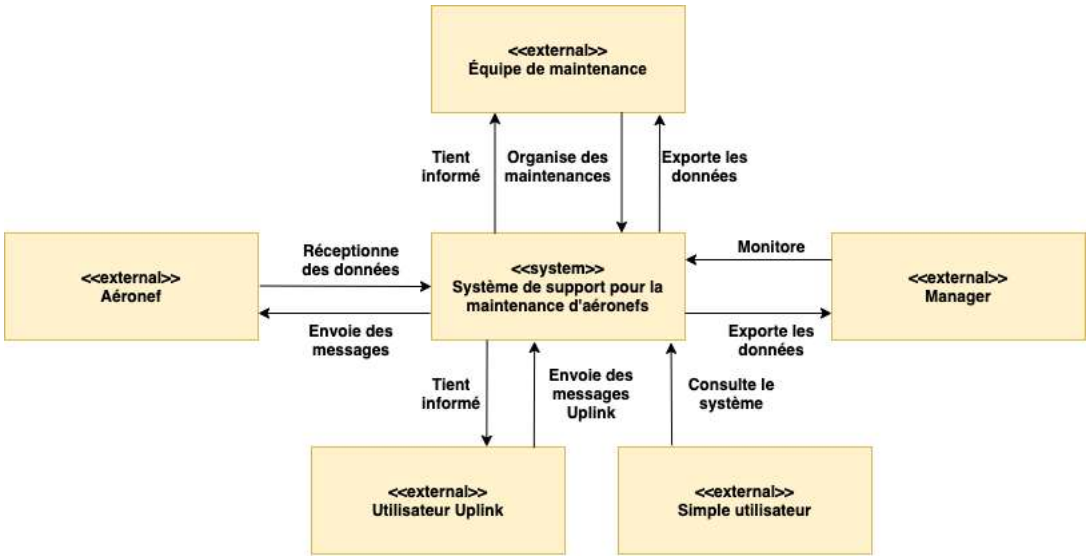

FIGURE 1.9 – Diagramme de contexte de l'application *Airline 1*.

1.4.2 Analyse de la méthode de gestion de projet

L'organisation du projet est faite au travers de la méthode SCRUM. Nous retrouvons ici les différents acteurs relatifs à cette méthode :

- **Le Product Owner** : le représentant des utilisateurs de l'application
- **Le Scrum Master** : le "chef d'orchestre" de l'équipe
- **L'équipe de développement**

Les principes de la méthode sont résumés dans le schéma ci-dessous :

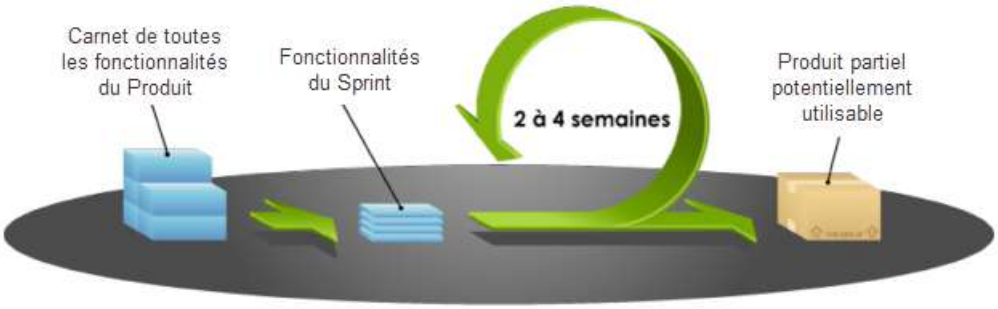

FIGURE 1.10 – Principes de la méthode SCRUM.
(Source : La taverne du testeur [Cha17])

Concernant la mise en pratique de la méthode SCRUM, les communications avec l'Inde se font au travers d'un kit Agile permettant les échanges par visio-conférence. Afin de faciliter le suivi des activités au quotidien de l'équipe, un outil partagé a été mis en place : JIRA. Dans cet outil, chaque User Story embarquée dans le backlog est découpée en différentes tâches. En fonction de l'état d'avancement, le statut de la tâche est mis à jour.

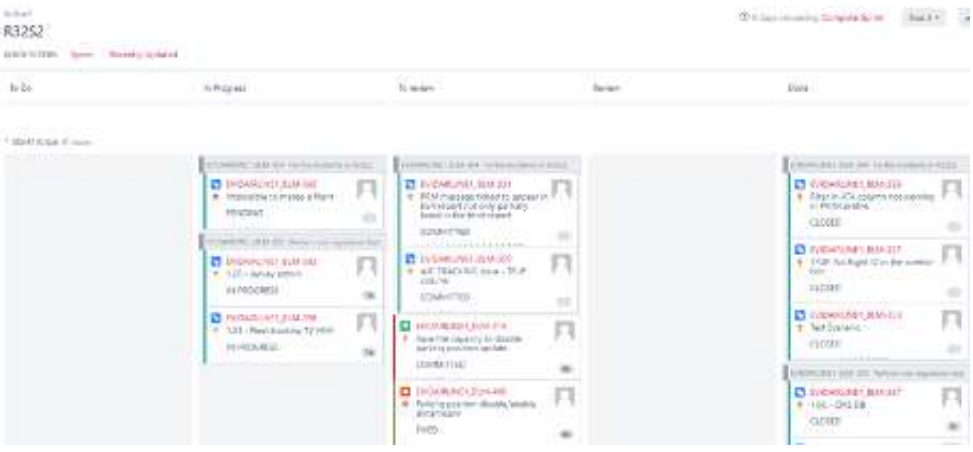

FIGURE 1.11 – Illustration de l'outil JIRA.
(Source : Sopra Steria)

1.4.3 Analyse conceptuelle du système existant

L'environnement du système *Airline 1*, permet de poser un cadre sur les questions :

- Qui est amené à utiliser le système ?
- Qui doit communiquer avec le système existant ?

Il est essentiel d'analyser en profondeur les différentes fonctionnalités du système. Les diagrammes de cas d'utilisations permettent de mettre en évidence les différentes fonctionnalités du systèmes et à qui elles sont destinées.

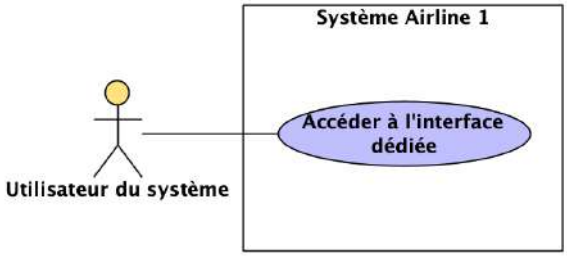

FIGURE 1.12 – Diagramme de cas d'utilisation d'accès au système.

Chaque utilisateur du système a besoin selon son rôle :

- D'une URL particulière, afin de pourvoir accéder à une interface dédiée à son rôle.

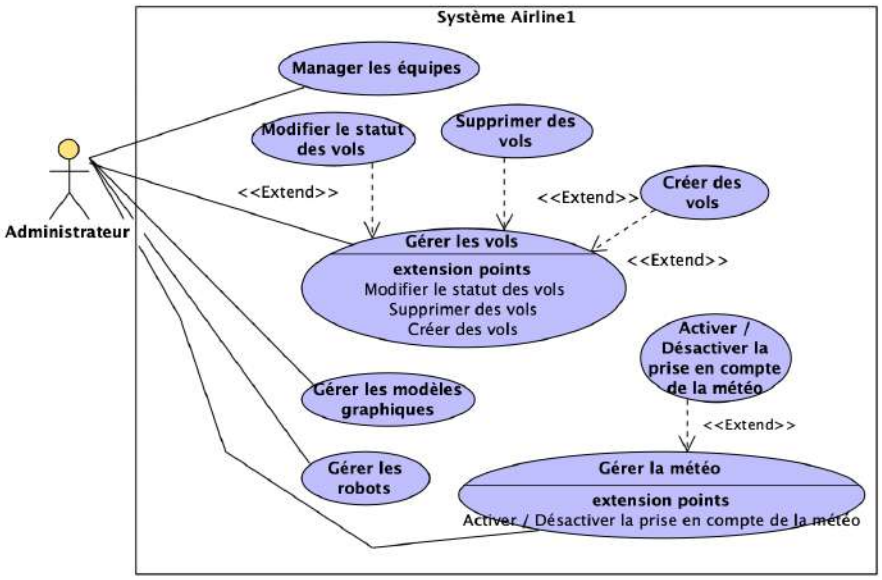

FIGURE 1.13 – Diagramme de cas d'utilisation de l'administrateur.

L'administrateur a différents types de besoins sur l'application, que le système doit pouvoir lui permettre :

- Sur la partie de gestion des vols il pourra, modifier le statut des vols existants en *in* en séance de maintenance, *out* en phase de roulage et enfin *off* l'aéronef a décollé. Il a la possibilité de créer de nouveaux vols et enfin dans la fonctionnalité de gestion des vols il peut supprimer des vols.
- L'administrateur peut gérer l'affichage et les dimensions des différents modèles graphiques.
- L'administrateur peut simuler le comportement d'un aéronef via l'intermédiaire de robots qui se substituent à un aéronef.
- Il a également la possibilité de gérer les différents niveaux d'attributions du personnel qui est amené à utiliser le système.
- Enfin comme dernier besoin technique l'administrateur doit pouvoir décider d'activer ou de désactiver la prise en compte de la météo au sein du système.

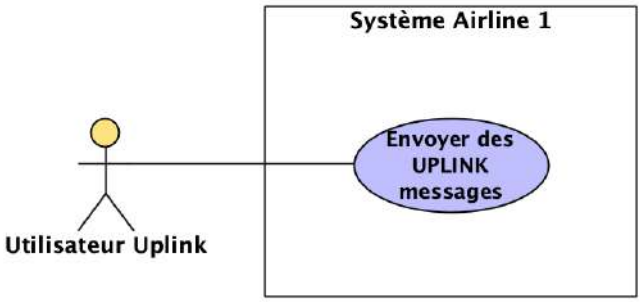

FIGURE 1.14 – Diagramme de cas d'utilisation de l'utilisateur *uplink*.

Le système pour le rôle de l'utilisateur *uplink* permet :

- D'envoyer un message à l'appareil pour, par exemple, l'avertir qu'à son prochain atterrissage il devra subir un contrôle, dans les plus brefs délais.

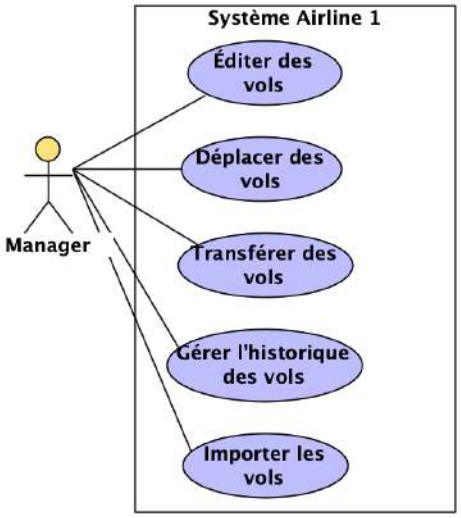

FIGURE 1.15 – Diagramme de cas d'utilisation du manager.

Le système pour le rôle du manager permet :

- Dans la gestion des vols le manager peut éditer des vols et, au besoin, compléter les informations manquantes. Il ne peut pas créer des vols.
- Il peut également dans cette même partie de gestion des vols, exporter les données de vols pour une exploitation ultérieure.
- Il a également la possibilité de transférer les différentes données de vols afin que les équipes techniques puissent avoir une meilleure précision en cas de contrôle approfondie de l'appareil.
- Il a la possibilité de consulter un historique de l'ensemble des vols dont il a la charge.
- Enfin, il a également la possibilité d'importer des vols.

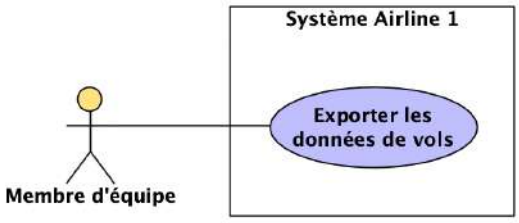

FIGURE 1.16 – Digramme de cas d'utilisation des membres d'equipe.

Pour chaque membre de l'équipe de maintenance, le système offre uniquement la possibilité :

- D'exporter les données de vols.

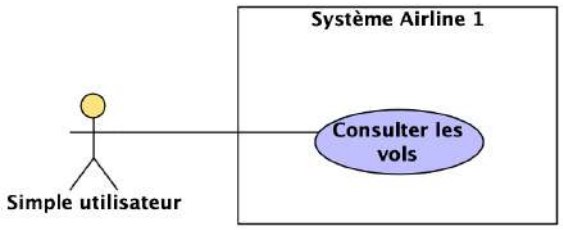

FIGURE 1.17 – Digramme de cas d'utilisation d'un simple utilisateur.

Enfin, pour un simple utilisateur le système lui offre uniquement la possibilité :

- De consulter l'intégralité du système sans toutefois pouvoir lui permettre de modifier les données du système.

Nous avons maintenant une vue statique de l'ensemble des fonctionnalités qui sont couramment utilisées par les utilisateurs. Mais afin de voir tout le processus qu'engendre l'ensemble de ces fonctionnalités, il est important d'avoir une vue dynamique du système.

1.4.4 Analyse dynamique du système existant

Afin de pouvoir exprimer plus en détail, les besoins exprimés par les différents diagrammes de cas d'utilisation, l'utilisation de digrammes de séquence permet d'avoir une vue dynamique du système qui est à l'étude.

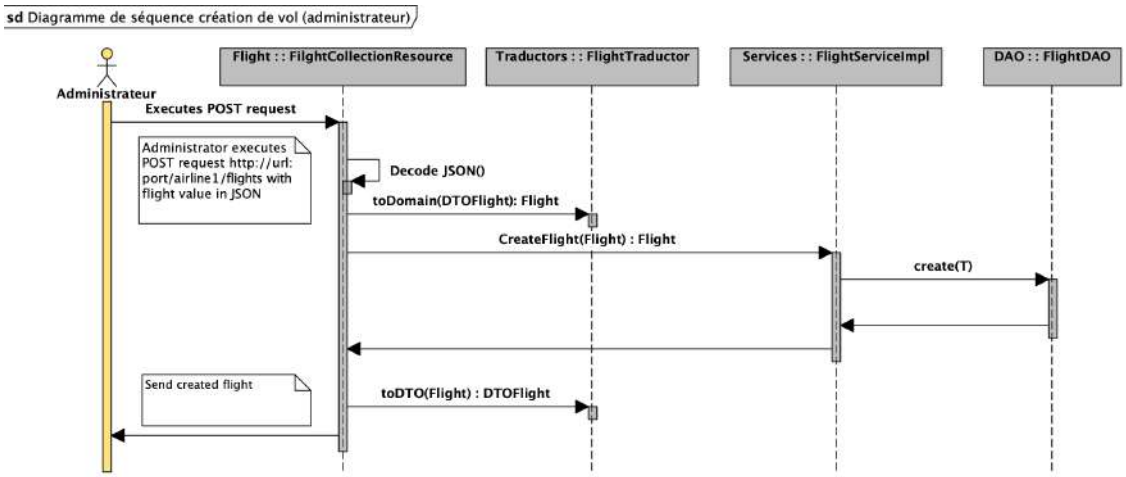

FIGURE 1.18 – Diagramme de séquence de création de vol.
(Source : Airbus)

D'après la Figure 1.18, nous constatons que pour le scénario de création de vol l'administrateur entre une description de vol afin de créer un vol. Lorsque l'administrateur valide, une requête AJAX (POST) est envoyée au serveur. La description du vol est ainsi sérialisée au format JSON.

L'URL est alors décryptée, via des mécanismes REST, et elle est dirigée vers la méthode "createFlight" de la classe "FlightCollectionRessource". Suite à l'appel de cette méthode, le JSON est alors désérialisé, un objet "FlightDTO" est alors créé. La classe "FlightTraductor" est utilisée pour avoir un objet de domaine "Flight".

Le service "FlightService" est appelé, qui appelle "FlightDAO" afin de conserver le vol dans la base de données. Le nouveau vol est renvoyé à la couche REST qui convertit cet objet en objet DTO et sérialise en JSON le nouveau vol. Le nouveau vol est envoyé à l'administrateur. Enfin, une notification est affichée à l'administrateur avec un message "le vol est crée".

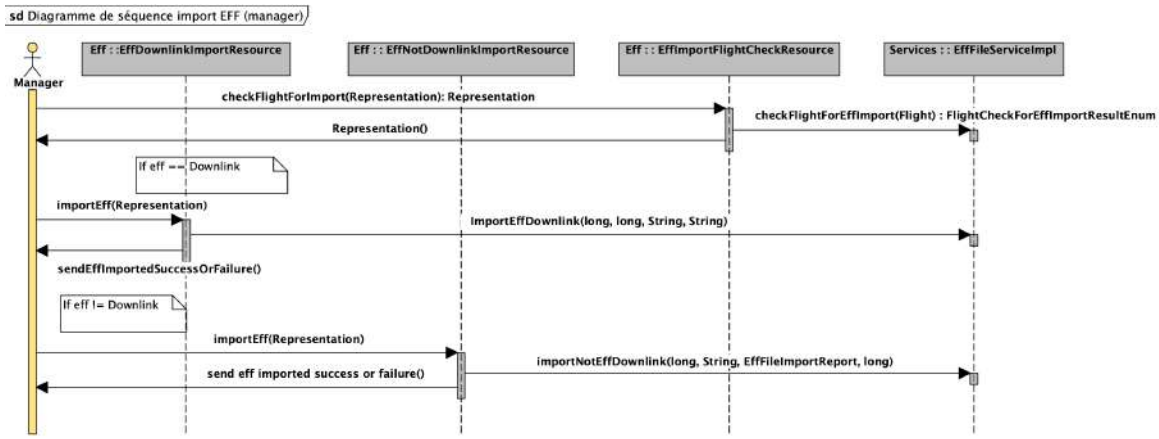

FIGURE 1.19 – Diagramme de séquence d'import d'un EFF.
(Source : Airbus)

Pour une illustration de l'aspect dynamique d'une fonctionnalité, la Figure 1.19, nous permet de voir toutes les interactions que l'utilisateur, ici le manager, a avec le système existant. Tout d'abord, le manager va chercher un vol compatible dans les classes de ressources des vols. Par la suite, le système va renvoyer une représentation du vol au manager. Une fois que la représentation du vol importé est faite, le système invite le manager à télécharger l'import. Une fois que cet import est fait un message est alors renvoyé au manager pour indiquer le succès ou l'échec du téléchargement.

1.4.5 Analyse des problèmes observés

Le besoin de réalisation d'une solution innovante découle de la finalité de *Airline 1*. Or pour apporter une réponse précise aux différents utilisateurs il est indispensable d'identifier les faiblesses du processus métier qui nécessite l'apport d'un nouveau système.

Aujourd'hui, le système *Airline 1* est d'ores et déjà en exploitation, il en résulte que très régulièrement nous avons des retours des utilisateurs sur les modifications souhaitées afin de rendre leur travail le plus efficace. Or, comme nous l'avons vu dans la sous-section 1.4.4 avec les différents diagrammes de séquences, voir Figure 1.18 et la Figure 1.19, chaque fonctionnalité développée représente plusieurs étapes dans le processus qui nécessitent d'être testées.

Le cycle de vie d'un logiciel fait que l'aspect des mises à jours, avec l'apport de nouvelles fonctionnalités, est important. La qualité est assurée par les activités de tests. Or c'est cette activité de test qui semble faire défaut et pour cela il faut établir un état des lieux pour choisir la stratégie adéquate qui permettra d'avoir un processus optimisé.
Le diagramme d'Ishikawa permet de mettre en évidence les causes pour aboutir au problème à résoudre.

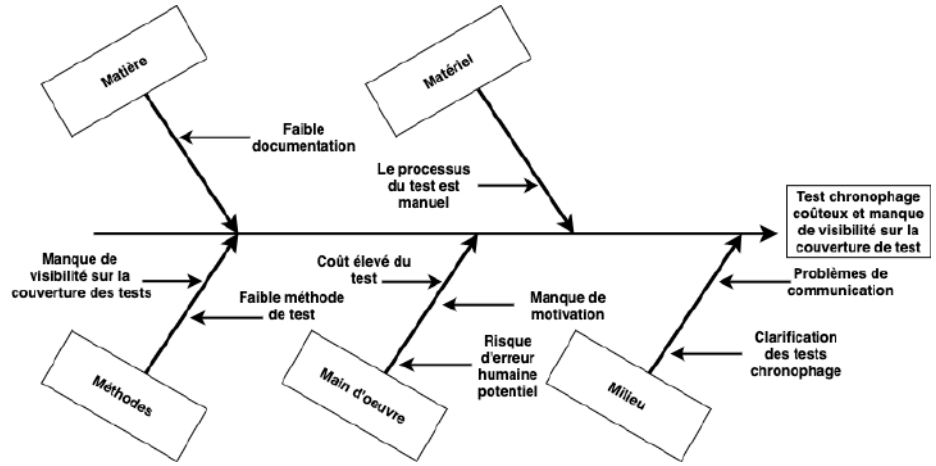

FIGURE 1.20 – Diagramme d'Ishikawa des causes et d'effets de l'application *Airline 1*.

Nous pouvons voir avec le diagramme d'Ishikawa, qu'il est nécessaire de mettre en place une solution nouvelle qui permettra d'avoir :

- Un test moins chronophage.
- Un test qui mobilise moins de ressources.
- Un test avec une couverture plus précise qui améliorera la qualité du test.

Avec le constat actuel, le fait que l'intégralité des tests soient faits manuellement, la solution la plus évidente serait de mettre en place des script de tests afin de rendre l'exécution automatique. Or comme le montre le Tableau I.2 ci-dessous, l'utilisation de scripts de tests permet de répondre à un certain type des problèmes posés mais ne permet pas de traiter le problème dans sa globalité.

Spécificités du test manuel	Spécificités du test automatique par script à la 1ère exécution
- La conception à un coût normal. - L'écriture à un coût réduit. - L'exécution à un coût très élevé (manuelle). - L'analyse à un coût moyen voir faible. - La maintenance à un coût faible. - L'archivage à un coût normal.	- La conception à un coût normal. - L'écriture à un coût très élevé. - L'exécution à un coût très faible (automatique). - L'analyse à un coût moyen voir élevé. - La maintenance à un coût élevé. - L'archivage à un coût normal.

TABLE I.2 – Tableau comparatif des spécificités des tests.
(Source : Tout sur le test logiciel [Hag19])

C'est pour cela qu'il est nécessaire de prendre un niveau d'abstraction plus élevé afin que l'intégralité des problèmes soient pris en compte.

Dans cette sous-section, nous avons pu voir que chaque nouvelle fonctionnalité proposée représente plusieurs éléments qui nécessitent d'être testés. La méthode de test actuelle fait que nous devons nous interroger pour pouvoir trouver une nouvelle méthode plus performante.

1.4.6 Finalité de la solution recherchée

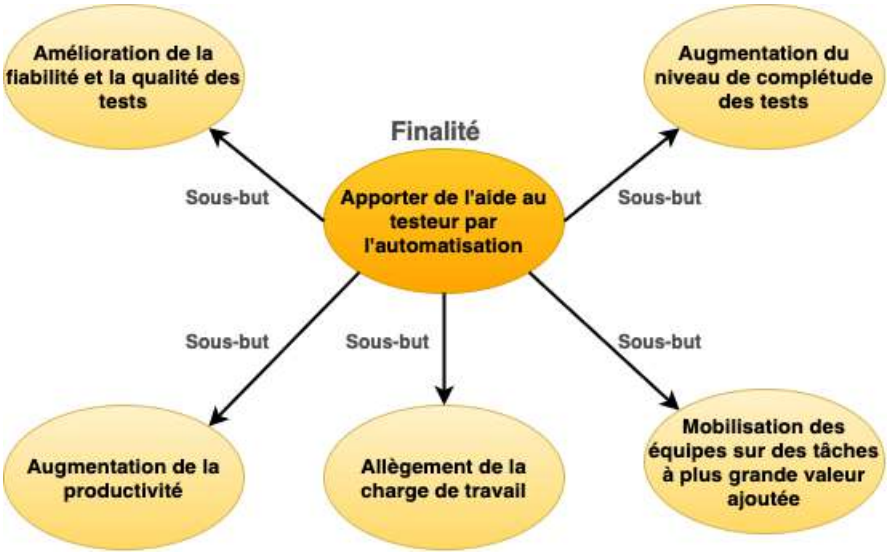

FIGURE 1.21 – Finalité de la solution.

Comme le montre la Figure 1.21 ci-dessus, il faut que la solution permette :

- Une amélioration de la fiabilité et de la qualité des tests, par un rapport qui répertorie toutes les étapes de tests.
- Une augmentation de la productivité, par des tâches qui peuvent être déléguées à la machine.
- Un allégement de la charge de travail par la réutilisabilité.
- Une augmentation du niveau de complétude des tests.
- Une mobilisation des équipes sur des tâches à plus grande valeur ajoutée, grâce à l'automatisation des tests.

Maintenant que le système existant a été vu dans dans globalité. Dans la prochaine section nous aborderons une première piste de solution envisagée pour améliorer le processus.

1.5 Solution envisagée

La solution envisagée réside à générer des tests par des modèles qui s'exécuteraient sur les tests de non-régression. Le résultat serait l'affichage de la liste des différents tests effectués sous la forme d'un rapport avec le statut du test afin de savoir si le test a été un succès ou non.
Cela implique que les différentes exigences au niveau des tests, exprimées par le client, doivent être traduites sous la forme de modèles. Une fois ce travail réalisé le testeur pourra intégrer ce modèle dans un générateur de test qui va alors produire les scénarios d'exécution. Ces scénarios d'exécution seront alors utilisés par un automate afin de rendre la génération comme l'exécution automatiques. Une fois l'exécution terminée un rapport sera publié.

1.5.1 Objectif de la solution

L'objectif de la solution doit permettre, à travers l'utilisation de modèles de tests, d'encapsuler différentes méthodes, de tests de non-régression, afin de pouvoir les exécuter de manière automatique.

Le diagramme de cas d'utilisation ci-dessous, montre les différentes étapes dont seule la création des modèles est à réaliser manuellement par l'ingénieur de tests. La création de modèles est une étape chronophage mais ces modèles seront réutilisables et les autres étapes seront elles automatisées.

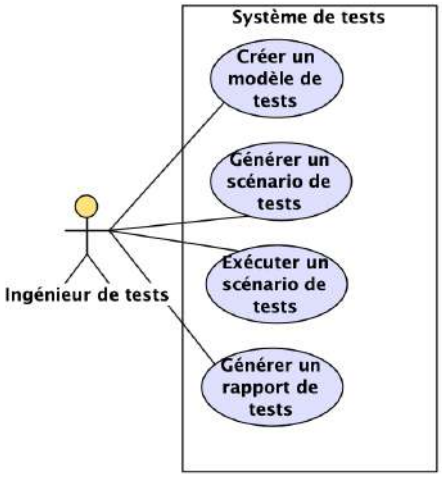

FIGURE 1.22 – Diagramme de cas d'utilisation de la solution.

1.5.2 Analyse fonctionnelle

D'après le digramme de cas d'utilisation de la solution à réaliser, voir la Figure 1.22, nous avons pu en déduire les exigences du système [RR18] représentées par le diagramme ci-dessous.

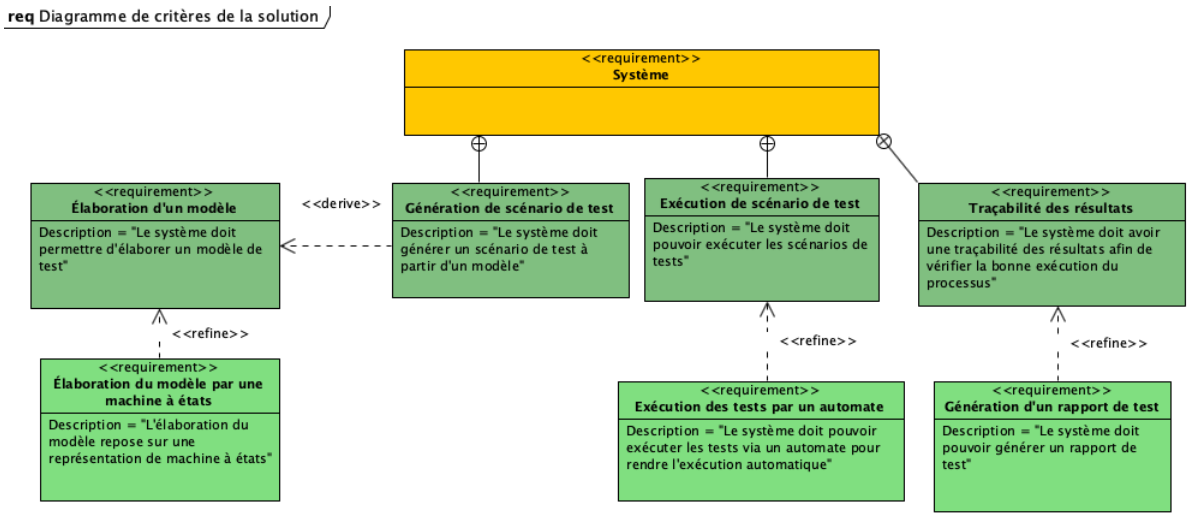

FIGURE 1.23 – Diagramme d'exigences de la solution.

La solution doit permettre :

- D'élaborer un modèle à partir des spécifications du test.
- De générer sur le système cible un scénario de tests grâce à une machine d'états.
- D'exécuter le scénario au moyen d'un automate.
- La rédaction automatique d'un rapport de résultat de tests qui permettra de valider la spécification de test.

Le diagramme de séquence ci-dessous montre l'utilisateur qui élabore le modèle de test suite aux différentes exigences du test. Cela permet d'avoir un système sous test (SUT). Une fois le modèle de test réalisé, il sera utilisé par le générateur afin de créer les différents scénarios d'exécution. Une fois que ce scénario est réalisé, il sera utilisé par un automate afin de rendre l'exécution du test automatique. Enfin, une fois que le test aura été exécuté, un rapport de test sera généré, afin de permettre à l'utilisateur de visualiser les résultats de la campagne de test.

La demande d'exécution de l'automate de test lancera un navigateur dont l'automate aura le contrôle. L'automate à l'aide du navigateur interrogera le Document Object Model (DOM) afin d'interagir avec les différents éléments de l'application suivant la logique du scénario de tests. Une fois le test terminé, l'automate fermera le navigateur et s'auto-détruira avant de produire

un rapport de test.

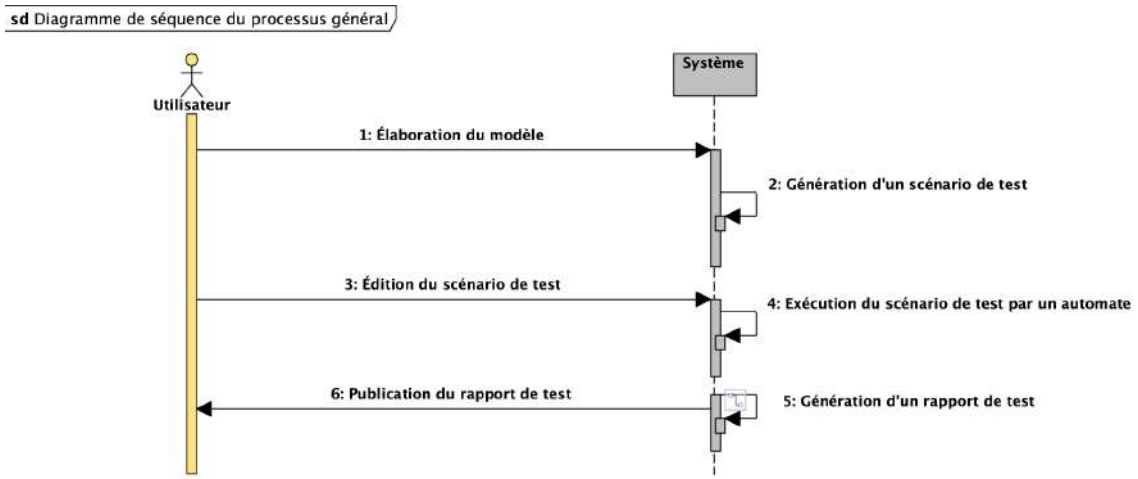

FIGURE 1.24 – Diagramme de séquence du processus général.

Afin d'avoir une idée "simplifiée" de la solution le schéma en boîte noire montre ce qui entre dans le système et ce qui en sort, sans trop détailler le système en lui-même. Elle se compose de :

- **En entrée du système :** Les différentes exigences de tests du clients.
- **À l'intérieur du système :** Les différents critères de sélection de tests, un client léger servant d'automate de test ainsi qu'un générateur de scripts de tests.
- **En sortie du système :** Un rapport de test est généré.

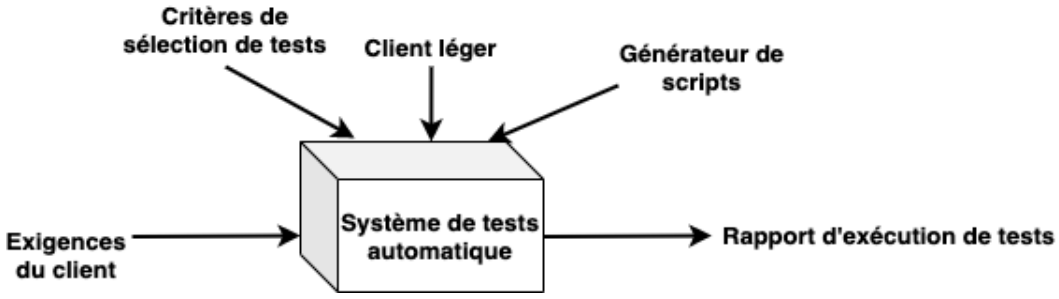

FIGURE 1.25 – Schéma en boîte noire de la solution envisagée.

1.5.3 Apport de la solution envisagée

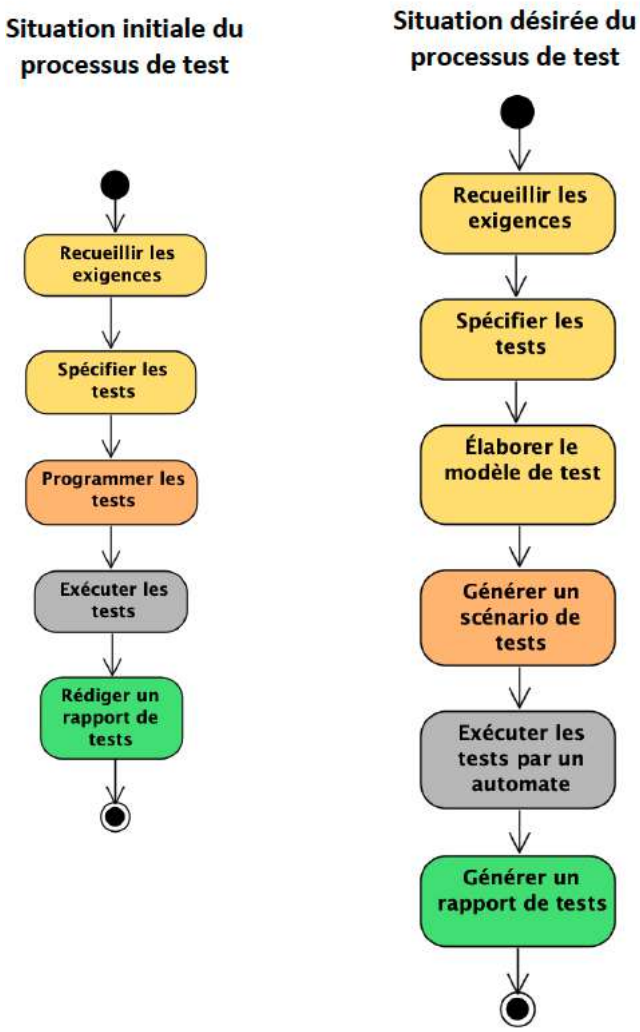

FIGURE 1.26 – Apport de la solution sur le processus initial.

Nous pouvons voir que dans la situation initiale les tests sont faits manuellement, alors que dans la solution proposée on constate que différentes étapes, celle de génération et d'exécution de test, seraient faites automatiquement, même celle de la génération du rapport de test.

1.6 Problématique

Afin de mettre en place la solution envisagée, les tests devront être générés automatiquement. Pour arriver à cet objectif, le système devra répondre à certains verrous technologiques :

- Pour fonctionner, le moteur de génération aura besoin d'un modèle de test.
- Les différentes fonctionnalités de l'application sont sujettes à des changements d'états. Il faudra trouver une solution pour que cela soit matérialisé à travers le modèle.
- Le moteur d'exécution doit pouvoir exécuter les différents tests de manière automatique.
- Enfin, pour rendre l'ensemble du processus "quasi-autonome", le système doit pouvoir générer un rapport de test.

La problématique pour répondre à ces différents verrous technologiques sera :

Comment générer et exécuter des tests automatiquement afin d'éviter la régression du système existant, vue la grande quantité de tests à effectuer, tout en limitant les coûts ?

Chapitre II

Génération automatique de tests dirigée par les modèles

Dans tout problème d'ingénierie l'objectif principal est de résoudre une problématique métier. Dans ce mémoire, il s'agit d'une mise en oeuvre d'une méthodologie de tests d'acceptance automatisés. Suivant ce postulat, nous recherchons une méthodologie permettant la génération automatique de tests d'acceptance à partir d'un artefact. Une fois que la problématique à résoudre est clairement définie, il devient naturel de faire l'inventaire de ce dont nous aurons besoin pour mettre en oeuvre cette méthodologie. Une fois cet inventaire réalisé, nous allons utiliser des modèles qui sont un des moyens qui vont nous permettre de décomposer et d'extraire un sujet de test en système sous test (SUT).

2.1 Principes généraux

Dans cette section seront abordés différents aspects des principes généraux, en premier lieux les tests logiciels, puis l'ingénierie dirigée par les modèles ainsi que celle axée sur le comportement et en dernier sera abordé le Domain Specific Language.

2.1.1 Les tests logiciels

Nous avons constaté lors du chapitre I qu'au vu du contexte il est nécessaire d'avoir un niveau d'abstraction plus élevé pour pouvoir prendre en considération tous les aspects des problèmes métier. L'ingénierie des modèles semble être une sérieuse piste pour mettre en place la solution de notre problème. Nous pouvons alors observer que la solution peut être vue comme un pipeline, un ensemble d'artefacts qui sont réutilisables les uns avec les autres. Cela signifie que chaque partie peut être modifiable afin de pouvoir garantir l'interopérabilité des différents éléments.

Au niveau du test logiciel, dans un test, il y a deux composants essentiels :

- La spécification de l'exigence du test.
- Les différentes étapes du test.

Ces deux constituants forment les éléments d'un scénario de test et, de ce fait, nécessite une compétence technique afin que les spécifications de l'exigence de test à réaliser soient pertinentes. Nous ne devons pas oublier qu'un scénario de test doit avant tout répondre aux questions : "Que fait réellement cette fonctionnalité ?" et "Quel est son processus de fonctionnement ?"

Cependant avant de nous intéresser en détail à l'approche des tests dirigés par les modèles, il est important de se rappeler pourquoi les tests, au sein d'un logiciel, sont importants et quels sont leurs différents aspects.

2.1.1.1 Les différents niveaux de tests logiciels

Un test permet d'évaluer un comportement d'exécution d'un logiciel, il en fait donc partie intégrante [Hag19]. Le choix des tests à réaliser se fait suivant des points stratégiques du système à étudier. Les systèmes actuels sont devenus si complexes qu'il est rare de pouvoir tout tester. Le test sert à vérifier que le développement est correct et également à vérifier que le logiciel en production correspond aux différentes exigences voulues par les parties prenantes.
Une application peut être vue comme une succession de couches, il en va de même pour les tests. Il existe différents niveaux de tests : unitaires, d'intégration, de validation et d'acceptation [Com18].

- *Les tests unitaires*, appelés également tests de composants, se concentrent uniquement sur les composants unitaires. Ces composants doivent pouvoir être isolés pour pouvoir être testés, ils doivent aussi pouvoir être remplacés.
- *Les tests d'intégration* concernent les interfaces qui relient les composants unitaires. Les choix architecturaux sont très importants parce que les interfaces doivent être clairement définies afin de pouvoir être testées et il est nécessaire de bien choisir l'ordre dans lequel ces tests pourront être effectués.
- *Les tests de validation et tests de système* vont tester une application, un système ou une fédération de systèmes, dans leurs globalités. Il est à noter que la différence fondamentale entre les tests de validation et les tests système concernent les spécifications : pour les spécifications fonctionnelles, qui correspondent aux attentes du logiciel, on utilisera les tests de validation ; pour les spécifications ou exigences non-fonctionnelles, qui correspondent à la manière dont les spécifications fonctionnelles doivent être réalisées, on utilisera les tests de système.
- *Les tests d'acceptation* Les tests d'acceptation concernent la préparation du système avant son utilisation, ils sont à égal niveau des tests de validation ou des tests système mais ils n'ont pas les mêmes buts. Il en existe de différents types :
  - *Les tests d'acceptation par les utilisateurs* permettent de s'assurer de l'utilisabilité du système par les utilisateurs finaux.
  - *Les tests d'acceptation opérationnelle* s'adressent surtout aux différents administrateurs du système et vérifient les opérations backup et les différentes restaurations.
  - *Les tests d'acceptation contractuelle* concernent l'acceptation par le client.
  - *Les tests d'acceptation réglementaire* concernent les obligations légales, gouvernementales ou de sécurité.

2.1.1.2 Les différents types de tests

S'il est assez simple de définir les différents niveaux de tests, la définition des types de tests est beaucoup plus floue. D'après le International Software Testing Qualifications Board (ISTQB), il s'agit d'un : *"Moyen de définir clairement l'objectif d'un certain niveau pour un programme ou un projet. Un type de test est focalisé sur un objectif de test spécifique (exemple : test de*

fiabilité, d'utilisabilité, de régression, etc.) et peut couvrir un ou plusieurs niveaux de tests et une ou plusieurs phases de tests" [Chr+15].

Il existe un très grand nombre de types de tests. Ils ne sont pas nécessairement corrélés ni à une technique spécifique ni aux différents niveaux de tests. Cette grande variété a été normée par l'Organisation Internationale de Normalisation (ISO).
Il a été normalisé huit familles de types de tests chacune correspondant à des caractéristiques spécifiques : fontionnels, de performance, de compatibilité, d'utilisabilité, de fiabilité, de sécurité, de maintenabilité et enfin tests de portabilité.

Vous trouverez ci-dessous un schéma illustrant cette grande variété de types de tests.

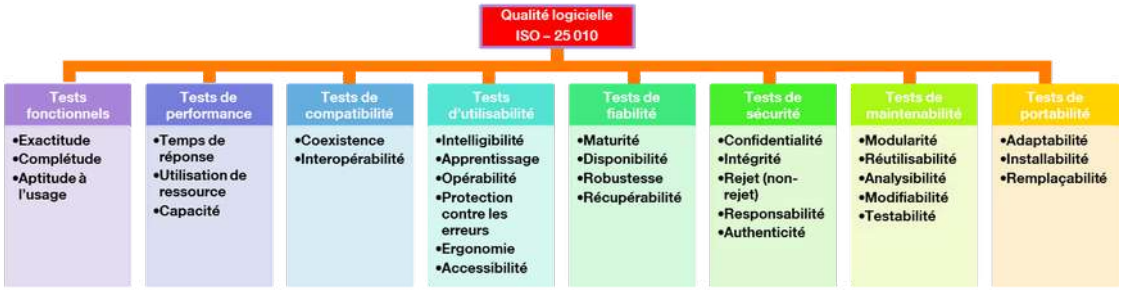

FIGURE 2.1 – Schéma des différents types de tests.
(Source : Organisation internationale de normalisation [Org17])

2.1.1.3 Les différents niveaux d'accessibilité

Nous avons vu les tests correspondant à différents niveaux ainsi qu'aux différents types, il existe aussi des tests en fonction de l'accessibilité. Nous pouvons citer : les tests en boîte noire, en boîte blanche et en boîte grise.

- **Le test de la boîte noire (également appelé test de fonctionnement) :** Ce test est basé sur l'idée que le testeur ne connaît pas le code source. Pour appliquer le test de la boîte noire, l'utilisateur fournit des entrées et vérifie les sorties en fonction des exigences du système à tester. Un des avantages des tests en boîte noire est que les non-programmeurs peuvent les réaliser car aucune connaissance du programme est requise. Un des inconvénients des tests en boîte noire, du fait que le code source est inconnu, est que tester toutes les entrées est impossible. Il existe plusieurs techniques pour tester la boîte noire, tels que

le test de partitionnement d'équivalence, les tests de valeurs limites, les graphiques de cause à effet et les tests aléatoires.

- *Test de partitionnement d'équivalence :* Tests dans lesquels l'entrée d'une série est divisée en différentes parties avec l'hypothèse que tester un élément de cette série équivaut à tester la série entière. Un exemple serait que, dans une série de nombres entiers, choisir à tester une valeur équivaudrait à tester toute cette série.
- *Test de la valeur limite :* Tests dans lesquels les limites des entrées et des sorties sont testées. Puisqu'un programme peut avoir un grand nombre d'entrées et sorties, le test de la valeur limite évitera que ces limites soient dépassées.
- *Représentation graphique cause-effet :* Cette représentation peut être appliquée aux tests en convertissant les exigences en causes et en effets. Par exemple, pour enregistrer un fichier, la longueur du nom du fichier doit être de dix caractères sinon une erreur est affichée. Cette exigence peut être convertie en une seule cause et en deux effets. La cause serait la longueur du nom de fichier se compose de dix caractères, tandis que les effets seraient : en premier, le fichier est enregistré puis en deuxième, un message d'erreur "longueur de nom de fichier invalide" s'affiche. Après la conversion, chaque cause et effet reçoit un identifiant. La réalisation d'un graphe booléen reliant ainsi les causes et les effets permet la conversion des identifiants pour construire des cas de test. En utilisant ce graphe une table de décision sera créée. Cette représentation est plus claire et réduit le temps nécessaire pour rechercher la cause d'une erreur puisque l'erreur peut être lié à un effet, qui est directement lié à ses causes possibles dans le graphique cause-effet.
- *Test aléatoire :* Cette technique est à utiliser uniquement en combinaison avec d'autres techniques de test parce que la probabilité de trouver une erreur est faible, bien que, en choisissant cette méthode une erreur improbable peut être détectée.

- **Les tests en boîte blanche :** Ces tests sont appelés également tests structurels puisqu'ils sont basés sur la structure interne du SUT. Ils ne se concentrent pas sur la spécification des exigences logicielles mais sur le code source lui-même qui est analysé à l'aide de la logique. Les différentes techniques de tests en boîte blanche couvrent : les instructions, les portions de code, les conditions et les portions/conditions.

  - — *Le test de couverture des instructions :* Le but est de s'assurer que chaque ligne de code, instruction, est exécutée au moins une fois. Bien que l'ensemble du programme puisse être testé, les portions peuvent être manquées. La couverture des instructions est donc une approche de couverture de code faible.
  - — *Le test de couverture de portion de code :* dans ce test, également appelé test de couverture de décision, toutes les portions de code du programme sont testées. Le testeur choisit la portion de code par laquelle il veut commencer le test. Généralement, ces portions de code sont construites avec des structures conditionnelles.
  - — *Le test de couverture des conditions :* Ce test est similaire au test de couverture des instructions. Au lieu que chaque portion de code soit à évaluer comme vrai ou faux, chaque condition au sein d'une portion de code doit maintenant être couverte.
  - — *Le test de portion/couverture de code :* La combinaison de ces deux tests est une méthode solide qui compense les éventuelles manques de chacune des deux techniques utilisées individuellement.

- **Le test en boîte grise :** Les tests en boîte grise combinent deux techniques de tests. Celle en boîte noire qui se concentre sur les spécifications des exigences et les fonctionnalités du système à tester, et celle en boîte blanche qui se concentre sur le code source et la logique.

Après avoir vu différents niveaux de tests, types de tests et différentes accessibilités des tests, il est à signaler que les différents types de tests sont dans une liste non-exhaustive. Nous pouvons en citer encore quelques-uns succinctement :

- **Tests de régression (appelé aussi de non-régression) :** Ils vérifient l'absence de "bugs" après modification.
- **Tests vitaux :** Ils testent fréquemment des petites fonctionnalités vitales de l'application.
- **Tests de bout en bout :** Ils testent toute l'application.
- **Tests exploratoires :** Ils sont réalisés lors d'une utilisation réaliste de l'application. Ils ne sont pas automatisables.
- **Tests Alpha :** Ils sont des tests d'acceptation "interne" réalisés par le testeur.
- **Tests Beta :** Ils sont des tests d'acceptation "externe" réalisés par le client.

Afin d'avoir une vue plus globale de ce que nous avons abordé jusqu'à présent dans cette sous-section, il est intéressant de trouver dans le Tableau II.1, ci-dessous, le maillage qu'il y a entre les différents niveaux de tests, les différents types de tests et les niveaux d'accessibilité.

Les familles de tests l'ISO-25 010, les tests de régression, les tests vitaux et les tests exploratoires peuvent être effectués sur tous les niveaux de tests. Ceux en boîte noire aussi alors que nous les voyons plus rarement sur le niveau des tests unitaires. Les tests boîte blanche, sur tous les niveaux mais rarement au niveau des tests d'acceptation. Les tests de "bout en bout" sont réalisés au niveau des tests systèmes et d'acceptations. Pour finir, les tests Alpha et Beta sont uniquement des tests d'acceptation.

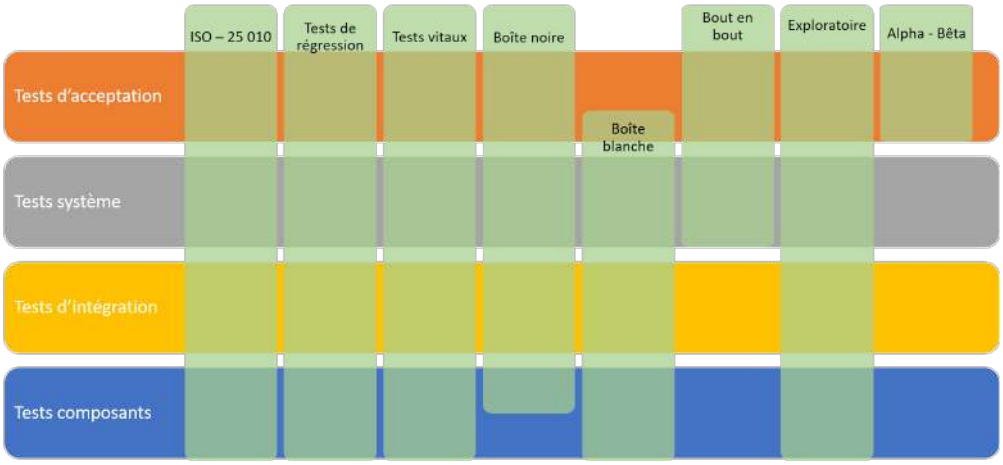

TABLE II.1 – Tableau des niveaux vs types de tests.
(Source : La taverne du testeur [Cha19])

Nous avons vu jusqu'ici que le test n'était pas seulement qu'une simple étape dans le cycle de vie du logiciel, il est comme l'implémentation de la solution un monde à part entière. En effet, cela ne se résume pas à tester la fonction que nous avons précédemment implémentée mais bien à voir le code sur différents aspects, son type, son accessibilité... De ce fait il serait intéressant de pouvoir automatiser ces différents aspects.

2.1.1.4 Les tests automatiques par scripts

Le test logiciel est une phase cruciale du processus de cycle de vie du développement logiciel. Il consomme près de 50%, voir plus, du processus de développement du logiciel. Ceci ne doit pas en impacter la qualité. Lors de la conception d'un système des étapes de validations et de vérifications garantissent la qualité. La norme internationale ISO 9000 :2015, standardise cette qualité.

La norme ISO 25051 :2014 [Org14] définit les tests comme un travail dans lequel une exécution se fait dans des conditions déterminées sur un système ou un composant. Ces mêmes tests exécutés produisent alors des résultats qui seront évalués afin de vérifier certains aspects du système [GHL16].

Cependant, comment vérifier si les exigences qui ont été commandées par le client ont bien été respectées lors de l'élaboration du produit final ? C'est justement à cela que servent les tests, à faire le lien entre réalisation et conformité. Or, l'activité de test est justement celle qui est la plus consommatrice en terme de ressources (humaine, temps et coût). C'est pour cette raison que les entreprises du numérique cherchent à mieux appréhender ce processus, afin de le rendre plus efficace, adaptable et enfin le plus rentable possible. Or, parmi ces différents facteurs, celui sur lequel l'on peut influer est celui du temps passé sur les tests. En effet, grâce à l'automatisation des tests par scripts, le développeur se consacre à l'élaboration du test et se décharge de l'exécution, vu que cette dernière sera faite automatiquement.

Pour le développeur ou le testeur, la mise en oeuvre d'un test automatique nécessite un effort de mise en application. Il faut donc s'interroger sur le bien-fondé d'une automatisation de test. Celle-ci aura un coût plus important au début, mais il y aura un gain certain au niveau du temps passé sur la réalisation et l'exécution du test, ces étapes pouvant être automatisées.

L'automatisation de test doit se baser sur certains critères, les facteurs ne doivent pas être choisis au hasard. Ci-dessous une liste, non exhaustive, des critères à retenir pour choisir d'automatiser des tests :

- Vérifier en premier lieu la faisabilité technique.
- Estimer si la fréquence d'utilisation sera importante.
- Prévoir la réutilisabilité des éléments.
- Au niveau de la criticité, quand le test manuel est trop complexe.

- Quand la quantité de ressources manuelles nécessaires sont trop importantes.

Pour aborder l'automatisation des tests, il existe différentes approches. Celle qui sera choisie dépendra bien entendu de ce que l'on souhaite tester.

Dans ce mémoire nous allons nous intéresser à l'approche d'automatisation dirigée par les modèles.

2.1.2 L'ingénierie dirigée par les modèles

Face aux différentes activités du cycle de développement du logiciel, l'ingénierie des modèles (IDM) à un rôle d'unificateur. Les systèmes que l'on souhaite construire sont de complexité croissante et on doit toujours trouver le bon compromis entre les trois critères opposés : temporel, financier, sans pour autant négliger la qualité. Le but est de faire un bon logiciel en respectant le planning de réalisation et le budget attribué.

Actuellement, il faut maîtriser les variabilités spatiales et temporelles qui sont des enjeux majeurs de l'ingénierie du logiciel. En effet, les problèmes se concentrent sur le fait qu'il faut toujours s'adapter à des nouvelles spécifications et souvent, au lieu de créer un produit, il est demandé d'en faire toute une gamme de façon simultanée. Microsoft est un bon exemple puisqu'il adapte ses logiciels à des dizaines de plates-formes tout en s'adaptant à des centaines d'environnements différents.

En effet, les fiches de tests, le code sources, les divers diagrammes sont différents artefacts qui entourent la pratique du développement d'une solution informatique. Ces artefacts permettent d'avoir différentes vues du système que l'on souhaite développer. De nombreuses solutions technologiques ont été développées pour montrer que ces différents artefacts sont bien liés entres eux. Nous pouvons citer par exemple : le langage orienté objet, l'UML, etc. L'ingénierie dirigé par les modèles permet justement d'harmoniser ces différentes transformations d'artefacts par l'utilisation de modèles. Elle propose une technologie et un cadre méthodologique pour rendre le processus cohérent.

Seulement, chaque solution à développer amène son propre lot de complexités et les différentes approches possibles sont standardisées par l'Object Management Group (OMG). Cette standardisation a pour objectif de décomplexifier la conception des systèmes. Au même titre qu'il a standardisé IDM, en anglais MDE, l'OMG a aussi normalisé les autres approches dirigées

par les modèles telles que : Model Driven Architecture (MDA) et Model Driven Development (MDD) sans oublier le Model-Based Testing (MBT). Cette dernière approche est orientée vers les tests logiciels basés sur les modèles.

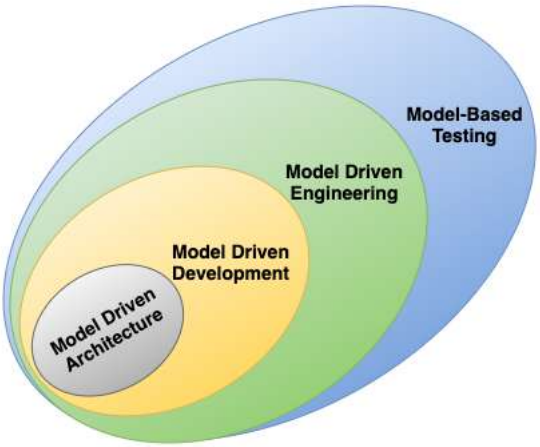

FIGURE 2.2 – Schéma des différents modèles proposé par l'OMG.

Model Driven Architecture (MDA) : En 2000, l'OMG a défini le MDA dans le but de déterminer : quoi ? quand ? comment ?, et pourquoi ? modéliser. Cette approche est plus globale dans laquelle il devient primordial la notion de modèle, parce que l'interopérabilité dans les systèmes ne peut se faire qu'à l'aide de la standardisation. Le principe fondamental du MDA est l'utilisation de modèles aux différentes phases du cycle de développement d'un logiciel. Afin d'obtenir un gain de productivité et de générer automatiquement la totalité du code des logiciels, le but de MDA est la création de modèles pérennes qui soient indépendants des plates-formes d'exécution et de leurs détails techniques tout en les prenants en compte [BS05 ; JCV12].

Model Driven Development (MDD) : Le développement piloté par modèle (MDD) est une ingénierie d'approche logiciel qui utilise les modèles et les technologies de modèles pour élever le niveau d'abstraction. Ce niveau d'abstraction plus élevé permet aux développeurs notamment de créer et d'évoluer des logiciels, afin de les simplifier et les formaliser, ou normaliser, pour que soit possible l'automatisation les différentes tâches et activités composant le cycle de vie du logiciel. Les exigences, l'analyse, la conception et l'implémentation sont les principales bases de cette approche. Pour spécifier le système à étudier cette approche définit des langages de

modélisations qui permettront les transformations qui eux mêmes pourront améliorer la qualité des processus de développement logiciel et améliorer la productivité [HT06].

Model-Driven Engineering (MDE) : en français l'Ingénierie Dirigée par les Modèles (IDM), met le modèle au premier plan pour le processus d'abstraction à haut niveau des applications. Cette abstraction permet de développer des systèmes plus complexes que la méthode de programmation classique. Si les modèles sont suffisamment spécifiés, ils pourront être utilisés par des machines qui, en les transformant, généreront d'autres modèles en sortie. Nous pouvons en déduire que IDM se fait à partir de modèles, de plus c'est une ingénierie génératrice puisqu'elle génère d'autres modèles. L'approche IDM a pour but de créer une application sous forme de code en partant de modèles et permet d'utiliser ces modèles de manière intensive tout en étant souple et interactive. Souple parce que cette approche s'adapte aux spécifications sans cesse changeantes, et interactive parce que les transformations et les raffinements successifs enrichissent le processus de développement.

Model-Based Testing (MBT) : Un test logiciel se doit d'être autant efficace qu'efficient et l'automatisation par l'approche MBT le permet. De plus, un meilleur contrôle sur les cas de tests est possible grâce aux modèles de MBT. L'automatisation du processus de test est le fondement de cette approche. En effet, le test logiciel fait partie intégrante du cycle de vie d'un logiciel. Les tests manuels réclament une spécification du design, là où le Model-Based Testing crée un modèle qui est certes chronophage à établir mais les autres phases de tests, au lieu d'être manuelles, sont automatiques et en conséquence il y a un gain de temps certain pour la réalisation des tests. Nous verrons dans la sous-section 2.3.2 plus en détail la méthode MBT.

Pour mieux comprendre, nous allons développer l'approche MDA et la transformation des modèles.

2.1.2.1 Model-Driven Architecture au centre de l'ingénierie

En 2001, l'OMG a proposé l'approche MDA qui est en fait une complémentarité du MDD. Celui-ci est un paradigme flexible n'appliquant pas les normes de OMG, là où MDA les utilise car tous les éléments sont considérés comme des modèles pour que leurs transformations successives aboutissent au code.

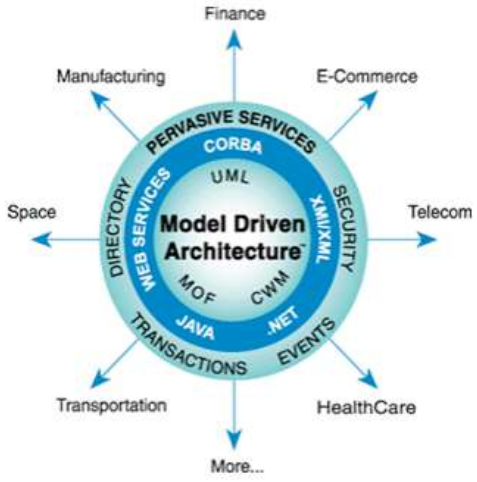

FIGURE 2.3 – Model Driven Architecture de l'OMG.
(Source : L'Object Management Group [Gro])

La figure ci-dessus, montre que l'OMG s'est basée sur le standard CWM (Common Warehouse Metamodel), le MOF (Meta-Object Facility) et UML (Unified Modeling Language). Un autre standard, XMI (XML Metadata Interchange) permet de dialoguer entre les différents middlewares (Java, CORBA, .NET et web services) se trouvant en deuxième couche. Dans la couche suivante se trouvent les services permettant de gérer la sécurité, les transactions, les évènements et les répertoires. La dernière couche propose différents domaines d'applications et leurs outils (commerce électronique, Espace, Finance, manufacture, médecine, Télécommunication, Transport...).

MDA est à la fois une démarche de développement et une architecture. Cette démarche de développement se base sur la différenciation des spécifications fonctionnelles de son élaboration sur une plateforme donnée. Son architecture est construite sur quatre niveaux :

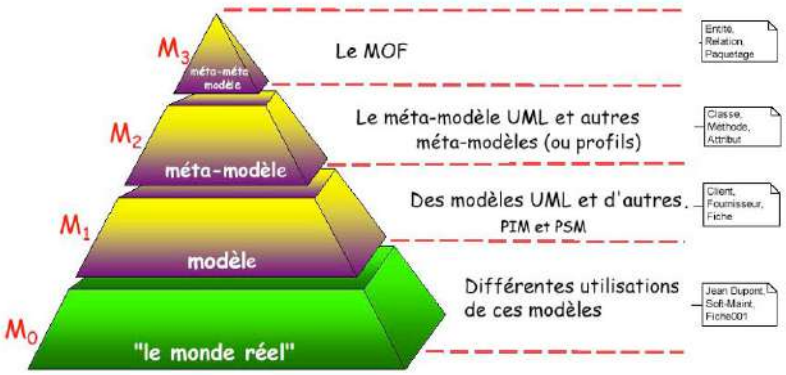

FIGURE 2.4 – Schéma d'architecture à quatre niveaux.
(Source : MDA (model driven architecture) principes et états de l'art [And04])

- Le niveau M0 comprend les informations réelles de l'utilisateur, il correspond au monde réel et c'est l'instance du modèle de M1.

Les trois niveaux suivants font parties du domaine de la modélisation.

- Le niveau M1 correspond au modèle, il est formé de modèles d'information. Il détaille les informations du niveau M0. Les modèles qui correspondent à ce niveau sont : UML, PIM et PSM (ces modèles seront décrits ultérieurement). Les modèles M1 contiennent des instructions des méta-modèles du M2.
- Le niveau M2 correspond au méta-modèle. La représentation des modèles M1 est définie par la grammaire et le langage de modélisation de M2. C'est à ce niveau qu'appartiennent les méta-modèles UML Les méta-modèles sont des instances du Meta-Object Facility (MOF).
- Le niveau M3 correspond au méta-méta-modèle. Il est formé d'un seul constituant appelé MOF. Le MOF décrit les structures des méta-modèles, ce qui permet de pouvoir modifier les existants.

L'approche MDA modélise les applications informatiques. L'élaboration d'une application a besoin de modèles qui représentent les informations nécessaires qu'elles soient totales ou partielles. La modélisation de l'approche MDA sépare les aspects métiers des aspects techniques. En partant des modèles, la génération automatique permet d'obtenir le code source. Les modèles deviennent des éléments productifs qui sont au coeur du processus.

Les différents modèles sont :

- *CIM (Computation Independant Model) :* Modèle indépendant de tout système informatique qui utilise un vocabulaire familier au maître d'ouvrage. Il permet d'avoir une vision de ce qui est attendu du système, sans détailler ni sa structure ni son implémentation. Il a une longue durée de vie puisqu'il est seulement modifié quand il y a des changements dans les connaissances ou dans les besoins métier.
- *PIM (Platform Independant Model) :* Modèle qui décrit la logique métier ainsi que le fonctionnement des entités et des services. C'est un modèle qui ne contient pas d'information sur les technologies qui seront utilisées pour déployer l'application. Il représente le fonctionnement des entités et des services. Il décrit le système, mais ne montre pas

les détails de son utilisation sur la plateforme. Il est constitué de diagrammes de classe du langage UML avec des contraintes en langage OCL. L'indépendance technique de ce modèle lui permet de garder tout son intérêt au cours du temps.

- *PDM (Platform Dependant Model) :* Bien que cette notion ne soit pas correctement définie par l'OMG, il s'agit d'une piste de recherche. Le PDM contient les informations pour la transformation de modèles vers une plateforme particulière qui est spécifique à celle-ci. Ce modèle de transformation permet le passage d'un modèle PIM vers un modèle PSM grâce à la définition de plateforme correspondant, par exemple, à : DotNet, le Web, CORBA, etc...
- *PSM (Platform Specific Model) :* Modèle dépendant de la plateforme technique spécifiée. Il sert essentiellement de base à la génération de code exécutable vers la ou les plateformes techniques cibles. Il existe plusieurs niveaux de PSM [Abd16]. Le premier est issu de la transformation d'un PIM avec un PDM, tandis que les autres sont obtenus par transformations successives jusqu'au code dans un langage spécifique. Les schémas des tables, les bibliothèques utilisées, le code du programme et autres sont décrits dans le dernier niveau : le PSM d'implantation.

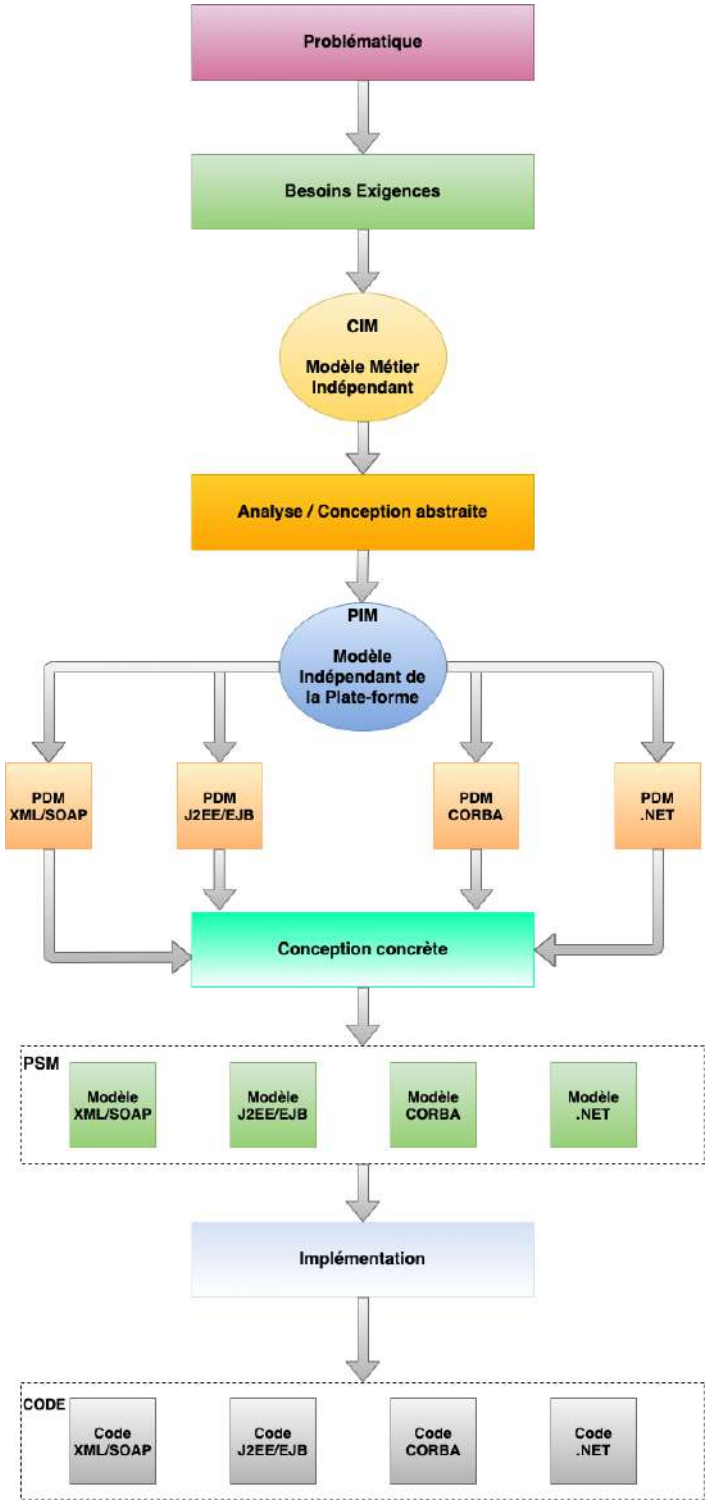

FIGURE 2.5 – Schéma des transformations successives de l'approche MDA.

Dans MDA, l'ensemble des éléments est considéré comme des modèles, quel que soient leurs niveaux hiérarchiques. Cela implique que même les niveaux les plus abstraits, comme ceux du méta-méta-modèles et méta-modèles, sont également des modèles [Lac17]. Afin d'éviter toute ambiguïté une modélisation sur quatre niveaux a été apporté par l'OMG.

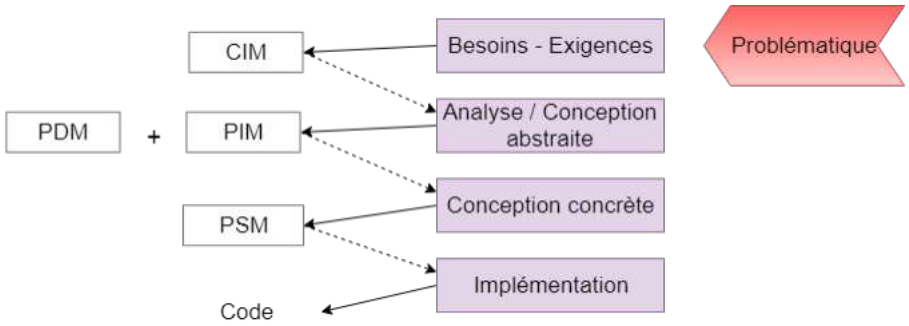

FIGURE 2.6 – Schéma de l'architecture de MDA à 4 niveaux d'abstraction.
(Source : Création d'un générateur de code générique, dirigé par les modèles, permettant la génération de vues, afin de représenter des scénarii de configuration, mettant en oeuvre un sous-système antenne [Fou18])

En résumé, nous pouvons dire que MDA est un processus qui met en oeuvre des outils pour concevoir et spécifier un système séparément de la plateforme qui le supporte en réalisant un PIM [BB02]. Des étapes successives vont pouvoir enrichir ce modèle. MDA spécifie alors les plateformes et choisit pour le système la plateforme adéquate pour transformer les PIM, la spécification du système en une nouvelle spécification pour une plateforme déterminée (PSM). MDA peut réaliser la transformation d'un CIM en un PIM et même un PIM à l'aide d'un PSM. La transformation se terminera par l'obtention d'une implémentation exécutable par le raffinement du PSM.

2.1.2.2 Transformation de modèle

Dans l'Ingénierie Dirigé par les Modèles (IDM), l'étape incontournable est celle de la transformation. En effet, une transformation de modèle, qui peut être multiple, permet à un modèle de passer d'un état abstrait à un état productif [SK03]. Tout programme comportant en entrée des "modèles source" et en sortie des "modèles cible" est qualifié de "transformation" en l'IDM. Elle est appelée "Model to Model" (M2M) quand en sortie il y a un modèle cible et "Model to Text" (M2T) quand en sortie il y a production de code. Un modèle cible est un modèle qui a subi une transformation tout en se conformant à un langage de transformation d'un méta-formalisme.

Un même espace technique dans lequel évoluent les modèles source et modèles cible n'est pas une contrainte. Nous pouvons tout à fait envisager la possibilité d'avoir ces modèles dans deux espaces techniques distincts, du moment que nous disposons d'outils qui nous permettent de faire la connexion entre ces deux espaces. A ce sujet, il existe déjà une multitude d'outils IDM afin de permettre la réalisation des connexions et des transformations de modèles.

Il est important de bien rappeler quelques définitions, que nous pouvons trouver dans divers ouvrages [MV06 ; KWB03], afin de mieux comprendre :

- Transformation : *"Une transformation est la génération automatique d'un modèle cible à partir d'un modèle source". "Une transformation doit suivre un ensemble de règles de transformation qui décrivent comment un modèle dans le langage source peut être transformé en modèle dans le langage cible".*
- Règle de transformation : *"Une règle de transformation est une description de la méthode à suivre pour que une ou plusieurs constructions dans le langage source puissent être transformées en une ou plusieurs constructions dans le langage cible".*

Transformation de modèle typé :

Une transformation de modèle typé nécessite des méta-modèles sources et cibles mais surtout, dans le méta-modèle source, un modèle typé. Ces méta-modèles, source et cible, ne partagent pas forcément le même mêta-modèle puisqu'il n'est pas nécessaire qu'ils soient dans le même espace technique. Quand l'espace technique n'est pas partagé, la définition de la transformation se fait dans un langage dont le méta-modèle se conforme soit au méta-formalisme source, soit au méta-formalisme cible. Une transformation est spécifiée (définie) au niveau des méta-modèles, M2, alors que son exécution s'effectue au niveau des modèles, M1. En d'autres termes, une transformation peut être vue comme une fonction d'un système qui a des éléments d'un ensemble de départ associé à des éléments d'un ensemble d'arrivée.

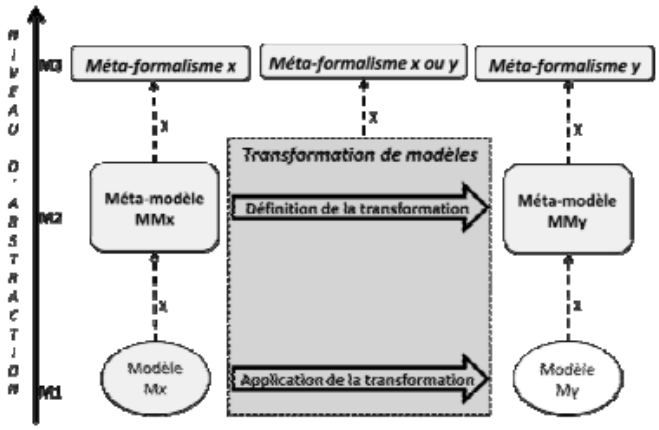

FIGURE 2.7 – Schéma de transformation de modèle type.
(Source : Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation [Gar13])

La Figure 2.7 illustre un exemple de transformation type dans lequel un modèle My est généré (conforme à son méta-modèle MMY) à partir d'un modèle Mx (conforme à son méta-modèle MMX) et de règles de transformations du niveau M2 exécutées en M1.

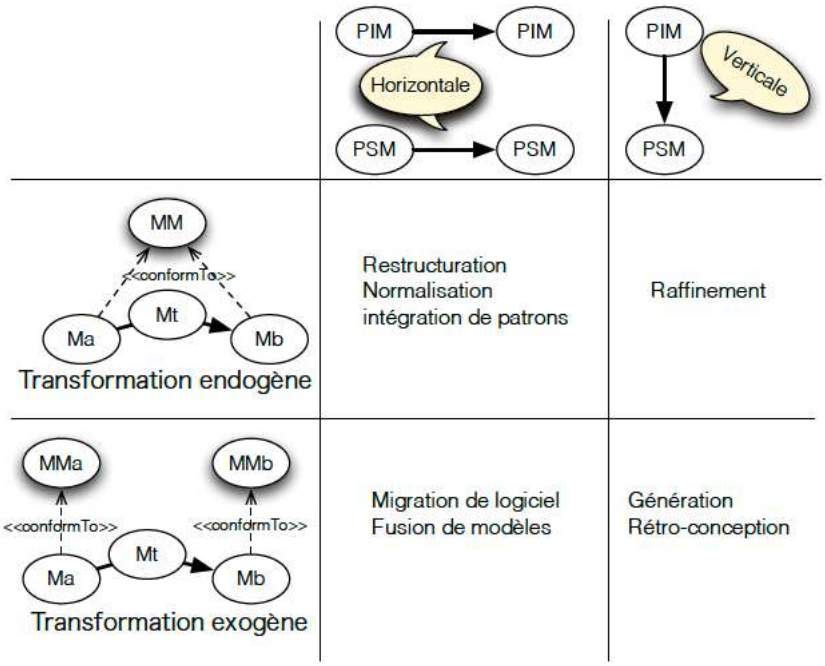

TABLE II.2 – Tableau des différents types de transformation et leurs principales utilisations.
(Source : Ingénierie dirigée par les modèles [JCV12])

Un exemple de transformation est montré dans le tableau ci-dessus. Un modèle Ma correspondant à son méta-modèle MMa génère un modèle Mb correspondant à son méta-modèle MMb avec des règles exécutées au niveau M1 alors qu'elles sont établies au niveau M2.

Il est à noter qu'une relation entre transformation et son inverse est comparable, en mathématiques, ce qui lie une fonction à sa réciproque. Quand un modèle source peut être retrouvé à partir d'un modèle cible, la transformation est appelée *réversible*.

Il est qualifié d'*endogène* la transformation concernant un modèle source possédant le même méta-modèle que le modèle cible et d'*exogène* la transformation comprenant des méta-modèles source et cible différents, ils ne sont pas partagés. La Figure 2.7 montre une transformation *exogène*. La transformation endogène sera utilisée par exemple pour l'optimisation de modèles, pour une simplification ou bien pour un *refactoring*. Alors que, la transformation exogène sera utilisée pour la génération de code ou pour effectuer des migrations de modèles sans changer de niveau d'abstraction. Elle est utilisé aussi pour la rétro-conception ou la rétro-ingénierie, *reverse engineering*.

Il est à noter que la transformation endogène est appelée *rephrasing*, reformulation en transformation de programme, et que la transformation exogène est appelée *translation*.

Une transformation de modèle, comme le montre le Tableau II.2, est appelée *horizontale* lorsqu'elle se réalise sur le même niveau d'abstraction, à contrario elle sera qualifiée de *verticale* lorsque différents niveaux d'abstraction sont concernés.

Il ne faut pas confondre les niveaux de modélisation avec les niveaux d'abstraction. Une transformation verticale sera toujours spécifié en M2 et sera exécutée sur un modèle M1 afin de produire un nouveau modèle M1. Une transformation impliquant deux modèles situés au même niveau M1 peuvent très bien avoir différents niveaux d'abstraction. Par exemple : un modèle générique sera le modèle source, là où le modèle de code spécifique à la plateforme Java sera le modèle cible : la transformation verticale est représentée par la génération de code.

Pour conclure ce que nous avons vu précédemment, voici une illustration d'une transformation de modèle incluant les règles de transformation que nous détaillerons dans la sous-section suivante.

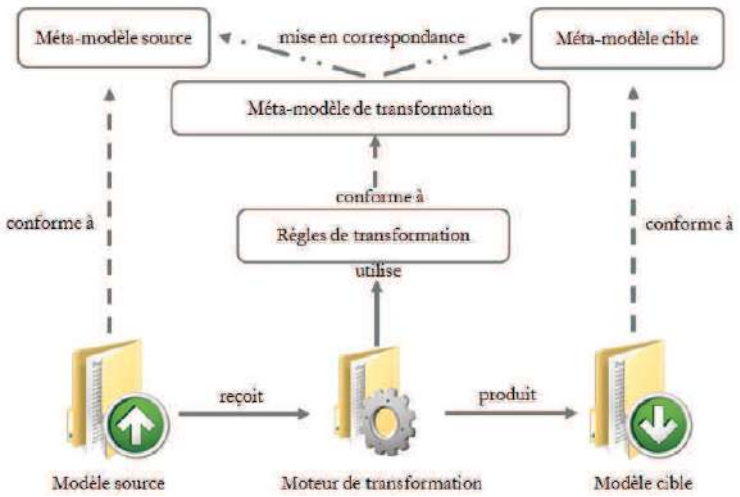

FIGURE 2.8 – Principe de transformation de modèle.
(Source : Aide à la conception de système d'information collaboratif, support de l'interopérabilité des entreprises [Tou07])

La figure ci-dessus montre que le moteur de transformation reçoit un modèle source conforme à son méta-modèle source. Ce moteur de transformation produira un modèle cible qui sera conforme à son méta-modèle. Pour ce faire, il utilise des règles de transformation qui sont conformes au méta-modèle de transformation qui met en correspondance le méta-modèle source avec le méta-modèle cible.

2.1.2.3 Structure de la transformation de règles

Nous avons vu la transformation des modèles dans sa globalité, nous allons voir maintenant que la structure de cette transformation, composée toujours d'un modèle source et d'un modèle cible, comporte des règles qui permettent de relier ces éléments.

La structure d'une règle de transformation de modèle est composée de deux membres :

- Un membre de gauche, "Left-Hand Side" (LHS), qui communique avec le modèle source.
- Un membre de droite, "Right-Hand Side" (RHS), qui communique avec le modèle cible.

Ces règles possèdent différents éléments qui peuvent être optionnels comme les variables, ou obligatoires comme les patterns et les expressions logiques. [CH03] :

- Les *variables* sont optionnelles et proviennent aussi bien du modèle source que du modèle cible, voir même d'éléments entre-deux
- Les *patterns* peuvent, ou pas, contenir des variables. Ils sont aussi appelés patrons et

se sont des fragments de modèles. Il en existe de plusieurs sortes, ils peuvent être des graphes, des chaînes de caractères, des termes...

- Les *expressions logiques* sont les éléments obligatoires qui permettent, par exemple, de décrire des calculs ou exprimer des exigences, contraintes, dans les modèles source et dans les modèles cible. On peut les exécuter de manière déclarative ou bien impérative. Les expressions OCL qui vont récupérer des éléments source seront exécutés de manière déclarative ; alors que les modèles directement manipulés par un code informatique seront exécutés de manière impérative, par exemple : la modification, l'ajout ou la suppression d'éléments. Ces expressions peuvent être spécifiés à l'aide des langages comme ATL, et ce de manière déclarative et/ou impérative. Elle peut être aussi non exécutable dans le cas où elles décrivent seulement une relation entre les modèles.

La figure ci-dessous montre la structure d'une règle de transformation.

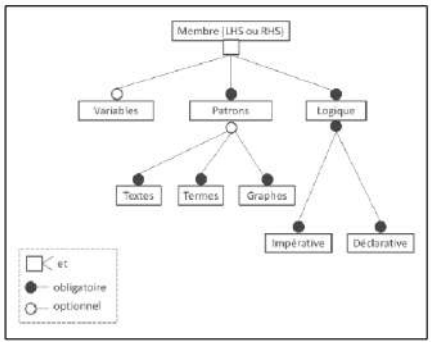

FIGURE 2.9 – Schéma d'une structure d'une règle de transformation.
(Source : Feature-Oriented Domain Analysis (FODA) Feasibility Study [Kan+90])

2.1.2.4 Différentes approches de transformation de modèles

Il existe deux manières pour aborder les transformations de modèles : une approche appelée M2M, de modèle vers modèle, et une autre approche appelée M2T, de modèle vers texte (code). Dans l'absolue un code est assimilé à un modèle, c'est pourquoi d'après l'IDM l'approche M2M ne devrait pas se différencier d'une approche M2T. La cible méta-modèle dans un langage de programmation est implicite à la transformation d'un modèle vers du texte (M2T).
Différentes approches sont possibles dans la partie M2M et seront abordées ci-dessous, de même que les différentes approches de génération de code de la partie M2T.

Le Tableau II.3 ci-dessous montre les différentes approches que nous allons aborder dans cette partie.

Approches de transformation	
Transformation M2M	Transformation M2T
Par manipulation directe	Génération de code par parcours de modèle
Opérationnelle	Génération de code par template
Par template	
Dirigée par la structure	
Par graphes	

TABLE II.3 – Tableau des différentes approches de transformations.
(Source : Approche de métamodélisation pour la simulation et la vérification de modèle [Com08])

Approches M2M :

L'approche par manipulation directe est située à un niveau bas d'abstraction et elle est de type impératif. Elle montre une représentation interne du modèle et des API pour manipuler les transformations. L'infrastructure des transformations est fournie par l'implémentation réalisée dans un environnement orienté objet.

L'approche opérationnelle est une variante de l'approche par manipulation directe tout en se situant dans un niveau d'abstraction plus élevé. Elle est aussi de type impératif. Il est habituel d'utiliser par exemple OCL pour étendre le méta-formulisme. On retrouve cette approche dans des langages comme Model Transformation Language (MTL) [VJ04 ; Fle06].

Les règles de transformation tiennent aussi compte de certains aspects qualifiés de remarquables : l'aspect déclaratif/impératif, la directionnalité, le paramétrage, l'appel aux structures intermédiaires, la séparation syntaxique, la stratégie d'application, la planification (ordonnancement) des règles et la composition de règles.

L'approche relationnelle spécifie les liens d'une relation entre source et cible à l'aide de contraintes. Il s'agit ici du sens mathématique du terme de relation. Parmi différentes approches relationnelles on peut citer : le plugin Eclipse Tefkat [LS05], Kent Model Transformation Language (KMTF), MOF QVT Relations, Model Transformation Framework (MTF), open-source.

L'approche par template, un template de modèle est un modèle qui intègre un méta-code chargé de calculer, d'évaluer, les variables. Ce meta-code est un OCL, là où les templates sont généralement dans le langage du modèle cible, dans sa syntaxe spécifique. Cette approche est aussi

très souvent utilisée en M2T [CA05].

L'approche dirigée par la structure se compose de deux phases différenciées : la première est celle qui crée la structure du modèle cible, là où la seconde instruit les références et les attributs. L'environnement de transformation déterminera les stratégies d'application ainsi que l'ordonnancement des règles de transformation. Auparavant, l'utilisateur aura seulement fourni ces règles. Dans cette approche, tout est orienté cible, même l'organisation des règles, puisqu'il en existe une pour chaque type d'élément cible et leur imbrication dépend de la structure du méta-modèle cible.

L'approche par graphe peut exprimer la transformation du modèle de façon déclarative tel que le font les approches relationnelles. Elle concerne principalement les environnements du secteur de la recherche académique [And+99]. Les modèles de type UML sont représentés par des graphes typés, étiquetés, attribués. Ils sont composés d'un modèle de graphe de membre de gauche (LHS) et d'un modèle de graphe de membre de droite (RHS). Le modèle par graphe peut être dans le langage source ou le langage cible dans une syntaxe concrète ou bien dans la syntaxe abstraite MOF. La syntaxe concrète est davantage familière aux développeurs. Dans les langages comme UML, qui sont relativement complexes, les modèles dans une syntaxe concrète sont plus concis que ceux dans la correspondante syntaxe abstraite. De plus, lorsque qu'il n'y a pas de syntaxe concrète spécialisée disponible, un rendu en syntaxe abstraite aura l'avantage de fonctionner pour tout méta-modèle. Le modèle de gauche (LHS) correspond au modèle en cours de transformation et est remplacé par le modèle de droite (RHS) en place. Le membre de gauche, en plus de contenir le modèle de gauche, contient des conditions comme, par exemple, des conditions négatives. Afin de calculer les valeurs d'attributs cibles, comme les noms d'éléments, une logique supplémentaire est nécessaire, par exemple dans les domaines numérique et chaînes de caractères. Certains outils proposent une forme étendue d'un modèle avec des multiplicités sur les arrêts et noeuds. Généralement dans les approches, la planification est externe et les mécanismes d'ordonnancement incluent les conditions explicites, les sélections non déterministes et les itérations, y compris celles des points fixes. Ces itérations de points fixes sont très utiles pour les calculs des fermetures transitives.

Approches M2T :

L'approche de génération de code par parcours de modèles est une des plus basiques qui parcourt le modèle source, sa représentation interne, et fournit le code du modèle cible.

L'approche de génération de code par template est celle préconisée par l'OMG, de ce fait, elle est la plus utilisée des approches M2T. Le membre droit d'un modèle (RHS) utilisera un texte fixe admettant des parties variables. C'est le modèle source qui fournit les informations pour ces variables. L'approche de génération de code par template ne génère pas uniquement du code puisque de la documentation sur les modèles peut être générée également. Il existe un standard OMG pour générer du code à partir de templates : MOFM2T, MOF Model To Text Transformation Language. Ce n'est pas un langage concret c'est un standard, une spécification. Différents outils IDM/MDA ont ce standard implémenté. On peut en trouver sous la forme de plugins sur la plateforme EMF.

Les approches hybrides, des approches M2M particulières :

Leur puissance d'expression font que ces approches fassent partie des plus encourageantes. En fonction du contexte elles permettent de définir des règles de façon déclarative, impérative ou alors de manière à combiner les deux. C'est l'environnement de transformation qui gère la stratégie implicite d'application des règles, alors que c'est l'utilisateur qui gère la stratégie explicite. Les approches hybrides permettent de composer avec les avantages des différentes approches puisqu'elles peuvent être sur plusieurs approches de transformation à la fois [Gar13].

QVT :

Les modèles à transformer n'ont pas forcément été créés avec des langages identiques ni ne sont pas toujours à un même niveau d'abstraction. Pour permettre les transformations, l'OMG a spécifié le standard QVT, Query View Transformation, qui couvre l'ensemble des transformations, des requêtes et des vues des transformations M2M, de modèle vers modèle. Les transformations à partir ou vers des modèles texte doivent s'orienter vers MOF2Text, puisque à chaque modèle correspond un metamodèle MOF. Le standard QVT normalise trois langages :

- *QVT-Operational*, pour les transformations unidirectionnelles, c'est un langage à caractère impératif.
- *QVT-Relations*, pour les transformations uni- et multidirectionnelles, c'est un langage à

caractère déclaratif.

- *QVT-Core*, employé comme la linguistique de QVT-Relations ou bien en tant que cible de transformation venant de cette dernière. C'est un langage simple qui est proche de la transformation de graphes.

En dernier lieu, signalons que pour déterminer des fonctionnalités déclarées dans des langages tels que XQuery, XSLT et autres, il faut faire intervenir le mécanisme *QVT-Black-Box*. Celui-ci sert à appeler des programmes externes durant l'exécution de la transformation.

Rappelons que QVT n'est pas vraiment un langage mais un des standards de l'ingénierie dirigée par les modèles. Tout langage comporte des caractéristiques, des contraintes et des spécifications ; il en est de même pour le standard QVT. Pour faciliter la génération de modèles à partir d'autres modèles, QVT propose une architecture Model Driven Architecture (MDA) et des langages de transformations de modèles dédiés.

2.1.3 L'ingénierie axée sur le comportement

Le Behavior-Driven Development (BDD) est une méthode de développement utilisée en mode agile [Pou12 ; Sma15]. C'est une méthode de développement qui permet d'avoir une communication facilitée entre les parties prenantes d'un projet et qui décrit le comportement du système cible [Che10]. Elle se base sur le développement piloté par les tests en rapport avec le comportement, sur la conception pilotée par le domaine. Mais avant de pouvoir parler plus en détail du BDD dans la sous-section 2.2.2, nous devons d'abord nous intéresser aux concepts sur lesquels il est fondé : DDD, TDD, ATDD.

2.1.3.1 Domain-Driven Design

Le DDD est une méthode, une approche en conception logiciel guidé par le métier. Elle propose de formaliser et d'implémenter du code dans un langage métier, elle intègre la démarche agile et elle prend en considération que le métier peut évoluer très rapidement et facilement. Les concepts que met en avant cette approche sont les suivants : un langage ubiquitaire, un contexte, un domaine et un modèle.

Un langage ubiquitaire : C'est un langage qui peut être utilisé et compris par tous les membres de l'équipe, il est structuré dans un domaine spécifique. Quel que soit notre rôle à l'intérieur d'une équipe nous parlons tous le même langage que l'on soit développeur ou product owner.

On cherche à faire abstraction des aspects spécifiques de chaque métier pour utiliser tous le même langage.

Un contexte : La zone dans laquelle un mot dans une phrase a une signification. Ce même mot peut exister dans d'autres domaines, contextes, mais avoir une toute autre signification. Par exemple : BDD pour un militaire c'est une Base De Défense, pour un informaticien ce sera une Base De Données, pour le développeur ça va être le Behavior Driven Development. La méthodologie du DDD a pour principe la création de modèles qui peuvent être très différents, qui peuvent ou pas avoir des relations entre eux mais concernent chacun un domaine du métier, des points de vue différents. En DDD, le contexte sera le regroupement de tous ces différents modèles.

Un domaine : Concerne le métier et les informations qui le caractérisent pour l'application à développer. Il regroupe les éléments clefs qui permettront d'établir les modèles.

Un modèle : Pour créer la solution qui corresponde aux besoins du client, il est nécessaire de construire un modèle de domaine qui reprend les éléments clefs du domaine et qui seront compris par toutes les parties prenantes. Il doit répondre aux deux questions principales, à savoir : À qui est destinée la solution ? Pour satisfaire quels besoins ? Un modèle de domaine peut concerner plusieurs domaines à la fois. Il peut aussi prendre différentes formes comme par exemple : diagrammes UML, cartes mentales, graphes...

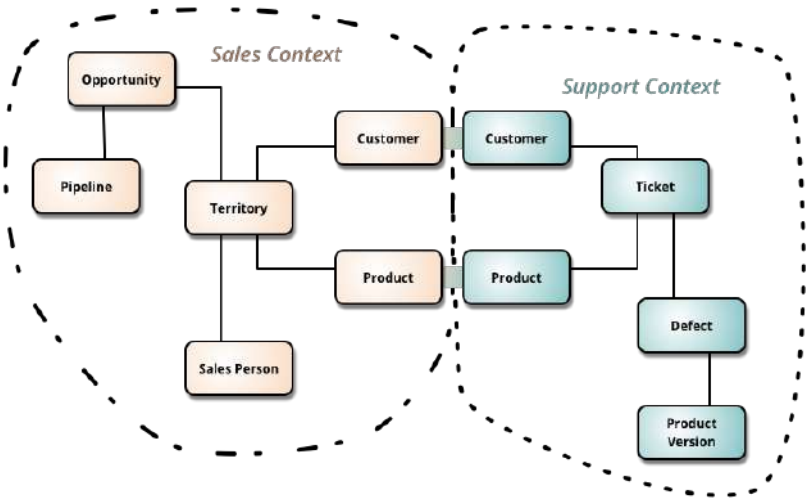

FIGURE 2.10 – Illustration d'un Bounded Context.
(Source : martinfowler.com [Fow14])

Martin Fowler montre dans la Figure 2.10 ci-dessus, que la stratégie de conception du DDD

permet de gérer de grands modèles en les divisant en plusieurs contextes bornés et il explique, en les montrant, qu'il existe des interrelations entre les différents éléments.

En résumé, nous avons vu que le Domain-Driven Design est une méthodologie permettant d'avoir une vue globale de l'application à développer. Elle permet d'avoir un langage commun compréhensible par tous. La stragégie de DDD, afin de décomplexifier le projet, est de le décomposer en différents contextes métiers.

2.1.3.2 Test Driven Development

Le TDD est une technique pour concevoir et mettre oeuvre un logiciel. Lorsque l'on développe avec Test Driven Development, le développeur, travaille par de courtes itérations parce que l'architecture du logiciel sera déterminée au fur et à mesure de l'avance. C'est dans le cadre du processus de génie logiciel Extreme Programming que TDD a été décrit pour la première fois [BA05]. Pour développer en TDD, les itérations comprennent trois étapes :

La première étape de TDD est celle de *création d'un test unitaire*. Le fait de commencer par la création d'un test unitaire permet de vérifier si la fonctionnalité, qui est déjà implémentée dans le système, est conforme aux différentes exigences. Ce travail nécessite qu'au préalable, nous devons bien nous questionner sur la signification de l'implémentation de la fonctionnalité. En d'autres termes, un test unitaire se comporte comme une spécification de conception. Ainsi, si l'on souhaite lancer un test dans cet état, il sera voué à l'échec car il n'a pas encore été implémenté.

La seconde étape de TDD est celle de l'*implémentation*. Une fois l'étape de création du test unitaire faite, la seconde étape est celle de l'implémentation de la fonctionnalité et ceci de manière simple et élémentaire. Cette implémentation reste dans un cadre de conception très restreint. Quand les tests unitaires sont passés avec succès, le travail d'implémentation est terminé.

Dans la troisième et dernière étape, la *refactorisation*, le développeur analyse ce que les changements ont pu provoquer dans le système, pour cela il prend du recul pour avoir une vision d'ensemble sur les fonctionnalités implémentées. En effet, il est possible que la modification qui a été introduite puisse apporter des duplications de fonctionnalités ou bien des incohérences. Afin d'éviter cette situation, une amélioration de la conception peut être apporté par le développeur, pour pouvoir mettre à jour son système sans impacter les fonctionnalités. Cette pratique est ap-

pelée "refactoring". Les tests unitaires élaborés lors de la première étape, implémentés lors de la deuxième étape, seront activés par cette refactorisation ce qui permettra d'avoir l'assurance de la bonne continuité de fonctionnement du système.

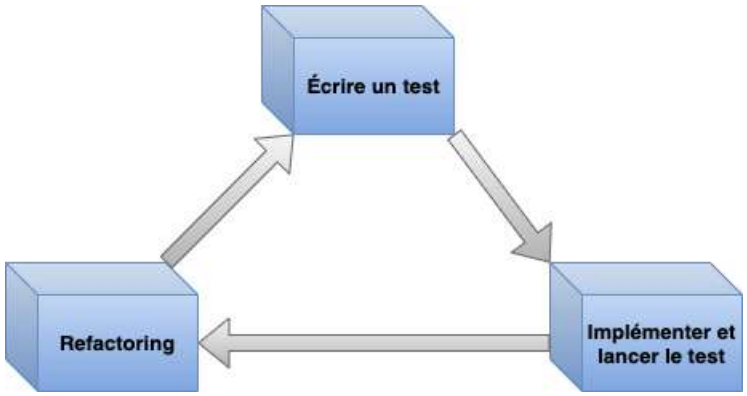

FIGURE 2.11 – Schéma du processus du Test Driven Development.

Pour que toutes les fonctionnalités soient implémentées, ces différentes étapes sont répétées, sur les différents éléments, afin que toutes les modifications nécessaires soient aussi implémentées. La fonctionnalité créée est très petite, puisqu'elle est faite à chaque itération, et qui ne dure généralement que quelques minutes [Kos08 ; JM07]. Ainsi la première étape permet de créer les tests qui vont piloter la mise en œuvre et la conception, c'est ce qui caractérise le TDD.
Le Test Driven Development n'amène pas obligatoirement à une conception montante ou descendante [Bec03] ; c'est au développeur de choisir le cas de test avec lequel il souhaite débuter. Par exemple, si l'on souhaite avoir une stratégie de test descendante, nous pouvons commencer par un cas de test simple du système. D'autres classes de bas niveau sont ajoutées par le développeur, afin de prendre en compte le comportement au fur et à mesure qu'il conceptualise le comportement détaillé nécessaire. Une stratégie ascendante peut être utilisée par le développeur, en démarrant par les tests pour les classes de bas niveau. Ainsi de plus en plus de classes sont ajoutées, ce qui permet de voir les abstractions qui ont été décelées par refactorisation. Cet ensemble forme ainsi la conception d'un niveau supérieur.
La mise en application et les tests sont présents à chaque itération. Néanmoins, tous les besoins d'un projet logiciel, en termes de conception et de test, ne sont pas tous couverts, comme nous allons le voir ci-dessous.

Pour l'élaboration d'une architecture adéquate, le Test Driven Development ne suffit pas toujours. Lorsque les développeurs pensent qu'il existe un point bloquant dans l'architecture, ils

encouragent les réunions d'équipes, pendant les cycles TDD, pour sortir de ce point bloquant et ainsi améliorer la conception [BA05]. De petites refactorisations permettent aussi d'introduire des modifications de conception pour l'amélioration du système.

La grande majorité des systèmes requiert des campagnes de tests supplémentaires. Même si les tests unitaires sont élaborés par les équipes de développement, d'autres tests sont nécessaires. Les tests supplémentaires dont il est question, sont les tests d'acceptation et d'intégration qui sont réalisés en dehors de TDD.

En général, d'après les évaluations du TDD, la qualité du système, par rapport au nombre de défauts, s'améliore bien qu'il soit à noter également que l'effort y est en augmentation la plupart du temps [GW04 ; JS05 ; JM07]. D'après une étude, la mise en œuvre de TDD permettrait d'avoir une conception améliorée d'un système, en utilisant des petites unités, plus simples [JS08].

2.1.3.3 Acceptance Test-Driven Development

Le développement piloté par les tests d'acceptation (ATDD) [JS08 ; Kos07] se concentre sur la spécification compréhensible pour le client à travers des exemples, nous nommons d'ailleurs cette pratique la spécification par l'exemple [Adz11]. ATDD est un processus de vérification et de spécification des exigences pour le logiciel. Le principe est que l'acceptation automatisée des tests permette à toutes les parties prenantes de se focaliser sur les objectifs du projet logiciel par les tests d'acceptation qui améliorent la communication avec des exigences qui sont lisibles pour l'ensemble des membres du projet. ATDD concerne avant tout trois acteurs : le développeur, le client et le testeur [Coh04 ; Kos07] et il est composé d'une suite de tests d'acceptation exécutables. Ce processus se découpe en trois étapes :

La *rédaction* d'un test d'acceptance est la première étape du processus. Une exigence correspond avec précision au comportement attendu du système. L'exigence est élaborée par le client et le testeur conjointement. Les autres types de comportement sont spécifiés à l'aide d'exemples simples avec des données en entrées et en sorties clairement identifiées [Adz11]. Elles seront ajoutées à la spécification originale en tant que critères d'acceptation. Les exigences étant rédigées conjointement par le client et le testeur, la documentation qui accompagnera les tests sera compréhensible par le client.

L'*automatisation* des tests d'acceptance est la seconde étape du processus. En principe, les tests ne peuvent pas être utilisés comme oracle de test, même s'ils contiennent des exemples concrets. Au fur et à mesure que les fonctionnalités attendues sont implémentées, le développeur peut alors automatiser les exemples en lien avec une et une seule exigence. Le développeur travaillera donc de pair avec le testeur pour mapper les concepts illustrés à travers des exemples et ainsi faire leur implémentation dans le système.

L'implémentation des tests d'acceptance est la troisième étape du processus. Dans cette dernière étape, la fonctionnalité qui permettra au système de correspondre à l'exigence, lorsqu'il réussit le test d'acceptation, est conçue puis mis en œuvre par le développeur.

Dans ATTD, pour toutes les exigences d'une seule itération, peuvent être écrits également en une itération, puis dans un second temps ils peuvent être automatisés et mis en œuvre un par un [Sma15].

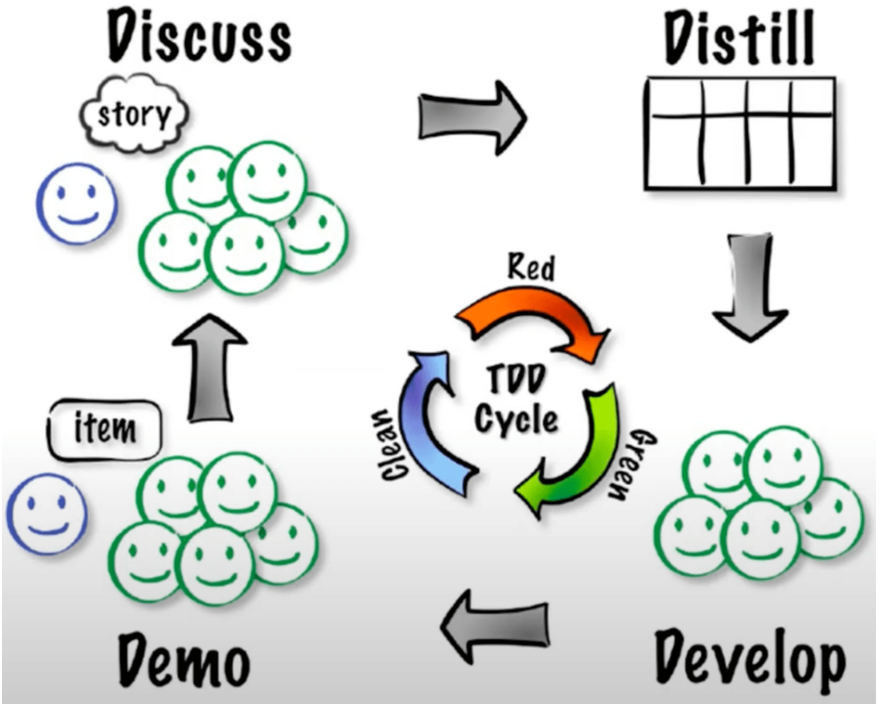

FIGURE 2.12 – Schéma du processus d'Acceptance Test Driven Development.
(Source : Présentation de l'ATDD par Chinthaka Chandrasekara [Cha16])

2.1.4 Domain Specific Language

Le langage spécifique à un domaine (DSL), Domain Specific Language, est un langage de programmation informatique d'une expressivité restreinte orientée vers un domaine particulier [FP11]. Ce domaine métier répond à un besoin grandissant exprimé dès les années 1980, où le concept DSL est apparu [Nei84]. John Bentley [Ben86] l'avait associé à la notion de "petit langage", partant du principe que plus les langages sont orientés vers un domaine en particulier et plus ils sont restreints du point de vu syntaxique. Ces langages sont d'autant plus faciles à concevoir, implémenter, documenter, mais aussi à apprendre et à utiliser. Les DSL sont contraires aux langages de programmation généralistes (General-Purpose Language, GPL) qui, comme leur nom l'indique, servent à la conception de logiciels sans domaine métier en particulier mais, au contraire, dans une large variété de domaines d'application, citons par exemple Python, Java ou C. Certains DSL sont bien connus, comme HTML, utilisé pour les documents sur internet, Gherkin pour définir des tests fonctionnels, ou encore SQL qui permet de communiquer, exploiter ou modifier des bases de données.

Pour Martin Fowler [FP11] la définition de DSL, Domain Specific Language, en tant que "Langage spécifique au domaine" n'est pas suffisamment explicite. Il estime la formule et la notion intéressantes mais trouve que le cadre n'est pas très clair. Parfois certains concepts sont visiblement des DSL, mais d'autres fois ils sont discutables. L'expression existe depuis quelque temps déjà mais n'a jamais bénéficié d'une définition bien précise, comme nombre de logiciels. La définition actuelle est : "langage de programmation informatique d'une expressivité limitée axée sur un domaine particulier". Selon lui, quatre notions clés composent cette définition :

Langage de programmation informatique : de même que les GPL, les DSL ont pour vocation de faciliter le travail de codage des humains, leurs structures sont facilement compréhensibles.

La nature du langage : les DSL ont un langage fluide et compréhensible aussi par les experts du domaine métier.

Expressivité limitée : les GSL possèdent de très nombreuses fonctionnalités là où les DSL proposent justement un minimum de fonctionnalités, très ciblées, qui offrent la possibilité à un expert métier de réagir face à un problème sans pour autant avoir des notions de programmation.

Focalisé sur un domaine : la valeur d'un langage DSL est inversement proportionnel à la gran-

deur du domaine, c'est-à-dire, plus le domaine d'application est réduit, plus le DSL aura de la valeur.

L'ambition de ces langages est avant tout de minimiser la barrière qui existe entre les experts du domaine en question et les développeurs. Bien entendu, pour certaines actions de codage, il est souhaitable de les confier aux développeurs. Il s'agit de réaliser des applications qui puissent répondre aux problèmes énoncés, aux attentes et besoins, grâce à une meilleure communication et à la confiance instaurée entre les différentes parties.
Il est nécessaire de spécifier d'abord les différentes propriétés des DSL, avant de montrer les bénéfices qui en sont attendus ainsi que les différences qui existent entre les DSL et les GPL ; puis les différents DSL en fonction de leur domaine d'application et, pour finir, les différents types de DSL.

2.1.4.1 Propriétés et bénéfices attendus d'un DSL

Pour la création d'un bon Langage Spécifique au Domaine, il faut considérer les différentes dimensions de conception des DSL, leurs propriétés [Völ10].

L'expressivité : L'avantage des DSL par rapport aux GPL est celle d'avoir une expressivité beaucoup plus importante. Il en résulte des programmes plus petits avec une sémantique plus accessible pour les outils de traitement.

La couverture du domaine : Généralement ce sont plutôt des sous-domaines qui sont définis pour éviter les concepts informatiques ou algorithmiques qui pourraient dépasser les connaissances des experts du domaine.

Les sémantiques : Il est à distinguer la sémantique *statique* de celle d'exécution, la sémantique *opérationnelle*. La première s'adapte aux règles de typage et aux contraintes. La seconde vérifie le comportement attendu d'un programme lors de son exécution qui a pu se faire grâce à son interprétation et à sa transformation en code exécutable.

La séparation des préoccupations : Un domaine n'est souvent pas composé que d'une seule préoccupation, chacune couvrant un aspect différent du domaine. Le système devra couvrir tous ces aspects du domaine, soit avec un seul DSL, soit avec des DSL encore plus spécifiés couvrant des préoccupations plus détaillées ; chaque fragment du programme pouvant y répondre séparément. Cette manière facilite aussi la réutilisabilité des fragments de codage.

La complétude : La complétude correspond au niveau avec lequel un langage peut énoncer des actions qui contiennent toutes les informations utiles pour pouvoir être exécutées. Un DSL qui n'est pas complet impose d'ajouter d'autres informations, tels que des fichiers de configuration ou du codage réalisés dans un langage de niveau inférieur, pour qu'il puisse être exécutable.

La modularité des langages : Cette modularité a pour but de faciliter la création d'autres langages. Le réemploi de certaines fonctionnalités dans d'autres contextes évite d'avoir à les développer de nouveau permettant même l'association de plusieurs autres DSL. Il y a quatre notions possibles à retenir lors de la composition d'un langage, il faut qu'un langage puisse être réutilisé, référencé, embarqué, voir même être étendu pour qu'il soit efficace.

La syntaxe concrète : Pour qu'une syntaxe concrète soit utile, il est important que le langage puisse être adopté par l'ensemble des experts du domaine sans qu'ils aient besoin d'avoir des connaissances en programmation. La notation d'un DSL doit être suffisamment simple et concise, tout en exprimant les besoins qui ont un rapport direct avec le domaine, ce qui facilite aussi la lecture du code généré. La syntaxe concrète peut se décliner sous différentes formes : textuelle, symbolique, graphique, matricielle ou encore tabulaire. Le choix de la forme, ou des formes, sera fait en fonction des experts du domaine en tenant compte de leurs besoins et de leur meilleure compréhension.

Après avoir vu les caractéristiques des DSL, voyons les bénéfices qui en sont attendus. Le bénéfice majeur des DSL est d'augmenter l'efficacité d'une partie ou de la totalité du code qui exécute une tâche déterminée. Cette augmentation de l'efficacité est le produit des autres bénéfices et avantages introduits par les DSL.

Les connaissances apportées : Elles sont apportées aux développeurs lors de la conception d'un langage spécifique d'un domaine, grâce à l'enquête qui est réalisée auparavant.

La garantie : Les DSL offrent la garantie aux futurs utilisateurs de l'application que celle-ci sera dédiée aux besoins spécifiques du domaine.

La communication : Elle est accrue entre les experts du domaine et les développeurs. Les caractéristiques des DSL rendent ce langage bien plus accessible que ne le serait son équivalent en langage générique, en GPL.

2.1.4.2 Limites entre les GPL et les DSL

La frontière entre ces deux langages n'est pas très claire, elle dépend du contexte d'utilisation et des notions dans la linguistique existante. Un langage à usage general (General-Purpose Language, GPL), comme son nom l'indique [Far16], est un langage qui répond à une large diversité de problèmes à l'opposé d'un langage spécifique au domaine (Domain Specific Language, DSL). Quelques exemples de ces langages bien connus sont Java, C, C++, COBOL. Quand un GPL peut répondre à tous types de problèmes qui puissent être résolus par un machine de Turing (ex : un ordinateur), il est qualifié *turing complet*. Il en résulterait qu'en théorie ces langages soient interchangeables. Néanmoins, l'intégration de diverses fonctionnalités peut être différente d'un GPL par rapport à un autre, comme par exemple celle de la gestion de mémoire en Java n'est pas compatible avec celle du langage C parce que celle de Java s'appui sur un autre algorithme. L'amélioration des langages informatiques est à créditer aussi à l'apparition de nouvelles programmations, tels que : orientée objet, orientée aspects, fonctionnelle, impérative, en logique, par contrats, ou bien, réflexive (des formalismes de méta-programmation).

Les langages spécifiques du domaine (DSL) ne sont guère *turing-complets*, ou très rarement, puisqu'ils sont prévus pour ne répondre qu'à des problèmes dans un domaine précis, à une catégorie spécifique. Malgré cela, l'amélioration par ajouts de transformations successives fait que ces langages concis deviennent des langages plus larges, des fois même *turing-complets*, jusqu'à devenir des GPL. Ce qui rend le développement plus simple d'un DSL est que habituellement il n'accepte pas des concepts algorithmiques comme des structures conditionnelles ou des boucles, ni n'autorise l'intégration d'abstractions nouvelles, des fonctions ou des sous-programmes.

D'après Martin Fowler [FP11] les principales propriétés d'un DSL, la nature du langage et l'expressivité, représentent le mieux les frontières entre les deux langages, même si ces propriétés concernent un domaine spécifique. Un GPL peut être en fait un DSL quand il correspond à des problèmes autres que ceux pour lesquels il a été conçu. Prenons comme exemple XSTL, langage consacré à la transformation de documents XML en des formats différents comme des pages HTML ou des documents PDF. Même s'il a toutes les particularités propres à un GPL, il est vu comme un DSL parce qu'il est surtout employé pour la transformation de documents XML. Il peut aussi être vu comme un GPL quand il est utilisé pour résoudre un autre genre de problèmes.

2.1.4.3 DSL en fonction du domaine d'application

Les DSL peuvent être classés dans plusieurs formes qui chacune couvre des besoins particuliers. Ils peuvent être utilitaires, techniques, architecturaux, ou bien métiers.

Les DSL utilitaires : Ils ont pour objectif de simplifier la tâche des développeurs pour des fonctionnalités bien précises. Nous pouvons citer Make comme exemple. Ce logiciel à la particularité de construire de manière automatique des bibliothèques avec des éléments de base, comme du code source, ou bien des fichiers généralement exécutables. De plus, ce logiciel n'exécute les commandes que si elles s'avèrent nécessaires.

Les DSL techniques : Ce sont ceux qui s'adressent principalement aux programmeurs. Comme les GPL, ils gèrent la logique d'une application dans sa totalité mais ces DSL proposent une syntaxique qui accélère la réalisation du code. Habituellement ils sont ajoutés à un GPL déjà existant. Prenons par exemple le GPL Ruby qui permet de rédiger de manière efficace des applications à objets, alors que son DSL associé Ruby on Rails permet de rendre très clair l'emploi du modèle d'architecture Model-Vue-Controler (MVC).

Les DSL architecturaux : Ils procurent les notions nécessaires lors de l'établissement d'une architecture logicielle spécifique. Les langages comme Unified Modeling Language (UML) permettent tous types d'architectures [BOO17] contrairement au Domain Specific Modeling Language (DSML) [LKT04]. Un programme rédigé à partir d'un DSL architectural ne produit qu'une partie du squelette du code. Pour pouvoir structurer l'application, il faudra que le programmeur implémente du code dans ce squelette, tout en respectant une architecture spécifique, tels l'architecture orientée objet, celle en couche, etc [GS93]. Ces DSL sont très intéressants, par exemple, pour la mise en place de modèles de conception ou pour développer des systèmes quand leurs architectures sont "multi-tiers", architectures Modèle-Vue-Contrôleur (MVC). Un exemple de DSL architectural serait AUTomotive Open System Architecture (AUTOSAR), créé en 2003, pour l'industrie automobile qui propose une architecture logicielle standardisée. Son architecture en couche le rend accessible aux différents constructeurs automobiles ainsi qu'à leurs équipementiers [Fre08].

Les DSL métier : Ce sont ceux qui s'adressent à des métiers de manière très spécifique en construisant des applications ciblées. Ce genre de DSL expose les aspects du domaine et la logique métier avec pour finalité que les experts du domaine puissent les utiliser sans que pour

autant ils aient les connaissances des informaticiens. Ceci implique des efforts de la part des programmeurs pour que le résultat corresponde aux pratiques et aux besoins des experts métier, avec un langage compréhensible et facile pour ces derniers. Dans le domaine de la finance, un exemple de DSL métier serait Quant DSL, qui est prévu pour les analyses quantitatives. Citons aussi Peyton Jones et al. pour leurs recherches sur un DSL de normalisation d'autres instruments financiers et de contrats dans le domaine des assurances [PES00].

2.1.4.4 Différents types de DSL

Il est possible de classer les DSLs en deux types : les internes ou DSLs enchâssés et les externes ou DSLs autonomes [FP11].

Le DSL interne : Il se base sur la syntaxe du langage GPL hôte, si celui-ci accepte une expansion en ce sens. Le DSL disposera de la puissance du GPL. Le but est de transformer un GPL en DSL, en le simplifiant et en essayant de masquer les parties qui ne sont pas intéressantes pour le domaine de l'expert métier. Du fait de créer un DSL dans un GPL, il est commode de pouvoir utiliser les outils de traitement du langage hôte pour traiter le DSL. Les langages les plus appréciés pour mettre en place un DSL interne sont : Ruby [Cun08], Lisp ou Scala [KA17]. Les bénéfices des langages internes sont la rapidité et la facilité de programmation de ces langages, néanmoins ils s'adressent principalement aux développeurs parce qu'ils font usage des fonctionnalités des langages génériques.

Le DSL externe : Il ne cherche pas à se baser sur un langage hôte, au contraire, c'est un langage autonome et unique, comme par exemple SQL (Structured Query Language). Un DSL externe a la liberté de faire usage de la forme qu'il souhaite en s'exprimant de manière simple et en modifiant les concepts d'un domaine particulier. La seule contrainte est d'avoir à créer un outil pour la transformation du DSL en code praticable par la machine, ou bien en code d'un langage générique, qui devra par la suite être compilé ou seulement interprété. C'est dans un métalangage syntaxique que sera définie une syntaxe spécifique sur lequel sera basée la création d'un DSL externe. Le plus populaire de ces langages est Backus-Naur Form (BNF), dont l'extension est Extendet Backus-Naur Form (EBNF). Les syntaxes générées pourront être traitées par différents générateurs d'analyseurs syntaxiques, tels Lex et Yacc ou un autre exemple serait ANTLR (ANother Tool for Language Recognition)) ou encore Bison.

D'après Eelco Visser et Markus Voelter [Völ10] de l'Université Technologique de Delft : *"Les*

DSL sont des outils puissants pour l'ingénierie logicielle, car ils peuvent être conçus sur mesure pour une classe spécifique de problèmes, mais en raison du grand degré de liberté dans la conception des DSL et parce qu'ils sont censés couvrir le domaine en cours de manière cohérente, et au bon niveau d'abstraction, la conception DSL est également difficile".

Avec les différents artefacts expliqués dans la section précédente, nous pouvons voir que deux approches, méthodes, se dégagent pour la mise en oeuvre d'une méthodologie, à savoir : l'approche par le biais de l'ingénierie du comportement avec la méthode du Behavior Driven Development (BDD) ainsi que par le biais de l'ingénierie dirigée par les modèles avec notamment la méthode de Model-Based Testing (MBT). La méthode axé sur le comportement Behavior Driven Development (BDD) utilise un DSL contrairement à MBT qui utilise un modèle.

2.2 Méthode de tests axée sur le comportement

Dan North est à l'origine de Behavior Driven Development [Sma15]. Ayant observé que le développement piloté par les tests (TDD) était assimilé à une technique de test afin d'éviter toute confusion il a remplacé le "T" de *"Test"* par "B" de *"Behavior"*, comportement. Dan North a ainsi relevé que BDD concerne la conception et non pas les tests.

2.2.1 Behavior Driven Development

BDD est une méthode complète en étant une extension de TDD tout en englobant aussi les principes de DDD et ATDD. Dan North définit BDD par ces trois principes :

- *"Enough is enough"* : Assez c'est assez, pas d'efforts gaspillés, au début ne faire que les analyses et la conception seulement nécessaires.
- *"Deliver stakeholder value"* : Délivrer de la valeur aux parties prenantes, ne faire que créer ou augmenter la capacité de créer de la valeur.
- *"It is all behavior"* : Tout est comportement, le comportement du système sera décrit en utilisant un même langage à chacun des niveaux de granularité.

Les *"user stories"* expriment les exigences du logiciel, on y ajoute les critères d'acceptations sous la forme de scénarios de tests. Ceux-ci seront par la suite automatisés pour devenir les tests d'acceptation automatisés nécessaires pour ATDD. C'est dans des cycles TDD que vont se dérouler la conception et la mise en oeuvre des fonctionnalités des *"stories"*.

Bien que DDD ait plusieurs aspects, ce n'est que celui du langage ubiquitaire, omniprésent, qui a été adopté par BDD.

Nous avons vu dans la sous-section 2.1.3 ces différentes notions et l'on constate qu'elles sont toutes reliées entre-elles.

2.2.2 Procédé d'après le Behavior Driven Development

Le succès de BDD réside sur le langage ubiquitaire compréhensible par toutes les parties prenantes et la figure ci-dessous montre à quel point chacune des parties se réfère sans cesse aux exemples compréhensibles par tous.

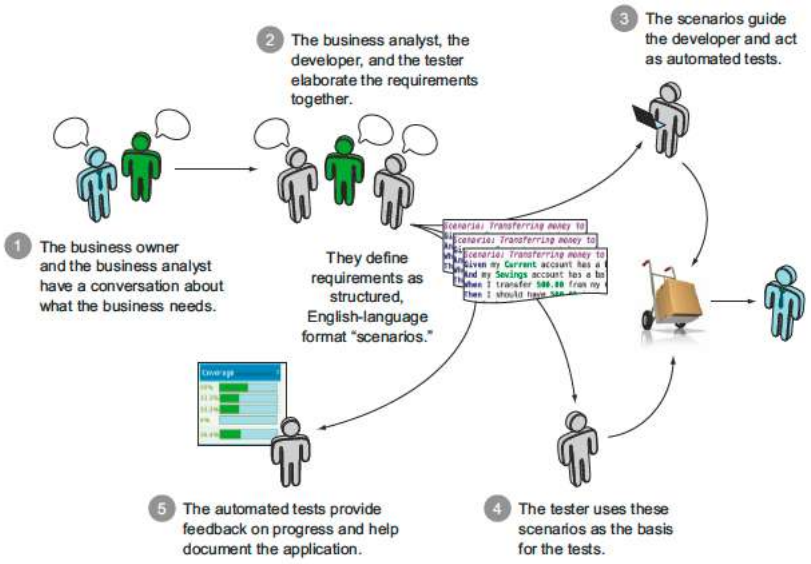

FIGURE 2.13 – Schéma du processus BDD.
(Source : BDD in action [Sma15])

Cette image nous montre que le client et l'analyste métier s'accordent sur les besoins de l'entreprise. Ces entretiens permettent à l'analyste métier, au développeur et au testeur d'élaborer des exemples qui seront définis dans un langage compréhensible au format "scénarios". Les scénarios guideront le développeur qui réalisera les tests automatiques. Le testeur utilise les mêmes scénarios comme base pour les tests. Les tests automatiques fournissent les réponses et aident à la rédaction de la documentation de l'application.

2.2.2.1 Contexte du procédé

L'étape initiale du projet commence par des conversations entre le *product owner* et l'analyste de l'entreprise pour déterminer les objectifs en énumérant les exigences pour définir une vision globale des résultats attendus.
Une fois les exigences énumérées, l'analyste et le développeur définiront les fonctions de haut-niveau. Cela permettra de créer des *stories* qui guideront vers des scénarios dans le langage utilisant *Given-When-Then*, qui aboutiront à des exemples compréhensibles par tous.

2.2.2.2 Conception et implémentation

Après avoir pris connaissance de ce qui doit être implémenté, la phase de conception permettra de savoir de quelle manière les fonctionnalités pourront être implémentées. Dans cette étape, les développeurs vont créer l'architecture du système tout en définissant les tests unitaires et le *refactoring*. Durant la phase de mise en oeuvre, l'implémentation des scénarios permet d'ajouter au logiciel les *stories* qui contiennent le comportement spécifié. Les tests d'acceptance seront habituellement pilotés à travers l'interface utilisateur, quand cela est possible. Dans le cas contraire il est conseillé de réaliser l'automatisation de manière sélective, toujours à l'aide de l'interface utilisateur, et les scénarios restant seront pilotés en partant d'un niveau inférieur.

Ci-dessous la Figure 2.14 montre les principales activités et résultats du BDD.

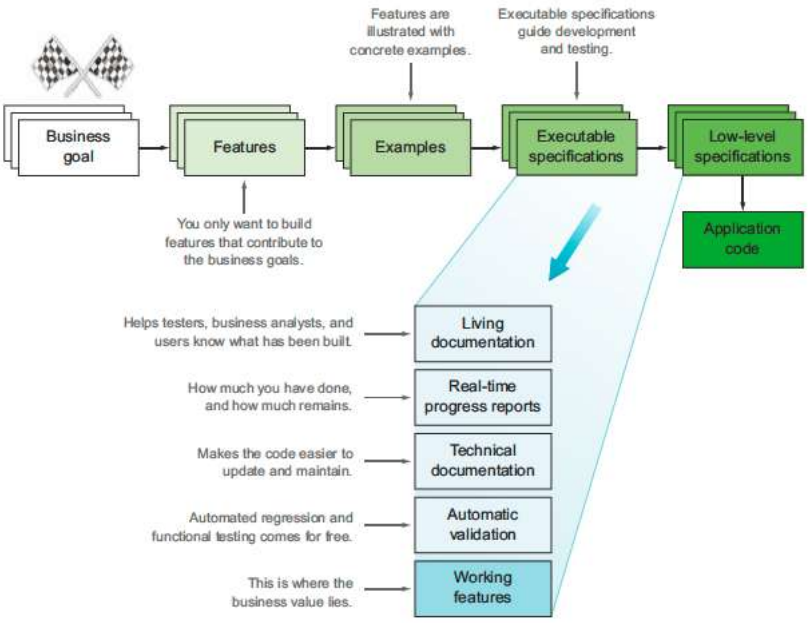

FIGURE 2.14 – Schéma récapitulatif des activités du processus BDD.
(Source : BDD in action [Sma15])

2.3 Méthode de tests dirigée par les modèles

Le concept du Model-based Testing (MBT) remonte aux années 70. MBT est né du fait que l'on cherche à surélever les tests basés sur des spécifications. L'idée était que, quand une notion devient trop complexe, afin de mieux comprendre, il serait intéressant de la découper en des structures plus petites. Des représentations graphiques peuvent être utilisées pour créer des structures plus petites et pour l'élaboration d'un modèle abstrait pour lequel des tests concrets pourront être utilisés sur le système cible. A la suite de cette génération de tests, un comparatif montrera les résultats obtenus et les résultats attendus. Cette technique de tests consiste à se focaliser sur l'élaboration d'un modèle de tests qui servira pour l'exécution des tests suivant une chronologie déterminée.

2.3.1 Procédé sous influence du Model-Driven Engineering

Nous avons vu dans la sous-section 2.1.2, que l'Ingénierie Des Modèles (IDM, en anglais MDE) met le modèle au premier plan pour le processus d'abstraction à haut niveau des applications. Cette ingénierie à pour but de créer des applications sous forme de code en partant des modèles. Elle a pour principe de constituer différents modèles pour mieux appréhender le système que l'on souhaite construire. Le modèle est une expression complexe, dont le but est d'organiser les activités et informations d'un système comme le sont, par exemple, le flux d'information, les traitements ou les données. Mais comme chaque solution à développer comporte son lot de complexités, l'Object Management Group (OMG), a aussi standardisé d'autres approches dirigées par les modèles ainsi que l'approche orientée vers les tests logiciels basés sur les modèles, à savoir, le Model-Based Testing (MBT).

2.3.2 Procédé d'après le Model-Based Testing

Les systèmes deviennent de plus en plus complexes tant par leurs envergures mais aussi par les technologies employées ce qui augmente leur distributivité ainsi que leur interconnectivité. Les flux de quantités de données générées sont conséquents, aussi venant de plate-formes externes. Par conséquent, l'importance du test n'a cessé de grandir au fil des années. Les tests peuvent être effectués manuellement ou bien être partiellement automatisés par script.
Il existe toutefois une méthode pour automatiser les scénarios de tests grâces à des outils. Cette méthode est le Model-Based Testing dans lequel seulement le modèle est créé manuellement,

mais ceci de manière très spécifique.

La planification des tests a subi l'impact des exigences du monde de l'industrie tant par l'augmentation des coûts que les délais de plus en plus restreints pour effectuer ces mêmes tests. Du fait que les cadences de l'activité de test ne peuvent pas être augmentées, d'autres solutions ont été envisagées par les entreprises de tests. C'est pour cette raison que les entreprises de tests ont commencé à s'intéresser à l'automatisation des tests basés sur les modèles. De cette manière ils pourraient adopter une nouvelle démarche et ainsi améliorer leur efficacité [Chr+15].

Le Model-Based Testing (MBT) est une des techniques d'ingénierie des modèles appliquée au test logiciel dans laquelle le comportement attendu, décrit dans le modèle, peut être comparé au comportement réel du logiciel. MBT fragmente les structures au point de les rendre abstraites, plus ils sont petits, plus ils sont précis et plus ils peuvent être réitérés. Pour ce faire, les représentations graphiques restent la meilleure approche pour aborder cette technique basée sur les modèles abstraits. Le modèle est dit abstrait car il ne donne qu'une vue partielle, mais très détaillée, du comportement attendu d'un système ou d'un logiciel. Un langage textuel approprié, comme par exemple UML/OCL, fournira les textes pour compléter les représentations graphiques. Après la création et la spécification du modèle abstrait, MBT va générer automatiquement des scénarios de cas de tests concrets puis les exécute et les valide pour, au final, produire un rapport. MBT fait partie de la catégorie des tests "Boîte noire".

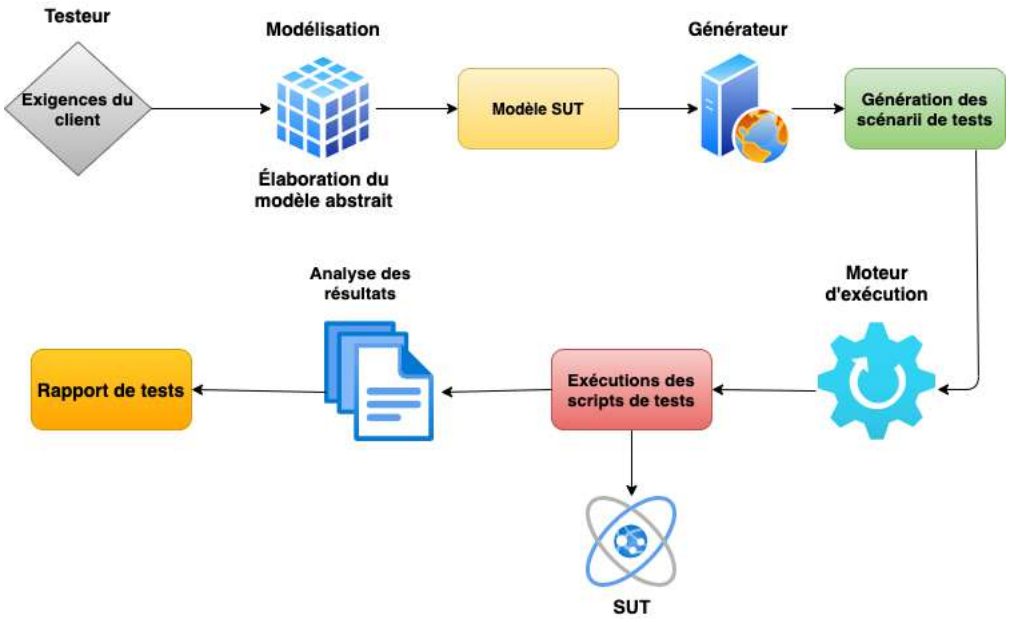

FIGURE 2.15 – Schéma de la méthodologie MBT.

La possibilité de tester la conception de logiciels, à partir de modèles d'automates finis, était déjà étudiée par Tsun S. Chow, dans les années 70 [Cho78]. Son travail, qui est encore très considéré aujourd'hui, représente le fondement de ce domaine de recherche qui est toujours très actif et productif. Ce fondement comporte la génération automatique des tests, avec l'utilisation de la représentation par des diagrammes d'état, des diagrammes UML et des automates finis.

Depuis plus d'une dizaine d'années, différentes approches d'ingénierie des systèmes ont vu le jour. Les modèles y sont utilisés pour les étapes de spécification du système. Ce n'est plus seulement le code la donnée à utiliser pour concevoir des tests, mais les modèles avec leurs spécifications. Ces modèles sont conçus de manière à ce qu'ils calculent automatiquement des séquences abstraites qui seront par la suite concrétisées pour pouvoir être exécutées. Mais aussi, les modèles sont suffisamment précis et formels pour ne permettre aucune ambiguïté afin que la génération automatique soit reproductible. Les modèles sont au même niveau de détail que les séquences abstraites.

Le modèle devient un oracle de test puisqu'il prédit le résultat attendu du système et le verdict, réussite/échec, est donné en le comparant avec le résultat obtenu sur le système. Il montre aussi une vue dynamique du système en précisant son état initial, il capte les points d'observation de contrôle et il formalise le comportement du système sous test. Pour pouvoir interpréter sans ambigüité et ainsi avoir un scénario de test qui soit reproductible automatiquement, il est primordial que le modèle soit formel et très précis. La complexité des systèmes informatisés augmente sans cesse, ce qui explique que les milieux industriels et scientifiques soient autant intéressés à la technique de génération de tests automatiques. De plus, cette technologie en constante évolution, garantit une fiabilité et une qualité de services maximales.

L'utilisation du critère de couverture de la génération de test par les modèles permet de respecter scrupuleusement le cahier de charges et les exigences exprimés. Ceci instaure un haut niveau de confiance, ce qui explique aussi que cette technique soit de plus en plus sollicitée. La réalisation de certains projets de développement peut prendre, des fois, très longtemps. Il est donc très intéressant de pouvoir suivre et valider le système étape par étape. Cela est possible parce que, dès le départ du cycle de réalisation du système, on intègre ces modèles. Ce domaine de technologie est toujours très productif, malgré le nombre de résultats sans cesse en augmentation, parce qu'il faut constamment correspondre aux conditions du marché, que ce soit à l'évolution de la normalisation ou au souhait de réduire les coûts, ou encore à l'amélioration de la qualité

de service souhaitée. L'innovation et la recherche dans ce domaine sont toujours très actives pour garantir la sécurité des systèmes, la preuve en est la quantité d'articles dans les magazines spécialisés et le grand nombre de conférences sur ce sujet.

Le MBT se fonde sur des techniques et procédés qui permettent, à partir de modèles abstraits, de créer automatiquement des cas de test concrets pour les rendre exécutables. Cet enchaînement se fait en quatre étapes. Il en résulte que les bases de MBT sont la modélisation de cas de test pour leur génération automatique, en tenant compte de la réutilisabilité de ces modèles, de leurs critères de sélection, des techniques et des stratégies à mettre en oeuvre. L'ambition de MBT est de diminuer, autant que possible, la barrière qui existe entre les ingénieurs de test et les analystes. Les modèles doivent correspondre véritablement aux besoins de l'entreprise. Le but étant d'approcher la couverture fonctionnelle totale des besoins. La suite des quatre étapes du procédé du MBT est comparable à un pipeline, composé de quatre éléments : un modeleur, un générateur, un moteur d'exécution et une analyse des résultats avec la constitution d'un rapport de test.

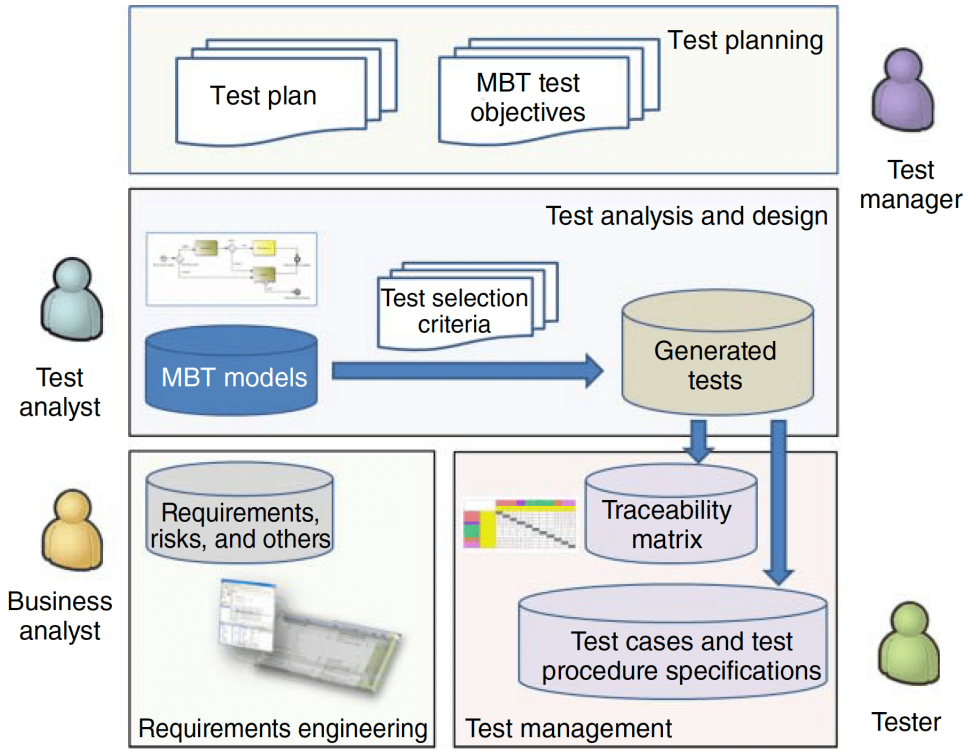

FIGURE 2.16 – Schéma du processus MBT dans un contexte entreprise.
(Source : Model-based testing essentials-guide to the ISTQB certified model-based tester : foundation level [KL16])

Pour compléter cet enchaînement, il est possible d'utiliser les sept dimensions de la taxonomie proposée par Marc Utting [UL07]. Celui-ci propose de classifier les trois premières étapes du MBT en les faisant correspondre à sept dimensions [UPL12]. Pour le modèle, il tient compte du sujet, de la redondance, des caractéristiques et du paradigme ; pour la génération des tests, il se base sur les critères de sélection de tests et sur la technologie utilisée ; et pour l'exécution des tests il différencie les tests *online* des tests *offline*.

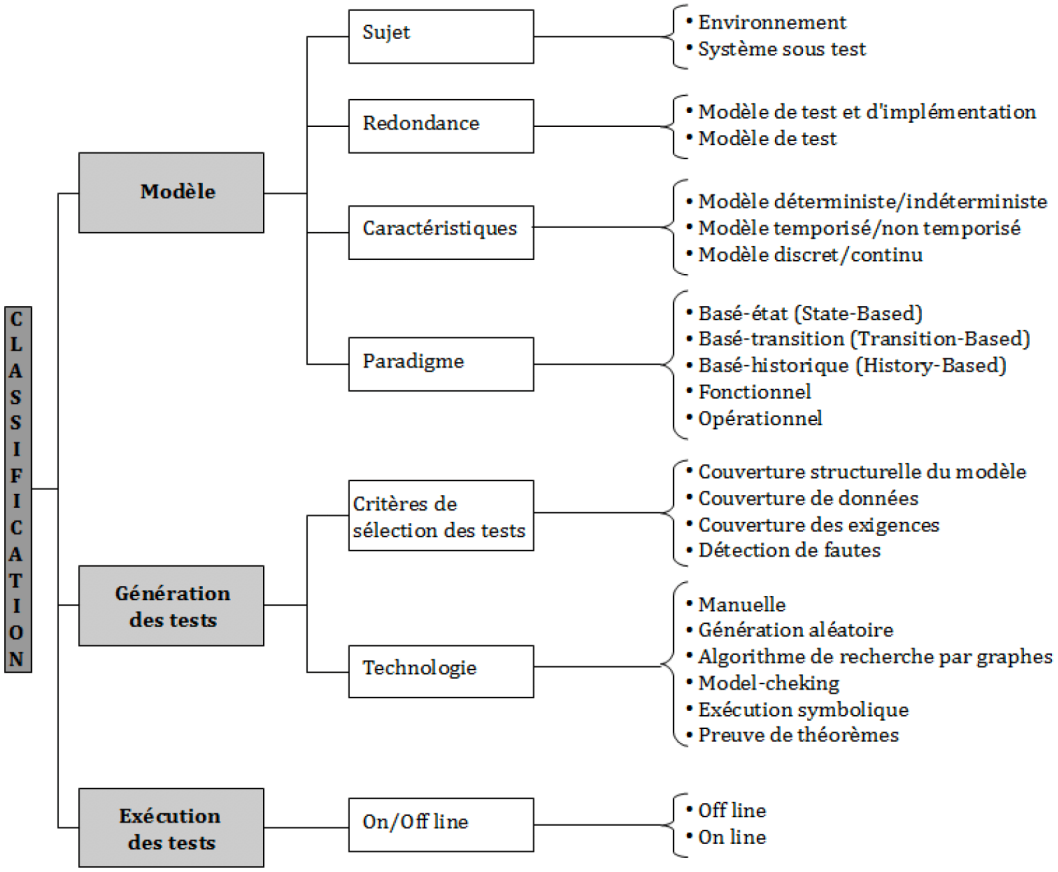

FIGURE 2.17 – Schéma de la classification du Model-Based Testing.
(Source : A taxonomy of model-based testing approaches [UPL12])

2.3.2.1 Modeleur

Le modeleur reçoit les exigences, les fonctions et le code, pour générer un modèle abstrait. Ce modèle de test exprime le comportement prévu du système. Il existe des langages de modélisation, comme UML, qui formalisent les aspects observation et contrôle ainsi tous les éléments donnés pour la configuration initiale du modèle. La traçabilité bidirectionnelle entre le modèle et les exigences, est assurée par les constituants du modèle, comme les décisions ou les transitions. Le modèle de test doit donc être très complet et précis. La modélisation consiste à créer ce modèle qui pourra exécuter des cas de tests sur le système sous test. Les points de contrôle et d'observations qu'il contient permettront une comparaison entre les valeurs attendues et les valeurs réelles. De plus, sa structure sera utilisée pour la génération des cas de tests garantissant une certaine couverture de tests. Ce modèle est issu d'une combinaison de facteurs issue de différents domaines dont il faut tenir compte, à savoir : le sujet du modèle, sa redondance, ses caractéristiques, et son paradigme. Il existe différents différents modèles applicables pour le MBT tels que les UML/OCL, les BPMN, la méthode B et les machines à états finis.

2.3.2.2 Générateur

Le générateur reçoit les modèles de test abstraits et génère des scénarios de tests concrets et exécutables. Cette génération est entièrement automatisée. Chacun des scripts de test est un découpage d'actions de haut niveau contenant les données d'entrée ainsi que les valeurs prévus en sortie. Les scénarios de tests générés sont complets et facilement compréhensibles et ils pourraient même être effectués manuellement directement sur le SUT.
En même temps que le MBT génère automatiquement des scripts exécutables, il importe des informations d'interfaces graphiques (Graphical User Interface, GUI). Dans le cadre du MBT, chacune des phases du modèle doit être seulement dirigée vers l'élément de l'interface graphique attribué. En fonction du type de modèle, plusieurs critères de sélection de tests et différentes technologies sont à la disposition de l'utilisateur. Pour générer des cas de tests avec le générateur de MBT, l'ingénieur en charge de réaliser les tests doit introduire des modèles de tests très détaillés et/ou des cas de tests pour chacun des mot-clé. Les scripts de tests produits par le générateur du MBT seront directement exécutables [AM12]. Le générateur de cas de tests va réaliser des ensembles de cas de tests avec des tests qui partagent les mêmes propriétés. Ces ensembles sont appelés "suites de tests". Le critère de sélection de ces ensembles permettra de mesurer la qualité de ces suites de tests grâce à leurs niveaux de couverture. Cette couverture

utilise les stratégies d'un modèle structurel comme : la couverture du cycle de processus (process cycle coverage), le test par paires (pair-wise testing), le test de cause à effet (cause-effect testing), ou encore l'analyse de la valeur limite (boundary value analysis). Il faut bien déterminer le nombre de tests à générer en fonction de la criticité de chacune des exigences, ainsi que faire une implémentation qui se base sur les exigences et les risques. Quand une application est de type non-critique, il est possible de seulement générer un test par comportement nominal et par cas d'erreurs principaux du modèle. Par contre, quand l'application est de type critique il faut réaliser un critère plus couvrant pour être sûr que les procédés métier qui y sont associés soient testés en profondeur.

2.3.2.3 Moteur d'exécution

Cette troisième dimension concerne le moment d'exécution des cas de tests générés, soit en ligne (On line), soit hors ligne (Off line). Avec MBT, l'exécution des tests peut être réalisée automatiquement, par exemple, avec Rational Functional Tester d'IBM [Leg15].

En ligne, On line : quand le système sous test (SUT) n'est pas déterministe, il est des fois nécessaire d'effectuer des cas de tests en ligne pour que les algorithmes puissent réagir directement avec les sorties du SUT, de manière à ce que le générateur de tests puisse visualiser les chemins empruntés par le SUT et ainsi pouvoir suivre ces mêmes chemins dans le modèle de cas de tests. C'est une approche de MBT par lequel les tests sont générés et exécutés en même temps.

Hors ligne, Off line : ce n'est que quand le SUT n'est pas en ligne que les cas de tests générés peuvent être exécutés. Les modèles non-déterministes avec les représentations d'arbres et graphiques sont plus difficilement exécutables que des séquences de tests. Il reste cependant des avantages à exécuter des tests hors-ligne puisque dans ce cas ils sont plus adaptés à l'exécution de tests. On peut générer un ensemble de cas de tests et l'exécuter plusieurs fois sur des machines différentes, à des moments différents et dans des environnements différents. De même, hors ligne on peut plus facilement d'utiliser des outils existants. Pour réduire la taille des ensembles de tests, il est possible de réaliser séparément des minimiseurs de suites de tests.

Si le Model-based Testing nécessite une grande connaissance de la part de l'ingénieur de tests et suppose un travail considérable pour la réalisation des modèles, il est indéniable qu'il réduit le temps d'exécution des tests. Off line c'est l'approche du MBT par lequel les cas de tests sont générés dans un référentiel pour une exécution future [All ; Chr+15].

2.3.2.4 Analyse des résultats

Le rapport de tests effectué grâce au Model-Based Testing fournira de précieux renseignements à l'ingénieur de tests, pas seulement de manière booléenne, "test réussi" ou "test échoué", mais aussi, dans le cas où une ou des erreurs seraient détectées, le rapport de tests signalera à quel niveau l'erreur se situe. Si l'erreur se situe dans le système, cela indiquerait que le système sous tests a été mal développé. Si l'erreur est détecté au niveau du modèle cela pourrait signifier que, par rapport aux comportements du logiciel, le modèle a été mal réalisé. L'analyse et la qualité du rapport sont un gage de fiabilité autant du système que du modèle réalisé.

2.3.3 Différents types de modèles pour le Model-Based Testing

Après la spécification des quatre étapes du Model-Based Testing, nous aborderons plus en détail les différents types de modèles applicables pour cette méthode : Machine à états finis, UML/OCL, BPMN et Méthode B.

2.3.3.1 Machine à états finis

Pour mieux comprendre ce que sont les machines à états, il est utile de mettre en évidence le côté pratique de leur définition ainsi que leur explication. Tous les développeurs voudraient augmenter la maintenabilité du code ainsi que sa fiabilité tout en réduisant les délais de mise en oeuvre. La machine à états serait une bonne solution. Si elle est bien connue, elle n'est guère utilisée surtout par les développeurs autodidactes. Et pourtant, s'il y a bien des outils utiles et simples à disposition ce sont bien les machines à états. Elles donnent lieu à la possibilité de diviser les algorithmes longs et complexes en des éléments plus petits et donc des éléments plus faciles à gérer. Quand un algorithme peut être représenté par un organigramme ou un digramme d'états il peut être implémenté grâce à l'architecture de programmation des machines à états.
Quand les machines à états sont petites on les représentes par des diagrammes à bulles, ou des diagrammes d'états/transitions, qui modélisent le système du point de vue dynamique [DV16 ; Sch13].

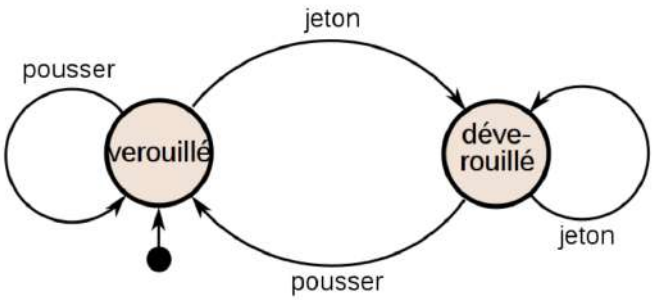

FIGURE 2.18 – Graphe orienté d'un portillon d'accès.
(Source : Cours Qt Automate fini ou Machine à états [Vai17])

Les bulles représentent les états de repos, des "états simples", illustrant les différents états dans le système et les lignes qui les relient sont les lignes de transitions qui modélisent les actions. Ces actions, transactions, peuvent être soit externes soit internes. Externe quand l'action se situe entre deux états et interne quand l'action reste dans un même état [Fou12 ; UL07]. Quand un

changement d'état du système est la réponse à un évènement il s'agit d'une transition. Celle-ci peut être définie par trois éléments : évènement, garde et action. L'évènement, appelé aussi "trigger", l'évènement est annoncé en tant qu'une opération de la machine d'état. La garde, appelée "guard", est sollicitée pour activer la transition au moyen d'une expression booléenne qui doit être satisfaite. Comme il n'est pas possible à une machine à états d'avoir plusieurs états en même temps, elle doit suivre les conditions signalées et les transitions précisées au préalable pour transiter d'un état à l'autre. Quand la garde est satisfaite, une action peut être exécutée. À partir de l'objet utilisé l'action peut, sur les objets visibles, agir de manière directe ou indirecte. Il est important de savoir que le code examine uniquement les conditions de sortie d'état, peu importe comment elle était arrivée à l'étape précédente, le code s'occupera seulement de l'état suivant.
Cette situation est comparable à une course d'orientation, un parcourt imposé et des balises à passer dans un ordre prédéfini (numérotées). Exemple : Si l'on a bien respecté le parcourt on sait que depuis la balise sept tout ce qu'on doit faire c'est d'aller à la balise huit et on ne s'occupe plus des balises trois, cinq ou deux, elles sont déjà faites.
Les machines à états font de même, elles s'occupent de ce qu'elles doivent faire sans se soucier de comment elles sont arrivées à un état précis. Elles sont de bons outils pour tout problème dans lequel il y a des états de repos bien définis. De plus, leur conception accepte facilement des modifications d'un état sans avoir à craindre une quelconque régression, il n'est pas difficile d'ajouter de nouvelles fonctionnalités.

Pour aborder un modèle de Model-Based Testing, on doit garder à l'esprit qu'il doit être le plus petit possible et donc on doit savoir quoi éliminer afin que la vue que l'on conserve soit assez abstraite pour que la machine d'états finis puisse la traiter.
Il existe deux sortes de machines d'états finis : celle déterministe à états finis et celle non déterministe à états finis. La première, déterministe, n'accepte qu'une entrée pour deux options possibles : "Si" alors "faire ceci" sinon "faire cela". La seconde, non déterministe, pour une entrée bien plus d'options à exécuter sont possibles[Gri+02 ; Bro05].
Pour illustrer ces propos, voici un exemple simplifié de deux graphes : l'un montrant une machine à café déterministe proposant seulement du café, l'autre montrant une machine à café non déterministe proposant café ou thé [Mon15].

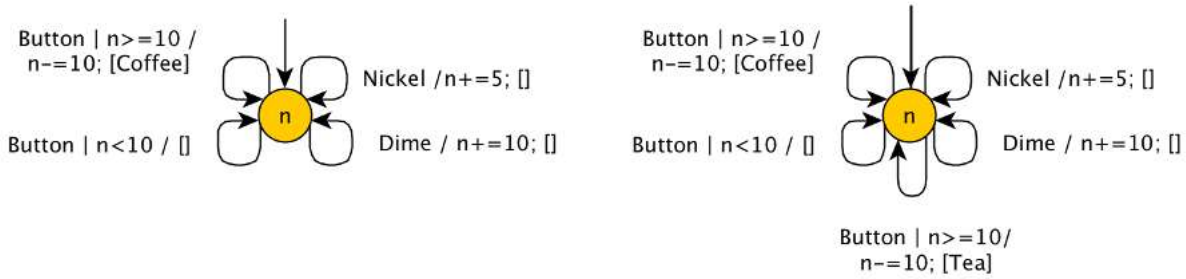

FIGURE 2.19 – Machine à café déterministe (à gauche) et non déterministe (à droite).
(Source : Model-Based Testing of web applications [Mon15])

Dans la pratique, les tests de conformité des machines à états finis sont très utiles et ce malgré la simplicité de la notation "Finite State Machine" (FSM). Les circuits numériques, les systèmes de contrôle intégrés et d'autres types de systèmes, comprenant les protocoles, ont été directement définis par les FSM. Pour définir des systèmes ou des parties de systèmes, il existe différentes notations formelles très semblables aux machines à états finis et parmi elles on peut citer les diagrammes d'états UML[Bro05] .

2.3.3.2 UML/OCL

UML n'est pas une méthode mais un langage de modélisation qui n'a cessé d'évoluer depuis les années '80. Ce langage est composé essentiellement de graphiques, diagrammes et pictogrammes. La version UML 2.5 offre le choix parmi quatorze diagrammes. Les différents diagrammes correspondent aux différents points de vue d'un modèle. On laisse à chacun le choix des diagrammes à utiliser. On peut comparer ces différents points de vue aux différents plans d'une construction de maison. Les corps de métier intervenants ont chacun leur plan pour un même bâtiment, le plombier aura un plan différent du maçon qui sera lui aussi différent de celui de l'électricien [Aud14].
Le langage UML permet surtout d'exprimer des contraintes structurelles.
Pour mieux expliquer UML, on peut ajouter qu'il est composé de trois parties :

- Les vues, qui correspondent aux différents points de vue qui peuvent décrire le système. Ces points de vue sont très variés et adoptés en fonction de leur utilité. On peut citer : les points de vue logiques, architectural, dynamique, temporel, géographique, organisationnel, etc.
- Les diagrammes, qui sont les descriptions graphiques de la composition des vues.

- Les modèles éléments, qui sont les composants graphiques formant les diagrammes.

Les vues, elles peuvent se superposer pour améliorer la compréhension du système.
La vue de cas d'utilisation, "use-case view", décrit les attentes de chaque acteur. Répond aux questions "quoi ?" ou "qui ?".
La vue logique c'est la description vue de l'intérieur expliquant comment satisfaire les attentes des acteurs. Répond à la question "comment ?"
La vue d'implémentation détermine les différentes dépendances entres les différents constituants.
La vue de processus est la vue technique et temporelle qui spécifie les tâches concurrentes, le contrôle, la synchronisation...
La vue de déploiement qui correspond à la question "où ?" pour situer chaque composant du système géographiquement et décrit l'architecture physique des composants.
UML ne définit pas la réponse à la question "pourquoi ?".
Les diagrammes, ils se complètent tout en étant dépendants de façon hiérarchique. Depuis la version 2.3 on distingue quatorze diagrammes : sept statiques ou de structure, trois comportementaux et quatre dynamiques [RR18].
Les diagrammes de structure ou statiques : Diagrammes de classes, d'objets, de composants, de déploiement, des paquets, de structure composite, de profils [Aud14].
Diagrammes de comportement : Diagrammes des cas d'utilisation, état-transitions, d'activité.
Diagrammes d'interaction ou diagrammes dynamiques Diagrammes de séquence, de communication, global d'interaction, de temps [Aud14].

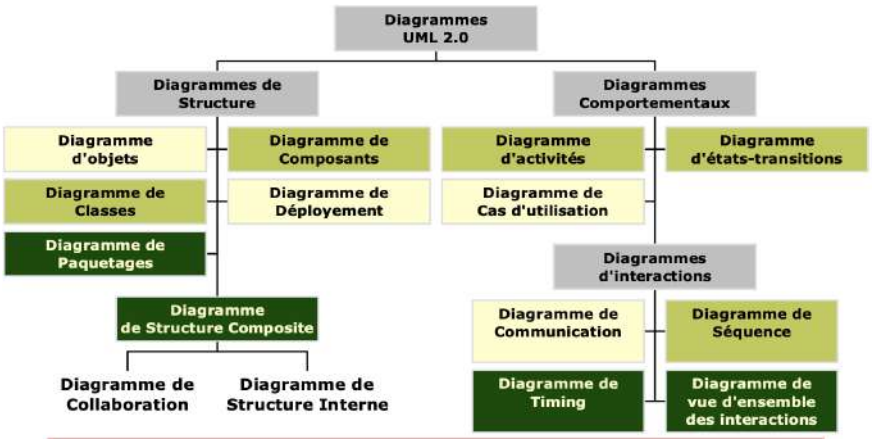

FIGURE 2.20 – Les différents diagrammes UML.
(Source : Cours IUT Conception par objets [Per15])

Modèles d'éléments

Un stéréotype, à définir entre guillemets.

Un classeur, un rectangle regroupant des éléments ayant la même structure, comportement.

Un paquet qui regroupe des unités ou des diagrammes.

Chaque objet ou classe est défini par " : :". Exemple : Une classe X hors de son classeur ou de son paquet sera exprimé par "Paquet A : :Classeur B : :Classe X".

Ce langage présente quand même quelques insuffisances qui sont comblées par le langage OCL (Object Contraint Language). Le langage est utilisé quand les contraintes exprimées s'avèrent être trop complexes pour bien les exploiter en UML et n'y sont pas déjà prédéfinis. OCL est un sous ensemble de UML, davantage orienté vers l'aspect dynamique du système. Pour le Model-Based Testing il existe des langages adaptés, qui découlent de UML et OCL, il s'agit respectivement de UML4MBT et OCL4MBT. [Cab13].

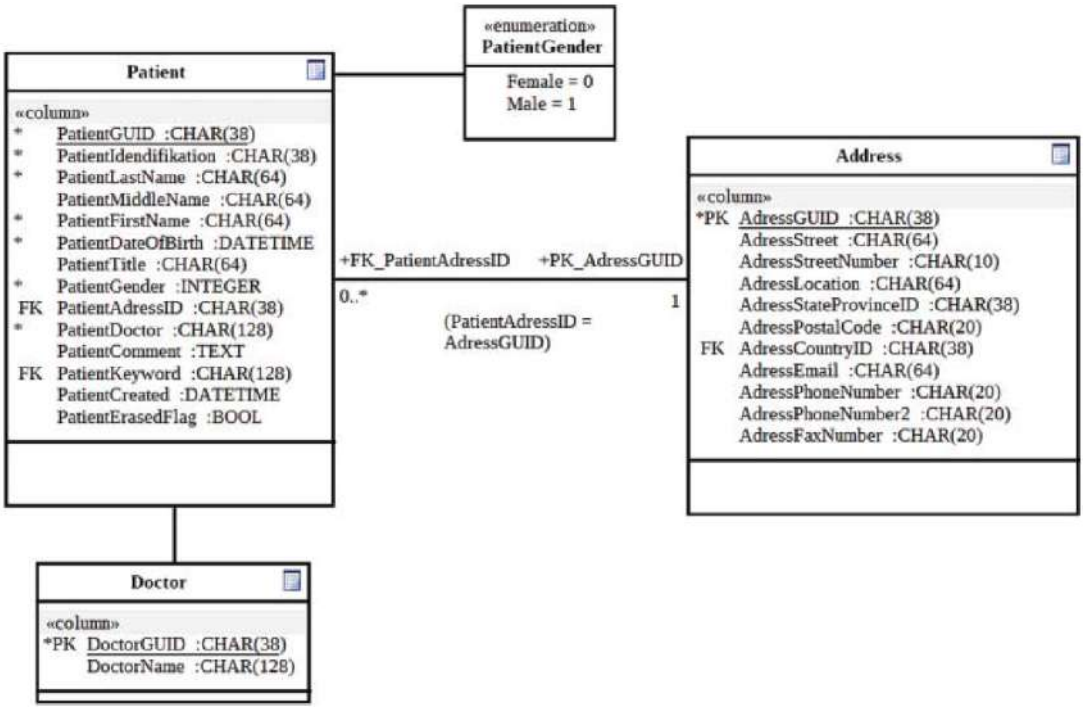

FIGURE 2.21 – Exemple d'un modèle de données (diagramme de classes).
(Source : Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester : Foundation Level [KL16])

La Figure 2.21 ci-dessus montre un extrait d'un modèle de données en UML utilisé durant l'écriture d'une modélisation d'un SUT. Les types de données sont spécifiés. Il s'agit d'un docteur, de ses patients et leurs adresses avec spécification de genre [KL16].

2.3.3.3 BPMN

Le langage UML conserve beaucoup d'avantages mais il persiste quelques inconvénients comme l'écriture et la lecture complexes pour les non-informaticiens, c'est un langage chronophage et de plus c'est un langage opposé au codage habituel des développeurs.
Il existe une solution plus simple, à savoir, le *Business Process Model and Notation* (BPMN) qui est un langage orienté vers les besoins métiers tout en étant concret [Vey18].
Tout système peut s'avérer complexe dans leur mise en oeuvre, dû notamment à un grand nombre d'acteurs ayant chacun leurs opinions qui peuvent paraître contraires à la solution proposée. Il peut arriver par exemple dans un projet informatique que certaines parties prenantes s'attardent plus au déploiement technique (point de vue d'administrateur système) tandis que d'autres se préoccupent des schémas de données de la solution informatique (point de vue administrateur de données). Toutes ces préoccupations nous amènent à considérer les différents points de vue pour l'élaboration d'un système. C'est dans un souci de ne rien omettre que la modélisation BPMN a été créée. Toutes les parties prenantes utilisent ce même langage et c'est un des avantages du BPMN : la communication [KL16].

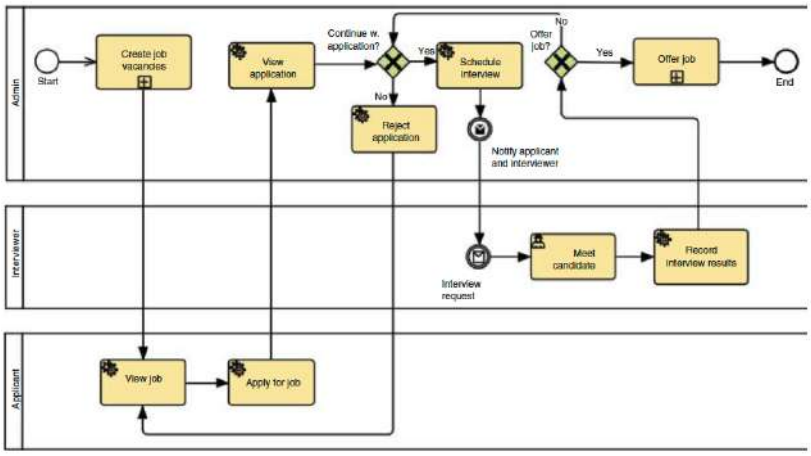

FIGURE 2.22 – Exemple de modèle comportemental (en BPMN).
(Source : Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester : Foundation Level [KL16])

La Figure 2.22 ci-dessus montre un modèle en BPMN. Il s'agit d'un procedé de recrutement. Dans ce procédé interviennent trois "acteurs" : l'administration, l'interviewer et le postulant, ils sont situé à gauche de l'image et chacun à son couloir. Les activités sont représentées dans les rectangles à coins arrondis, les flèches représentes les flux et les "début" et "fin" du processus sont clairement visibles [KL16].

2.3.3.4 Méthode B

En utilisant son propre langage formel, la méthode B permet de formaliser des programmes et des spécifications. C'est un langage pour la théorie des ensembles dans une version simplifiée. La méthode B a pour but de démontrer qu'il n'y a pas de différences entre le code réalisé et la spécification, là où les testeurs avec les méthodes basées sur les tests, peuvent seulement indiquer qu'ils n'en ont pas trouvé. Les modèles créé par la méthode B sont validés par la démonstration, la preuve. C'est d'une méthode surtout de spécification mais aussi potentiellement de programmation. La méthode B est structuré et basée sur un formalisme mathématique : booléens, fonctions, relations, prédicats, etc [Ham12].

La confiance que l'on peut avoir dans la méthode B fait qu'il a été utilisée dans de nombreux programmes industriels, surtout dans les transports ou le logiciel embarqué a été largement développé dans les pilotes automatiques. Après le succès de l'automatisation de la ligne 14 du métro parisien, la RATP à de nouveau utilisé la méthode B pour la ligne 1. En faite, depuis 1995, de nombreuses villes dans le monde entier ont utilisé la méthode B pour les pilotes automatiques des métros : New-York, Madrid, Séoul, Lausanne, Pékin, Delhi, pour n'en citer que quelques uns. La méthode B emploie une linguistique claire et le formalisme mathématique la rend sans ambiguïté possible. Ceci conduit à des descriptions des logiciels précises. La méthode B ne propose pas une abstraction de bas niveau, seulement de haut niveau. Son avantage mathématique peut être un inconvénient puisque ce formalisme peut refroidir certains ingénieurs [Eng05].

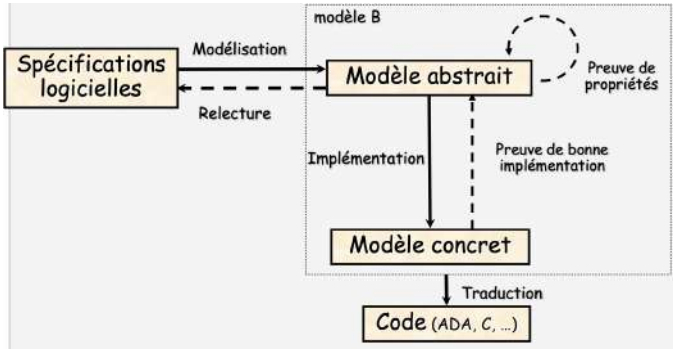

FIGURE 2.23 – Illustration de la méthode B.
(Source : Méthode B - Formation [Eng05])

L'illustration ci-dessus nous montre le concept de la méthode B avec le cheminement d'un modèle B en partant des spécifications logicielles jusqu'à la traduction en langage de program-

mation.

2.4 Évaluation des méthodes

Nous avons détaillée dans la section précédente deux méthodes qui pourraient amener à résoudre la problématique évoquée dans la section 1.6 de ce mémoire. Pour avoir une solution qui corresponde le mieux aux différentes parties prenantes, il est important de voir les côtés positifs, comme les côtés négatifs de ces deux méthodes.

2.4.1 Évaluation du Behavior-Driven Development

Après avoir expliqué le fonctionnement de BDD, dans la sous-section 2.2.2, l est intéressant de lister les avantages et les limites de cette méthode.

Avantages :

- BDD réduit les efforts des développeurs en focalisant seulement sur les fonctionnalités qui valorisent l'entreprise de telle manière que les retours d'information permettent d'effectuer des ajustements plus précoces.
- La réduction de ces "pertes de temps" réduit d'autant les coûts.
- L'utilisation d'un langage ubiquitaire pour les spécifications de bas niveau facilite la documentation.
- La réduction de risque de régression avec les tests unitaires et d'acceptance automatisés.

Limites :

- BDD nécessite une collaboration étroite entre les parties prenantes pour profiter pleinement de tous les avantages.
- BDD est plus efficace dans un contexte itératif avec la méthodologie Agile, ce qui empêche que les différentes parties soient indépendantes.
- Les coûts de maintenance peuvent se révéler importants quand les tests ne sont pas créés avec la bonne expressivité et le bon niveau d'abstraction.

2.4.2 Évaluation du Model-Based Testing

Le procédé du Model-Based Testing a été expliqué dans la sous-section 2.3.2. Cette méthode de tests existe depuis plusieurs dizaines d'années mais elle n'est utilisée que depuis une quinzaine d'années quand des industries du numériques ont cherché à développer les outils pour MBT. Les acteurs du secteur ont estimés que des nombreux avantages incitaient à utiliser cette méthode de test :

- Model-Based Testing permet des tests automatiques intensifs et à grande échelle.
- Une meilleure communication entre les équipes.
- La possibilité d'effectuer des vérification par la Méthode B.
- Meilleure détection des défauts des exigences lors de la création du modèle.
- Meilleure détection des défauts dans le SUT.
- Une meilleure traçabilité.
- Gain de coût et de temps par rapport aux tests manuels.
- Amélioration de la qualité des tests par la génération automatisée des scripts de tests.
- La satisfaction des manageurs.
- La satisfaction des équipes de développement.

Après tous ces avantages, il est nécessaire d'énumérer les limites de MBT :

- L'évolution constante du secteur fait que MBT nécessite toujours d'avantage d'outils et de formations pour améliorer la qualité d'automatisation.
- Ne pas construire un modèle d'après les exigences car avec l'évolution du logiciel, elles sont vite dépassées, obsolètes.
- Enfin comme d'autres méthodes de tests, MBT ne détecte pas non plus tout les défauts.

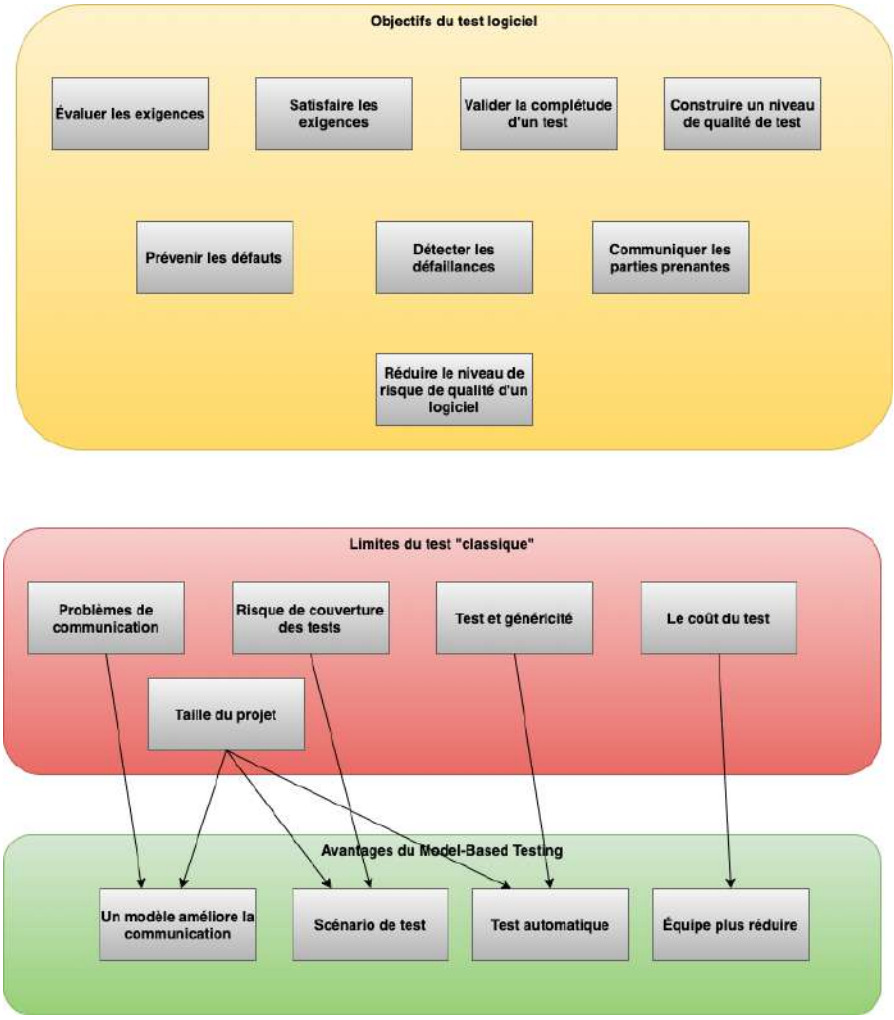

FIGURE 2.24 – Schéma récapitulatif des motivations pour le Model-Based Testing.

Après avoir explicité les avantages et les inconvénients de deux approches de tests, l'approche par Model-Based Testing semble être la mieux adaptée pour répondre à la problématique initiale, dans la section 1.6. Nous aborderons dans la section suivante si réellement la méthode MBT est la mieux adaptée à notre problème métier.

2.5 Génération de tests dirigée par les modèles appliquée à une application web

2.5.1 Types de formats de fichiers pour la transformation de modèle

Les attributs structurels de l'information sont décrits dans les méta-données et pour faciliter leur encodage, les formats d'échange de données ont subi des évolutions. De même, des formats standard d'échange de données ont été développées pour que les exigences puissent prendre en chargent les données d'échange au niveau du système. Les deux formats d'échange de données sont : XML et JSON, chacun ayant des objectifs qui leur sont propre.

2.5.1.1 Format XML

XML est un langage de balisage extensible. C'est bien un sous-ensemble du Standard Generalized Markup Language (SGML) mais à cause de la complexité de ce standard, XML a évolué. Grâce a sa grande compatibilité avec tous les systèmes, XML est réputé comme étant la référence en informatique.

La complexité de SGML est à l'origine d'XML qui, grâce à sa description universelle de l'information, se veut simple dans sa structure et intégré afin de le rendre le plus lisible possible pour les utilisateurs. Selon l'organisme de normalisation W3C le format XML doit :

- Être facilement utilisable.
- Rendre les documents beaucoup plus lisible par l'homme.

XML a comme principales objectifs la sérialisation d'objets afin de pouvoir transférer des données entre les différents systèmes, comme ici avec les modèles, et l'autre principale utilisation est Remote Procedure Calls (RPC). Ce langage est utilisé pour créer des balises définies par l'utilisateur sur les schémas et les documents d'encodage et sera défini grâce à un autre système automatisé ainsi que par un autre *"user"*.

```
<name>
    <first>Gautier</first>
    <last>LEBOURG</last>
</name>
```

FIGURE 2.25 – Illustration d'un encodage XML.

2.5.1.2 Format JSON

De même que XML, le format JSON est aussi facile à utiliser et à analyser, mais de plus il convient mieux aux applications JavaScript parce qu'il est directement intégré dans ce dernier. Des études ont montré que JSON offre des performances significatives par rapport à XML. Ce dernier a besoin de bibliothèques supplémentaires pour, à partir du Document Object Model (DOM), récupérer des modèles de données. C'est une des raisons pour lesquelles XML est moins performant que JSON.

FIGURE 2.26 – Illustration d'un encodage JSON.

2.5.2 Méthodologie d'implémentation du Model-Based Testing

Pour résoudre la problématique évoquée dans la section 1.6 de ce mémoire, la méthode retenue s'appuie sur le procédé du Model-Based Testing. En effet, pour pouvoir créer un script de test, après avoir défini les différentes exigences de la part du client sur la fonctionnalité à tester, le modèle sous-test est élaboré à l'aide d'une machine d'états finis. Ces éléments du modèle de test vont permettre de pouvoir garder le contrôle sur le changement d'état du système cible. Cette machine d'états finis sera générée à l'aide du générateur. Pour pouvoir faire ce procédé, le générateur va alors parcourir le modèle, qui a été converti au préalable au format XML ou JSON. De cette manière, le générateur va pouvoir identifier les éléments du graphe orienté qui correspondent aux différents sommets et ceux qui correspondent aux différentes arcs du graphe. Les sommets vont représenter l'état souhaité du système et les arcs vont représenter toutes les actions que nous devons faire pour atteindre cet état souhaité [Kri19].

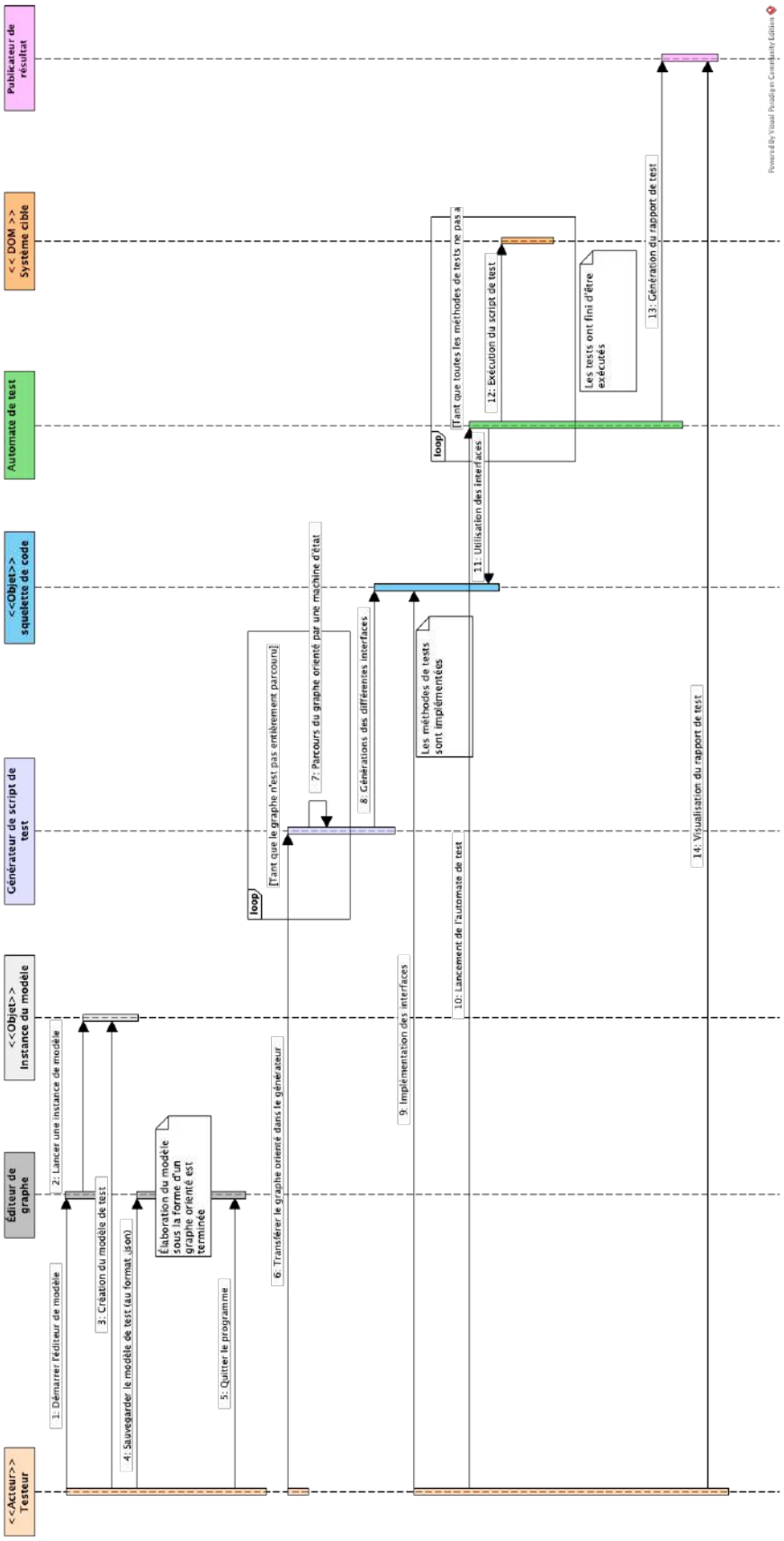

FIGURE 2.27 – Schéma d'application du Model-Based Testing.

2.5.3 Architecture de la solution

2.5.3.1 Architecture fonctionnelle

L'exécuteur de script de test pour l'élaboration de test automatique est constitué d'une multitude de composants : d'un éditeur de modèles, d'un compilateur, d'un conteneur de squelettes de code, d'un exécuteur de scripts et enfin d'un publicateur de résultats.

- **L'éditeur de modèle :** va permettre à la personne chargée de l'élaboration du test de pouvoir créer un modèle sous-test.
- **Le compilateur :** va permettre de faire la "conversion" du modèle graphique vers de la génération de code, Transformation de Modèle vers Texte (M2T).
- **Le conteneur de squelette de code :** va contenir les différentes bibliothèques faites pour faciliter le travail du testeur par l'intermédiaire de méthodes d'actions pré-implémentées.
- **L'exécuteur de script :** va être l'automate qui va être appelé afin de pouvoir exécuter le test.
- **Le publicateur de résultat :** va être l'élément qui sera généré afin de s'assurer du bon déroulement du test.

Le composant du compilateur sera lui-même composé d'un analyseur de modèle afin de pouvoir traduire les sommets ainsi que des arcs du graphique en différentes méthodes de tests.
Le composant exécuteur du script sera également composé d'un autre composant, celui-ci externe, afin de pourvoir appeler le "driver de test" qui exécutera le test.

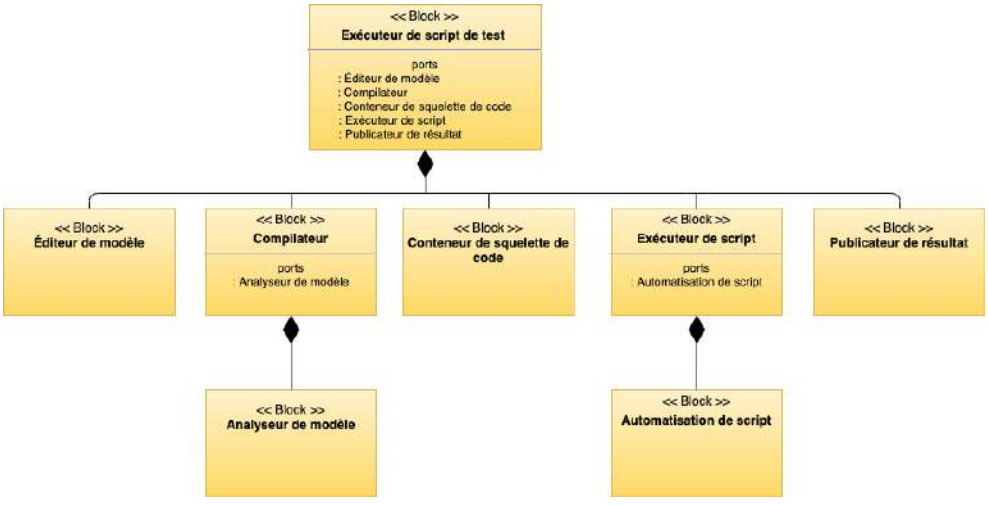

FIGURE 2.28 – Schéma d'architecture du système.

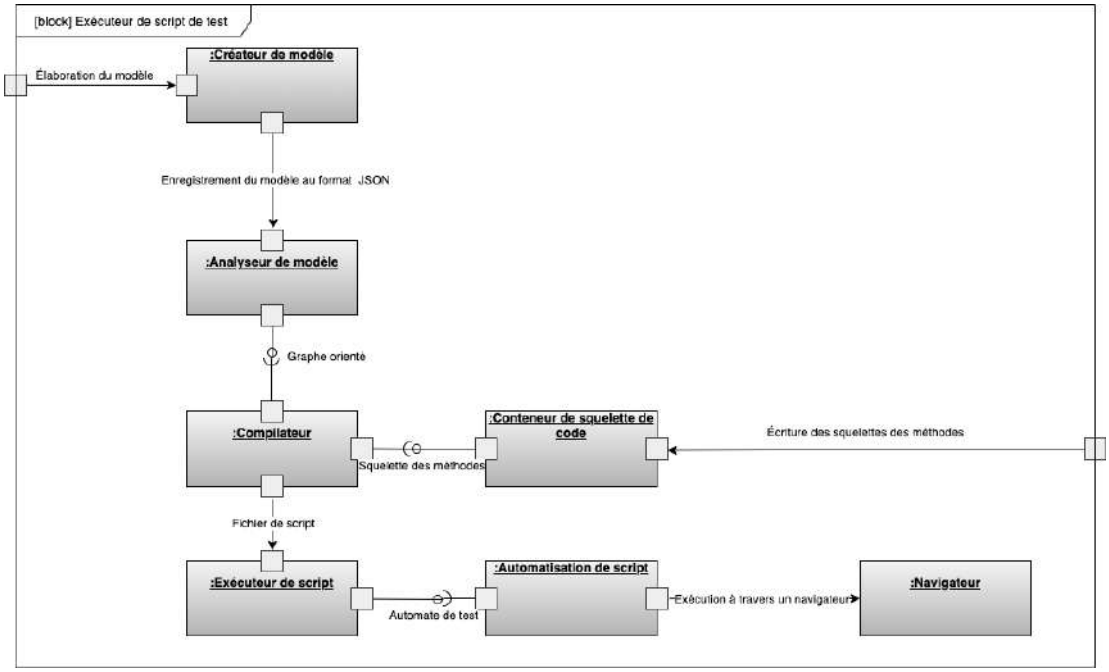

FIGURE 2.29 – Schéma en blocs d'architecture du système.

l'éditeur de modèle

L'éditeur de modèle va permettre au testeur d'élaborer le modèle de test qui sera exécuté. Pour pouvoir faire cette étape, le testeur sera aidé d'un outil de modélisation. C'est cet élément qui sera compilé et introduit dans le générateur qui va permettre de sortir les différentes méthodes de test par rapport à ce modèle.

Le compilateur

Le composant de compilation va permettre d'extraire d'un modèle de test, lu alors en XML ou en JSON par le compilateur, les différentes interfaces de tests qui vont être alors implémentées par le testeur. La compilation ainsi que la génération sera faite par une machine à états finies qui va être lue par l'outil de génération qui va parcourir le graphe de manière aléatoire, afin de s'assurer que chaque sommet soit couvert.

L'exécuteur de script

L'exécuteur de script de test, via un automate de test, va exécuter le script de test via une première classe générique de la classe RemoteWebDriver, qui va alors produite un fichier DLL. Ce fichier va par la suite être transformé en requêtes HTTP POST suivant les différents protocoles du standard W3C qui sont suivis par les différents navigateurs du marché. Enfin l'application

à tester va alors rendre visible les différents points d'entrées et de sorties afin de communiquer directement avec le navigateur automate.

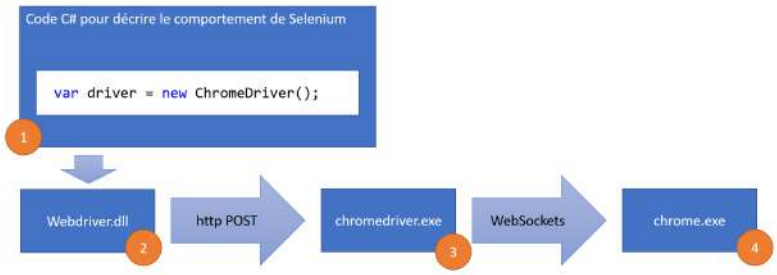

FIGURE 2.30 – Schéma du fonctionnement de Selenium WebDriver.
(Source : www.infinitesquare.com [Viv18])

Le publicateur de résultat

Le composant de publication des résultats, va permettre de lire les différentes méthodes de tests qui ont été exécutées et ainsi va permettre de savoir quelle méthode de test a une réussite et quelle méthode a été un échec.

2.5.3.2 Architecture de déploiement

L'architecture de déploiement permet de relier le scénario de déploiement à la structure physique de l'entreprise, de telle manière qu'il soit possible de vérifier que l'ensemble des exigences soient respectées. Dans notre cas, le système sera incorporé à l'infrastructure du projet *Airline 1*.

Pour pouvoir déployer la solution, un poste devra être affecté à l'ingénieur de tests afin qu'il puisse élaborer les différents modèles de tests. C'est directement sur ce poste que seront mis en place les différents outils qui lui permettront de faire l'édition de modèles et la génération de scripts de tests.

Dès que le script de tests est généré, il est sera exporté vers le répertoire de stockage des scripts de tests, qui est hébergé par le serveur de la société. A partir du répertoire de stockage des scripts de tests, des fichiers exécutables pourront être lancer vers le système cible, le système à tester, dans notre cas *Airline 1*.

Une fois que l'ensemble des scripts auront été effectués dans le système cible, un rapport de tests sera crée sous forme de fichier HTML qui pourra être conservé dans le serveur au niveau du répertoire de stockage des rapports de tests.

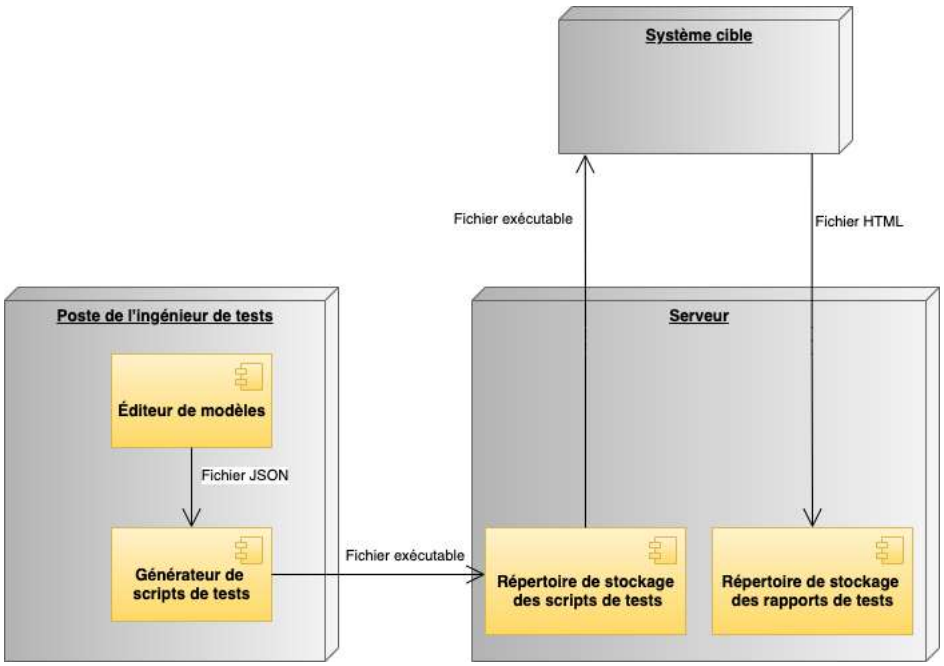

FIGURE 2.31 – Diagramme de déploiement de la solution.

2.5.4 De l'architecture de déploiement vers le modèle de test abstrait

Pour avoir une idée plus précise de ce a quoi ressemble un modèle de tests, de type machine à états finis, nous voyons ci-dessous, un tableau et son graphe associé.
Le tableau ci-dessous représente les différentes étapes d'un scénario de tests d'*Aircraft Tracking validation*, dont une des principales fonctionnalités est l'*Uplink Capability*. Les messages de type *Uplink* sont les seuls messages que peut envoyer le système *Airline 1* vers un appareil. Les autres types de messages étant seulement des messages de réception.

A/C Tracking validation: non regression

1.01 Uplink Capability		
Date	25/06/2019	
Tester	EBA	
Initial conditions:		
Step	Actions	Status
1	The list of available uplink is complete (on all A/C type)	OK
2	The list of available uplink is the same from the message list and from the aicraft list.	OK
3	A320 - Request an AIDS report	OK
4	A350 - Send an ACMS RQP Excel template	OK
5	A350 - Send an ACMS RQP ACU parameters	OK
6	A350 - Send an ACMS RQP RAW parameters	OK
	FINAL STATUS	OK

TABLE II.4 – Scénario de tests des Uplink Capability du système *Airline 1*.

Ci-dessous, le graphe associé à ce tableau de scénario de tests, voir le Tableau II.4. Ce graphe est le modèle abstrait comprenant des sommets qui représentent les états attendus à examiner et des arrêtes représentant les actions nécessaires pour aller d'un sommet à l'autre. Dans ce modèle abstrait de tests, il est montré en vert quels sont les sommets qui sont déjà parcouru par le générateur, et en gris ceux qui ne le sont pas encore.

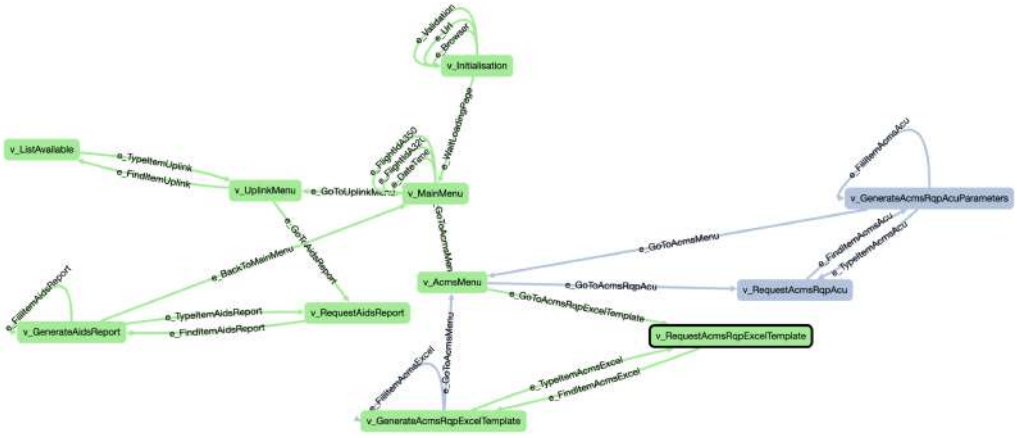

FIGURE 2.32 – Modèle abstrait d'Uplink Capability du système *Airline 1*.

Chapitre III

Mise en oeuvre de la solution

Le facteur primordial de réussite dans les industries du logiciel, dans de nombreux cas, réside dans le domaine de l'ingénierie produit. En effet, la qualité, la fiabilité, les moindres coûts ainsi que le déploiement de systèmes logiciels toujours dans des délais plus courts sont les exigences des différents clients qui nécessitent de modifier la pratique pour apporter de la qualité logiciel. La demande toujours plus grande de solution numérique contraint les entreprises de concevoir de nouveaux produits avec des délais toujours plus courts.

Depuis quelques dizaines d'années, nous observons une réduction significative du délai de mise sur le marché mais également des clients exigeant des systèmes plus flexibles, ce qui pour conséquence une complexité toujours plus grande de l'ensemble des systèmes. Le retard engendré par un délai de mise sur le marché peut avoir un impact négatif considérable sur les bénéfices de l'entreprise. Malheureusement, ce changement de rythme, plus rapide, contraint les entreprises de réduire le temps consacré aux tests des produits. Cette situation exige de nouvelles méthodes et outils puissants. Les tests dans les industries d'aujourd'hui sont traditionnellement effectués manuellement par des humains et sont un processus très long.

De ce constat fait, nous constatons fréquemment des erreurs lors des différentes phases de conception des produits pour le clients final. Ces différentes anomalies, ont pour origine un non-respect d'une ou plusieurs exigences pourtant décelées dans le cahier des charges du client. Elles auraient dues ainsi être détectées et corrigées, avant la phase de livraison, lors de l'étape de contrôle de qualité. Par conséquent, le processus nécessite un effort plus important dans le temps consacré, notamment dans cette activité. Le principe d'automatisation des tâches répé-

titives représentent un effort non négligeable à réaliser en amont afin de développer un outil de génération des scripts de tests automatique. En raison de ce niveau d'abstraction plus élevé, nous alors pouvoir créer différents artefacts réutilisables et ainsi réduire les coûts des différents tests d'acceptences. Il est à noter également que, le fait d'avoir un niveau d'abstraction élevé va permettre de mieux distinguer la séparation entre les aspects métier et les aspects purement techniques afin qu'ils soient correctement employés. Posséder un moyen systémique et automatisé afin de tester les solutions numériques permettrait de réduire les dépenses consacrées au domaine du test de façon significatives. De la même manière, cela permettrait également aux entreprises de consacrer leurs efforts et leurs temps à la phase de conception et de mise en oeuvre du logiciel plutôt qu'au domaine du test.

Le test basé sur un modèle (MBT) est une méthode de test logiciel qui est en plein essor depuis quelques années. L'idée fondamentale du MBT est que les tests découle automatiquement de différents types de modèles permettant de décrire le comportement du système testé (SUT).

Même si des efforts sur la recherche dans ce domaine ont été fait et, elles ont permis de concevoir des techniques de test plus complexes basées sur des modèles. Cependant, du fait du manque d'outils et d'une faible perspective ces techniques demeurent très peu ont été utilisées dans un contexte industriel. Il n'en demeure pas moins que dans un contexte industriel ou non, il peut sembler difficile de savoir quand arrêter les tests. En règle générale, lorsque toutes les exigences sont couvertes par des tests nous clôturons cette activité et nous pouvons alors la considérer comme terminé. En pratique, ce n'est pas toujours une règle immuable. Dans le monde de l'entreprise nous sommes confronté fréquemment à de multitude de grands projets avec des milliers d'exigences. Par conséquent, le client finale ne peut être garantie d'un niveau de couverture des exigences de 100 %.

3.1 Présentation du cas d'étude

La troisième partie de ce mémoire présente la mise en oeuvre des notions de Model-Based Testing dans le but d'appliquer la génération de tests dirigée par les modèles abordée dans la section 2.5. Pour ce faire, il sera utilisé, la transformation de modèle vers du texte (M2T) de l'approche MDA. Les scripts de tests générés, contiennent des méthodes de tests qui déclenchent des actions à effectuer sur le SUT comme par exemple, le traitement des informations qui ont été capturé sur le SUT ainsi que l'édition d'un rapport les résultats des différentes méthodes de tests exécutées. Les modèles génères des méthodes abstraites, des blocs de codes, qui vont contenir différents éléments qui doivent être implémentés. Le modèle qui aura été créé dirigera l'implémentation de ces éléments qui sont des fragments de code.

3.1.1 Contexte du COVID-19

Du fait de la situation sanitaire mondiale dû au virus COVID-19, le projet *Airline 1* a dû être repris par le client. Ce revirement a fait qu'il était impossible de pouvoir illustrer la méthodologie sur le cas d'étude *Airline 1* étant donné qu'il n'y avait plus d'accès possible à l'application. Cependant, afin de pouvoir illustrer la méthodologie du Model-Based Testing dans ce mémoire, il a été décidé d'un commun accord avec mon tuteur du mémoire Monsieur Lotfi CHAARI ainsi que mon tuteur d'entreprise Monsieur Laurent THULL d'utiliser la méthodologie expliquée dans le chapitre II sur une autre application web, comme celle *d'Amazon*.

3.1.2 Une autre application pour cette même problématique

L'entreprise *Amazon* est une multinationale dont le siège social est à Seattle aux États-Unis. Si la société au départ était spécialisée dans la vente de livres en ligne, de nos jours *Amazon* est devenu un site de e-commerce incontournable dans le monde avec plus de :

— 34 milliard de dollars de chiffres d'affaires.

— 137 millions de clients par semaine.

— En moyenne un client rapporte à l'entreprise 189 $.

— 10 % du e-commerce aux États-Unis.

3.2 Outillage pour la mise en oeuvre

Dans cette section, nous aborderons la comparaison des formats de transformation, dans le but de choisir celui qui nous conviendra le plus pour notre cas d'étude. Ensuite nous énumérerons les différents outils de Model-Based Testing et nous finirons avec les choix techniques que nous aurons fait pour la mise en oeuvre.

3.2.1 Comparaison des formats de transformation de modèles

Nous avons vu dans la sous-section 2.5.1 qu'il existait deux types de format pour la transformation de modèle : le format XML et le format JSON. Or, dans l'étude réalisée par Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, Clemente Izurieta [Nur+09], il a été démontré que le format JSON était bien plus performant que le format XML, comme on peut le voir dans le tableau ci-dessous.

	JSON	XML
Number Of Objects	1000000	1000000
Total Time (ms)	78257.9	4546694.78
Average Time (ms)	0.08	4.55

TABLE III.1 – Tableau comparatif du format JSON et XML.
(Source : Comparison of JSON and XML data interchange formats : a case study [Nur+09])

Nous pouvons voir d'après ce tableau que JSON est beaucoup plus rapide dans l'exécution d'un grand nombre de tests, 1 million. En moyenne, JSON est presque 57 fois plus rapide que XML. Par conséquent, pour notre cas d'étude, nous utiliserons JSON comme format de transformation de modèles.

3.2.2 Les outils de Model-Based Testing

Pour pouvoir choisir les meilleurs outils pour Model-Based Testing, nous allons nous appuyer sur le tableau comparatif ci-dessous réalisé par Gijs van der Meijde, de l'Université Radboud de Nijmegen aux Pays-Bas. Ce tableau énumère un grand nombre d'outils et les conditions à remplir pour vérifier leurs performances.

Tool	publicly available	Compatible with model	Offline testing	Export functionality	Coverage based	Multi-platform
Lutess	Yes	No	No	No	Yes	-
Lurette	Yes	No	No	No	Yes	-
GATeL	Yes	No	Yes	-	-	-
AutoFocus	Yes	No	Yes	Yes	Yes	-
Conformance kit	No	-	-	-	-	-
PHACT	No	-	-	-	-	-
TVEDA	Yes	Yes	Yes	-	Yes	-
Cooper	Yes	Yes	-	-	Yes	-
AsmL	Yes	Yes	Yes	Yes	Yes	No
TGV	Yes	-	Yes	Yes	Yes	-
TorX	Yes	Yes	No	No	Yes	Yes
STG	No	-	Yes	Yes	-	-
AGEDIS	No	No	Yes	Yes	Yes	-
TestComposer	Yes	Yes	Yes	Yes	Yes	Yes
AutoLink	Yes	Yes	Yes	Yes	Yes	-
Jumbl	Yes	Yes	Yes	Yes	Yes	Yes
GraphWalker	Yes	Yes	Yes	Yes	Yes	Yes
HADS	Yes	Yes	Yes	Yes	Yes	Yes

TABLE III.2 – Tableau comparatif des outils de tests.
(Source : A Practical Application of Model Learning and Code Generation [Mei+18])

JUMBL :

JUMBL, J Usage Model Building Library, gère les différentes étapes de Model Based Testing, de la modélisation à l'analyse des résultats. Pour ce faire, même s'il peut agir avec des méthodes de transition standard, il adopte les chaînes de Markov pour choisir quel sera le groupe de tests le plus profitable et en génèrera le chemin à emprunter. Ces systèmes ont des états de début et de fin bien déterminés, or ce n'est pas toujours le cas de nos modèles. Pour la construction des séquences de tests JUMBL utilisera l'algorithme du facteur chinois, un graphe connexe non orienté, étant donné qu'il se fonde sur la probabilité d'utilisation d'un modèle.

GraphWalker :

Le test dirigé par les modèles peut utiliser plusieurs algorithmes pour la génération des cas de tests. Ceci peut être exploité par l'outil GraphWalker. Les différents algorithmes pour la génération des tests continuent de s'exécuter tant qu'une condition d'arrêt ne soit atteinte. La couverture de l'intégralité des sommets peut être une exigence. GraphWalker peut également avoir ce type d'exigences grâce à une bibliothèque Java dédiée. La génération et l'exécution se font à l'aide de machines à états finis étendues (EFSM). GraphWalker peut être exécuté *online* et *offline*.

Hybrid adaptive distinguishing sequences :

L'outil Séquences de distinction adaptatives hybrides (HADS), permet de générer des cas de test à l'aide de machines d'états finies. Pour réaliser la génération de ces différents suites de tests, l'outil se base sur l'approche démontrée par Lee et Yannakakis, en 1994, à l'aide de spécifications de tests sous la forme de machines Mealy. Si l'outil à la possibilité d'utiliser de nombreuses variante pour la génération des cas de tests comme par exemple la méthode en W directement hérité du monde du management, il doit pour que la génération soit complète, utiliser des graphes déterministes, finies.

3.2.3 Choix techniques pour la mise en oeuvre

Après avoir vu quelques-uns des outils pour appliquer Model-Based Testing, comme il s'agit d'une application web et par conséquent sujet à des changements d'états, les organismes ISTQB et CFTL, son équivalent en France, préconisent l'utilisation de machines d'états pour la vérification des réponses que les changements d'états procurent sur le système cible à tester, comme le montre le tableau ci-dessous.

Objectif de test	Exemple de modélisation MBT	Sujet du modèle	Focus du modèle
Vérifier que le flot métier est implémenté correctement	Un modèle de processus métier décrivant le flot	Système	Comportemental
Vérifier que le système fournit des réponses correctes à des stimulations dans des états spécifiques	Une machine à états UML	Système	Comportemental
Vérifier la disponibilité d'une interface	La description de la structure de l'interface	Système	Structurel
S'assurer que l'objet de test est adapté aux usages effectifs	Un modèle d'usage décrivant le comportement de l'utilisateur	Environnement	Comportemental
Vérifier les configurations d'un système	Un modèle de données utilisant un arbre de classification	Données	Structurel

TABLE III.3 – Tableau ISTQB sur la modélisation MBT en fonction des objectifs.
(Source : ISTQB Testeur Certifié Niveau Fondation Model-Based Testing Syllabus [Chr+15])

Pour le choix des outils à utiliser pour la mise en œuvre de notre cas d'étude, en ce qui concerne les parties modeleur et générateur, nous allons nous orienter vers GraphWalker ; et ce pour plusieurs raisons :

- GraphWalker accepte le format JSON.
- C'est un outil "open source".

- Sur le tableau de Gijs Meijde, voir le Tableau III.2, toutes les colonnes des caractéristiques sont au vert.
- Les organismes ISTQB et CFTL préconisent l'utilisation de machines d'état pour les tests d'applications.
- GraphWalker est maintenu régulièrement, la dernière mise à jour date de 2020.

3.2.3.1 GraphWalker

Nous avons pu voir que GraphWalker est un outil open-source. Il est utilisé pour créer des modèles en termes de graphes et il est aussi capable de générer des cas de test à partir de ces graphes basés sur des machines à états finis.

Il contient deux modules utilisés à cet effet : l'outil CLI, Command Line Interface, et la version GUI, Graphical User Interface, appelée GraphWalker Studio. Le module CLI est un outil pour travailler avec des graphiques. Ce module a différents sous-ensembles de commandes pour son mode hors ligne qui peuvent générer une séquence de test qui peut être exécutée ou utilisée pour vérifier l'exactitude du modèle.

Le module CLI gère la génération des méthodes abstraites et l'exécution du modèle SUT car il ne peut agir que sur des modèles déjà créés. Il est assimilable aux formats graphml et JSON. Le mode online permet de se connecter directement au SUT et de manière dynamique effectuer les tests. CLI peut agir sur le changement de l'élément de point de départ de la machine d'états. Les modèles contiennent différents arcs orientés, ce qui permet de montrer que GraphWalker se dirige vers tous les autres chemins possibles afin de permettre la création des cas de tests.

L'outil GraphWalker Studio possède une interface graphique qui lui permet de faire l'édition des modèles. Les sommets ainsi que les arêtes sont les deux constituants d'un modèle. Puisque les machines d'états finis sont la base de ces modèles, le principe d'action et de garde peuvent également s'y appliquer. Les actions représentent l'exécution du modèle quand ce dernier arrivera à l'élément cible. Quand à la garde, elle ne permet l'accès à l'élément, que si, et seulement si, la condition d'accès à l'élément est remplie. Ces deux constituants, ainsi que les propriétés d'action et de garde, sont également présents dans la version CLI de GraphWalker. L'état attendu d'un système est représenté par un sommet alors que les transitions sont quand à elles représentées par des arêtes, des arcs. L'exécution du modèle de test peut également se faire grâce à l'outil graphique de GraphWalker : GraphWalker Studio. De cette manière, le cheminements

entre les différents sommets est apparent et permet ainsi d'avoir un premier contrôle visuel du modèle. A la suite de ce premier contrôle, le générateur pourra choisir comment rendre possible le chemin préalablement visualisé. La génération peut se faire aléatoirement à l'aide de différents algorithmes. Les cibles aussi sont sélectionnées par l'outil de manière aléatoire du moment que la condition d'arrêt aura été spécifiée clairement.

Pour mieux visualiser, ci-dessous le schéma explicatif de l'architecture du fonctionnement de GraphWalker Studio.

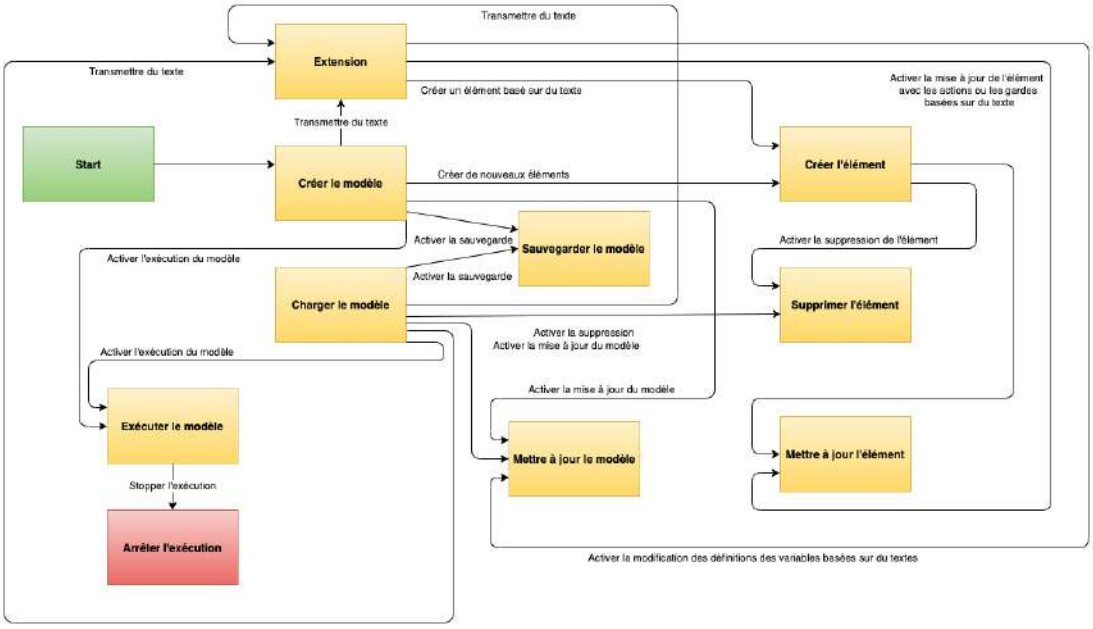

FIGURE 3.1 – Architecture du fonctionnement de GraphWalker Studio.

3.2.3.2 Selenium

Afin de pouvoir exécuter les différentes méthodes de tests que génère le modèle de manière automatique, nous nous appuierons sur des frameworks de tests qui permettent de répondre à cet objectif, notamment avec la solution Selenium.

L'outil Selenium fait parti des frameworks d'automatisation de suites de tests les plus populaires et il est également open source. Il a été créé par Jason Huggins en 2004. Selenium peut être utilisé par divers systèmes d'exploitation mais il peut également prendre en charge les tests appliqués aux applications mobiles.

Pour mieux comprendre l'articulation du projet Selenium, l'illustration ci-dessous donne une vue d'ensemble des composants de Selenium.

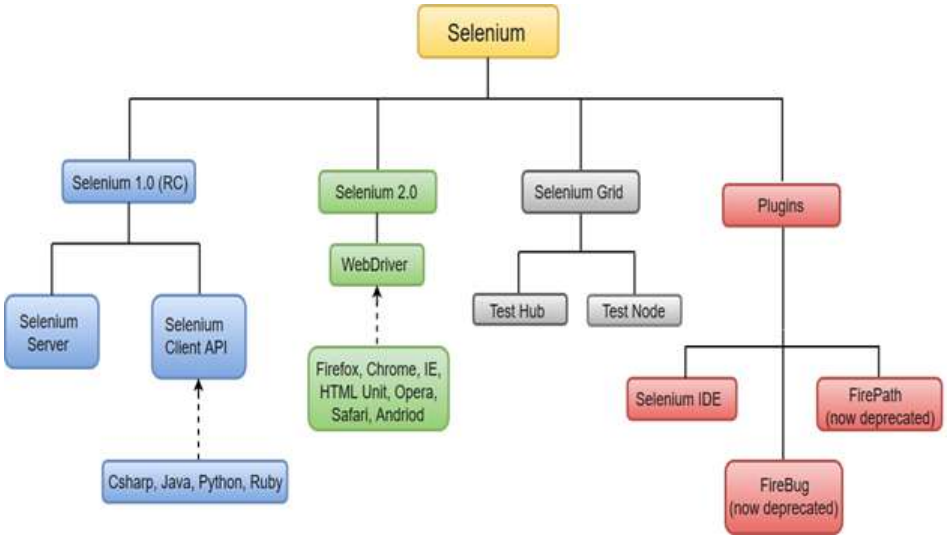

FIGURE 3.2 – Illustration de l'arborescence du projet Selenium.
(Source : www.javatpoint.com [Sona])

D'après la Figure 3.2, nous pouvons voir que les débuts de Selenium WebDriver, en tant qu'outil de test pour les navigateurs web, ont été réalisés avec la version 2.0 du projet. Auparavant la première version de Selenium comprenait seulement RC, Grid et IDE.

Il est possible d'écrire des scripts de tests avec de nombreux langages de programmation tel que : Java, Python, C#, PHP et Ruby.

L'utilisation Selenium WebDriver plutôt que Selenium RC fait que le framework interagit directement avec les navigateurs plutôt que d'interroger un serveur "intermédiaire" comme c'était le cas dans la version initiale.

Afin de mieux comprendre comment Selenium communique avec les différents navigateurs, l'illustration ci-dessous représente ce processus de communication.

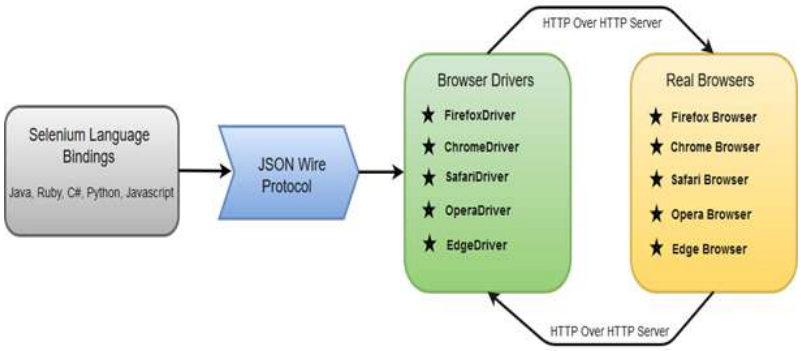

FIGURE 3.3 – Illustration du processus de communication de Selenium WebDriver.
(Source : www.javatpoint.com [Sona])

Cette illustration nous permet d'observer le processus de communication qu'utilise Selenium WebDriver. Il se compose de quatre composants :

- Un langage de programmation pour le script de test.
- Un protocole d'échange de données en JSON.
- Un pilote de navigateur web.
- Un navigateur web.

Nous pouvons observer qu'une fois que le développeur a écrit son script de test, un ensemble de bibliothèques clientes, fourni avec l'outil Selenium, va permettre de rendre compatible les différents langages utilisés par le développeur pour l'exécution automatique de son test. Un fois que cette compatibilité est faite, le standard d'échange de données JSON va permettre interopérabilité entre le serveur et le client. Pour plus de détails sur ce standard veuillez vous référer à la sous-sous-section 2.5.1.2. Enfin, Selenium établit une communication avec le navigateur pour que le test puisse se réaliser sans toutefois révéler la logique du test, afin de garantir un certain niveau de confidentialité.

3.2.3.3 TestNG

Pour que le processus du Model-Based Testing soit complet, une fois que l'étape d'exécution est réalisée, nous devons être capable de générer un rapport de test en lien avec le modèle d'entrée. Pour ce faire nous utiliserons un autre framework de test open source : TestNG.

TestNG est un framework de test qui permet, en partant des cas de tests exécutés par Selenium, ainsi que des données de tests fourni par exemple par des fichiers XLS, des rapports HTML et, en cas de problèmes d'exécution de TestNG, des rapports de logs.

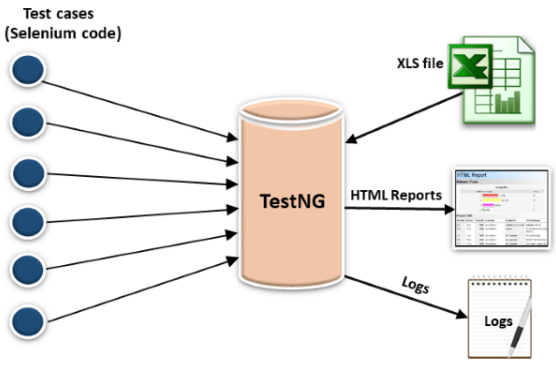

FIGURE 3.4 – Vue d'ensemble de TestNG.
(Source : www.javatpoint.com [Sonb])

Il a été développer afin de sillonner tous les différents types de tests, dans leur globalité. Il possède de nombreux caractéristiques notables à savoir :

- Le regroupement de plusieurs cas de tests.
- Il n'est pas un outil dépendant des classes java.
- Un minima d'annotations est requis pour son utilisation comme par exemple : *@BeforeClass* et *@AfterClass*.
- Il a la possibilité de réaliser des tests de dépendances sur les méthodes.
- L'exécution des tests en parallèle est possible comme pour le test paramétré.
- L'ordonnancement des tests est possible.
- Il a la possibilité de générer un rapport de test en HTML.

Après cette vue d'ensemble sur différents outils qui sont retenu pour le cas d'étude, le schéma ci dessous montre quand est-ce que ces différents outils et quels sont leur rôle respectif dans la méthode du Model-Based Testing.

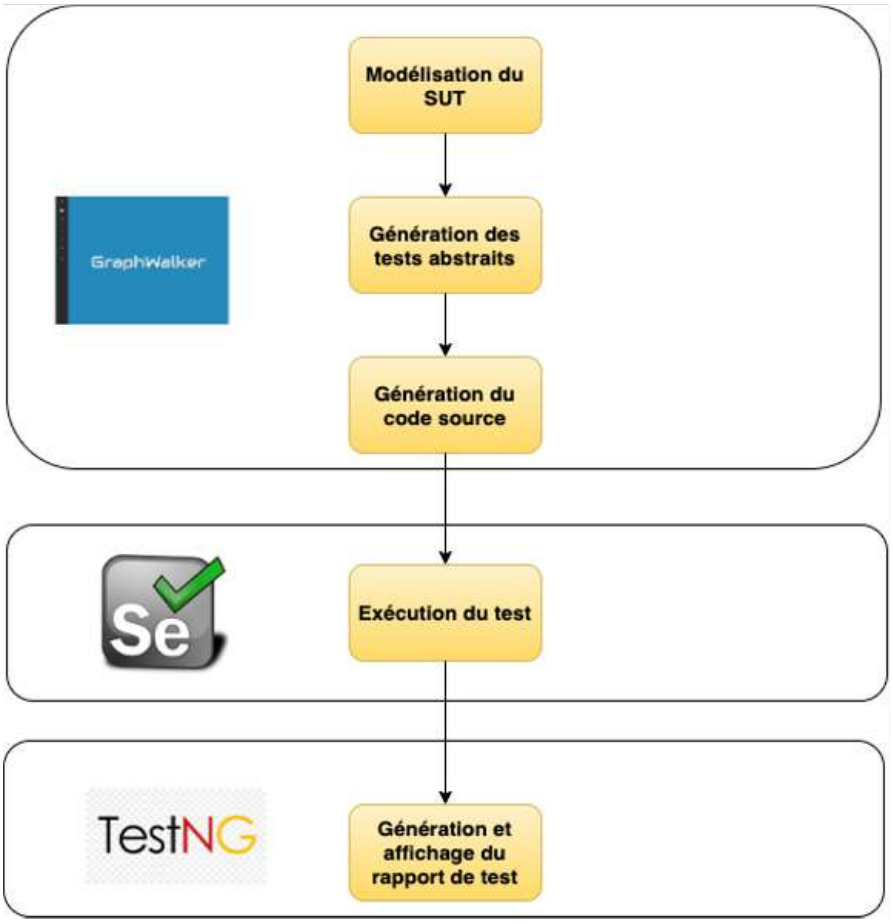

FIGURE 3.5 – Récapitulatif des outils utilisés pour la mise en oeuvre.

3.3 Méta-modélisation et exigences du système sous-test

Dans cette section, nous allons aborder plus en détails la création du méta-modèle du système à tester ainsi que les différentes exigences du modèle sous-test de notre cas d'étude.

3.3.1 Création du méta-modèle

Dans tout système que l'on doit tester, il est indispensable d'élaborer un méta-modèle. En effet, le méta-modèle permettra d'avoir :

- Une représentation abstraite, une cartographie, du système à tester.
- Un modèle de base qui permettra d'élaborer les différents modèles SUT du systèmes, qui eux, permettront de tester les différentes fonctionnalités du système.

Le méta-modèle est le composant qui va orchestrer l'élaboration des modèles. La manipulation ainsi que la précision du modèle doivent être pratiques pour faciliter leur utilisation. Cela permettra d'enrichir le modèle abstrait pour y incorporer des modèles intermédiaires qui permettront de faire la jonction entre les éléments du modèle abstrait et les autres composants. Pour réaliser le méta-modèle, nous devons représenter les différentes parties de l'application web que nous devrons tester. Pour construire ce méta-modèle nous utiliserons une machine d'états finis.

De cette manière, cette représentation du méta-modèle a pour avantage de rendre aisé la compréhension de l'application web à tester. De plus, étant donné que les différents modèles SUT, qui auront pour but de tester les différentes fonctionnalités de l'application, proviennent directement du méta-modèle produit précédemment, ils seront également construits à l'aide de machines d'états finis pour que leurs élaborations soient facilitées. Donc, le fait que les modèles et les méta-modèles soient de natures identiques, machine d'états finis, cela facilitera : leur compréhension, la correspondance entre les deux, l'ajout de nouveaux paramètres, la maintenance, etc.

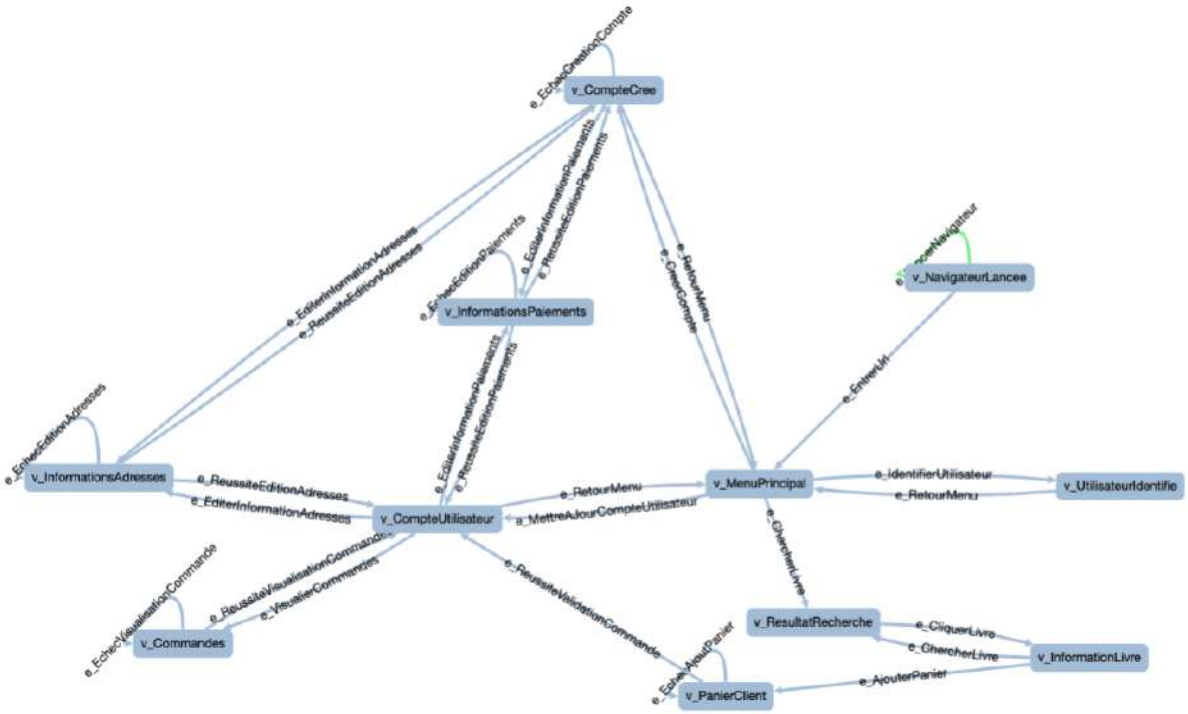

FIGURE 3.6 – Méta-modèle du système *Amazon*.

D'après l'illustration du méta-modèle ci dessus, nous pouvons voir que le système peut être représenté par deux artefacts distincts mais complémentaires :

- Les *vertex*, sommets, qui représentent les états attendus du système.
- Les *edges*, arcs, qui représentent les différentes transitions, les actions pour aller d'un état attendu à l'autre, d'un sommet à l'autre.

Dès à présent, l'ensemble des éléments qui permettent de caractériser l'application web est matérialisé par le méta-modèle. Ce composant peut être amené à évoluer au fur et à mesure que de nouvelles exigences s'ajouteront au niveau du méta-modèle. Nous avons donc comme principaux composants :

- *v_NavigateurLancee :* L'état du navigateur web, ce sommet, permet de vérifier que l'action du navigateur a été bien lancée par l'automate de test.
- *v_MenuPrincipal :* L'état de l'arrivée du client de l'application web est matérialisé, afin de pouvoir représenter la page d'accueil de l'application sur laquelle le client pourra se "diriger" pour s'authentifier, réaliser une première recherche ou encore gérer son compte client.

- *v_UtilisateurIdentifie :* L'état de l'authentification du client lorsqu'il réussit à s'authentifier.
- *v_CompteCree :* L'état de création du compte client, lorsque ce dernier n'en possède pas déjà un, va permettre de vérifier si le compte client a bien été créé.
- *v_InformationsPaiements :* L'état d'informations sur le paiement illustre toutes les informations que doit rentrer le client dans le système, pour tout ce qui peut se rapporter au paiement d'une commande.
- *v_InformationsAdresses :* L'état d'informations sur les adresses illustre toutes les informations que doit rentrer le client dans le système, pour tout ce qui peut se rapporter aux informations liées aux adresses de livraison ainsi que celle de la facturation d'une commande.
- *v_CompteUtilisateur :* L'état du compte de l'utilisation représente toutes les actions que l'utilisateur, le client, peut faire sur son compte client.
- *v_Commandes :* L'état des différentes commandes représente toutes les informations et actions que le client peut faire une fois qu'une commande est réalisée et validée.
- *v_ResultatRecherche :* L'état des résultats de recherches représente l'affichage des résultats de la recherche du client.
- *v_InformationLivre :* L'état du choix du client représente l'affichage des informations sur un livre donnée.
- *v_PanierClient :* L'état de validation du choix du client dans son panier d'achats.

Maintenant que nous avons vu les différents changements d'états dû aux arcs et sommets, la relation entre un scénario de test et un des modèles sous test, qui fait partie du méta-modèle, sera abordé dans la sous-section suivante.

3.3.2 Exigences du modèle sous-test

Du fait de son envergure et du nombre de clients que compte le site, il y a, à chaque ajout de fonctionnalité, de nombreux tests à effectuer. Afin de pouvoir répondre aux besoins de tests et à la qualité du test, nous allons l'illustrer avec le test d'ajouter un article dans le panier d'un client au moyen de la fonctionnalité "ajouter panier". Bien entendu, tous les autres tests de fonctionnalités suivront le même cheminement.

La première étape réside dans le recueil des exigences de la fonctionnalité à tester, afin d'être certain que l'ensemble des aspects soient couverts.

1.04 Ajouter au panier			
Date	01/09/2020		
Tester	Gautier		
Initial conditions:			
Step	Actions	Expected result	Status
1	Aller sur le site web.	Le cataloge du site est visible.	OK
2	Cliquer sur le bouton "Identifiez-vous".	Le site nous redirige vers une page secondaire afin de nous permettre de nous identifier.	OK
3	Renseigner l'adresse e-mail du client.	On peut voir l'adresse mail dans la zone de texte.	OK
4	Cliquer sur le bouton "Continuer"	Le site nous redirige vers une autre page secondaire afin de nous permettre de renseigner le mot de passe.	OK
5	Rensigner le mot de passe du client.	On peut voir le mot de passe en crypté dans la zone de texte.	OK
6	Cliquer sur le bouton "Continuer"	Le site nous redirige vers la page d'acceuil du site.	OK
7	Renseigner un ouvrage dans la barre de recherche	On peut voir le titre de l'ouvrage dans la zone de texte.	OK
8	Une liste de contenu est alors affiché	La liste du contenu est affichée.	OK
9	Cliquer sur le titre de l'ouvrage choisi	Le site nous redirige vers une autre page secondaire afin de nous permettre de voir les informations sur l'ouvrage.	OK
10	Cliquer sur le bouton "Ajouter au panier"	L'ouvrage s'ajoute au panier du client, le compteur du panier s'incrémente. Par la suite, une sous-fenêtre apparaît.	OK
FINAL STATUS			OK

TABLE III.4 – Scénario de tests de l'ajout du panier du système *Amazon*.

Le tableau ci-dessus montre objectivement les différentes étapes pour tester la fonctionnalité *ajouter panier*. Ces différentes étapes sont dans l'ordre :

- Se rendre sur le site *Amazon*.
- S'identifier.
- Cette identification entraîne la saisie de l'e-mail.
- Une fois que l'e-mail est saisi, il faut par la suite renseigner le mot de passe.
- Une fois que les informations du client sont saisies, on clique sur le bouton de validation.
- Une fois la validation faite, le système redirige le client vers la page d'accueil afin qu'il puisse renseigner le nom d'un ouvrage.

- Le système va alors afficher à l'écran un ensemble de résultats en rapport avec la recherche du client.
- Le client clique sur l'ouvrage désiré et le système le redirige vers le contenu spécifique du choix.
- Pour finir, le client va cliquer sur le bouton *ajouter panier*.

Après avoir vu l'ensemble des aspects à tester pour cette fonctionnalité, dans la section suivante nous verrons plus en détails comment tester par le procédé du Model-Based Testing avec les différents constituants du pipeline de la méthode MBT.

3.4 Conception de la solution par Model-Based Testing

Dans la sous-section 2.3.2 nous avons expliqué l'ensemble du procédé Model-Based Testing avec les composants de la méthodologie : l'édition du modèle sous test, la génération des scripts de tests, l'exécution des scripts de tests et enfin la publication des résultats. Nous allons maintenant aborder la manière de l'appliquer à notre étude de cas.

3.4.1 Composant : Éditeur du modèle sous test

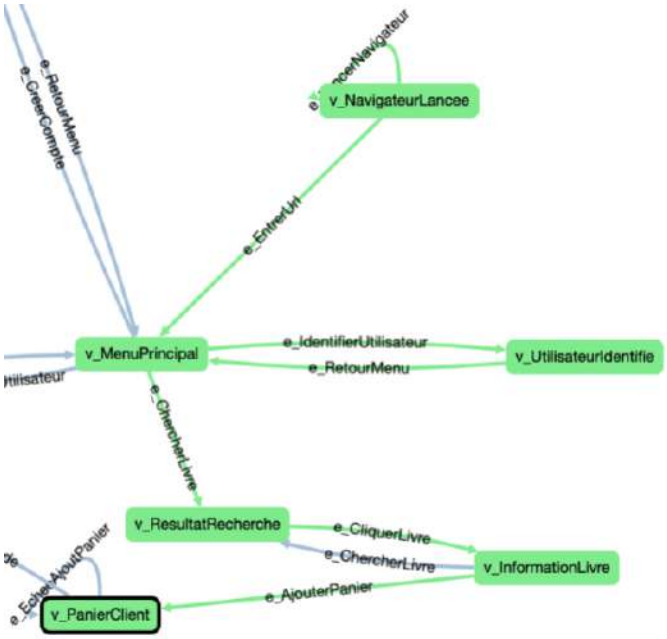

FIGURE 3.7 – Le modèle SUT de la fonctionnalité *Ajouter au panier*.

Pour illustrer le composant "Éditeur du modèle sous test", ci-dessus une partie du méta-modèle représentant le modèle SUT pour l'ensemble des fonctionnalités nécessaires à couvrir pour tester la fonctionnalité *ajouter panier*. Comme expliqué dans la sous-section précédente, différentes opérations sont à respecter pour couvrir les les exigences. Un des composants de cette couvertures d'opérations est représentée ci-dessous avec la transition *e_IdentifierUtilisateur*. Nous pouvons observer que pour réussir cette opération, on doit satisfaire la garde, *guard*, qui est un ensemble de contraintes pré-établies, une série de conditions à respecter. Dans notre cas, il s'agit :

- D'avoir saisi le bon URL de l'application.
- De posséder un compte utilisateur.
- Et enfin que l'utilisateur ne ce soit pas encore identifié.

FIGURE 3.8 – Illustration d'une transition sur *e_IdentifierUtilisateur*.

Une fois que l'ensemble des critères de la garde est satisfaisante, la garde autorise le changement d'état et ainsi permet que l'action *Utilisateur_identifie* se mette à jour et permet ainsi l'action de l'identification de se réaliser et permettra la redirection du client identifié vers la page d'accueil de l'application web. La réalisation de l'identification comprend plusieurs étapes qui faudra également tester : entrer l'email puis cliquer sur le bouton de validation puis renseigner le mot de passe puis cliquer sur le bouton de validation.

Nous pouvons voir dans la figure suivante l'illustration d'une partie du méta-modèle faite à l'aide du premier module de *GraphWalker*, à savoir, *GraphWalker Studio*. Le modèle sous test, graphique, est enregistré au format JSON. Le fichier JSON contient la description textuelle du

modèle que nous venions de créer. Nous pouvons également observer que les gardes et les actions sont bien matérialisées. Cette description va être ensuite utilisé par le deuxième module de l'outil *GraphWalker* pour la partie de génération, à savoir *GraphWalker* lui-même. Celui-ci va permettre par la suite la génération des méthodes abstraites et le parcours du modèle sous test.

```
{
  "models": [
    {
      "edges": [
        {
          "id": "750b3c00-1eb0-11eb-bd1c-a59832f35e1b",
          "name": "e_LancerNavigateur",
          "sourceVertexId": "664a39f0-1eb0-11eb-bd1c-a59832f35e1b",
          "targetVertexId": "664a39f0-1eb0-11eb-bd1c-a59832f35e1b",
          "weight": "0",
          "actions": [
            "navigateur = true;"
          ]
        },
        {
          "id": "1ceb4f00-1eb1-11eb-bd1c-a59832f35e1b",
          "name": "e_EntrerUrl",
          "sourceVertexId": "664a39f0-1eb0-11eb-bd1c-a59832f35e1b",
          "targetVertexId": "0ebe2880-1eb1-11eb-bd1c-a59832f35e1b",
          "guard": "navigateur == true",
          "weight": "0",
          "actions": [
            "url = true;"
          ]
        },
        {
          "id": "1f960730-1eb2-11eb-bd1c-a59832f35e1b",
          "name": "e_IdentifierUtilisateur",
          "sourceVertexId": "0ebe2880-1eb1-11eb-bd1c-a59832f35e1b",
          "targetVertexId": "f350b710-1eb1-11eb-bd1c-a59832f35e1b",
          "guard": "url == true && possede_compte == true && utilisateur_identifie == false",
          "weight": "0",
          "actions": [
            "utilisateur_identifie = true;"
          ]
        },
```


FIGURE 3.9 – Illustration du méta-modèle au format JSON.

L'illustration suivante est une autre manière de montrer la description du modèle sous test, de manière plus lisible pour l'utilisateur du modèle au même format JSON. Nous constatons que le modèle devra prendre en compte plusieurs types d'opérations, d'actions.

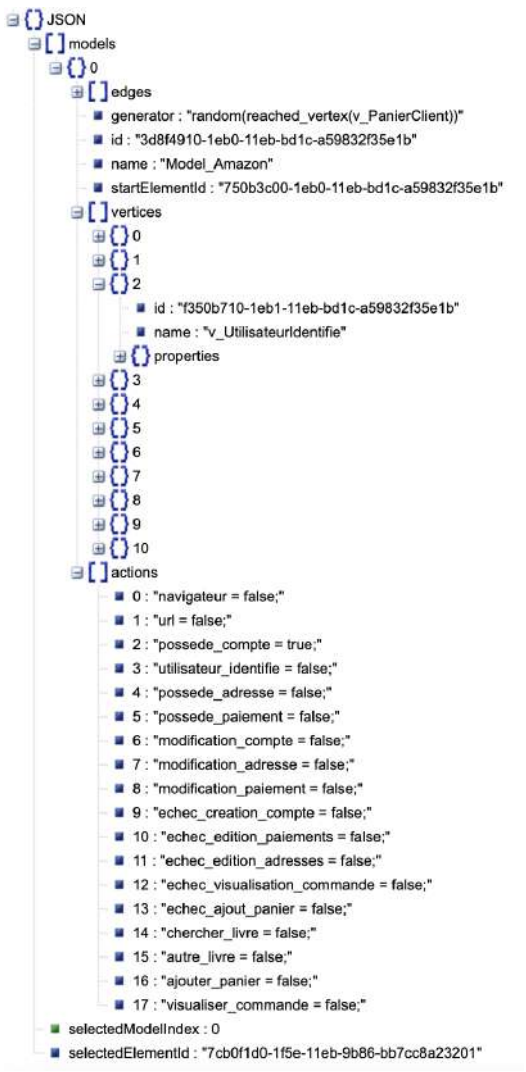

FIGURE 3.10 – Illustration de l'arborescence du méta-modèle avec ses actions au format JSON.

Nous avons vu dans cette sous-section la partie de l'édition du modèle sous-test ainsi que les gardes et les actiosn correspondantes que le générateur des scripts de tests devra prendre en ligne de compte afin de poursuivre le procédé du Model-Based Testing. Dans la sous-section suivante nous verrons l'étape de compilation et de génération des méthodes abstraites.

3.4.2 Composant : Compilateur et générateur

Une fois que nous avons encadré notre système sous test par les différentes gardes et actions correspondantes nécessaires à la mise à jour du système sous test en prévision de l'exécution du cas de test, nous pouvons maintenant procéder à l'étape de génération des méthodes abstraites du modèles afin que ces dernières, lors de l'étape d'exécution, soient successivement exécutées par l'automate de test suivant le cheminement préalablement établi par le générateur de test en

fonction de tous les arcs et les sommets du modèle.

Pour pouvoir réaliser cette étape de génération des différentes méthodes abstraites, l'éditeur de la solution *GraphWalker* propose de générer un projet *maven* à partir d'un archétype. Cela permettra de génener un projet *maven* valide avec toutes les dépendances requises pour le bon fonctionnement du générateur *GraphWalker*.

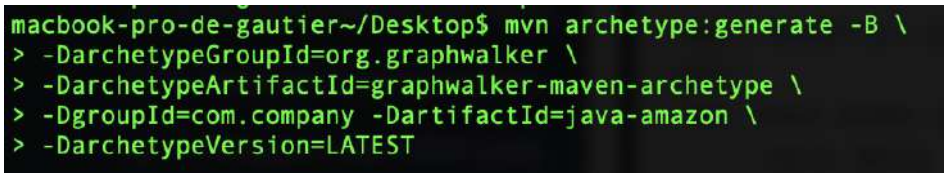

FIGURE 3.11 – Illustration de la commande de génération de l'archétype du projet.

Une fois archétype du projet créé, nous n'avons plus qu'à introduire notre modèle sous test, préalablement créé par l'outil graphique, dans le répertoire *resources* afin que le générateur puisse trouver le modèle et ainsi créer l'ensemble des méthodes abstraites.

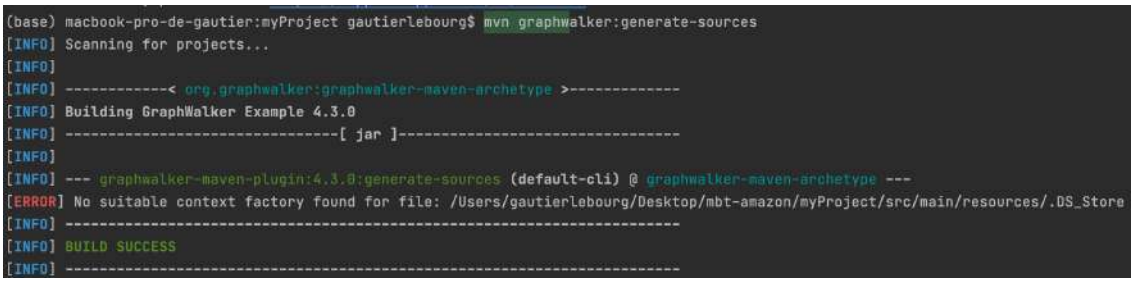

FIGURE 3.12 – Illustration de la commande *mvn graphwalker :generate-sources*.

L'étape de génération se fait avec la commande maven *mvn graphwalker :generate-sources* et va ainsi permettre de générer l'arborescence des méthodes abstraites que nous voyons ci-dessous à la Figure 3.13. Cette arborescence créée par le générateur contient un ensemble de répertoires :

- *target*.
- *generated-sources*.
- *graphwalker*.
- *com*.
- *company*.

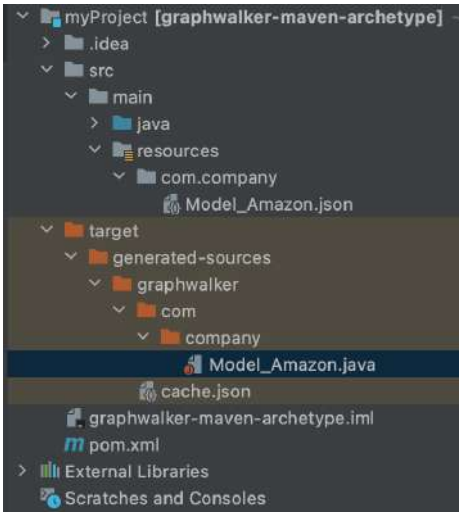

FIGURE 3.13 – Illustration de l'arborescence générée.

Dans cet ensemble de répertoires, à l'intérieur de *company* nous retrouvons notre fichier *JAVA Model_Amazon* contenant les différentes méthodes abstraites qui sont illustrées dans la Figure 3.14.

```java
// Generated by GraphWalker (http://www.graphwalker.org)
package com.company;

import org.graphwalker.java.annotation.Model;
import org.graphwalker.java.annotation.Vertex;
import org.graphwalker.java.annotation.Edge;

@Model(file = "com/company/Model_Amazon.json")
public interface Model_Amazon {

    @Edge()
    void e_LancerNavigateur();

    @Vertex()
    void v_NavigateurLancee();

    @Edge()
    void e_EntrerUrl();

    @Vertex()
    void v_MenuPrincipal();

    @Edge()
    void e_IdentifierUtilisateur();

    @Edge()
    void e_ChercherLivre();

    @Vertex()
    void v_ResultatRecherche();

    @Edge()
    void e_CliquerLivre();

    @Vertex()
    void v_InformationLivre();

    @Edge()
    void e_AjouterPanier();

    @Vertex()
    void v_PanierClient();
```

FIGURE 3.14 – Illustration des abstractions de méthodes générées.

Maintenant que nous avons tous les éléments nécessaires à l'étape de l'exécution, nous allons pouvoir procéder à cette étape dans la sous-section suivante.

3.4.3 Composant : Exécuteur de script

L'étape d'exécution est celle qui nécessite une préparation pour pouvoir ensuite laisser faire l'automate de test. Dans notre cas, comme nous l'avons montré d'ors et déjà dans la sous-section 3.2.3, nous utiliserons automate de test fourni avec l'outil *Selenium webDriver*. Pour pouvoir mener à bien cette étape d'exécution nous devons cependant implémenter les différentes méthodes abstraites qui vont permettre à l'automate de test d'interroger l'arborescence du *DOM* afin d'exécuter le test et de mettre à jour le système cible. Ci-dessous, vous pouvez observer une partie de l'implémentation des méthodes de tests.

```
@GraphWalker(value = "random(reached_vertex(v_PanierClient))", start = "e_LancerNavigateur")
public class Model_Amazon_Impl implements Model_Amazon {
    WebDriver driver = null;
    WebDriverWait waiter = null;


    String NOMLIVRE = "Systèmes Architectures Intégration Pratique de l'Intégration des Systèmes et des Logiciels avec SysML";
    String NOMAUTEUR = "Yann Pollet";

    @BeforeExecution
    public void setup() {
        System.setProperty("webdriver.chrome.driver", "/Users/gautierlebourg/Desktop/Selenium/chromedriver");
    }

    @AfterExecution
    public void quit() {
        if (driver != null) {
            driver.quit();
        }
    }

    @Test
    public void e_LancerNavigateur() {
        driver = new ChromeDriver();
        waiter = new WebDriverWait(driver, 10);

    }
    @Test
    public void v_NavigateurLancee() {
        Assert.assertNotNull(driver);
    }

    @Test
    public void e_EntrerUrl() {
        String url = "https://www.amazon.fr/";
        driver.get(url);

    }

    @Test
    public void v_MenuPrincipal() {
        waiter.until(ExpectedConditions.urlContains("amazon.fr"));
    }

    @Test
    public void e_IdentifierUtilisateur() {
        String btnIdentification = "//a[@id='nav-link-accountList']/div/span";
        String emailLocator = "//div/input[@id='ap_email']";
        String btnContinue = "//span/input[@id='continue']";
        String btnIdentifie = "//span/input[@id='signInSubmit']";
        String passwordLocator = "//div/input[@id='ap_password']";
```

FIGURE 3.15 – Illustration de l'implémentation des méthodes abstraites de tests.

Nous pouvons observer grâce à l'illustration ci-dessus que notre classe de test, qui implémente les méthodes abstraites, possède une première annotation :
@GraphWalker(value="random(reached_vertex(v_PanierClient))", start = "e_LancerNavigateur")).
Cette annotation va permettre de faire reconnaître à l'outil *GraphWalker* que nous devons rechercher et terminer l'exécution par le sommet *v_PanierClient* en commençant l'exécution par l'arc *e_LancerNavigateur*.
Par la suite, de simples précautions d'usages sont à prendre pour le bon fonctionnement de l'outil, cette fois *Selenium*, comme par exemple vérifier qu'il n'y ait pas d'erreur au lancement du navigateur, dans ce cas nous devons prévoir que l'automate de test quitte prématurément le navigateur web. De façon analogue nous devons prévoir que lorsque le cas de test se termine, l'automate devra d'abord fermer le navigateur puis s'auto-détruire.

Une fois ces deux précautions prises, nous pouvons enfin implémenter nos différentes méthodes de tests avec le code permettant à l'outil *Selenium WebDriver* d'interroger les différents éléments du *DOM* en rapport avec le scénario de test que nous devons exécuter.

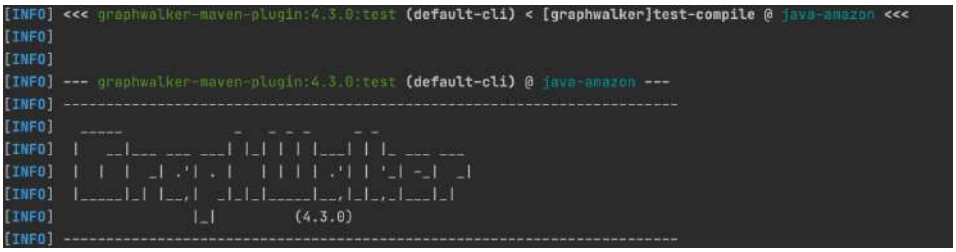

FIGURE 3.16 – Illustration de la commande *mvn graphwalker :test*.

L'illustration ci-dessus montre la commande permettant l'exécution de l'automate de test et ainsi laisser faire l'automate pour au final obtenir un rapport de test, que nous aborderons dans la sous-section suivante.

Avant cela, pour plus de clarté, ci-après une illustration montrant de manière simplifié, le processus que suit le test *e_ChercherLivre*. Nous partons d'une partie du modèle sous test sous forme d'une machine d'états finis comprenant le menu principal, l'action de chercher un livre et le résultat de la recherche. Comme nous l'avons spécifié précédemment, après avoir respecter les gardes d'identifications, l'automate pourra alors être redirigé vers le menu principal de l'application. Nous montrons avec l'illustration suivante la méthode *e_ChercherLivre* qui d'une

part demande à l'automate de test d'attendre que l'élément de la barre de recherche soit visible afin, d'autre part, de pouvoir lancer une requête qui va permettre de rechercher l'ouvrage désiré et valider la recherche. Une liste d'ouvrage sera affiché, un ouvrage pourra être choisi. L'illustration suivante montre que le *DOM* a bien reçu la requête de l'automate à savoir : "Systèmes Architectures Intégration Pratique de l'Intégration des Systèmes et des Logiciels avec SysML". Enfin, avant de cliquer sur l'ouvrage, l'automate exécutera la méthode *v_ResultatRecherche* qui va lui demander d'attendre le temps que la page s'affiche avec le résultat.

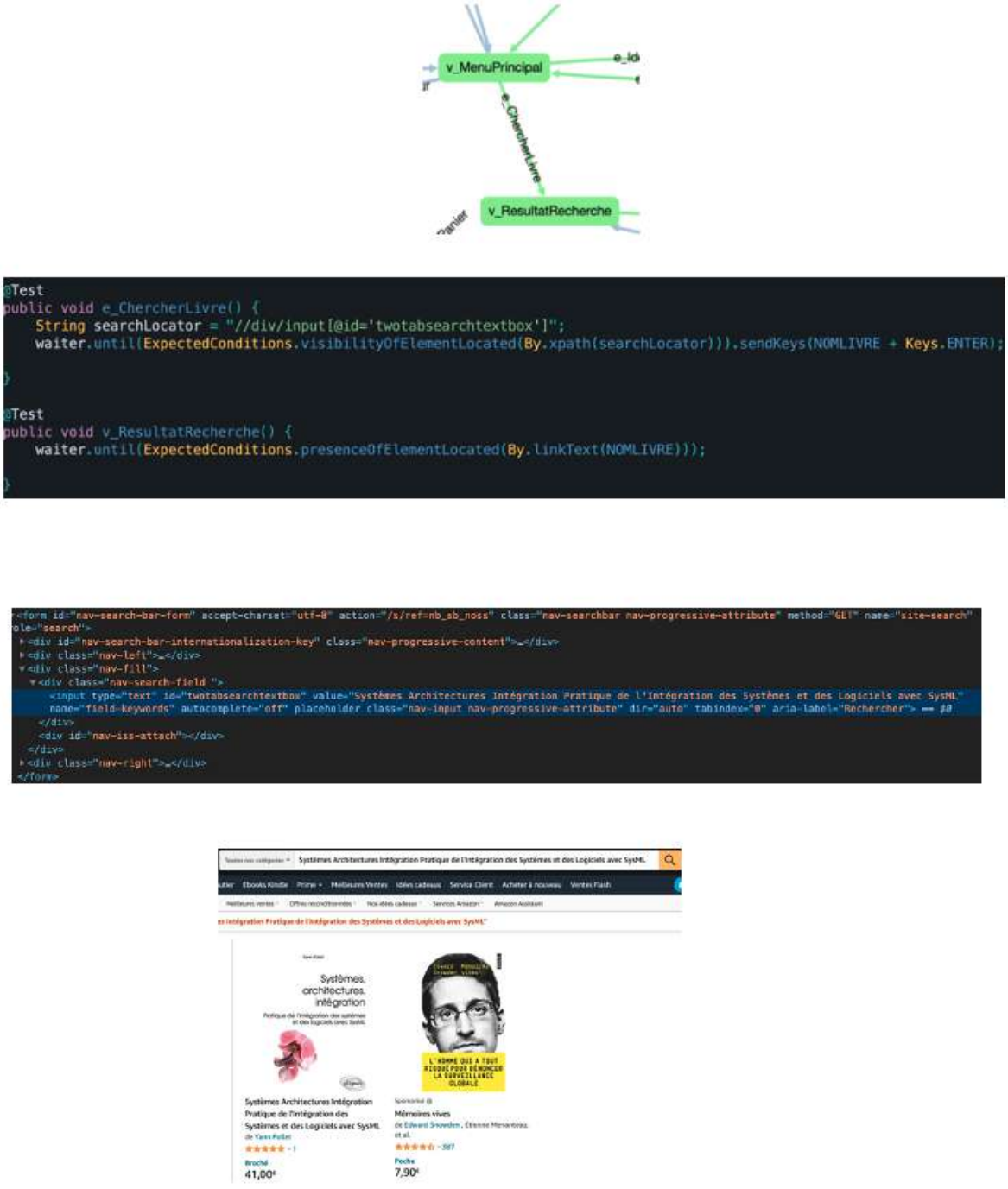

FIGURE 3.17 – Illustration d'un processus de test.

3.4.4 Composant : Publicateur de résultat

Lorsque l'étape d'exécution se termine, l'outil TestNG produit un rapport de tests. Ce rapport peut être consulté à l'aide du navigateur. Le rapport de test, comme le montre l'illustration ci-dessous, montre qu'il est composé de différentes parties :

- Une partie *Info* qui renseigne sur le nombre de cas de tests que l'outil a effectué mais également sur le temps d'exécution de chaque méthode de test. Nous avons également la possibilité de voir les méthodes de tests qui auraient pu être ignorées.
- Une partie *Results* qui renseigne sur le nombre de méthodes de tests total avec le nombre des méthodes de tests qui ont réussi leurs exécutions et dans le cas où il y aurait une présence d'erreur, nous en serions également informés.
- Une partie qui permet d'afficher les différentes informations complémentaires que nous souhaitons obtenir.

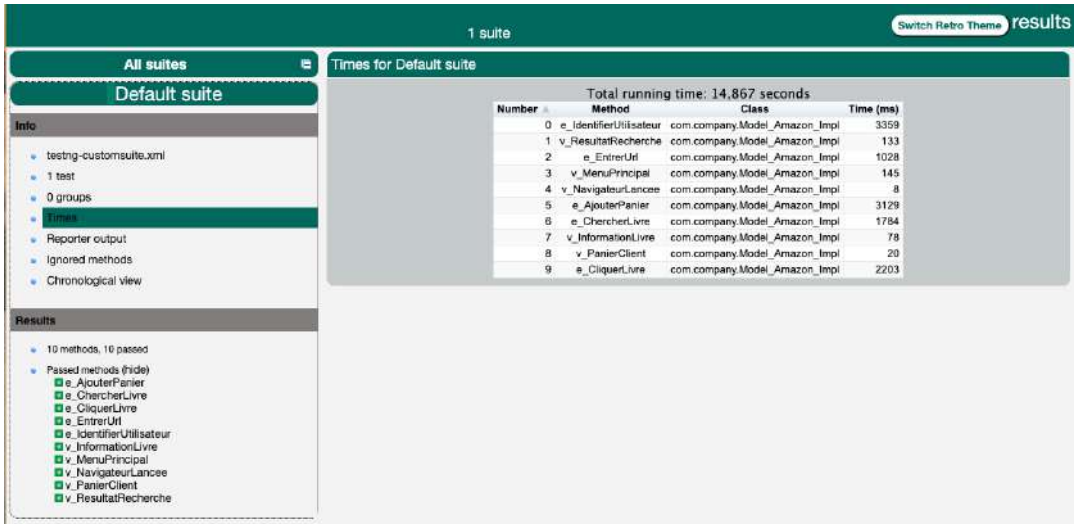

FIGURE 3.18 – Illustration du rapport de test généré.

Afin d'avoir une précision supplémentaire, nous pouvons observer, grâce à l'illustration ci dessous, le statut des méthodes qui ont été appelées durant l'exécution du cas de tests. Ces différentes méthodes de tests possèdent deux états matérialisés par deux icons qui sont : *success check icon* pour montrer le résultat de la méthode réussie et *failure check icon* pour montrer le résultat de la méthode qui a échouée. Toutefois, par défaut, elles sont classées par ordre alphabétique et non par ordre d'exécution.

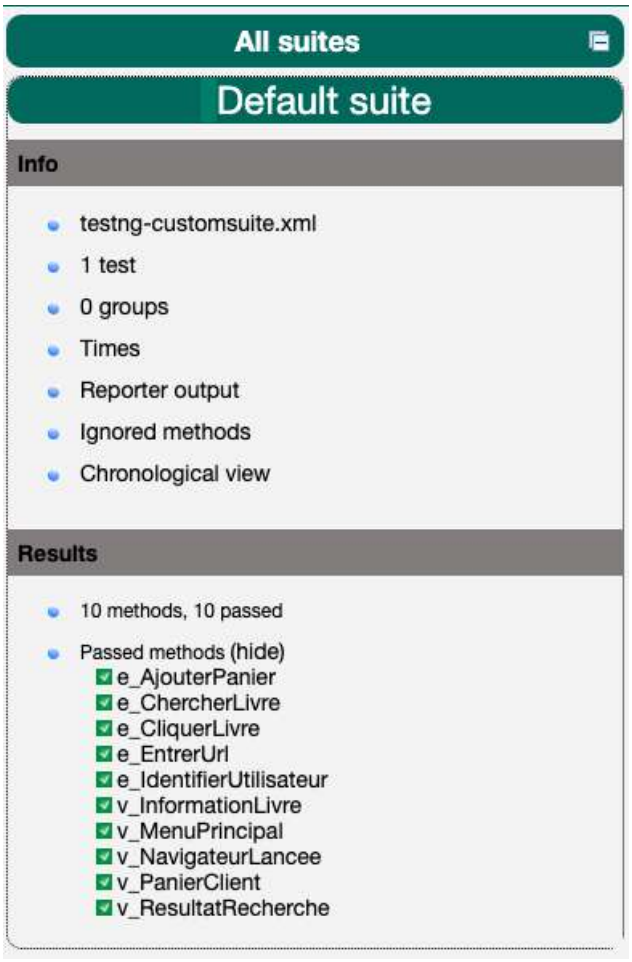

FIGURE 3.19 – Illustration des statuts des différents méthodes de tests.

En effet, le tableau ci-dessous nous montre les différentes méthodes et leurs temps d'exécution correspondants. Si des valeurs de temps sont très différentes, cela s'explique par le fait que les méthodes préfixées par un "e", *edge*, correspondent aux différentes actions à exécuter dans une même méthode, elles sont plus longues à effectuer que les méthodes préfixées par un "v", *vertex*, correspondant à des vérifications d'actions.

Total running time: 14,867 seconds

Number	Method	Class	Time (ms)
0	e_IdentifierUtilisateur	com.company.Model_Amazon_Impl	3359
1	v_ResultatRecherche	com.company.Model_Amazon_Impl	133
2	e_EntrerUrl	com.company.Model_Amazon_Impl	1028
3	v_MenuPrincipal	com.company.Model_Amazon_Impl	145
4	v_NavigateurLancee	com.company.Model_Amazon_Impl	8
5	e_AjouterPanier	com.company.Model_Amazon_Impl	3129
6	e_ChercherLivre	com.company.Model_Amazon_Impl	1784
7	v_InformationLivre	com.company.Model_Amazon_Impl	78
8	v_PanierClient	com.company.Model_Amazon_Impl	20
9	e_CliquerLivre	com.company.Model_Amazon_Impl	2203

TABLE III.5 – Tableau de la durée du cas de tests.

Après avoir vu dans le premier chapitre le contexte industriel, puis dans la seconde partie la génération automatique des tests dirigée par les modèles, dans la troisième partie la mise en oeuvre de la solution, la logique voudrait que l'on passe à la partie évaluation. Ceci sera abordé dans le chapitre suivant.

Chapitre IV

Évaluation du système

Nous rappelons que la solution qui a été proposée était destinée au système *Airline 1*, mais du fait de la situation sanitaire mondiale dûe au virus COVID-19, le projet *Airline 1* a dû être repris par le client. Ce revirement de situation a fait qu'il était impossible de pouvoir illustrer la méthodologie de la solution sur le cas d'étude *Airline 1* étant donné qu'il n'y avait plus d'accès possible à l'application. Cependant, afin de pouvoir illustrer la méthodologie du Model-Based Testing dans ce mémoire, il a été décidé d'un commun accord avec mon tuteur du mémoire Monsieur Lotfi CHAARI ainsi que mon tuteur d'entreprise Monsieur Laurent THULL d'utiliser la méthodologie expliquée dans le chapitre II sur une autre application web, comme celle *d'Amazon*.

Dans la section précédente, nous avons mis en application la génération de scripts de tests d'acceptance dirigée par des modèles. Pour la mise en oeuvre de cette solution nous avons fait un méta-modèle permettant de représenter l'environnement de l'application à tester. Par la suite, à partir de ce méta-modèle nous avons créé des modèles sous-tests permettant de tester l'ensemble des fonctionnalités du système cible. De plus, ce modèle sous test a généré des squelettes de méthodes abstraites de tests. Ensuite nous avons implémenté les différentes méthodes de tests par différents scripts pour rendre l'exécution et la publication d'un rapport de tests de manière automatique.

Avoir choisi l'application web *Amazon* montre bien que la solution est applicable à toutes sortes d'application web. Cette solution, répond effectivement à la problématique énoncée, puisque la génération et l'exécution de tests automatiques afin d'éviter la régression dans le système

permet également un gain de temps et par la même un gain financier.

Le système étant opérationnel, nous allons pouvoir évaluer l'ensemble pour savoir si la solution proposée est vraiment pertinente dans son environnement.

4.1 Pertinence de la solution

Dans cette section, nous allons pouvoir vérifier que la solution proposée est pertinente et répond bien aux préoccupations annoncées et aux exigences énumérées avec un protocole d'évaluation de tests.

4.1.1 Rappels de la problématique et des préoccupations

Afin de vérifier la pertinence de la solution apportée, nous rappelons la problématique :

Comment générer et exécuter des tests automatiquement afin d'éviter la régression du système existant, vue la grande quantité de tests à effectuer, tout en limitant les coûts ?

Ci-dessous, nous pouvons trouver le Tableau IV.1 qui rappelle les principaux problèmes initiaux et les réponses à ceux-ci apportées par la solution mise en oeuvre.

Problèmes observés	Apport de la solution
La qualité	La solution a permis d'avoir une meilleure couverture de test.
Les délais	La solution a réduit le délai engendré par les différents tests de non-régression.

TABLE IV.1 – Tableau des problèmes observés.

En effet, nous pouvons voir qu'au niveau de la qualité, le procédé du Model-Based Testing se caractérise par :

- La qualité syntaxique : Sur le langage utilisé ainsi que les règles employées.
- La qualité sémantique : Le modèle décrit correctement des tests exécutables par un automate de tests.
- La qualité pragmatique : Une fois le modèle réalisé, les étapes suivantes sont automatisées.

En ce qui concerne les délais, le gain de temps est quantifiable et notable.

4.1.2 Protocole d'évaluation de la solution

Pour pouvoir évaluer au mieux la solution proposée, nous avons fait un récapitulatif des finalités recherchées ainsi que les exigences que devaient être prise en compte par cette même solution. Nous pouvons voir dans les deux tableaux, Tableau IV.2 et Tableau IV.3, d'une part les finalités recherchées sont toutes conformes et que toutes les exigences ont été conformes.

Finalité recherchée	Conforme
Améliorer la fiabilité et la qualité des tests.	Oui
Augmenter la productivité.	Oui
Alléger la charge de travail par la réutilisabilité.	Oui
Augmenter le niveau de complétude des tests.	Oui
Automatiser les tests.	Oui
Améliorer la communication entre les différentes parties prenantes.	Oui

TABLE IV.2 – Tableau de la finalité recherchée.

Exigences de la solution	Conforme
Élaboration d'un modèle à partir des spécifications du test.	Oui
Génération de scénarios de tests par machine d'états finis.	Oui
Exécution des scénarios par un automate de test.	Oui
Rédaction automatique d'un rapport de résultats	Oui

TABLE IV.3 – Tableau des exigences de la solution.

4.2 Bilan de la solution apportée

Pour pouvoir réaliser un bilan de la solution apportée avec Model-Based Testing, nous verrons d'abord le comparatif de temps d'exécution des tests réalisés manuellement et nous les avons comparés aux tests d'exécution réalisés par un automate de tests.

4.2.1 Tests produits

Pour arriver à un bilan de la solution proposée, nous avons pu quantifier le gain de temps en comparant le temps nécessaire pour effectuer les tests manuellement et en les opposants au temps utilisé par les tests automatiques grâce au Model-Based Testing. En effet, le gain de temps apparait clairement dans le tableau ci-dessous.

Intitulé du test	Test manuel (En secondes)	Test automatique (En secondes)
Ajouter un ouvrage au panier	45,20	14,747
Information sur l'ouvrage	34,24	13,174
Résultat de la recherche	33,48	8,682
Accès à l'application	2,82	3,487
Identification utilisateur	26,04	7,050
Vérification liste commandes	28,34	10,392
Vérification profil utilisateur	27,59	8,109
Vérification informations paiements	33,11	8,873
Vérification informations adresses	30,78	9,106
Création d'un compte client	40,74	5,601
TOTAL	**302,34**	**89,221**

TABLE IV.4 – Tableau comparatif des exécutions des tests.

Ce tableau permet de mettre en évidence le gain de temps notable obtenu grâce à l'exécution des tests avec les automates de test du Model-Based Testing. Après avoir pris en compte les résultats, les tests en Model-Based Testing représentent 29,50 % du temps nécessaire pour les mêmes tests effectués manuellement. Le gain de temps est donc de 70,50 %.

Même si l'élaboration du modèle est chronophage, sa qualité réutilisable compense le temps consacré à cette première modélisation, de plus les étapes suivantes sont automatiques.

Il en résulte que plus on utilise le modèle effectué, plus le système devient rentable. Du fait de la précision du modèle abstrait et de l'automatisme des tests, la garantie est assurée par la large couverture des tests.

4.2.2 Limites et perspectives de la solution

Les attentes et bénéfices du Model-Based Testing ne doivent pas occulter ses limites. En effet, si MBT met en avant l'automatisation, il n'est cependant pas la réponse à tous les problèmes. Il faut une bonne maîtrise dans des techniques de modélisation pour la conception du modèle qui permettra l'automatisation des différentes étapes de tests. Model-Based Testing est davantage préconisé dans des campagnes de tests sur des systèmes conséquents [Chr+15].

Personnellement, ma principale préoccupation aura été d'utiliser uniquement des outils *open-source* qui ne sont pas toujours fiables à 100 % surtout en ce qui concerne le générateur. D'ailleurs, nombre de conférences et de thèses pourtant sur ce sujet convergent sur le fait que

le générateur est l'outil le plus "critique" et en concluent qu'il est plus intéressant d'utiliser un générateur payant.

De plus, les rapports de tests générés à la fin du processus sont actuellement stockés dans un répertoire qui est accessible pour consultation. Cependant toute analyse ultérieure des résultats est encore faite manuellement.

Il faut quand même reconnaître qu'il faut un niveau de connaissances élevé pour la création du méta-modèle, vu qu'il permet la création du modèle sous-test. Ce modèle doit être aussi précis que possible pour aboutir à une exécution fiable.

Concernant les perspectives, sur le dernier point des limites, il serait intéressant d'automatiser l'interprétation des résultats par une base de données et du *machine learning* ce qui permettrait de réaliser des statistiques et des projections.

Enfin une autre perspective serait de combiner *Model-Based Testing* avec *Behavior-Driven Development* pour pouvoir profiter des avantages de chacune des deux méthodes : MBT pour gérer la complexité et la communication et BDD pour l'utilisation d'un DSL afin d'avoir un niveau d'abstraction plus élevé sur les actions de tests avec notre propre DSL métier, compréhensible par toutes les parties prenantes.

Conclusion

Ce mémoire a présenté une solution à la problématique de la génération automatique de tests d'acceptance pour une application web par une approche de tests dirigée par les modèles. Initialement prévu pour le système *Airline 1*, nous avons pu l'adapter à une autre application web ce qui a permis de mettre en évidence que cette solution est adaptable à toute application web. Le modèle élaboré dans le chapitre III sous la forme d'une machine d'états finis, a l'avantage d'être compréhensible par l'ensemble des parties prenantes, dans notre cas un projet sur le e-commerce. Bien entendu, pour d'autres types de projet web, d'autres modèles sous forme de machine d'états finis peuvent être créés.

L'élaboration de ce modèle abstrait est certes chronophage, il est cependant réutilisable et les autres étapes du procédé de *Model-Based Testing* étant automatiques, il en résulte que le gain de temps de l'ensemble est notable ainsi que quantifiable. De plus, le taux d'automatisation rend ce processus fiable.

Comme nous l'avons vu au début de ce mémoire, dans le monde industriel les deux préoccupations majeures sont : la qualité et la réduction des coûts. Le procédé de notre solution, avec *Model-Based Testing* combiné à l'utilisation d'une machine d'états finis, permet de répondre à ces deux préoccupations étant donné que l'automatisation est garante de la couverture des tests, qualité, et le gain de temps constaté se traduit par la réduction des coûts.

Bibliographie

[Abd16] Nezha Nesrine ABDELLAOUI. « Application de l'approche d'ingénierie dirigée par les modèles pour le développement des applications mobiles (Interfaces graphiques). » French. PhD Thesis. Tlemcen : Université Abou Bakr Belkaid– Tlemcen, 2016.

[Adz11] Gojko ADZIC. *Specification by example: how successful teams deliver the right software*. English. Shelter Island, N.Y : Manning, 2011. ISBN : 978-1-61729-008-4.

[All] Patrick ALLGEYER. *Le MBT, Model Based Testing*. French. Site institutionnel.

[AM12] ARMIN BEER et Stefan MOHACSI. « How to make the most of Model-Based Testing ». English. In : (mar. 2012), p. 12.

[And+99] Marc ANDRIES et al. « Graph transformation for specification and programming ». In : *Science of Computer programming* 34.1 (1999). ISBN: 0167-6423 Publisher: Elsevier, p. 1-54.

[And04] Sylvain ANDRE. « MDA (model driven architecture) principes et états de l'art ». French. Probatoire. Lyon : Conservatoire National des Arts et Métiers centre d'enseignement de Lyon, nov. 2004.

[Aud14] Laurent AUDIBERT. *UML 2: de l'apprentissage à la pratique : présentation des diagrammes UML, cas d'utilisation ...* French. Paris : Ellipses, 2014. ISBN : 978-2-340-00204-3.

[BA05] Kent BECK et Cynthia ANDRES. *Extreme programming explained: embrace change*. English. 2nd ed. Boston, MA : Addison-Wesley, 2005. ISBN : 978-0-321-27865-4.

[BB02] Jean BÉZIVIN et Xavier BLANC. « MDA: Vers un important changement de paradigme en génie logiciel ». French. In : *Développeur référence v2* 16 (2002), p. 15.

[Bec03] Kent BECK. *Test-driven development: by example*. English. The Addison-Wesley signature series. Boston : Addison-Wesley, 2003. ISBN : 978-0-321-14653-3.

[Ben86] Jon BENTLEY. « Programming pearls: little languages ». English. In : *Communications of the ACM* 29.8 (1986). ISBN: 0001-0782 Publisher: ACM New York, NY, USA, p. 711-721.

[BOO17] GRADY BOOCH. *Unified modeling language user guide.* English. Place of publication not identified : ADDISON-WESLEY, 2017. ISBN : 978-0-13-485215-7.

[Bro05] M. BROY, éd. *Model-based testing of reactive systems: advanced lectures*. English. Lecture notes in computer science 3472. Berlin ; New York : Springer, 2005. ISBN : 978-3-540-26278-7.

[BS05] Xavier BLANC et Olivier SALVATORI. *MDA en action: ingénierie logicielle guidée par les modèles*. French. Paris : Eyrolles, 2005. ISBN : 978-2-212-11539-0.

[CA05] Krzysztof CZARNECKI et Michał ANTKIEWICZ. « Mapping features to models: A template approach based on superimposed variants ». In : *International conference on generative programming and component engineering*. Springer, 2005, p. 422-437.

[Cab13] Kalou CABRERA CASTILLOS. « Generation automatique de scenarios de tests à partir de proprietes temporelles et de modèles comportementaux ». French. PhD Thesis. 2013.

[CH03] Krzysztof CZARNECKI et Simon HELSEN. « Classification of model transformation approaches ». In : *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. T. 45. Issue: 3. USA, 2003, p. 1-17.

[Cha16] CHANDRASEKARA. *Acceptance Test Driven Development ATDD*. English. Course. Sept. 2016.

[Cha17] Marc Hage CHAHINE. *Le testeur dans la méthode Scrum*. French. Nov. 2017.

[Cha19] Marc Hage CHAHINE. *Quelle différence entre types et niveaux de test ?* French. Sept. 2019.

[Che10] David CHELIMSKY, éd. *The RSpec book: behaviour-driven development with RSpec, Cucumber, and Friends*. English. Lewisville, Tex : Pragmatic, 2010. ISBN : 978-1-934356-37-1.

[Cho78] T.S. CHOW. « Testing Software Design Modeled by Finite-State Machines ». In : *IEEE Transactions on Software Engineering* SE-4.3 (mai 1978), p. 178-187. ISSN : 1939-3520.

[Chr+15] Stephan CHRISTMANN et al. *ISTQB® Testeur Certifié Niveau Fondation Model-Based Testing Syllabus*. French. Comité Français des Tests Logiciels, 2015.

[Coh04] Mike COHN. *User stories applied: for agile software development*. English. Addison-Wesley signature series. Boston : Addison-Wesley, 2004. ISBN : 978-0-321-20568-1.

[Com08] Benoît COMBEMALE. « Approche de métamodélisation pour la simulation et la vérification de modèle ». French. Thesis. Toulouse : ENSEEIHT, 2008.

[Com18] COMITÉ FRANÇAIS DES TESTS LOGICIELS. *Testeur Certifié Syllabus Niveau Fondation*. French. Août 2018.

[Cun08] H. Conrad CUNNINGHAM. « A little language for surveys: constructing an internal DSL in Ruby ». English. In : *Proceedings of the 46th Annual Southeast Regional Conference on XX*. 2008, p. 282-287.

[Dut19] Guy DUTHEIL. « Boeing : six questions sur la crise ouverte par les deux crashes du 737 MAX ». French. In : *Le Monde.fr* (mar. 2019).

[DV16] Laurent DEBRAUWER et Fien VAN DER HEYDE. *UML 2.5: initiation, exemples et exercices corrigés*. French. St Herblain : Éditions ENI, 2016. ISBN : 978-2-409-00100-0.

[Eng05] Clearsy System ENGINEERING. *Méthode B - Formation niveau 1*. French. présentation. Juin 2005.

[Far16] FARLEX INTERNATIONAL. *The Farlex grammar book. examples, exceptions, exercises, and everything you need to master proper grammar. Volume 1, Volume 1,* English. 2016. ISBN : 978-1-5353-9920-3.

[Fle06] Franck FLEUREY. « Langage et méthode pour une ingénierie des modèles fiable ». French. Thesis. Rennes : Université de Rennes 1, 2006.

[Fou12] Elizabeta FOURNERET. « Génération de tests à partir de modèle UML/OCL pour les systèmes critiques évolutifs ». French. PhD Thesis. Université de Franche-Comté, 2012.

[Fou18] Gabin FOURCAULT. « Création d'un générateur de code générique, dirigé par les modèles, permettant la génération de vues, afin de représenter des scénarii de configuration, mettant en oeuvre un sous-système antenne. » French. Mém. de mast. Toulouse : Conservatoire National des Arts et Métiers centre d'enseignement de Toulouse, oct. 2018.

[Fow14] Martin FOWLER. *Bounded Context*. English. Jan. 2014.

[FP11] Martin FOWLER et Rebecca PARSONS. *Domain-specific languages*. English. Upper Saddle River, NJ : Addison-Wesley, 2011. ISBN : 978-0-321-71294-3.

[Fre08] Ulrich FREUND. « Mulit-level system integration based on AUTOSAR ». English. In : *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 2008, p. 581-582. ISBN : 1-4244-4486-1.

[Gar13] Stéphane GARREDU. « Approche de méta-modélisation et transformations de modèles dans le contexte de la modélisation et simulation à évènements discrets: application au formalisme DEVS ». French. Thesis. Corse : Université de Corse-Pascal Paoli, 2013.

[GHL16] Matthias GEIGER, Simon HARRER et Jörg LENHARD. « Process Engine Benchmarking with Betsy in the Context of ISO/IEC Quality Standards ». English. In : *Softwaretechnik-Trends (STT)* 36.2 (2016), p. 57-60.

[Gri+02] Wolfgang GRIESKAMP et al. « Generating finite state machines from abstract state machines ». English. In : *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 2002, p. 112-122.

[Gro] Object Management GROUP. *Model Driven Architecture (MDA) | Object Management Group*. English.

[GS93] David GARLAN et Mary SHAW. « An introduction to software architecture ». English. In : *Advances in software engineering and knowledge engineering*. World Scientific, 1993, p. 1-39.

[GW04] Boby GEORGE et Laurie WILLIAMS. « A structured experiment of test-driven development ». English. In : *Information and software Technology* 46.5 (2004). ISBN: 0950-5849 Publisher: Elsevier, p. 337-342.

[Hag19] Marc HAGE CHAHINE. *Tout sur le test logiciel: préparation à la certification ISTQB : profession testeur*. French. 2019. ISBN : 978-2-340-03021-3.

[Ham12] Samiya HAMADOUCHE. « Etude de la sécurité d'un vérifieur de byte code et génération de tests de vulnérabilité ». French. PhD Thesis. Boumerdès, Algérie : Université M'Hamed Bougara, 2012.

[HT06] B. HAILPERN et P. TARR. « Model-driven development: The good, the bad, and the ugly ». English. In : *IBM Systems Journal* 45.3 (2006), p. 451-461. ISSN : 0018-8670.

[JCV12] Jean-Marc JÉZÉQUEL, Benoît COMBEMALE et Didier VOJTISEK. *Ingénierie dirigée par les modèles des concepts à la pratique*. French. Paris : Ellipses, 2012. ISBN : 978-2-7298-7196-3.

[JM07] Ron JEFFRIES et Grigori MELNIK. « Guest Editors' Introduction: TDD–The Art of Fearless Programming ». English. In : *Ieee Software* 24.3 (2007). ISBN: 0740-7459 Publisher: IEEE, p. 24-30.

[JS05] David JANZEN et Hossein SAIEDIAN. « Test-driven development concepts, taxonomy, and future direction ». English. In : *Computer* 38.9 (2005). ISBN: 0018-9162 Publisher: IEEE, p. 43-50.

[JS08] David JANZEN et Hossein SAIEDIAN. « Does test-driven development really improve software design quality? » English. In : *Ieee Software* 25.2 (2008). ISBN: 0740-7459 Publisher: IEEE, p. 77-84.

[KA17] Md Rezaul KARIM et Sridhar ALLA. *Scala and Spark for big data analytics: tame big data with Scala and Apache Spark!* English. Birmingham, UK : Packt Publishing, 2017. ISBN : 978-1-78528-084-9 978-1-78355-050-0.

[Kan+90] Kyo C. KANG et al. *Feature-oriented domain analysis (FODA) feasibility study*. Rapp. tech. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[KL16] Anne KRAMER et Bruno LEGEARD. *Model-based testing essentials-guide to the ISTQB certified model-based tester: foundation level*. English. John Wiley & Sons, 2016. ISBN : 1-119-13001-8.

[Kos07] Lasse KOSKELA. *Test driven: practical tdd and acceptance tdd for java developers*. English. Manning Publications Co., 2007. ISBN : 1-932394-85-0.

[Kos08] Lasse KOSKELA. *Test driven: TDD and Acceptance TDD for Java developers*. English. Greenwich, CT : Manning, 2008. ISBN : 978-1-932394-85-6.

[Kri19] Karl KRISTIAN. *Creating a model using yEd*. English. dépôt git. Sept. 2019.

[KWB03] Anneke G. KLEPPE, Jos B. WARMER et Wim BAST. *MDA explained: the model driven architecture: practice and promise*. English. The Addison-Wesley object technology series. Boston : Addison-Wesley, 2003. ISBN : 978-0-321-19442-8.

[Lac17] Mohamed LACHGAR. « Approche MDA pour automatiser la génération de code natif pour les applications mobiles multiplateformes ». French. Thesis. Marrakech : Faculté des Sciences et Techniques Marrakech, 2017.

[Leg15] Bruno LEGEARD. *Model-Based Testing dans l'industrie usage et dissémination*. French. Diaporama séminaire. Juin 2015.

[LKT04] Janne LUOMA, Steven KELLY et Juha-Pekka TOLVANEN. « Defining domain-specific modeling languages: Collected experiences ». English. In : *4 th Workshop on Domain-Specific Modeling*. 2004.

[LS05] Michael LAWLEY et Jim STEEL. « Practical declarative model transformation with Tefkat ». In : *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2005, p. 139-150.

[Mei+18] Gijs van der MEIJDE et al. « A Practical Application of Model Learning and Code Generation ». English. Mém. de mast. Nijmegen, The Netherlands : Radboud Universiteit, août 2018.

[Mon15] J. R. MONSMA. « Model-Based Testing of web applications ». English. Mém. de mast. The Netherlands : Radboud University, mai 2015.

[MV06] Tom MENS et Pieter VAN GORP. « A taxonomy of model transformation ». English. In : *Electronic notes in theoretical computer science* 152 (2006). ISBN: 1571-0661 Publisher: Elsevier, p. 125-142.

[Nei84] James M. NEIGHBORS. « The Draco approach to constructing software from reusable components ». English. In : *IEEE Transactions on Software Engineering* 5 (1984). ISBN: 0098-5589 Publisher: IEEE, p. 564-574.

[Nur+09] Nurzhan NURSEITOV et al. « Comparison of JSON and XML data interchange formats: a case study. » English. In : *Caine* 9 (2009), p. 157-162.

[Org14] ORGANISATION INTERNATIONALE DE NORMALISATION. *Exigences de qualité pour le logiciel et son évaluation (SQuaRE)*. French. ISO/IEC 25051:2014. Fév. 2014. (Visité le 21/08/2020).

[Org17] ORGANISATION INTERNATIONALE DE NORMALISATION. *Exigences de qualité et évaluation des systèmes et du logiciel (SQuaRE)*. French. ISO/IEC 25010:2011. 2017.

[Per15] Christian PERCEBOIS. *Conception par objets*. French. Cours IUT Informatique. IUT Informatique Toulouse III, 2015.

[PES00] Simon PEYTON JONES, Jean-Marc EBER et Julian SEWARD. « Composing contracts: an adventure in financial engineering (functional pearl) ». English. In : *ACM SIGPLAN Notices* 35.9 (2000). ISBN: 0362-1340 Publisher: ACM New York, NY, USA, p. 280-292.

[Pou12] Philippe POUMAROUX. *Introduction au Behaviour Driven Developement - Poum*. French. Fév. 2012.

[RR18] Pascal ROQUES et Gaël RENAULT. *UML 2.5 par la pratique: études de cas et exercices corrigés*. French. 2018. ISBN : 978-2-212-67565-8.

[Sch13] Gerardo SCHNEIDER. *Model-Based Testing*. English. Course. Dept. of Computer Science et Engineering Chalmers | University of Gothenburg, 2013.

[SK03] Shane SENDALL et Wojtek KOZACZYNSKI. « Model transformation: The heart and soul of model-driven software development ». English. In : *IEEE software* 20.5 (2003). ISBN: 0740-7459 Publisher: IEEE, p. 42-45.

[Sma15] John Ferguson SMART. *BDD in action: Behavior-Driven Development for the whole software lifecycle*. English. Shelter Island, NY : Manning Publications, 2015. ISBN : 978-1-61729-165-4.

[Sona] SONOO JAISWAL. *Selenium WebDriver Tutorial - javatpoint*. English.

[Sonb] SONOO JAISWAL. *TestNG Tutorial - Javatpoint*. English.

[Tou07] Jihed TOUZI. « Aide à la conception de système d'information collaboratif, support de l'interopérabilité des entreprises ». French. Thesis. Albi Carmaux : École des Mines, 2007.

[UL07] Mark UTTING et Bruno LEGEARD. *Practical model-based testing: a tools approach*. English. Amsterdam ; Boston : Morgan Kaufmann Publishers, 2007. ISBN : 978-0-12-372501-1.

[UPL12] Mark UTTING, Alexander PRETSCHNER et Bruno LEGEARD. « A taxonomy of model-based testing approaches ». English. In : *Software testing, verification and reliability* 22.5 (2012). ISBN: 0960-0833 Publisher: Wiley Online Library, p. 297-312.

[Vai17] Thierry VAIRA. *Cours Qt Automate fini ou Machine à états*. French. Cours. IUT Arles, mai 2017.

[Vey18] Pierre VEYRAT. *Exemples de BPMN : comprendre la signification de 20 symboles*. French. Publication Title: HEFLO FR. Oct. 2018.

[Viv18] Fabing VIVIEN. *Selenium Driver : Derrière la scène !* French. Déc. 2018.

[VJ04] Didier VOJTISEK et Jean-Marc JÉZÉQUEL. « MTL and Umlaut NG-Engine and framework for model transformation ». In : (2004).

[Völ10] Markus VÖLTER. *DSL engineering: designing, implementing and using domain-specific languages*. English. Lexington, KY : CreateSpace Independent Publishing Platform, 2010. ISBN : 978-1-4812-1858-0.

Table des figures

Liste des tableaux

Annexes

Annexe I - Carte d'environnement du système *Airline 1*

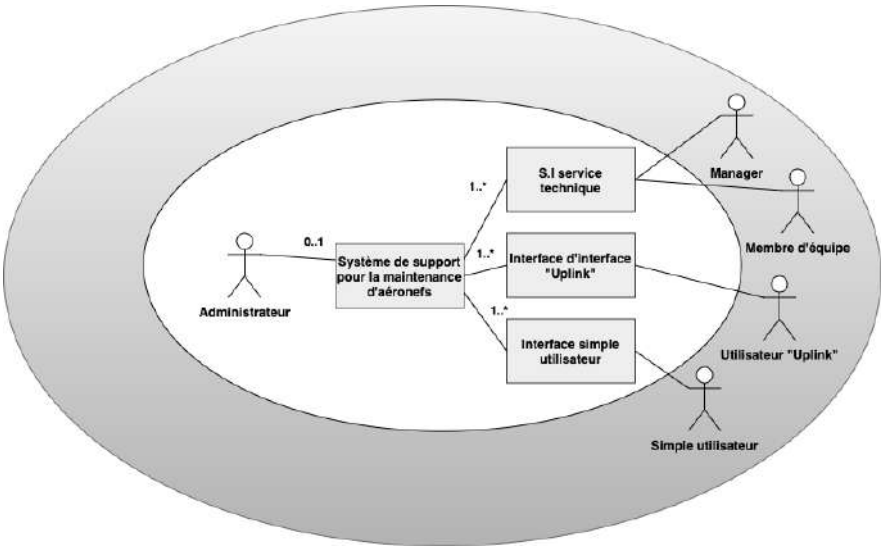

FIGURE 1 – Carte d'environnement du système Airline 1.

Annexe II - Cas d'utilisations d'Airline 1

ID	UC1.
Cas d'utilisation	Manager les équipes.
Acteurs	Administrateur.
Scénario nominal	L'administrateur peut manager les équipes en accédant à son interface dédiée. Par la suite il peut accéder à un sous-menu permettant de voir la liste des membres de son équipe et de leur envoyer des messages.
Flux	L'administrateur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 5 – Tableau de description textuelle UC1 *d'Airline 1*.

ID	UC2.
Cas d'utilisation	Créer des vols.
Acteurs	Administrateur.
Scénario nominal	L'administrateur peut gérer des vols et ainsi créer des vols en accédant à son interface dédiée. Par la suite il peut créer un vol en accédant à un sous-menu. Une fois arrivé dans ce sous-menu l'administrateur rentre dans le système l'indicatif de l'aéroport de départ et d'arrivée. Mais également l'indicatif de l'appareil et la date du vol.
Flux	L'administrateur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 6 – Tableau de description textuelle UC2 *d'Airline 1*.

ID	UC3.
Cas d'utilisation	Modifier le statut des vols
Acteurs	Administrateur
Scénario nominal	L'administrateur peut gérer des vols et ainsi modifier les informations des vols en accédant à son interface dédiée.
Flux	L'administrateur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 7 – Tableau de description textuelle UC3 *d'Airline 1*.

ID	UC4.
Cas d'utilisation	Supprimer des vols.
Acteurs	Administrateur.
Scénario nominal	L'administrateur peut gérer des vols et ainsi supprimer des vols en accédant à son interface dédiée.
Flux	L'administrateur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 8 – Tableau de description textuelle UC4 *d'Airline 1*.

ID	UC5.
Cas d'utilisation	Gérer les modèles graphiques.
Acteurs	Administrateur.
Scénario nominal	L'administrateur peut gérer les modèles graphiques en modifiant les droits de ceux qui peuvent voir les différentes interfaces, modifier leurs tailles et éventuellement les supprimer.
Flux	L'administrateur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 9 – Tableau de description textuelle UC5 *d'Airline 1*.

ID	UC6.
Cas d'utilisation	Gérer les robots.
Acteurs	Administrateur.
Scénario nominal	L'administrateur peut gérer utilisation des robots pour simuler le comportement d'un vol en accédant à son interface dédiée.
Flux	L'administrateur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 10 – Tableau de description textuelle UC6 *d'Airline 1*.

ID	UC7.
Cas d'utilisation	Activer/désactiver la prise en compte de la météo.
Acteurs	Administrateur.
Scénario nominal	L'administrateur peut activer/désactiver la prise en compte de la météo grâce à des informations partagées par Météo France. Il pourra ainsi activer/désactiver cette fonctionnalité pour les différents rôles qui utilisent cette application.
Flux	L'administrateur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 11 – Tableau de description textuelle UC7 *d'Airline 1*.

ID	UC8.
Cas d'utilisation	Accéder à l'interface dédiée.
Acteurs	Administrateur, Manager, Utilisateur Uplink, Membre d'équipe et simple utilisateur.
Scénario nominal	Chaque utilisateur accède à une interface dédiée via une URL propre au rôle.
Flux	L'acteur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 12 – Tableau de description textuelle UC8 *d'Airline 1*.

ID	UC9.
Cas d'utilisation	Envoyer des UPLINK messages.
Acteurs	Utilisateur Uplink.
Scénario nominal	L'utilisateur Uplink accède à l'interface « Uplink request » dans lequel il peut choisir quelle est la nature du message (ACMS, CMS, AOC, FMS et CC Mail) et qui sera envoyé à l'appareil préalablement sélectionné.
Flux	L'utilisateur Uplink peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 13 – Tableau de description textuelle UC9 *d'Airline 1*.

ID	UC10.
Cas d'utilisation	Éditer les vols.
Acteurs	Manager.
Scénario nominal	Le manager peut éditer les vols via l'interface qui lui est dédiée, il faut pour cela qu'il se rende vers la liste des vols, sans toutefois pouvoir les supprimer.
Flux	Le manager peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 14 – Tableau de description textuelle UC10 *d'Airline 1*.

ID	UC11.
Cas d'utilisation	Déplacer les vols.
Acteurs	Manager.
Scénario nominal	Le manager peut préparer l'exportation des données de vol.
Flux	Le manager peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 15 – Tableau de description textuelle UC11 *d'Airline 1*.

ID	UC12.
Cas d'utilisation	Transférer les vols.
Acteurs	Manager.
Scénario nominal	Le manager peut transférer les données de vols aux différentes équipes dont il a la charge.
Flux	Le manager peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 16 – Tableau de description textuelle UC12 *d'Airline 1*.

ID	UC13.
Cas d'utilisation	Gérer l'historique des vols.
Acteurs	Manager.
Scénario nominal	Le manager peut consulter l'historique des vols. Pour cela il rentrera, dans le système via une interface dédiée, une plage de dates et ainsi le système affichera la liste des vols prévus dans cette plage.
Flux	Le manager peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 17 – Tableau de description textuelle UC13 *d'Airline 1*.

ID	UC14.
Cas d'utilisation	Importer les vols.
Acteurs	Manager.
Scénario nominal	Le manager peut importer des vols, pour cela il intègre le fichier qu'il souhaite importer dans le dossier ELECTRONIC FLIGHT FOLDER. Par la suite le système rajoutera une nouvelle ligne dans laquelle apparaitra la nouvelle ligne de vol.
Flux	Le manager peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 18 – Tableau de description textuelle UC14 *d'Airline 1*.

ID	UC15.
Cas d'utilisation	Exporter les données de vols.
Acteurs	Membre d'équipe.
Scénario nominal	Chaque membre d'équipe accèdera au système via une interface dédiée. Ils pourront ainsi exporter les différentes données de vols auxquelles le manager leur permettra d'avoir accès.
Flux	Chaque membre peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 19 – Tableau de description textuelle UC15 *d'Airline 1*.

ID	UC16.
Cas d'utilisation	Consulter les vols.
Acteurs	Simple utilisateur.
Scénario nominal	Un simple utilisateur peut uniquement consulter l'ensemble de l'application sans aucun droit ni de modification, ni de suppression.
Flux	Le simple utilisateur peut accéder à cette fonctionnalité grâce à l'URL qui lui est dédiée.

TABLE 20 – Tableau de description textuelle UC16 *d'Airline 1*.

Annexe III - Planification initiale du projet

Le projet a véritablement commencé le 18 décembre 2019. La tâche que l'on m'a alors confiée était en adéquation avec l'équipe de développeurs à laquelle on m'a intégré. Du fait de la longueur de la tâche, il était indispensable de la découper en une multitude de phases afin de mener à un livrable dans les temps impartis, mais également afin de maximiser la valeur ajoutée de la solution. Tout d'abord, il est nécessaire de fournir, dans un temps relativement court, une première solution d'automatisation qui permettra de confirmer la première idée de solution. Puis, un second livrable fera suite au premier afin d'apporter une première solution aux équipes technique.

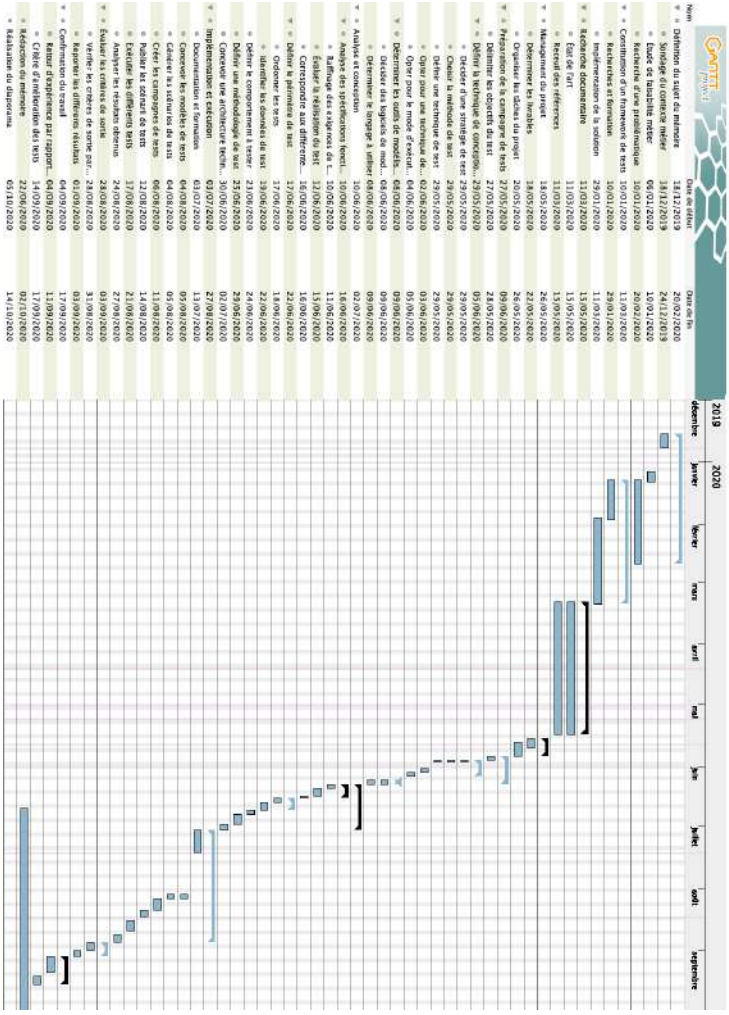

FIGURE 2 – Digramme de GANTT du projet.

Gestion de projet de l'alternance du Master II

Tâches

Nom	Date de début	Date de fin
Définition du sujet du mémoire	18/12/2019	20/02/2020
Sondage du contexte métier	18/12/2019	24/12/2019
Étude de faisabilité métier	06/01/2020	10/01/2020
Recherche d'une problématique	10/01/2020	20/02/2020
Constitution d'un framework de tests	10/01/2020	11/03/2020
Recherches et formation	10/01/2020	29/01/2020
Implémentation de la solution	29/01/2020	11/03/2020
Recherche documentaire	11/03/2020	15/05/2020
État de l'art	11/03/2020	15/05/2020
Receuil des références	11/03/2020	15/05/2020
Management du projet	18/05/2020	26/05/2020
Déterminer les livrables	18/05/2020	22/05/2020
Organiser les tâches du projet	20/05/2020	26/05/2020
Préparation de la campagne de tests	27/05/2020	09/06/2020
Délimiter les objectifs du test	27/05/2020	28/05/2020
Définir la technique de conception de test	29/05/2020	05/06/2020
Décider d'une stratégie de test	29/05/2020	29/05/2020
Choisir la méthode de test	29/05/2020	29/05/2020
Définir une technique de test	29/05/2020	29/05/2020
Opter pour une technique de test	02/06/2020	03/06/2020
Opter pour le mode d'exécution du test	04/06/2020	05/06/2020
Déterminer les outils de modélisation	08/06/2020	09/06/2020
Décider des logiciels de modélisation	08/06/2020	09/06/2020
Déterminer le langage à utiliser	08/06/2020	09/06/2020

TABLE 21 – Premier découpage des tâches de l'étude.

Gestion de projet de l'alternance du Master II

Tâches

Nom	Date de début	Date de fin
Analyse et conception	10/06/2020	02/07/2020
Analyse des spécifications fonctionnelles	10/06/2020	16/06/2020
Raffinage des exigences de tests	10/06/2020	11/06/2020
Évaluer la réalisation du test	12/06/2020	15/06/2020
Correspondre aux différentes exigences	16/06/2020	16/06/2020
Définir le périmètre de test	17/06/2020	22/06/2020
Ordonner les tests	17/06/2020	18/06/2020
Identifier les données de test	19/06/2020	22/06/2020
Définir le comportement à tester	23/06/2020	24/06/2020
Définir une méthodologie de test	25/06/2020	29/06/2020
Concevoir une architecture technique du système à tester	30/06/2020	02/07/2020
Implémentation et exécution	03/07/2020	27/08/2020
Documentation et formation	03/07/2020	13/07/2020
Concevoir les modèles de tests	04/08/2020	05/08/2020
Générer les scénarios de tests	04/08/2020	05/08/2020
Créer les campagnes de tests	06/08/2020	11/08/2020
Publier les scénarii de tests	12/08/2020	14/08/2020
Exécuter les différents tests	17/08/2020	21/08/2020
Analyser les résultats obtenus	24/08/2020	27/08/2020
Évaluer les critères de sortie	28/08/2020	03/09/2020
Vérifier les critères de sortie par rapport à la planification	28/08/2020	31/08/2020
Reporter les différents résultats	01/09/2020	03/09/2020
Confirmation du travail	04/09/2020	17/09/2020
Retour d'expérience par rapport aux résultats	04/09/2020	11/09/2020
Critère d'amélioration des tests	14/09/2020	17/09/2020

TABLE 22 – Deuxième découpage des tâches de l'étude.

Gestion de projet de l'alternance du Master II

Tâches

Nom	Date de début	Date de fin
Rédaction du mémoire	22/06/2020	02/10/2020
Réalisation du diaporama	05/10/2020	14/10/2020

TABLE 23 – Troisième découpage des tâches de l'étude.

Annexe IV - Diagramme de classes de domaine du EFF

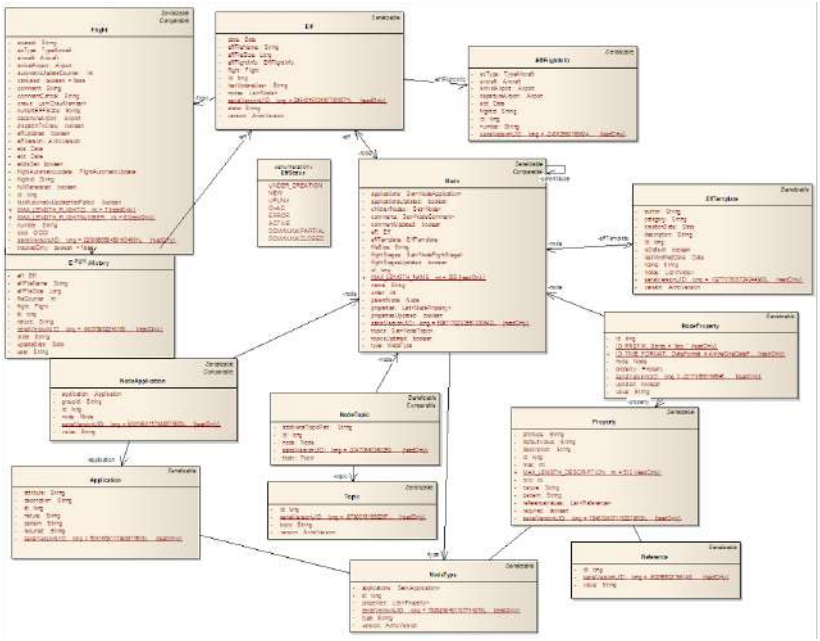

FIGURE 3 – Diagramme de classes de domaine du EFF.

Annexe V - Diagramme de cas d'utilisation d'Amazon

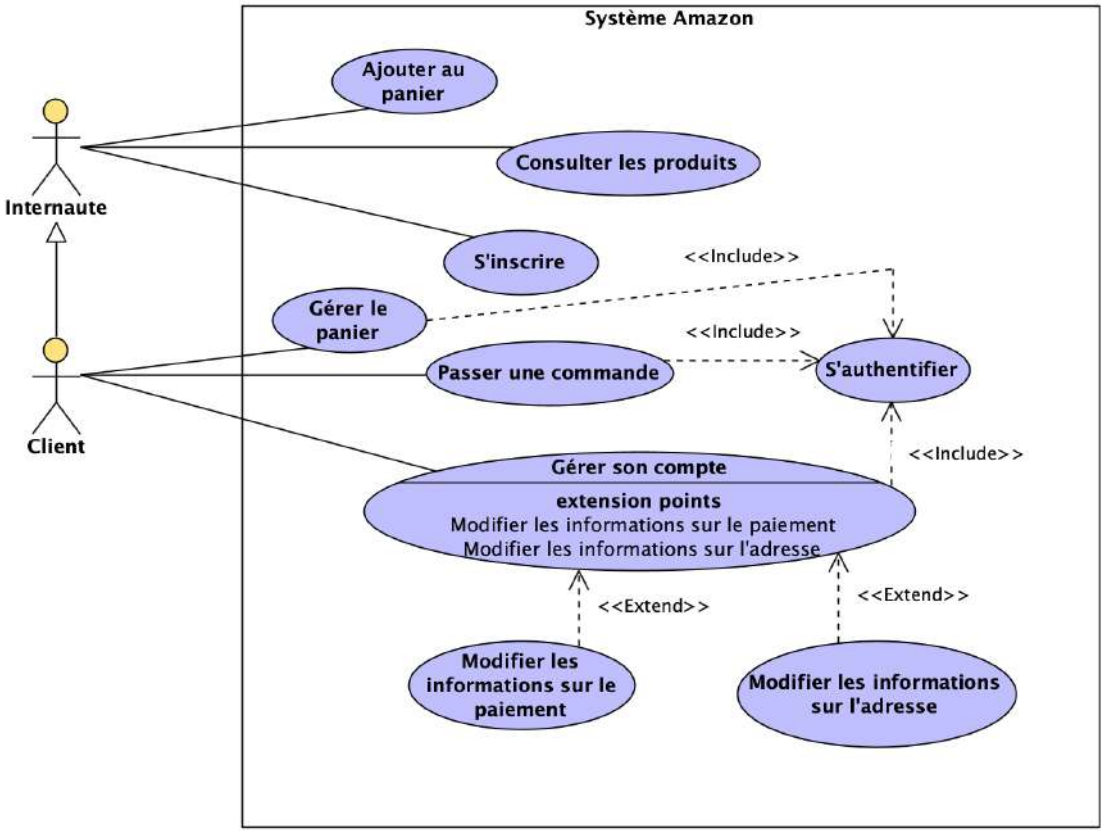

FIGURE 4 – Diagramme de cas d'utilisation *d'Amazon*.

ID	UC17.
Cas d'utilisation	Consulter les produits.
Acteurs	Internaute.
Scénario nominal	Un internaute peut consulter l'ensemble des produits proposés par l'application web dans le but de réaliser une commande par la suite.
Flux	L'internaute peut accéder à cette fonctionnalité grâce à l'URL de l'application.

TABLE 24 – Tableau de description textuelle UC17 *d'Amazon*.

ID	UC18.
Cas d'utilisation	Ajouter au panier.
Acteurs	Internaute.
Scénario nominal	Un internaute peut ajouter un article dans son panier. Pour cela il doit d'abord consulter un article désiré. Puis une fois que ce dernier a fait son choix, il doit cliquer sur le bouton « Ajouter au panier ».
Flux	L'internaute peut accéder à cette fonctionnalité grâce à l'URL de l'application.
Préconditions	L'internaute doit au préalable consulter un article.
Post conditions	Une fois que l'internaute aura cliqué sur le bouton « Ajouter au panier », le système va incrémenter le nombre d'articles ajoutés sur l'icône du panier. Par la suite, l'internaute sera redirigé vers son panier pour, s'il le souhaite, terminer le processus d'achat.

TABLE 25 – Tableau de description textuelle UC18 *d'Amazon*.

ID	UC19.
Cas d'utilisation	S'inscrire.
Acteurs	Internaute.
Scénario nominal	Un internaute peut s'inscrire à l'application web pour réaliser une commande. Pour cela il sera demandé de renseigner un e-mail ainsi qu'un mot de passe.
Flux	L'internaute peut accéder à cette fonctionnalité grâce à l'URL de l'application.

TABLE 26 – Tableau de description textuelle UC19 *d'Amazon*.

ID	UC20.
Cas d'utilisation	Gérer le panier.
Acteurs	Client. (Généralisation de l'acteur internaute).
Scénario nominal	Un client peut gérer son panier client dans le but de vérifier s'il s'agit bien du produit qu'il souhaite. Il peut aussi modifier la quantité de son ou ses produits présents dans le panier. Enfin, il peut supprimer un ou plusieurs produits dans le cas où il n'est plus désiré par ce dernier.
Flux	Le client peut accéder à cette fonctionnalité grâce à l'URL de l'application.
Post conditions	Le panier du client est mis à jour.

TABLE 27 – Tableau de description textuelle UC20 *d'Amazon*.

ID	UC21.
Cas d'utilisation	Passer une commande.
Acteurs	Client. (Généralisation de l'acteur internaute).
Scénario nominal	Un client peut passer une commande. Pour cela il doit consulter un produit, le rajouter à son panier client, cliquer sur le bouton de validation. Enfin il faut qu'il choisisse son moyen de paiement ainsi que son adresse de livraison.
Flux	Le client peut accéder à cette fonctionnalité grâce à l'URL de l'application.
Préconditions	Le client doit au préalable posséder un compte. Il doit également fournir un moyen de paiement ainsi qu'une adresse de livraison et de facturation.
Post conditions	Un email sera envoyé au client pour lui confirmer que sa commande a été réalisée avec succès.

TABLE 28 – Tableau de description textuelle UC21 *d'Amazon*.

ID	UC22.
Cas d'utilisation	S'authentifier.
Acteurs	Client. (Généralisation de l'acteur internaute).
Scénario nominal	Un client peut s'authentifier. Pour cela il renseigne au système son e-mail ainsi que son mot de passe de connexion.
Flux	Le client peut accéder à cette fonctionnalité grâce à l'URL de l'application.
Préconditions	Le client doit au préalable posséder un compte.
Post conditions	Le client sera redirigé vers l'interface d'accueil de l'application.

TABLE 29 – Tableau de description textuelle UC22 *d'Amazon*.

ID	UC23.
Cas d'utilisation	Gérer son compte.
Acteurs	Client. (Généralisation de l'acteur internaute).
Scénario nominal	Un client peut gérer son compte client. Il peut mettre à jour ses informations concernant le paiement ainsi que mettre à jour les informations qui concernent l'adresse de livraison et de facturation.
Flux	Le client peut accéder à cette fonctionnalité grâce à l'URL de l'application.
Préconditions	Le client doit au préalable posséder un compte.
Post conditions	Le client sera notifié par un pop-up du succès de la mise à jour.

TABLE 30 – Tableau de description textuelle UC23 *d'Amazon*.

Génération automatique de tests d'acceptance par une approche de tests dirigée par les modèles pour une application web

Mémoire d'Ingénieur C.N.A.M., Toulouse 2020

RÉSUMÉ

Ce mémoire traite de deux approches, la première approche est axée sur le comportement avec Behavior-Driven Development et la seconde dirigée par les modèles avec Model-Based Testing. En effet, la qualité du logiciel est un enjeu crucial du fait de l'intéropérabilité des objets. Ces logiciels sont toujours plus avides de fonctionnalités. Partant de ce postulat, l'axe de recherche de ce mémoire était d'étudier de nouvelles méthodes de tests afin de limiter les coûts et les ressources déployées sur ces activités de tests et plus particulièrement les tests d'acceptance. En conséquence, un comparatif de deux méthodes de tests a été réalisé pour aboutir à la mise en place d'un générateur de scripts de tests, d'après un modèle abstrait, reposant sur la philosophie de transformation de modèles ayant pour origine l'approche Model-Driven Architecture.

Mots clés : Tests d'acceptance, automatisation, Domain-Specific Language, Behavior-Driven Development, Transformation de modèles, Model-Based Testing

---

ABSTRACT

This master thesis deals with two approaches, the first one is behavior-driven with Behavior-Driven Development and the second model-driven with Model-Based Testing. Indeed, the quality of the software is a crucial issue because of the interoperability of objects. These software are always more feature-hungry. Starting from this postulate, the research axis of this master thesis was to study new test methods in order to limit the costs and resources deployed on these testing activities and more particularly on acceptance tests. Consequently, a comparison of two test methods was carried out to lead to the implementation of a test script generator, based on an abstract model, based on the philosophy of model transformation originating from the approach Model-Driven Architecture.

Key word : Acceptance test, automation, Domain-Specific Language, Behavior-Driven Development, model transformation, Model-Based Testing