Migration WPF vers Angular

TER

M1 Informatique - Génie Logiciel

Faculté des Sciences de Montpellier

Encadrants : SERIAI Abdelhak-Djamel et RIMA Bachar

TRINQUART Matthieu / SANCHEZ Martin / ROMERO Gaétan

23 mai 2022

Remerciements

Avant toute chose, il nous paraît nécessaire de commencer ce rapport par des remerciements. Aussi, nous tenons tout particulièrement à remercier nos encadrants M. SERIAI et M. RIMA pour leur gentillesse, leur pédagogie, pour nous avoir guidé tout au long du projet et nous avoir permis d'atteindre nos objectifs. Nous tenons aussi à remercier BEGOUG Mahi pour toute l'aide qu'il nous a apporté et pour avoir mis à notre disposition son travail réalisé sur la migration de GWT vers Angular.

Sommaire

1 Présentation du projet

1.1 Introduction

Les technologies de programmation, les langages et les Frameworks évoluent et se renouvellent très fréquemment. Il n'est pas rare alors de trouver au sein d'entreprises des logiciels, des outils développés avec les technologies obsolètes. Cela représente un coup massif pour une entreprise de redévelopper le même outil avec une technologie plus moderne. C'est pour cela qu'on s'intéresse de plus en plus à la migration. La migration peut être manuelle ou automatique. Le problème est que la migration manuelle est coûteuse, longue et risquée. C'est pour cette raison, que la migration automatique est une solution pour réduire les coûts, les délais et les risques.

Dans le cadre de notre projet, nous nous intéressons à la migration d'applications WPF, qui est un Framework datant de 2007 vers des applications Angular, qui lui date de 2016 et est donc bien plus récent et plus utilisé. De plus, aujourd'hui les applications web sont de plus en plus privilégiées aux applications de bureau, ce qui augmente l'intérêt de notre migration.

1.2 Contexte

Pour être dans les meilleures conditions pour comprendre ce rapport sur la migration de WPF vers Angular, il faut déjà connaître un minimum ces deux technologies. Nous ne nous étalerons pas sur l'explication profonde de leur fonctionnement ici, car nous le ferons un peu plus loin, mais citons au moins quelques points.

WPF(Windows Presentation Foundation) est un Framework créé par Microsoft en 2007 qui permet de faire des applications de bureau (cf. Documentation WPF Microsoft). Il nous permet de découper le développement d'applications en deux, avec une partie pour le visuel en XAML et une pour le fonctionnement en C#.

Angular est un Framework web utilisant les langages HTML,CSS et TypeScript créer par Google en 2016, qui permet de faire des applications web (cf. Description Angular). Il s'agit d'un Framework basé sur le principe de composant. C'est-à-dire que le fonctionnement d'une application est géré par des composants, qui sont en fait, des éléments réutilisables constitués d'une vue et d'un ensemble de traitements qui définissent la manière dont l'utilisateur peut interagir avec elle.

1.3 Motivations en lien avec notre formation

Tous les membres du groupe étant en Génie logiciel, il est important pour nous de connaître les principaux Frameworks qui permettent de créer des applications. En faisant notre projet TER, nous avons donc emmagasiné beaucoup de connaissances en Angular et WPF qui sont deux Frameworks que nous pourrions avoir à réutiliser dans nos futurs emplois.

Ce projet nous a aussi permis d'avoir une initiation dans la création de modèle, métamodèle ou workflow. Ces connaissances pourront aussi nous servir l'année prochaine quand nous verrons l'ingénierie des modèles.

De plus, dû a la rapidité des changements des technologies dans l'informatique, il est important pour nous d'avoir des bases dans la migration entre deux technologies.

1.4 Cahier des charges

L'application Angular rendue par la migration, doit être identique à l'application de base WPF. Pour cela, l'apparence du site doit se rapprocher le plus possible de l'apparence de base de l'application WPF, le positionnement de chaque élément doit être identique, l'apparence de chaque élément aussi ou presque, etc... Les fonctionnalités implémentées par l'application WPF doivent être aussi les mêmes dans le rendu Angular.

Il ne faut aussi pas perdre de vue que le code obtenu pour l'application Angular, doit pouvoir être lu et compris par un développeur quelconque et le nommage, la visibilité, etc..., doivent rester identique.

1.5 Déroulement du rapport

Dans le rapport, nous allons commencer par faire une étude de notre sujet en étudiant une migration de GWT vers Angular et aussi expliquer pourquoi la migration de WPF vers Angular a un intérêt. Nous allons ensuite expliquer le fonctionnement de WPF et de Angular en montrant leurs modèles et WorkFlow respectifs. Par la suite, nous parlerons de l'analyse du sujet et de la réflexion derrière sa conception, avant de montrer les avancés actuelles de la migration. Ensuite, nous consacrerons deux parties pour parler de l'expérimentation d'un test en particulier et de l'étude et l'interprétation des résultats obtenus. Enfin, nous parlerons de notre gestion du projet et ferons la conclusion en parlant des perspectives de notre projet.

2 Etude

Maintenant que nous avons présenté le sujet ainsi que le cahier des charges, nous allons pouvoir expliciter l'ensemble de l'étude que nous avons réalisée au préalable. L'utilité de cette partie est de nous permettre de comprendre les technologies utilisées mais aussi les enjeux y étant liés ainsi que les outils requis pour réaliser cette migration.

Cette partie sera découpée en deux parties distinctes. La première sera l'étude d'une migration déjà réalisée du Framework GWT vers Angular, et la deuxième sera l'étude de notre cas et des technologies utilisées.

2.1 Etude de la migration existante de GWT vers Angular

Avant de décrire cette migration, il serait judicieux d'expliquer pourquoi on fait l'étude d'une autre migration plutôt que de commencer la nôtre. La raison est très simple. Au commencement du projet, nous ne connaissions pratiquement pas cette partie de l'informatique qui consiste à migrer des applications créées dans une technologie qui devient obsolète vers une autre technologie plus récente et donc plus utilisée. De ce fait, nous n'avions pas connaissance des outils, des techniques et des étapes à suivre pour y parvenir, il fallait donc que nous nous renseignions sur ceux-ci. C'est pourquoi l'étude de cette migration a été importante.

Maintenant que nous avons expliqué pourquoi, nous allons pouvoir présenter ce que cette migration nous a apporté. Tout d'abord, nous avons eu une présentation de celle-ci dans laquelle on nous expliquait le déroulement du projet ainsi que les outils utilisés pour y parvenir. Donc, dans un premier temps, pour un projet de ce type, les étapes les plus importantes pour débuter sont, l'étude de toutes les technologies mises en jeu et donc la compréhension totale de celles-ci, ainsi que l'assimilation de toutes les différences structurelles entre celles-ci. C'est-à-dire toutes les différences qui existent que ce soit dans la manière dont une même application se comporte dans les deux technologies (puisque l'on passe d'un Framework vers un autre), ou que ce soit dans la structure de l'application elle-même. De plus, pour décrire le fonctionnement et la structure d'une application, nous devions construire le workflow ainsi que les modèles et méta-modèles de ces technologies. Le principe du modèle est que n'importe quelle application créée avec WPF ou Angular par exemple, doit être une instance de celui-ci. Il faut donc comprendre la totalité du Framework (ou de tout autre technologie utilisée) pour faire ces modèles, mais nous les verrons plus en détails un peu plus tard. Ensuite, nous avons eu accès au rapport de ce projet ainsi qu'au code sur Github pour nous familiariser avec les techniques et les outils. Cependant comme nous n'utilisons pas les mêmes Frameworks, nous ne pouvions pas utiliser les mêmes outils, mais le principe restait le même donc cela nous a grandement aidé à réfléchir et imaginer le déroulement du projet.

Maintenant que nous avons vu un cas réel de migration, nous allons pouvoir nous pencher sur notre cas d'étude. Dans un premier temps, nous parlerons de pourquoi avoir choisis WPF et Angular et ensuite nous verrons les différences structurelles entre eux.

2.2 Choix des technologies

Le choix des technologies constitue une étape importante et même cruciale dans une migration, si l'on souhaite allier la performance et l'intérêt. Et oui, car l'objectif ultime reste tout de même de migrer une application créée dans un framework(ou library) obsolète (ou dépassé) vers une application créée dans une technologie plus récente. Nous devons donc, à la fin, être en possession d'une application qui garde le même fonctionnement, mais aussi qui ne perd pas tout son sens comme par exemple une application de bureau qui devient une application mobile... Donc, dans notre cas, on peut se poser deux questions Pourquoi WPF ? et Pourquoi Angular ?

2.2.1 Pourquoi WPF ?

Commençons par décrire ce qu'est WPF. WPF ou Windows Presentation Foundation, est un Framework GUI (cf. WPF Tutorial) c'est-à-dire qu'il s'agit d'un Framework qui s'occupe de gérer en grande partie la partie graphique d'une application en nous fournissant des widgets déjà créés et nous permet de nous concentrer sur la partie métier de l'application. Il est utilisé avec .NET Framework afin de créer des applications de bureau. Le développement d'une application en WPF est découpé en deux parties. L'une s'occupe du visuel en XAML et l'autre de la partie métier en C#. De cette manière, nous pouvons, déjà, bien mieux visualiser l'interface utilisateur sans toucher à la partie métier et nous pouvons aussi mieux découper le déroulement du développement de l'application.

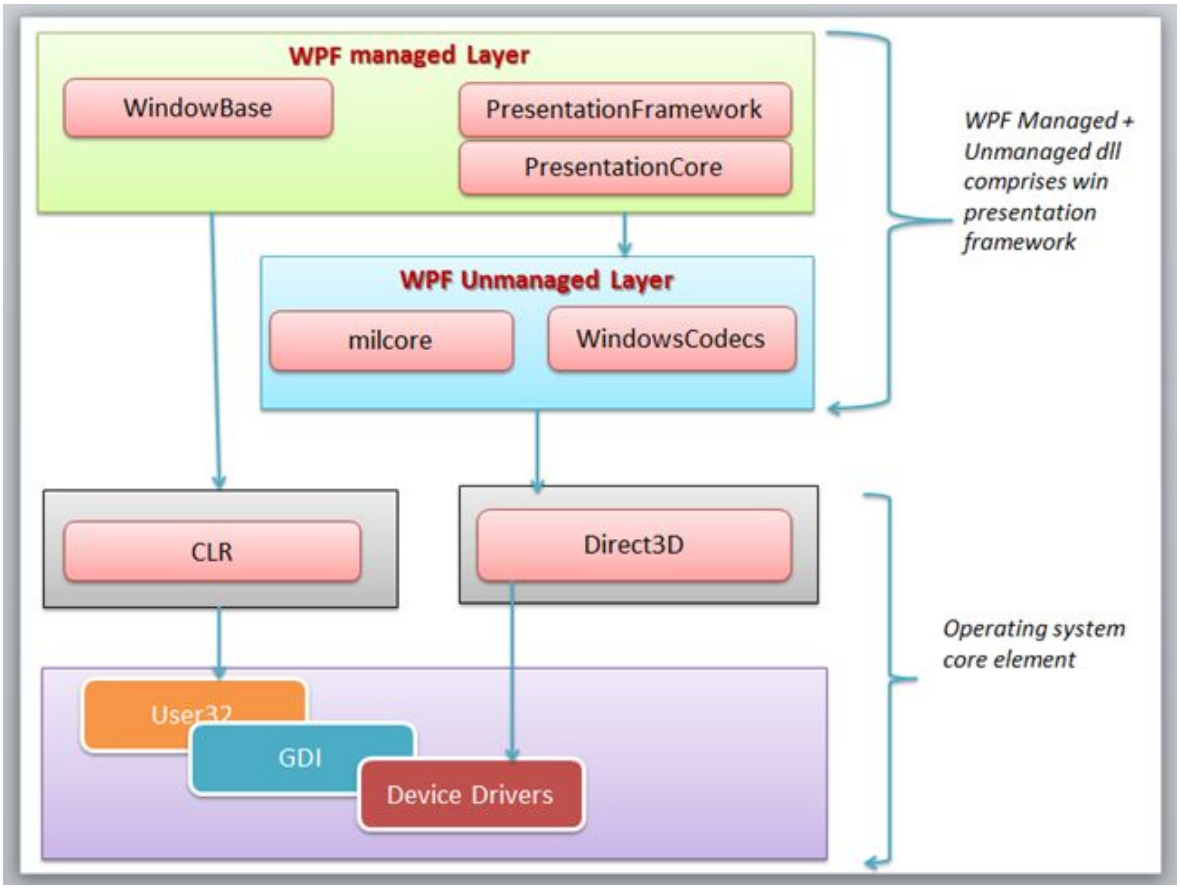


FIGURE 2.1 – Structure application WPF

Ce Framework est basé sur une architecture MVVM c'est à dire Model View ViewModel. Il s'agit d'un design pattern qui vise à séparer la gestion de l'application en trois parties :

- Model : s'occupe de la logique métier de l'application.
- View : s'occupe de l'interface graphique.
- ViewModel : fait le lien entre les deux. C'est à dire qu'il gère les données de la partie Model qui doivent être utilisées par View

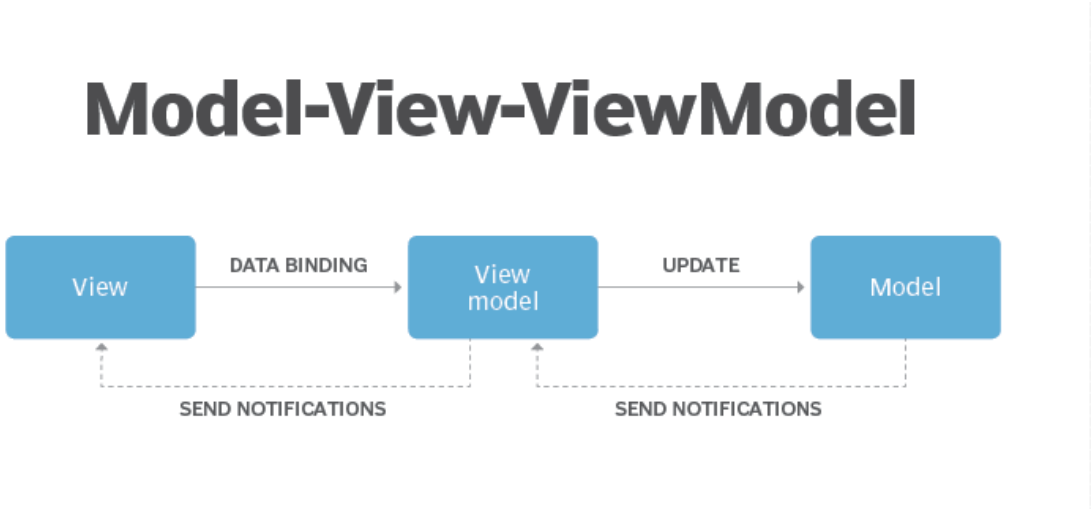


FIGURE 2.2 – Architecture MVVM

Malgré le fait que WPF soit encore utilisé par certains développeurs d'application de bureau, sa popularité baisse d'années en années et les applications créées avec, sont vouées à disparaître si elles ne sont pas migrées rapidement.

On peut voir ci-dessous un graphique représentant les tendances de questions autour de WPF sur StackOverflow depuis 2009.

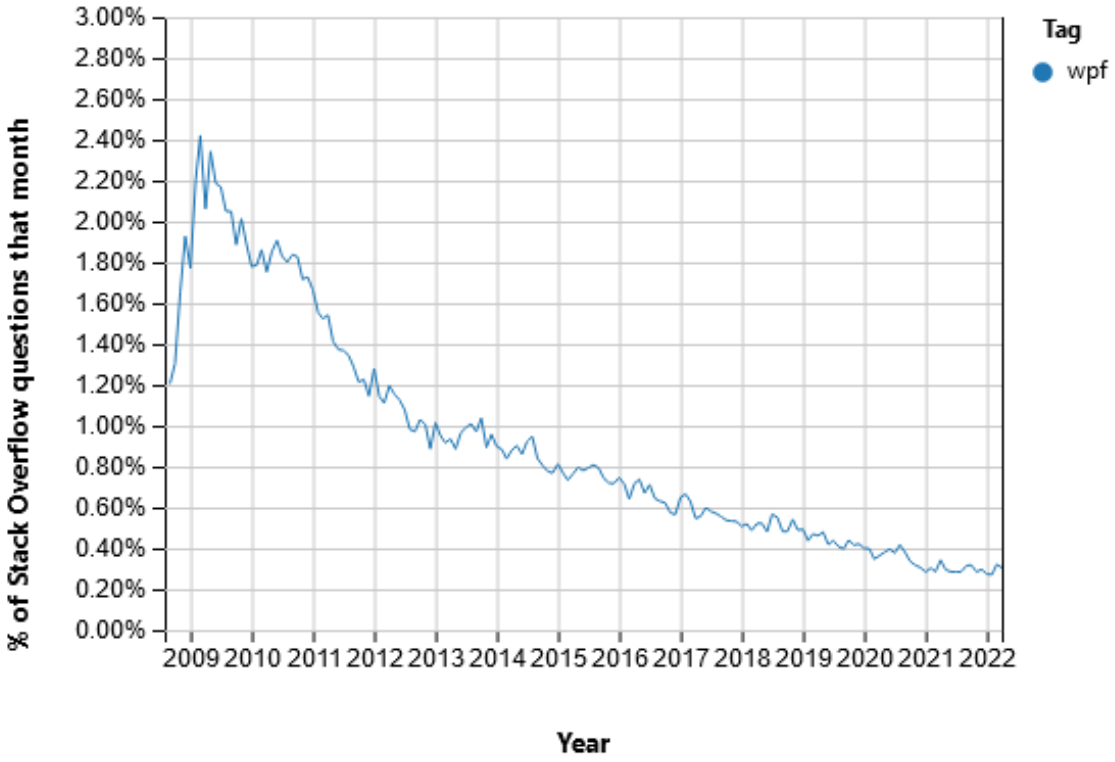


FIGURE 2.3 – Tendances des questions autour de WPF depuis 2009 sur StackOverflow

Sur cette figure, nous pouvons voir que la courbe descend continuellement depuis une dizaine d'années, ce qui prouve que WPF est de moins en moins utilisé avec le temps. Cela ne fait que confirmer le besoin de migration de plus en plus urgent pour les applications réalisées avec ce Framework. Maintenant que nous avons explicité les raisons pour lesquelles nous avons fait le choix de WPF, nous allons faire de même pour Angular.

2.2.2 Pourquoi Angular ?

Comme précédemment, nous allons faire une brève explication de ce qu'est Angular (cf. Page Angular Wikipédia). Angular est un Framework Web Open-Source permettant de créer des applications web évolutives c'est à dire que le contenu de la page change sans que la page elle-même soit rechargée. C'est le principe des SPA ou Single Page Applications mais nous verrons ça un peu plus tard. Nous devons préciser avant toute chose que le principe de fonctionnement de ce Framework repose sur les composants et les services. Tout d'abord nous avons les composants. Ils sont découpés en 2 parties distinctes. La partie graphique (template) qui est implémentée en HTML/CSS et la partie métier du composant implémentée en TypeScript. Tout ce qui sera affiché sur l'interface utilisateur et toute interaction qui en découlera, proviendra d'eux. Pour expliquer en quelques mots le principe d'une page Angular, il y a un composant par défaut qui s'appelle app-component. C'est par lui que nous pouvons afficher les autres composants, par l'intermédiaire d'une balise <router-outlet> qui permet de changer le composant afficher sur la page actuellement. Cela permet de changer l'interface sans recharger la page entièrement. Il existe plusieurs méthodes pour faire cela mais nous ne sommes pas ici pour en faire la description..

Il nous reste maintenant les services. Ces services sont des classes TypeScript qui vont nous permettre de réutiliser du code entre différents composants, mais aussi d'optimiser l'échange des données dans l'application. De ce fait, les responsabilités techniques et visuelles sont séparées.

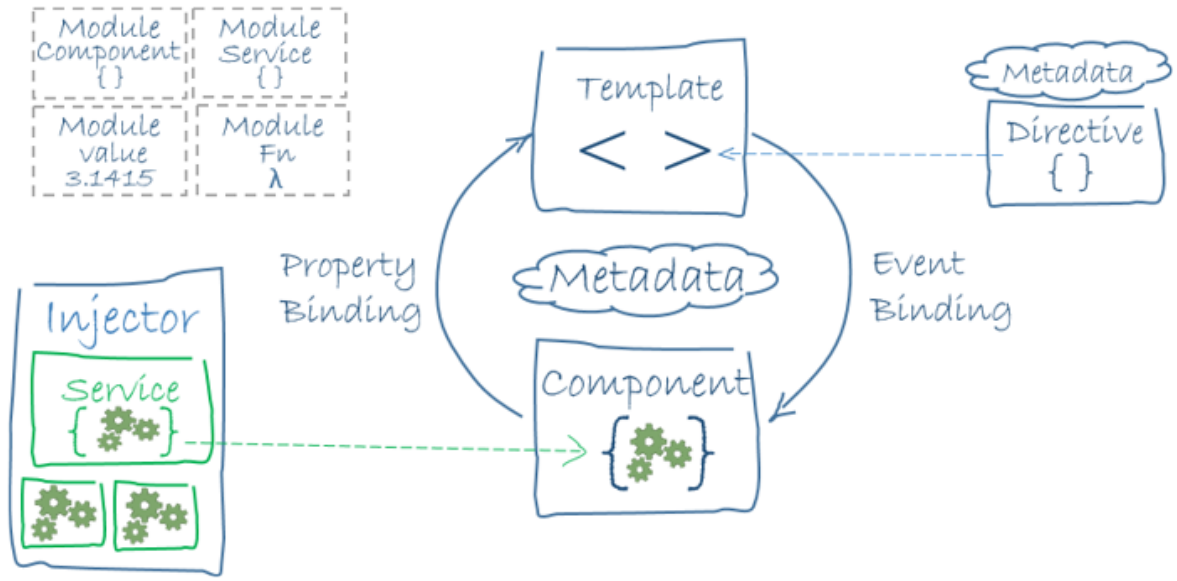


FIGURE 2.4 – Structure d'une application Angular

Angular est basé sur la même architecture que WPF, c'est à dire MVVM, donc la migration de l'un vers l'autre semble, dans un sens, moins ambigüe étant donné que les deux Frameworks sont découpés de façon semblable. Mais plusieurs questions se posent alors. Pourquoi avoir choisi de migrer WPF vers du Angular ? Est-ce judicieux de migrer une application de bureau vers une application web ? et est-ce possible ?

Pour répondre à la première, voici deux graphiques montrant le nombre de recherches sur Google tournant autour de WPF et Angular et les tendances des questions autour de WPF et Angular sur StackOverflow :

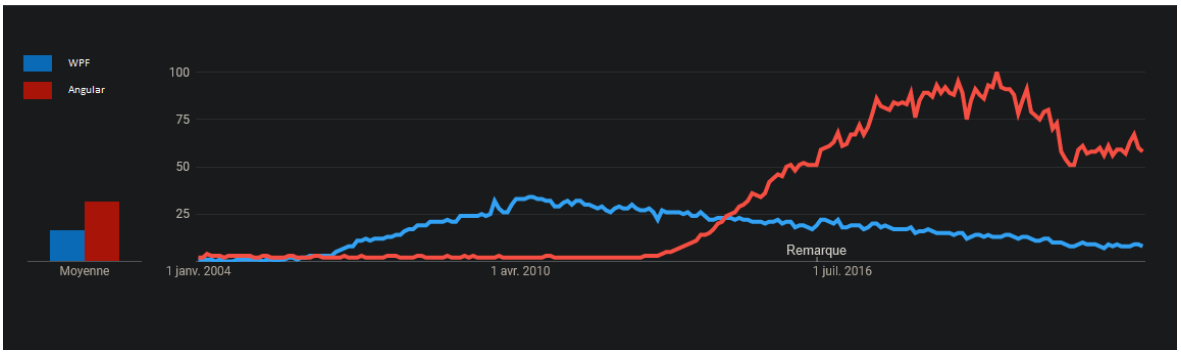


FIGURE 2.5 – Recherches Google autour de WPF et Angular

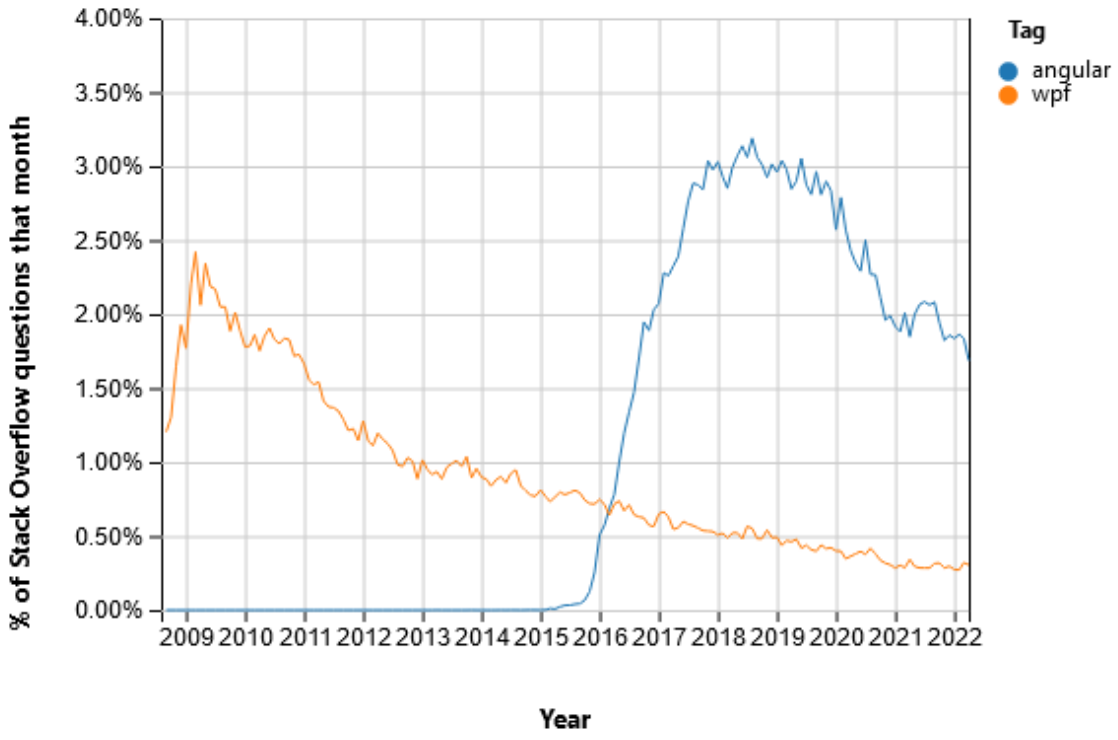


FIGURE 2.6 – Tendances des questions autour de WPF et Angular depuis 2009 sur StackOverflow

Nous remarquons, sur ces deux graphiques, qu'en plus du fait que WPF baisse continuellement, Angular fait plutôt l'inverse. Cela signifie qu'Angular est en plein essor comparé à WPF et est donc un choix plutôt idéal pour cette migration. Nous avons aussi un graphique montrant les technologies les plus utilisées par les développeurs en 2021 :

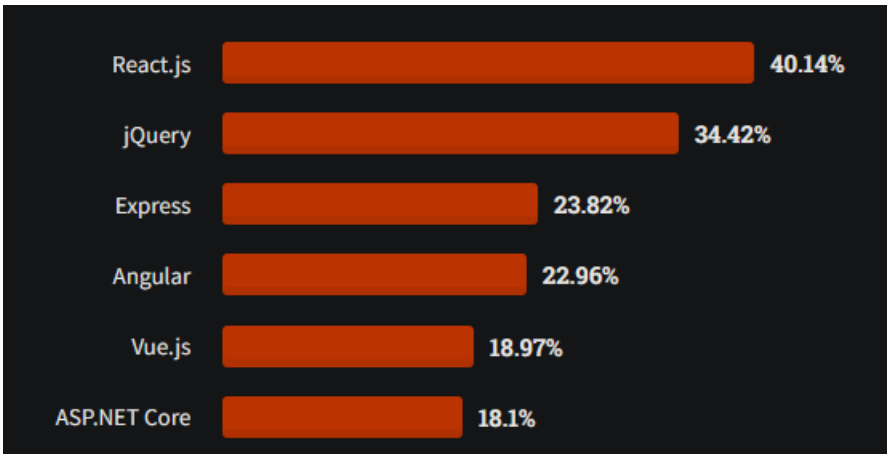


FIGURE 2.7 – Popularité des technologies en 2021

Ici nous pouvons voir qu'Angular fait partie des 5 Frameworks/Library les plus utilisés au monde.

Pour répondre maintenant aux autres questions, alors oui bien sûr c'est possible de faire une telle migration. Quant au fait de savoir si c'est judicieux, cela dépend dans un premier temps du sujet de l'application et ensuite du public/client visé. Si elle a pour but d'organiser nos documents par exemple, il vaut mieux ne pas changer sinon elle perdrait tout son sens. Mais s'il s'agit par exemple d'une application de génération d'url, la transformer en application web est tout de suite bien plus crédible.

2.3 Différences structurelles

Il reste un point à aborder maintenant que nous avons présenté les deux Frameworks et c'est celui des différences structurelles entre les deux. Nous avons vu, jusque-là, qu'ils faisaient tous les deux parties de la même architecture, qui est MVVM, mais nous n'avons pas abordé ce qui les différenciait. La principale différence réside dans le fait qu'une application WPF est une MPA (Multi Page Application) et une application Angular est une SPA (Single Page Application). Nous allons voir que ces deux types divergent beaucoup même si, au premier regard, nous pourrions les confondre.

2.3.1 MultiPage et SinglePage

Commençons par décrire le principe de fonctionnement derrière ces deux catégories d'applications.

Une SPA est une application qui n'a qu'une seule page comme son nom l'indique. En fait, la page est chargée au démarrage et après ce ne sont que des changements de contenu dynamiques. De ce fait, la structure de la page ne change pas et donc l'application est beaucoup plus rapide, fluide. Ces changements sont orchestrés par des API telles que HttpRequest en JavaScript.

Une MPA est, à l'inverse, une application composée de plusieurs pages. La plupart des sites web que nous connaissions, il y a de cela quelques années, en étaient. Il ne faut pas oublier que le terme SPA est très récent dans le monde de la programmation web, avant il n'existait que les applications MPA, seulement on ne leur donnait pas de nom car c'était le seul type d'application. La différence est donc qu'une MPA va recharger une nouvelle page à chaque interaction de l'utilisateur, même si seulement une partie de celle-ci change. Malgré cela, quelques points sont encore en faveur des MPA puisque le temps de chargement initial est bien plus rapide car il ne nécessite pas de charger de code en JavaScript (ou très peu comparé aux SPA) et d'analyser l'URL pour savoir quel composant afficher.

Une problématique se fait apparaître. Il s'agit de comment passer de plusieurs pages à une seule sans altérer le fonctionnement. Nous verrons dans une autre partie (cf. partie 3.1.1) comment nous avons décidé de résoudre cette problématique.

2.3.2 Modèles

Dans notre étape vers la compréhension de ces deux technologies, nous avons dû établir les méta-modèles et les workflows des deux. Nous verrons les workflows dans un second temps. Tout d'abord concentrons nous sur les modèles et méta-modèles.

Qu'est-ce qu'un modèle ? Un modèle est une façon structurée et simplifiée de représenter une technologie telle que chaque application créée avec cette technologie (ou domaine d'application), soit une instance de ce modèle.

Et un méta-modèle ? C'est un modèle dont les instances sont aussi des modèles. Le terme méta devant modèle signifie que l'on abstrait encore un peu plus de sorte que le résultat final permette de décrire un modèle et non le domaine d'application. Toutes ces techniques servent à conceptualiser une technologie.

WPF

Pour WPF, nous avons créé trois ViewPoint différents pour modéliser le méta-modèle. Le premier, "ViewPoint Global", décrit la structure globale d'une application. Nous pouvons voir qu'un projet WPF est bien découpé en deux parties : Logique Métier donc les classes C# et GUI donc la partie graphique en XAML. "Bin" et "Obj" sont des dossiers générés automatiquement lors de la création du projet, c'est pour cela que nous les avons mis d'une couleur différente. « Ressources » est un dossier contenant les possibles ressources de notre application mais n'est pas obligatoire d'où la couleur jaune correspondante à « Optionnal ». Il nous reste « Visual ». Celui-ci correspond à la super classe englobant tous les éléments graphiques(Widgets) qui constitueront les éléments mis dans la partie GUI de l'application. Nous avons créé un nouveau ViewPoint pour cette classe afin de mieux découper notre structure. Nous devons préciser deux points avant de passer au ViewPoint suivant. Premièrement, chaque fichier de logique métier est associé à un fichier GUI portant exactement le même nom que lui, et deuxièmement, chaque classe de la logique métier étend un élément de la classe Visual comme Window ou Page....

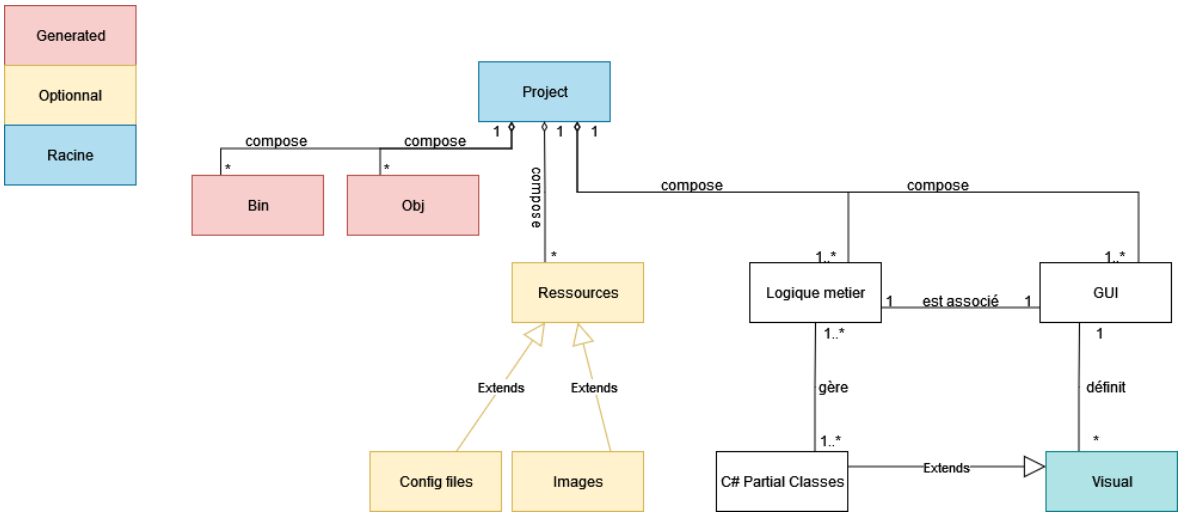


FIGURE 2.8 – ViewPoint Global de WPF

Le ViewPoint suivant est celui portant sur la classe Visual, comme expliqué juste au-dessus. Nous pouvons voir qu'elle contient trois classes. Chacune de ces classes permet d'utiliser des éléments différents. Premièrement, ContainerVisual, est une classe contenant tous les éléments permettant de structurer et contenir les autres widgets. Il ne peut y en avoir qu'un seul par page car ils ne peuvent pas se contenir eux-mêmes. Parmi les objets de cette classe, les plus utilisés sont Window, Page ...

Ensuite, ViewPort3DVisual. Cette classe nous permet d'utiliser les éléments 3D grâce à un Adapteur, dans un visuel 2D.

Pour terminer, il y a UIElement. Elle possède qu'une seule sous classe FrameworkElement, car elle permet surtout aux utilisateurs de créer leur propre classe de Widget en étendant celle-ci. C'est dans FrameworkElement que tous les éléments graphiques (ou widgets) sont présents. Nous les verrons dans le dernier ViewPoint.

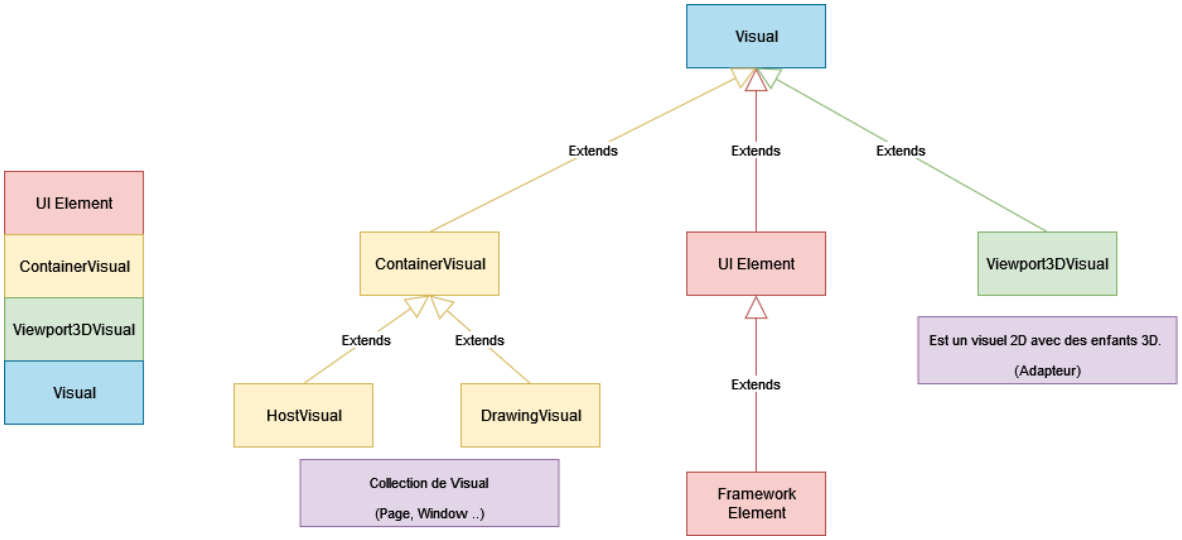


FIGURE 2.9 – ViewPoint Visual de WPF

Ce dernier ViewPoint liste l'ensemble des sous classes de FrameworkElement contenant les widgets. Nous avons par exemple Panel qui contient les widgets permettant de contenir et gérer les positions de certains éléments, Shape qui contient toutes les formes comme Rectangle / Line ...

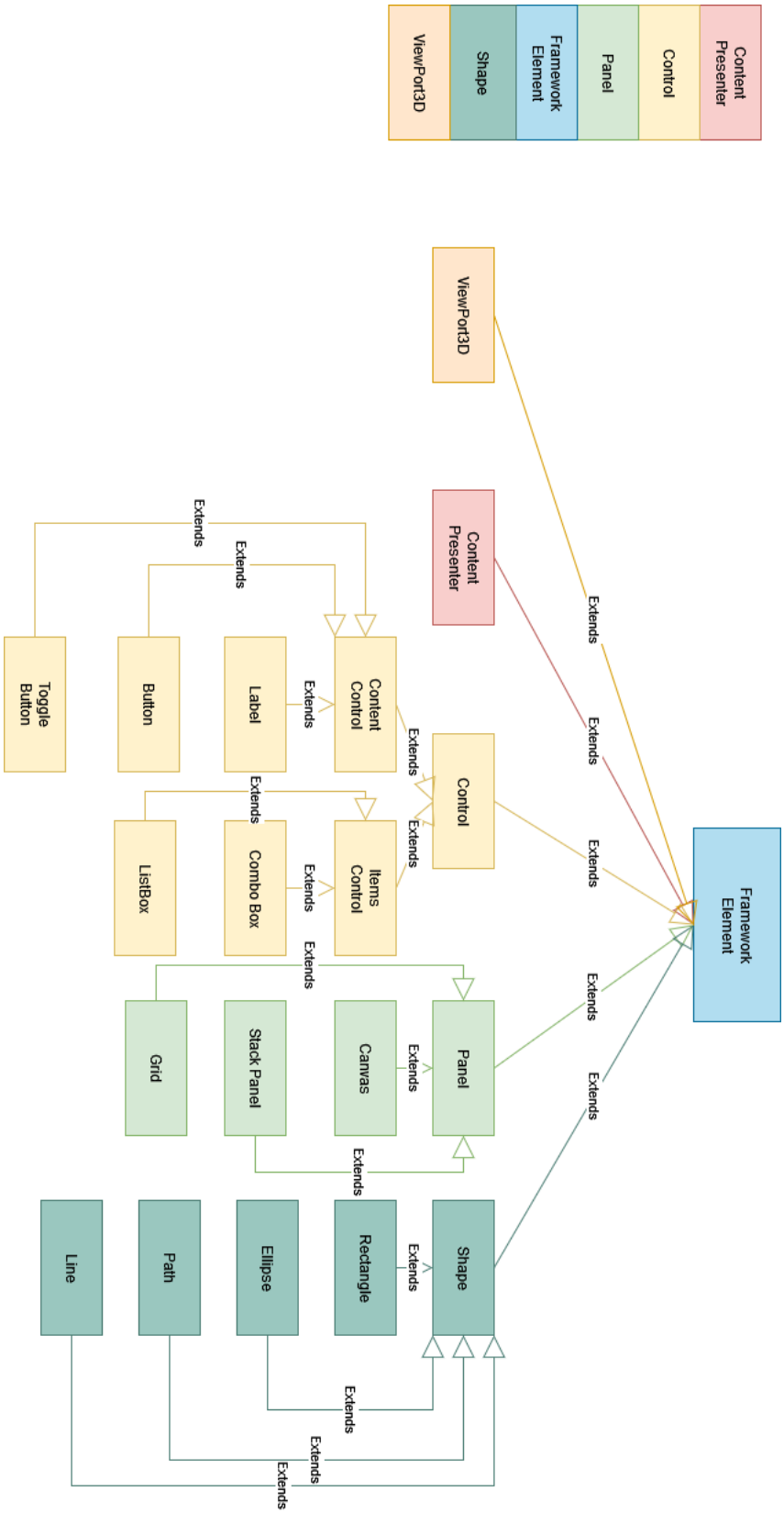


FIGURE 2.10 – ViewPoint FrameworkElement de WPF

Angular

Dans le cas d'Angular, nous avons fait deux ViewPoint. Le premier, comme pour WPF, nous montre la structure d'un projet. Ce projet contient des dossiers et des fichiers générés automatiquement à la création. Ils sont représentés en vert et rouge sur le ViewPoint. La partie la plus importante parmi ce modèle est le dossier App. C'est dans celui-ci que le fonctionnement global d'une application Angular est développé. C'est d'ailleurs le contenu du second ViewPoint que nous verrons juste après.

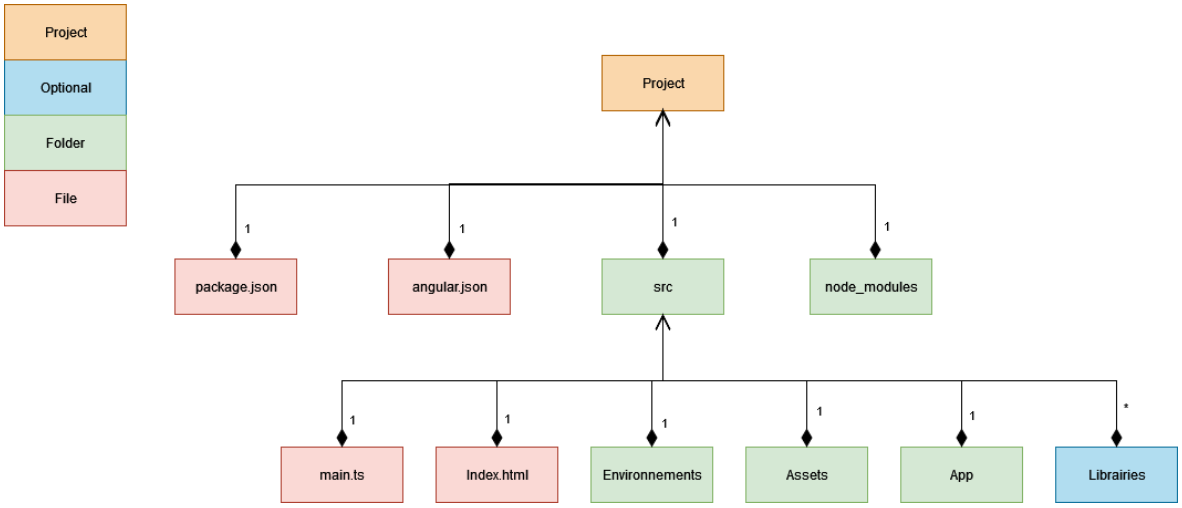


FIGURE 2.11 – ViewPoint Général d'Angular

Comme dit précédemment, ce ViewPoint décrit le dossier App. Parmi les deux parties optionnelles (en bleu), nous avons les services et les composants additionnels (car il y a un composant par défaut). Ils sont optionnels car une application peut très bien fonctionner sans eux, elle n'a besoin que du composant par défaut. En parlant de ce composant par défaut, il s'agit du RootComponent cité dans le ViewPoint. Dans une application, il s'appelle toujours AppComponent et nous pouvons voir qu'il contient 4 fichiers principaux :

— Les .html et .css
  Ils s'occupent de la partie graphique.
— Les .ts
  Ils s'occupent de la partie métier.

Le fichier "app-module.ts" est optionnel car nous avons la possibilité de choisir, à la création du projet, si nous voulons ajouter un fichier de module ou non. Pour notre migration, ce fichier sera toujours mis par défaut.

Nous devons préciser que les composants ont tous la même composition que le root component, c'est à dire les mêmes fichiers. La partie Behavior est gérée par les .ts, la partie Module par app-module.ts et le template par les .html et .css.

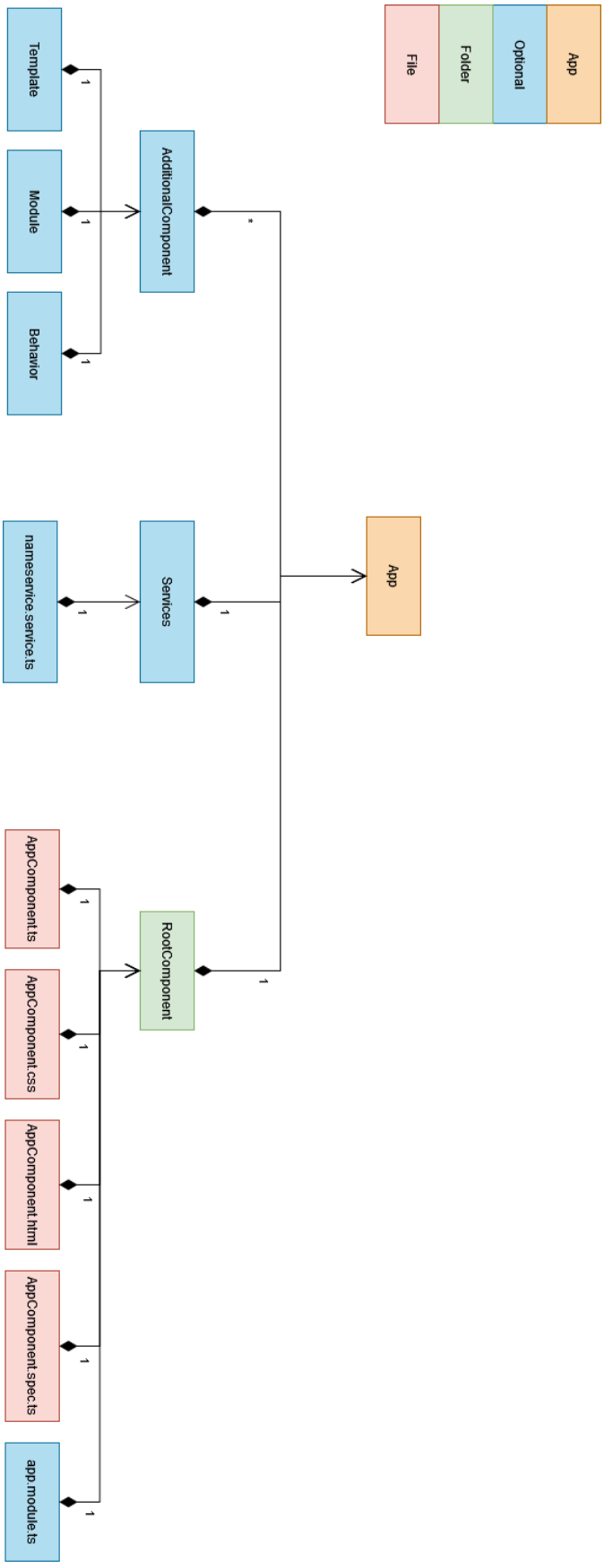


FIGURE 2.12 – ViewPoint App d'Angular

2.3.3 Workflow

WPF

Une fois les modèles terminés, afin de comprendre le fonctionnement des deux Frameworks, nous avons modélisé leur Workflow. Ceux-ci permettent de voir visuellement et conceptuellement le principe de fonctionnement d'un processus.

Sur ce schéma nous pouvons voir le fonctionnement général d'une application WPF. Pour transformer du code en une interface graphique, WPF procède de la facon suivante :

— Dans un premier temps, on extrait les interfaces grâce aux noms des fichiers.
— Pour chacune de ces interfaces, on lit le fichier C# qui correspond, c'est à dire celui qui porte le meme nom, en y ajoutant l'extension ".cs".
— Nous avons maintenant lu la partie logique métier (C#) et la partie vue (XAML). Nous sommes donc capable d'afficher les interfaces.
— Pour finir, WPF charge la première interface qui est renseignée dans le fichier "App.xaml" et l'affiche dans la fenêtre.

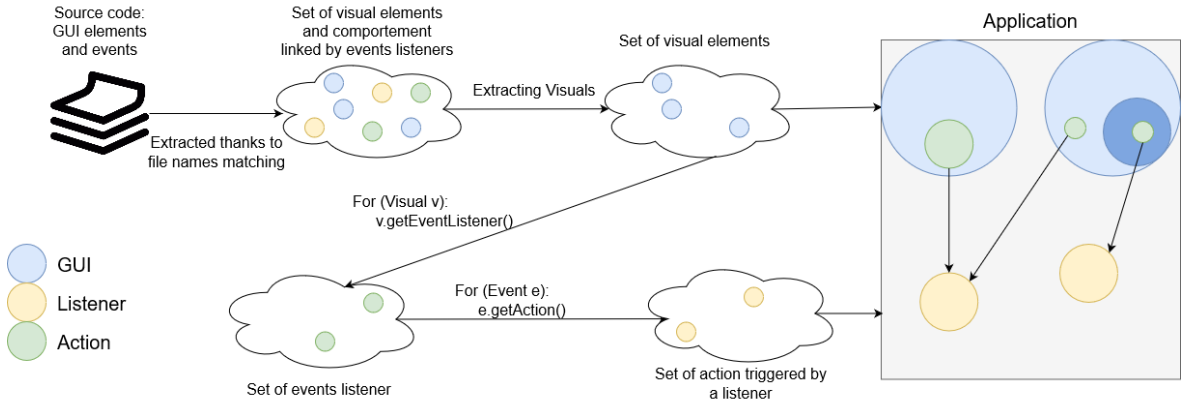


FIGURE 2.13 – Workflow de WPF

Angular

Sur ce schéma, nous pouvons voir le fonctionnement général d'une application Angular. Pour transformer du code en une interface web, Angular procède de la facon suivante :

— Dans un premier temps, on extrait les composants.
— Pour chacun de ces composant, on lit le fichier Typescript qui fait le lien avec la partie "Vue" du composant.
— Nous avons maintenant lu la partie logique métier (Typescript) et la partie vue (HTML / CSS). Nous sommes donc capable d'afficher tous les composants.
— Pour finir, Angular affiche les composants instanciés par le routeur dans la fenêtre web.

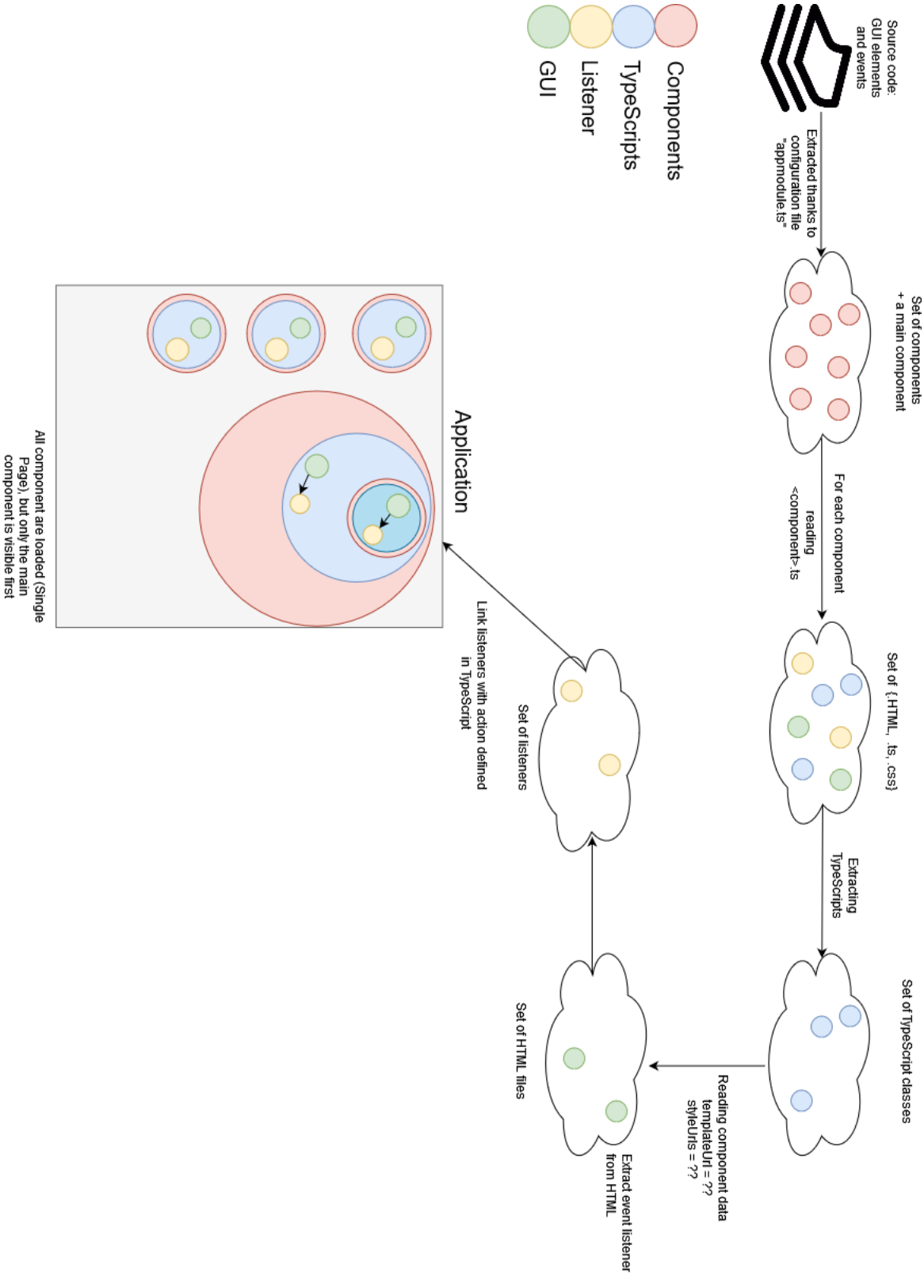


FIGURE 2.14 – Workflow d'Angular

3 Analyse et Conception

3.1 Réflexion

3.1.1 Page WPF vers Composant Angular

L'une de nos principales problématiques était de trouver comment convertir une interface WPF, en une interface Angular. La solution la plus évidente était de convertir chaque page XAML en une page HTML. Chaque interface WPF correspondrait à un composant Angular. L'avantage de cette approche est sa simplicité, en effet, c'est plutôt intuitif de procéder comme ceci, il nous suffit juste de traduire les éléments XAML, en éléments HTML.

Cependant, elle présente un défaut majeur. Il ne sera alors possible d'afficher qu'une seule interface à la fois. Cette contrainte nous est imposée par le caractère single page d'Angular. Or, WPF est un Framework multipage, il est donc envisageable d'afficher plusieurs interfaces à la fois comme nous pouvons le voir dans l'exemple ci dessous :

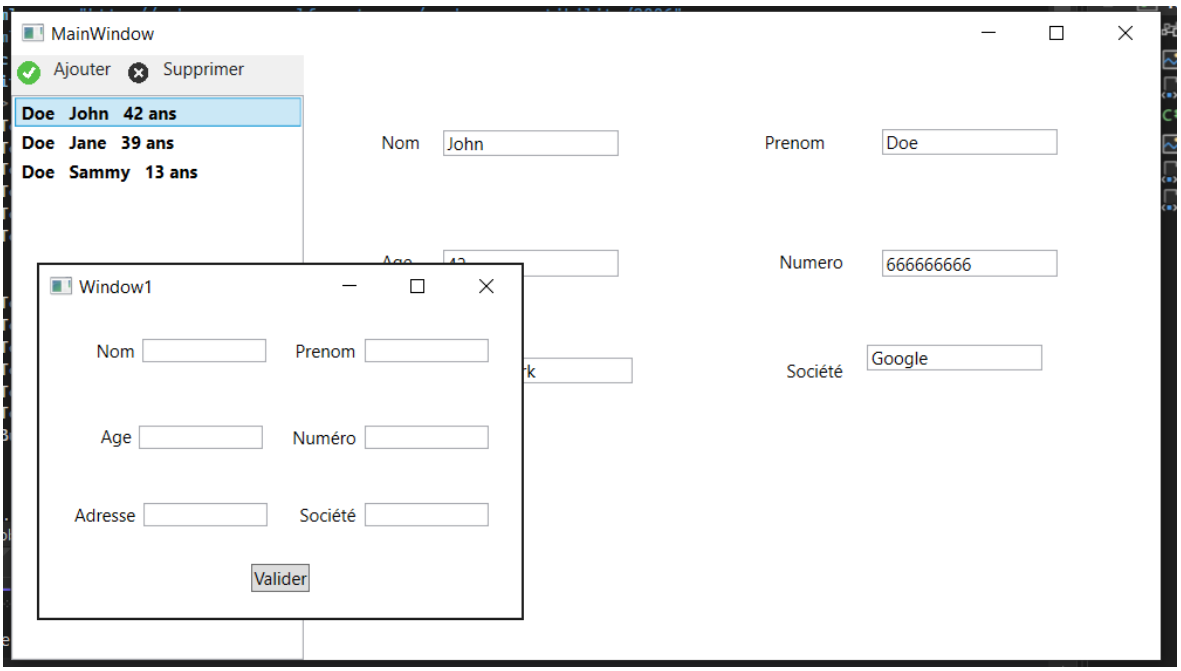


FIGURE 3.1 – Exemple d'une application affichant deux fenêtres à la fois

Ce cas de figure n'est pas traductible en Angular avec notre approche.
Il existe plusieurs cas possible :

— Une seule page est affichée en WPF.
Dans ce cas là, la traduction en Angular est triviale.

— Plusieurs pages sont affichées en WPF, cependant, une seule demande réellement l'attention de l'utilisateur.

Dans ce cas là, nous décidons d'afficher en Angular seulement la page qui nécessite l'attention de l'utilisateur. Et par la suite, quand cette page ne sera plus utile, nous afficherons à nouveau la page principale.

— Plusieurs pages sont affichées en WPF et toutes demandent l'attention de l'utilisateur.
C'est le cas le plus complexe à traiter, en effet Angular ne permet pas vraiment d'afficher plusieurs pages en même temps. Dans ce cas là, nous pensons utiliser un composant Angular qui permet d'afficher des boîtes de dialogue. Ce composant se nomme "Dialog Box". Il permet d'afficher une page dans une autre, cette solution nous permettrait d'afficher deux interfaces à la fois.

Avec cette approche nous sommes capables de traiter tous les cas de figure de composition des interfaces à l'écran. Un seul cas est plus compliqué à traiter mais d'après nos observations, il s'agit d'un cas isolé, rare. Nous faisons l'hypothèse que le dernier cas représente une minorité d'applications, et par conséquent notre approche résout la majorité des cas.

3.1.2 Garder la même interface

Autre question importante dans notre approche de la solution :
- Comment préserver l'interface utilisateur de l'application ?
Car en effet, une des contraintes dans la migration d'une application est de préserver au maximum l'utilisabilité de l'application, c'est-à-dire que les utilisateurs de l'application puissent facilement s'adapter à la nouvelle application après migration.

La principale différence entre les affichages WPF et les affichages en Web, est le scroll, c'est-à-dire la capacité à dérouler la page. Cet aspect est primordial en web, nous le retrouvons dans la majorité des sites web connus. Mais il est largement minoritaire dans le monde des applications de bureau. Pour répondre à ce problème, la migration se faisant de WPF à Angular, nous pouvons considérer que nos applications web ne seront pas scrollables.

L'autre principale différence, concerne le positionnement des éléments. Dans WPF, le positionnement se fait de manière fixé, les composants sont positionnés par rapport à des points fixes, le centre, les bords ou les coins de la fenêtre, en opposition avec le web où chaque élément est placé en fonction de l'emplacement de l'élément précédent.
Répondre correctement à cette problématique est primordial pour conserver l'utilisabilité de l'application, c'est pour cela que nous avons passé beaucoup de temps à réfléchir à plusieurs solutions, à leurs avantages et leurs inconvénients.

3.2 Utilisation d'un Framework pour le template

Il existe de nombreux outils pour faciliter la programmation de sites web. Parmi eux, il y a ceux que l'on appelle les Frameworks CSS. Ils permettent de travailler plus facilement sur l'esthétique de la page web. Pour notre projet, nous avons décidé d'en utiliser un pour l'esthétique des composants Angular. De nos jours, leur utilisation en programmation web est quasiment incontournable.

3.2.1 Quel Framework ?

Parmi tous ceux qui existent, notre choix s'est porté sur MaterializeCSS. Comme tous les autres, il permet de ne pas écrire du code CSS nous même, ce qui constitue, en soit, est un avantage et un gain de temps considérable. De plus, le fait de ne pas écrire de CSS nous permet de centraliser tout le code migré dans deux fichiers, Typescript et HTML.

3.2.2 Quelles limites ?

Cependant, les frameworks CSS sont conçus pour "designer" des pages web, et nous avons aussi besoin, dans notre problématique, de gérer le positionnement des éléments. Pour cet aspect là, nous allons devoir écrire nous même le CSS en utilisant les types de positionnement propres à HTML. Mais pour ne pas éditer un fichier CSS, nous mettons ce code CSS dans l'attribut “style” d'un nœud HTML.

3.3 Les règles de mapping

Pour mener à bien ce projet, nous avons dû longuement réfléchir aux règles de transformation pour obtenir du code Angular à partir de code WPF. Ces règles de transformation (ou mapping) permettent de traduire un framework en l'autre. Concrètement, en suivant ces règles, nous devrions être capable de créer un équivalent d'un code WPF en code Angular, et inversement. C'est une étape clé du projet, car la qualité des règles de mapping se répercute directement sur la qualité de la migration. Nous avons d'abord écrit ces règles dans un fichier Excel, pour les visualiser et surtout pour les améliorer à chaque fois que nous étions confronté à un élément que nous ne savions pas migrer. Nous détaillerons notre approche, qui sépare les règles en deux catégories. Celles qui tranforment l'affichage donc du XAML en HTML, et celles qui transforment la logique métier donc du C# en Typescript.

3.3.1 XAML to HTML

Nous nous sommes intéressés d'abord à la transformation de l'affichage, car cela concerne deux langages qui ne sont pas si différents. En effet HTML et XAML sont des dérivés du XML qui est un langage de formatage des données.

WPF Element		Angular Translation	WPF Element		Angular Translation	WPF Element		Angular Translation	WPF Element
Window		<html></html>	Grid			Button	<input type="button"> ou <button></button>		Unité de mesure
	x:Class						Content	value	px
	xmlns								in
	xmlns:x						Margin	margin	cm
	xmlns:d		TextBlock		<p><\p>		VerticalAlignmen	Materialize: valign-wrapper	pt
	xmlns:mc			Text	p.innerText		Click	(click) ou onClick	
	xmlns:local			HorizontalAlignm	left : 50%		vérifier button		
	mc:Ignorable			Margin	margin				
	Title	import { Title } fr… '@angular/platf…		VerticalAlignmen	top : 50%				
		private titleServic… Title		Name	id=""			class : BTN	
		titleService.setTit… "MainWindow");							
	Height	height=""					HorizontalAlignm	position: fixed;	
	Width	width=""						left: 0.5 * X - (Y / 2)	
TextBox		<input>	vérifier que name soit bien équivalent à id = OUI					X = window.width	
								Y = element.width	
			Menu						
ListView			MenuItem		<button>				
DockPanel		SAIS PAS ENCORE	""						
StackPanel		idem							
DataTemplate		idem							
WrapPanel									

FIGURE 3.2 – Tableau Excel du mapping de XAML à HTML

3.3.2 C# to Typescript

Ensuite, nous avons cherché à trouver des règles de traduction pour convertir du C# en Typescript. Cette partie est bien plus compliquée car ces langages sont bien plus complexes que les précédents. Ici, les règles étants plus compliquées, nous avons décidé de passer par un transpileur qui convertira automatiquement le code C# en Typescript. Il nous suffira de construire un AST et de lui envoyer les morceaux de codes à traduire.

C#		Angular Translation	C#		Angular Translation
window1 = new Window1();		import { Router }			
window1.show();		add "private router : Router" to constructor			
		router.navigate(window1)			
handler parameters					
d-the-source-element-in-an-event-handler?view=netframeworkdesktop-4.8					

FIGURE 3.3 – Tableau Excel du mapping de C# à Typescript

3.3.3 Critique de notre approche

Rapidement, nous avons compris que cette approche était bien plus efficace pour la partie interface graphique que pour la partie logique métier. C'est pour cela que nous avons bien plus approfondie la partie XAML to HTML. Pour la partie C# to Typescript, nous avons tout de même gardé certaines règles car la traduction grâce au transpileur n'est pas forcément exacte. Certains concepts comme l'instanciation de nouvelles fenêtres demandent un traitement particulier.

3.4 Le positionnement des éléments de l'interface

3.4.1 Framework CSS

Materialize, le Framework CSS que nous avons choisi, pourrait être utilisé pour le positionnement des éléments. Il propose une classe "row" qui permet de découper une ligne en 12 colonnes. Cependant, nous avons abandonné cette méthode car elle ne permettrait pas de placer précisément les éléments, et nous avons une contrainte qui est de conserver l'apparence de l'interface graphique. Donc, pour préserver l'utilisabilité de l'application migrée, nous avons décidé d'abandonner cette possibilité.

3.4.2 Utiliser le positionnement Absolute

Utiliser le positionnement "absolute" de HTML pour positionner les éléments selon les mêmes points fixes qu'en WPF.

Cette solution est la plus simple à mettre en place, en effet on utilise un type de positionnement en HTML. Celui-ci permet de positionner tous les éléments en fonction du point haut gauche de la page. La traduction est alors triviale, nous devons juste convertir les points d'ancrage en WPF en distances par rapport au coin haut gauche de l'application. Cela se fait par une simple conversion mathématique et nous permet de conserver les proportions de l'application. Cependant, l'inconvénient de cette solution est que nous n'utilisons pas les schémas d'arborescence d'une page HTML. Dans notre cas, tous les éléments sont positionnés selon l'élément "body".

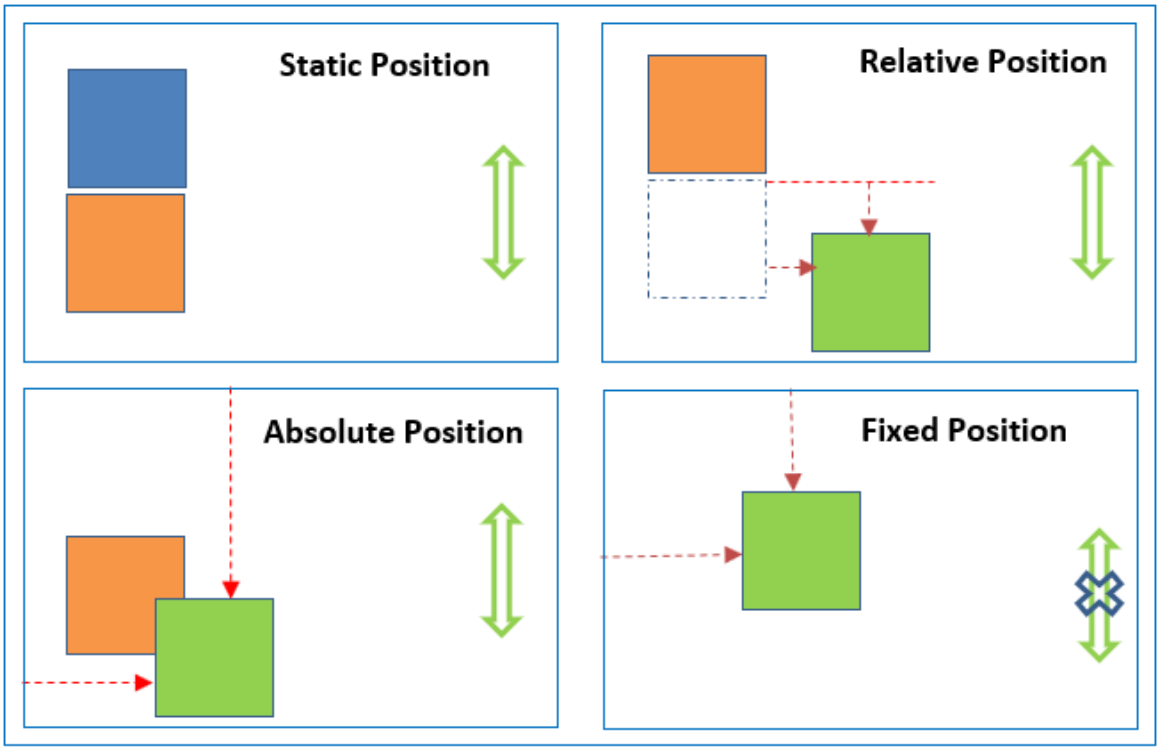


FIGURE 3.4 – Positionnement Absolute HTML

3.4.3 Reconstruire un RelativeLayout

Cette solution est déjà moins évidente, car elle demande de pré-traiter l'interface avant de la convertir en HTML. Ici, l'idée est de trier les éléments dans l'ordre de haut en bas, puis de gauche à droite. Cela permet de les transposer dans l'HTML en gardant la structure arborescente d'une page web. Ensuite, nous devons calculer la distance relative de chaque élément par rapport à celui qui est au-dessus, et celui qui est à gauche. Pour cela, il faut analyser le code XAML. Le problème de cette solution est que, vu que nous pré-traitons le fichier XAML, il est plus difficile d'assurer la conservation de l'esthétique de l'interface.

3.4.4 Construire un modèle de l'interface

Cette méthode est la plus complexe, mais la plus évolutive. Cette méthode a été évoquée en fin de projet, c'est pourquoi nous ne l'avons pas autant approfondie que les autres. Son inconvénient est qu'elle est particulièrement longue à mettre en place.

3.4.5 Critique de ces approches

Nous avons choisi la méthode qui utilise le positionnement absolute, car elle offre un bon compromis et elle est relativement efficace concernant la ressemblance de l'interface. D'un autre côté, en considérant les contraintes de temps du projet, cette approche se met en place assez rapidement.

Cependant nous pensons que la dernière méthode est la meilleure, car elle offre plus d'extensibilité. Cependant, nous n'avons pas eu le temps de la mettre en place.

4 Développement

4.1 Applications WPF

4.1.1 Premières applications

Dans l'objectif de tester notre logiciel qui fera la migration de WPF vers Angular, nous devions avoir des applications WPF à tester. Pour cela, nous avons nous même créé des applications que nous pourrions, par la suite, tester.

Notre première application est une simple application qui affiche un texte HelloWord et un bouton, qui affiche coucou quand on appuie dessus.

FIGURE 4.1 – Première Application WPF

Notre deuxième application est un formulaire dans lequel l'utilisateur fournit des informations concernant une personne. Ces informations sont :

— Prenom
— Nom
— Numero
— Age
— Domaine

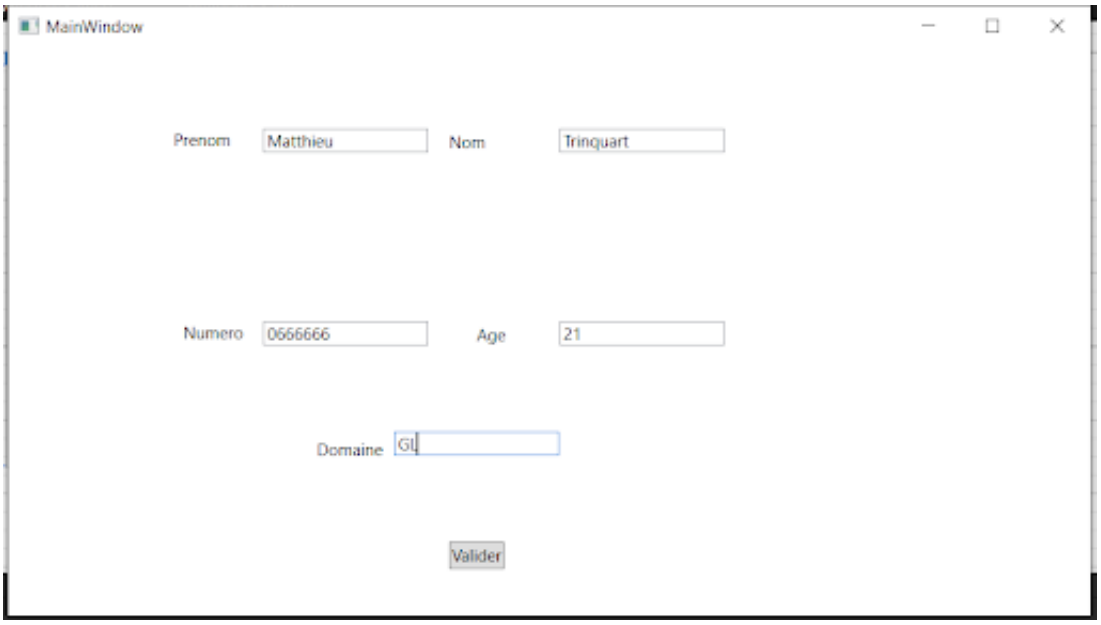


FIGURE 4.2 – Deuxième Application WPF saisi de formulaire

Cette application nous ouvre une deuxième fenêtre avec les informations saisies par l'utilisateur, une fois qu'il a validé son formulaire.

FIGURE 4.3 – Deuxième Application WPF affichage de formulaire

C'est cette application en particulier qui nous a permis de soulever une importante problématique pour la suite. Comment gérer l'affichage de deux fenêtres simultanément sur une seule page Angular ? Nous y avons répondu dans la partie précédente, mais nous tenions à montrer à quel moment précis cette question avait été soulevée.

4.1.2 Améliorations successives

Une fois que nous avions nos premières applications mises en place, nous pouvions tester notre migration (encore manuelle à ce stade du projet). Cependant, le constat est que ces applications ne suffisent amplement pas, pour tester notre migration. Il est évident que tester sur à peine 4 voir 5 éléments graphiques, ne permet pas de couvrir un grand échantillon de Widgets différents. C'est pourquoi, nous avons décidé d'améliorer, au fur et à mesure, ces applications pour couvrir de plus en plus de cas de figures différents. De plus, cela nous a permis de voir certains problèmes que nous n'avions pas eu auparavant, mais aussi les limites de notre migration.

Premièrement, nous avons fait en sorte que la première application affiche « Coucou » sur une page et non sur la même, afin d'essayer différentes méthodes pour faire la transition entre une application MPA vers une SPA (cf. 2.3.1).

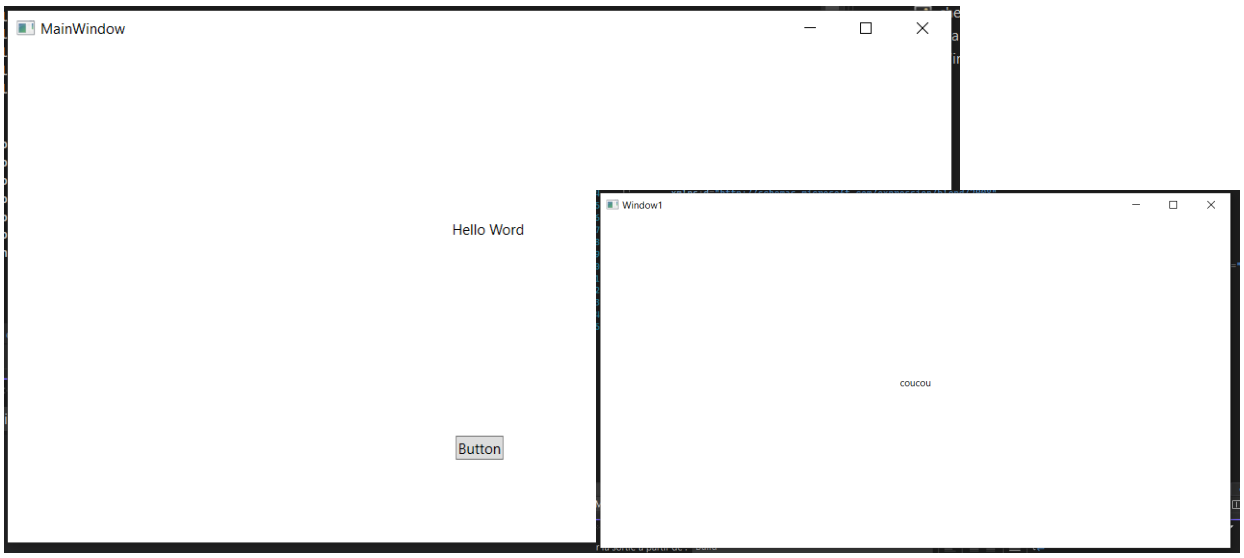


FIGURE 4.4 – Première Application WPF V2

Ensuite, nous nous sommes occupés d'améliorer la deuxième application. C'est celle-ci qui a connu le plus de changements. Elle permet toujours d'ajouter des informations pour une personne mais, cette fois ci, lors de la validation, la personne est enregistrée dans une listView (c'est-à-dire une liste scrollable qui permet d'afficher une liste d'éléments les uns à la suite des autres). D'autres fonctionnalités ont été ajoutées, telles que la suppression d'éléments déjà ajoutés à la liste, ou encore la possibilité de visionner une personne déjà enregistrée.

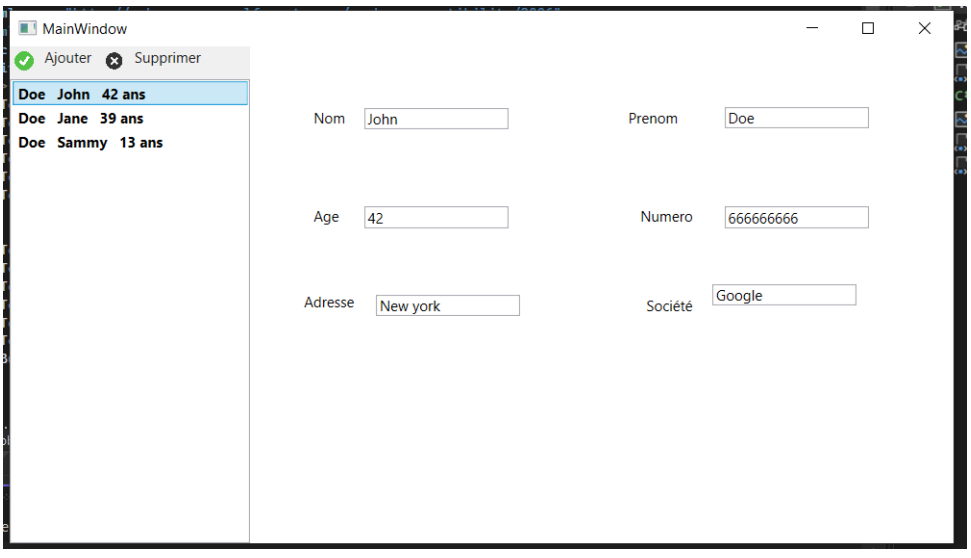


FIGURE 4.5 – Deuxième Application WPF V2 Fenêtre principale

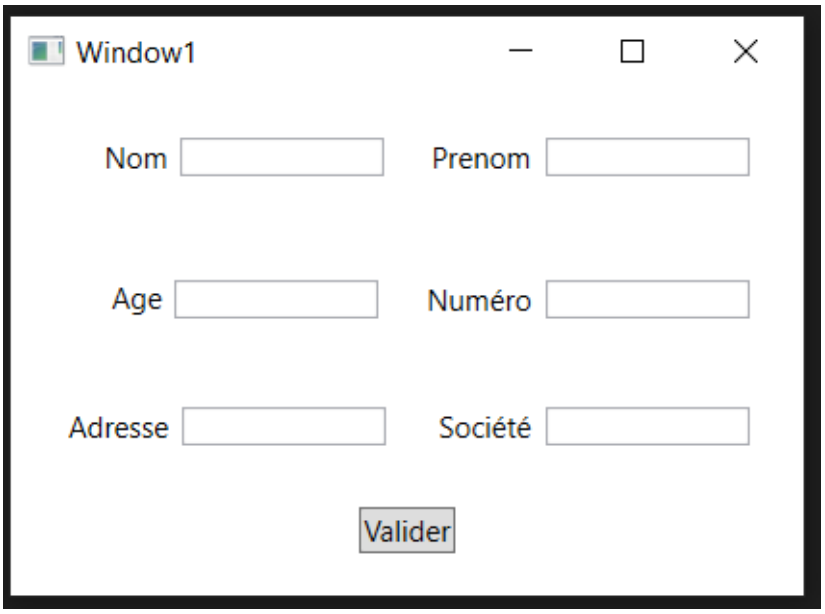


FIGURE 4.6 – Fenêtre d'ajout de formulaire

Bien que nos applications soient intéressantes, avoir seulement deux applications n'est en aucun cas suffisant. C'est pourquoi, une recherche d'applications sur le web s'est imposée, étant donné le manque de temps que nous avions pour créer nous-même plus d'applications. Certains critères devaient être respectés pour les applications. Elles devaient être toutes différentes, de tailles différentes, créées par des professionnels mais aussi par des particuliers, etc...

Le but était d'avoir finalement, un lot de 4 ou 5 applications permettant de couvrir un maximum de cas de figures. Pour cela, nos recherches se sont surtout portées vers Github. Grâce à cela, nous avons obtenu une cinquantaine d'applications que nous avons tripes pour n'en garder que 5. Voici un échantillon des applications trouvées :

— Un morpion (le jeu)
— Un template WPF
— Une application qui capture les Flux RSS d'un site Web
— Un générateur de code QR
— Une calculatrice
— etc ...

Parmi ces applications, voici celles que nous avons gardées. :

— Le jeu de morpion car c'est une petite application mais avec une interface assez complexe et une partie métier simple.
— Un template WPF car c'est une petite application avec une interface assez simple mais une partie métier complexe
— Nous allons aussi utilisé notre application WPF de formulaire car c'est une petite application avec une interface et une logique métier simple
— Une application WPF qui permet créer des diagrammes de gantt. C'est une grosse application avec une interface et une logique métier complexe.
— Une application pour générer des QRCodes qui est une grosse application avec une logique métier et une interface simple.

Voici les différences entre chaque application, sur un tableau :

	Morpion	template WPF	Formulaire	Gantt	QRCodes
Grosse application				x	x
Petite application	x	x	x		
Interface simple		x	x		x
Interface complexe	x			x	
Logique métier simple	x	x	x		
Logique métier complexe				x	x

4.2 Migration manuelle

4.2.1 Déroulement

Avant de mettre en place la migration automatique, nous avons d'abord essayé de la faire à la main en appliquant les règles de transformation (Mapping). Pour cela, nous créons d'abord un projet Angular puis un composant pour chaque fenêtre WPF. Ensuite, nous lisons le fichier XAML de l'application WPF et le traduisons à la main en respectant les règles que nous avions définies préalablement(une TextBox sera remplacée par un <input>, un TextBlock par une balise <p>, etc...). Pour la partie C#, le principe restait le même que pour XAML mais les règles étaient contenues dans un autre fichier.

Ce schéma explique la stratégie globale qui a été mise en place pour réaliser la migration. Le principe est que nous créons un AST du code source de l'application de départ (donc WPF), que nous transformons par des règles de mapping vers un autre AST mais dans le langage cible, pour, finalement, recréer le code de l'application d'arrivée (donc Angular). Par exemple, pour la transformation de XAML vers HTML, nous lisons chaque fichiers XAML de l'application WPF, puis nous les transformons en AST XAML. Une fois cela fait, nous transformons chaque nœud de cet AST pour qu'il corresponde à un nœud d'un AST HTML. De cette façon, l'AST XAML se transforme en AST HTML et il ne nous reste plus qu'à le retransformer en code dans un fichier HTML.

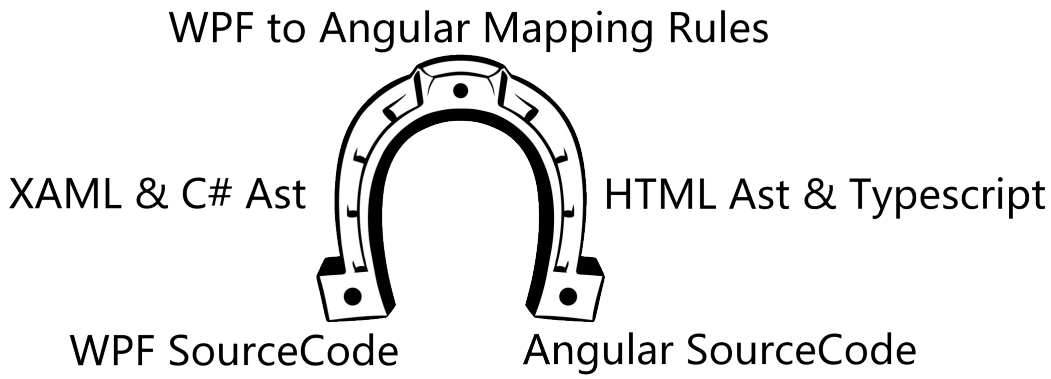


FIGURE 4.7 – Stratégie de Migration

4.2.2 Mise en place de pseudos algorithmes

L'étape suivante pour avancer sur le processus d'automatisation était d'écrire sous forme de procédures, les actions que nous faisions manuellement jusqu'a présent. Réaliser cette étape nous a conduit à un ensemble de pseudos algorithmes, que nous allons présenter ci-après :

Ces première lignes de code en bash nous permettent simplement d'initialiser l'environnement de migration.

```
1                   mkdir Migration
2                   cp <wpfProject> Migration/<wpfProject>
3                   cd Migration
4
```

Listing 4.1 – Première étape : Préparation à la migration

Par la suite, nous initialisons le projet Angular, pour ceci nous clonons un dépot git contenant un projet vide.

```
1                   ng new <migratedWpfProject> --routing --defaults
2
3                   git clone https://github.com/M1-TER-WPFtoAngular/AngularTemplate
4                   //  Vider les HTML par défaut
5                   //  Mettre Materialize dans index.html
6                   cp AngularTemplate/src/index.html
7                      <migratedWpfProject>/src/index.html
8
9                   //  Mettre le routeur outlet dans app.component.html
10                  cp AngularTemplate/src/app/app.component.html
11                     <migratedWpfProject>/src/app/app.component.html
12
13                  rm -rf AngularTemplate
14
```

Listing 4.2 – Deuxième étape : Initialisation du projet Angular

Une petite étape supplémentaire pour créer des variables contenant les chemins vers les différents projets est requise, car cela facilite la suite de la migration.

```
1                   pwd
2                   // Nous sommes dans le fichier Migration/
3                   cd <wpfProject> && $dirWPF = `pwd` && cd ..
4
5                   cd <migratedWpfProject> && $dirANGULAR = `pwd` && cd ..
6
7
```

Listing 4.3 – Troisième étape : Migration du projet WPF

Si nous choisissons l'approche par le positionnement en ABSOLUTE, voici l'algorithme qui va s'exécuter pour s'occuper de l'interface graphique :

```
1               cd $dirWPF
2               //  Pour chaque fichier .xaml
3                   $file = `pwd`
4                   cd $dirANGULAR
5                   ng generate component <nameOfXamlFile>
6               //  Ajouter une route vers ce composant (   /    si racine)
7                   cd /src/app/<nameOfXamlFile>
8                   $component = `pwd`
9
10                  cd $file
11              //  lire et construire  l ast  du fichier XAML
12                  cd $component
13              //  Pour chaque noeud de  l ast
14              //      Générer un identifiant unique : <id>
15              //      Ajouter son équivalent au fichier HTML avec #<id> id=   <id>
16              //      Gérer  l esth étique du noeud (voir 3.2)
17              //      Gérer la logique métier du noeud (voir 3.3)
18
```

Listing 4.4 – Dans le cas du positionnement ABSOLUTE

Même principe que précédemment, mais ici, il s'agit de l'approche par reconstruction d'un relative layout.

```
1               cd $dirWPF
2               //  Pour chaque fichier .xaml
3                   $file = `pwd`
4                   cd $dirANGULAR
5                   ng generate component <nameOfXamlFile>
6               //  Ajouter une route vers ce composant (   /    si racine)
7                   cd /src/app/<nameOfXamlFile>
8                   $component = `pwd`
9
10                  cd $file
11              //  lire et construire  l ast  du fichier XAML
12                  cd $component
13              //  Pour chaque noeud de  l ast
14              //      Calculer sa position dans le fenêtre
15              On transforme le HorizontalAlignment et le VerticalAlignment en pixels
    pour obtenir un positionnement par rapport au coin haut gauche.
16
17              //      Générer un identifiant unique : <id>
18              //  Trier les noeuds dans  l ordre  croissant de positionnement vertical,
    en cas  d  égalité trier par ordre croissant de positionnement horizontal
19
20              //  Pour chaque noeud de la liste triée
21              //      Ajouter son équivalent au fichier HTML avec #<id> id=   <id>
22              //      Gérer  l esth étique du noeud (voir 3.2)
23              //      Gérer la logique métier du noeud (voir 3.3)
24
```

Listing 4.5 – Dans le cas de la conversion en RELATIVE LAYOUT

Dans le code ci-dessous, nous venons recréer l'esthétique de l'application grace aux règles de mapping.

```
//  Dans le fichier TypeScript associé au composant :
//      - Ajouter les import
import { Title } from '@angular/platform-browser';
import { Router } from "@angular/router";
//  - Les instancier dans le constructeur
 constructor(private titleService: Title, private router: Router)

//  - Ajouter dans  l annotation  @Component
 host: {
   '(window:resize)': 'onResize($event)'
 }

//  - Ajouter les attribut privés correspondant à la taille de la fenêtre WPF
 private Height : number = 450;
 private Width : number = 800;

//  - Créer la fonction :
 private refreshGraphicalElements(initialize : boolean) : void{
    . . .
 }

//  - Créer la fonction onResize()
 onResize(event : any){
   this.refreshGraphicalElements(false);
 }

//  - Dans le onInit()
 ngOnInit(): void {
   this.refreshGraphicalElements(true);
 }


//  Pour un noeud donné
//  Ajouter  l attribut
 private <ID> : HTMLElement | null = null;

//  Dans la fonction refreshGraphicalElements(initialize : boolean) : void
   this.<ID> = document.getElementById("<ID>") ;


//  Récuppération et transformation de chaque attribut XAML
   if (this.<ID>) {  // Vérifier que  l  élément  n est  pas null
     if (initialize) {
         // Ici les attributs qui ne seront jamais modifiés
Ex :
         this.<ID>.innerText = "Button" ;
         this.titleService.setTitle("MainWindow");

}
        // Ici les attributs qui seront modifiés au     refresh
     Ex :
    // 1: Liste des CSS (POSITIONNEMENT)
     let style = "" ;
     this.<ID>.setAttribute("style", style) ;

    // 2: Liste des classes Materialize (Esthétique)
     let className = "" ;
     this.BTN1.setAttribute("class", className) ;

   }

```

Listing 4.6 – Positionnement et style des éléments HTML

Cette partie concerne la logique métier, nous utilisons le transpileur pour traduire du code C# en Typescript.

```
1              Pour chaque fichier .xaml.cs:
2              Construire l ast du fichier
3
4              Pour chaque noeud de l ast :
5              Si c est l instanciation d une nouvelle fenêtre:
6                  convertir en router.navigate
7
8              Si constructeur:
9                  convertir et mettre dans le onInit()
10
11             Le convertir à l aide du parser et l écrire dans le fichier TS
       correspondant
12
13
14             Lire le fichier App.xaml (Fichier racine d un projet WPF) ainsi
       que le fichier C# associé
15             Dans celui ci se trouve un attribut StartupUri , il représente la
       première vue affichée
16
17             Nous lisons ce fichier .xaml ainsi que le C# associé (Il est renseigné
       dans l attribut x:Class)
18             En parallèle, nous créons un projet angular (ng new <nom> --routing --
       defaults), avec le module de routing /!\
19
20             Ensuite on vient construire un AST du fichier XAML
21             On le parcourt, et associe chaque élément a une balise HTML, chaque
       attribut a une traduction en HTML / Typescript.
22             On crée un composant Angular, on l ajoute au module de routing (Si
        c est la première vue, l'associer à la racine) et on y applique les
           traductions .
23
```

Listing 4.7 – Transformation des fichiers C# vers des fichiers TYPESCRIPT

4.3 Migration automatique

4.3.1 Parser

Le parser est un outil indispensable pour notre migration. WPF et Angular n'utilisant pas les mêmes langages, nous devons être en possession de parser, afin de nous permettre de traduire le XAML et le C# en HTML et Typescript respectivement. Ils vont nous permettre de transformer le code en AST, que nous pourrons ensuite manipuler.

Il y avait deux solutions dans la création des parsers. Soit nous arrivions à trouver un parser déjà fait sur internet afin de l'intégrer à notre migration, soit nous devions le créer nous même. La première solution était la plus envisageable car elle nous permettait d'économiser un maximum de temps pour la suite.

XAML

Pour le parser XAML, nous avons eu quelques problèmes. Nous avons fait beaucoup de recherches sur internet et particulièrement sur Github, mais malheureusement, parmi tous les parsers que nous avons trouvés, aucun ne fonctionnait correctement avec notre migration. De ce fait, nous avons adopté une autre approche, qui était de rechercher un parser XML et non XAML, car le XAML n'est qu'un dérivé du XML et est donc très proche de celui-ci. Parmi nos résultats, un sortait du lot. Il s'agit d'une approche consistant à créer notre propre AST en utilisant une bibliothèque C# *System.Xml.Linq*. Cette blibliothèque nous permet de parser en AST un fichier et de récupérer chaque balise comme un nœud d'un AST, et chaque attribut de ces balises comme attribut de ce nœud.

Une fois que cet AST est créé, il nous suffit de le parcourir en transformant chaque nœud et ses attributs XAML, en HTML grâce aux règles de mapping. Après cette étape, nous sommes en possession d'un AST HTML et plus XAML.

Voici ci-après le déroulement de l'algorithme pour le parser. Dans un premier temps, nous pouvons voir le contenu du fichier window1.xaml. Dans cet extrait, nous montrons le déroulement sur 3 balises différentes pour des mesures de places mais le fichier en contient d'autres. Donc, nous pouvons remarquer les balises suivantes :

— Window
— Grid
— TextBlock

N'oublions pas que ces balises ont aussi des attributs.

```
<Window x:Class="Test_TER.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Test_TER"
        mc:Ignorable="d"
        Title="Window1" Height="249" Width="344">
    <Grid>
        <TextBlock HorizontalAlignment="Left"  TextWrapping="Wrap" Text="Nom" VerticalAlignment="Top"
                   Margin="[l:]39,[t:]21,[r:]0,[b:]0" RenderTransformOrigin="0.38,0.316"/>
```

FIGURE 4.8 – Fichier XAML pour l'exécution du parser

Maintenant, regardons l'exécution du parser. Nous voyons bien que chaque balise présente au départ a été trouvée et stockée dans l'AST par le parser, et nous pouvons apercevoir clairement que les balises (et leurs attributs) sont indentées de façon que chaque élément fils soit décalé par rapport à son parent. Cela est réalisé par un parcours en profondeur de l'arbre.

```
============ELEMENT XAML============
{http://schemas.microsoft.com/winfx/2006/xaml/presentation}Window
{http://schemas.microsoft.com/winfx/2006/xaml}Class : Test_TER.Window1
xmlns : http://schemas.microsoft.com/winfx/2006/xaml/presentation
{http://www.w3.org/2000/xmlns/}x : http://schemas.microsoft.com/winfx/2006/xaml
{http://www.w3.org/2000/xmlns/}d : http://schemas.microsoft.com/expression/blend/2008
{http://www.w3.org/2000/xmlns/}mc : http://schemas.openxmlformats.org/markup-compatibility/2006
{http://www.w3.org/2000/xmlns/}local : clr-namespace:Test_TER
{http://schemas.openxmlformats.org/markup-compatibility/2006}Ignorable : d
Title : Window1
Height : 249
Width : 344
============ELEMENT XAML============
  {http://schemas.microsoft.com/winfx/2006/xaml/presentation}Grid
============ELEMENT XAML============
    {http://schemas.microsoft.com/winfx/2006/xaml/presentation}TextBlock
    HorizontalAlignment : Left
    TextWrapping : Wrap
    Text : Nom
    VerticalAlignment : Top
    Margin : 39,21,0,0
    RenderTransformOrigin : 0.38,0.316
============ELEMENT XAML============
```

FIGURE 4.9 – Exécution du parser XAML

C#

Comme pour le parser XAML, nous avons d'abord chercher des parsers déjà faits sur internet mais malheureusement, nous n'avons pas trouvé de parser allant de C# vers TypeScript. Nous avons donc essayé de contourner le problème en cherchant un parser C# vers JavaScript, pour ensuite, convertir le JavaScript vers le TypeScript (En supposant que comme le JavaScript est très largement utilisé, un parser de ce genre serait plus facilement trouvable qu'un vers TypeScript). Mais malheureusement, nous n'en avons pas trouvé qui répondait à notre problématique. Nous avons donc dû créer nous-même notre parser, et pour cela il fallait créer l'AST d'un programme C#. Pour ce faire, nous avons trouvé la bibliothèque Roslyn qui permet de créer un AST d'un programme C#.

Ensuite, avec cet AST, nous parcourons l'arbre en profondeur et cherchons à voir si le nœud est une déclaration de méthode, une déclaration de classe, une déclaration de variable etc... Une fois le parcours de l'arbre fait, il nous suffit de traduire chaque nœud par la version en TypeScript, en utilisant les règles de mapping.

Pour appliquer les transformations, nous avons fait un programme C# qui contient plusieurs fonctions comme *addConstructeur(List<Attribut> attributs)* ou *AddMethode(Methode m)* qui permettent de créer des constructeurs ou des méthodes dans un fichier TypeScript d'un projet Angular. Ce programme va nous permettre de faire la transition entre l'AST généré par Roslyn vers le langage TypeScript.

Ci-contre, nous pouvons voir une classe C# correspondante au fichier MaiWindow.cs dans une application WPF. Le but ici est de parser en AST ce fichier en utilisant Roslyn.

```
public partial class MainWindow : Window {
    private List<Personne> items = new List<Personne>();
    private int select;
     Matthieu *
    public MainWindow() {
        InitializeComponent();
        items.Add( item: new Personne( Nom: "John",  Prenom: "Doe",  Age: 42,  NumeroTel: 0666666666,  Adresse: "New york",  Société: "Google"));
        items.Add( item: new Personne( Nom: "Jane",  Prenom: "Doe",  Age: 39,  NumeroTel: 0666666666,  Adresse: "Paris",  Société: "Facebook"));
        items.Add( item: new Personne( Nom: "Sammy",  Prenom: "Doe",  Age: 13,  NumeroTel: 0666666666,  Adresse: "londre",  Société: "Apple"));
        lvUsers.Items.Add(items[0]);
        lvUsers.Items.Add(items[1]);
        lvUsers.Items.Add(items[2]);
    }
     1 usage  Matthieu *
    private void Ajouter(object sender, RoutedEventArgs e) {
        Window1 win = new Window1();
        if (win.ShowDialog() == true) {
            var p = new Personne(win.getNom(), win.getPrenom(), win.getAge(), win.getNumeroTel(), win.getAdresse(),  Société: win.getsoc());
            items.Add(p);
            this.lvUsers.Items.Add(p);
        }
    }
```

FIGURE 4.10 – Fichier C# pour l'exécution du parser

Comme pour le parser XAML, voici, ci-dessous, le résultat de l'exécution de notre parser. Nous pouvons voir que l'ensemble des éléments présents dans le fichier de départ, ont été représentés ici. Par exemple, la variable *select* de type *int*. Dans ce résultat, nous avons une section "DECLARATION VARIABLE" qui nous montre que la variable *select* est bien de type *int* et qu'elle est *private*.

```
namespace : Test_TER
  Classe : MainWindow
    DECLARATION VARIABLE :
        Type : List<Personne>
        Visibility : private
        Variable : items = new List<Personne>()
    DECLARATION VARIABLE :
        Type : int
        Visibility : private
        Variable : select
    CONSTRUCTEUR :
    Nom Constructeur : MainWindow
    VISIBILTER : public
    METHODE :
        Type retour : void
        Visibility : private
        NOM : Ajouter
    DECLARATION VARIABLE :
       Nom paramètre sender
       Type paramètre object
```

FIGURE 4.11 – Exemple d'éxecution de l'AST C#

4.3.2 Script d'automatisation

Quel langage choisir ?

Pour implémenter l'automatisation du processus de migration, nous avons choisi d'utiliser le langage C#. Celui-ci nous impose quelques contraintes de part sa forte corrélation avec Windows, mais nous l'avons choisi car les parsers que nous avons trouvés étaient implémentés dans ce langage.

Structure du code

Sur la figure 4.12 ci-dessous, nous pouvons voir la structure du code dans un dépot Github. Nous devons, tout d'abord, expliquer en quelques mots, la correspondance de certains dossiers et fichiers.

— Migration.exe -> L'exécutable qui permet de lancer la migration. Il attend un paramètre qui est le nom du projet à migrer.
— WpfToAngular -> Le dossier de la migration. Il est créé par le programme.
— _src -> Le dossier contenant les sources du programme.
— martin -> Un projet WPF.

Dans le dossier "_src", nous retrouvons les fichiers ".cs" contenant le code ainsi que deux scripts bash qui permettent de compiler le projet et de l'executer.

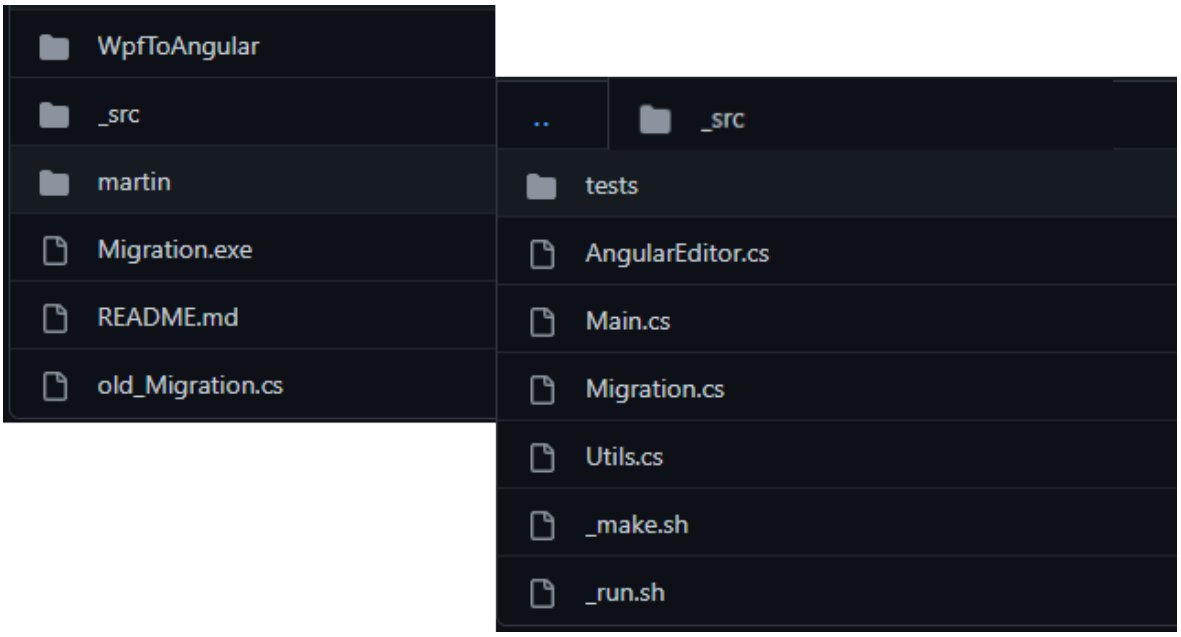


FIGURE 4.12 – Dépot Github du programme de Migration

Voici, ci-dessous (figure 4.13), un diagrame de classe qui représente les composantes du projet. Nous avons regroupé toutes les classes dans un namespace appelé "MigrationTool". Les éléments essentiels du projet sont, la classe "Migration" qui contient toutes les étapes liées à la transformation d'un projet WPF, et la classe "AngularEditor" qui nous permet d'éditer un projet Angular.

Fonctionnement du programme

Ce schéma (figure 4.14) explique le fonctionnement du programme. Nous pouvons voir les différents états des fichiers concernés par la migration. Les cases en jaune représentent ces états en detaillant l'arborescence des fichiers. En bleu, nous avons les méthodes invoquées par la classe "Migration" et en vert, celles invoquées par la classe "AngularEditor".

Dépendances du programme

Notre outil de migration dépend d'un certain nombre d'autres outils. Ceci est lié au fait qu'il a été développé sous Linux. Il n'est donc prévu pour être exécuté que sous celui-ci. En effet, il utilise le bash pour exécuter certaines commandes telles que *git clone*, *ng new* ou *ng generate component*. Nous devons donc avoir installé Git et le AngularCli auparavant.

Pour la compilation et l'exécution, nous utilisons un compilateur C# pour Linux. Pour l'installer, il suffit d'utiliser la commande suivante :

```
1                sudo apt install mono-complete
2
```

Listing 4.8 – Instaler le compilateur C

FIGURE 4.13 – Schéma UML du programme de Migration

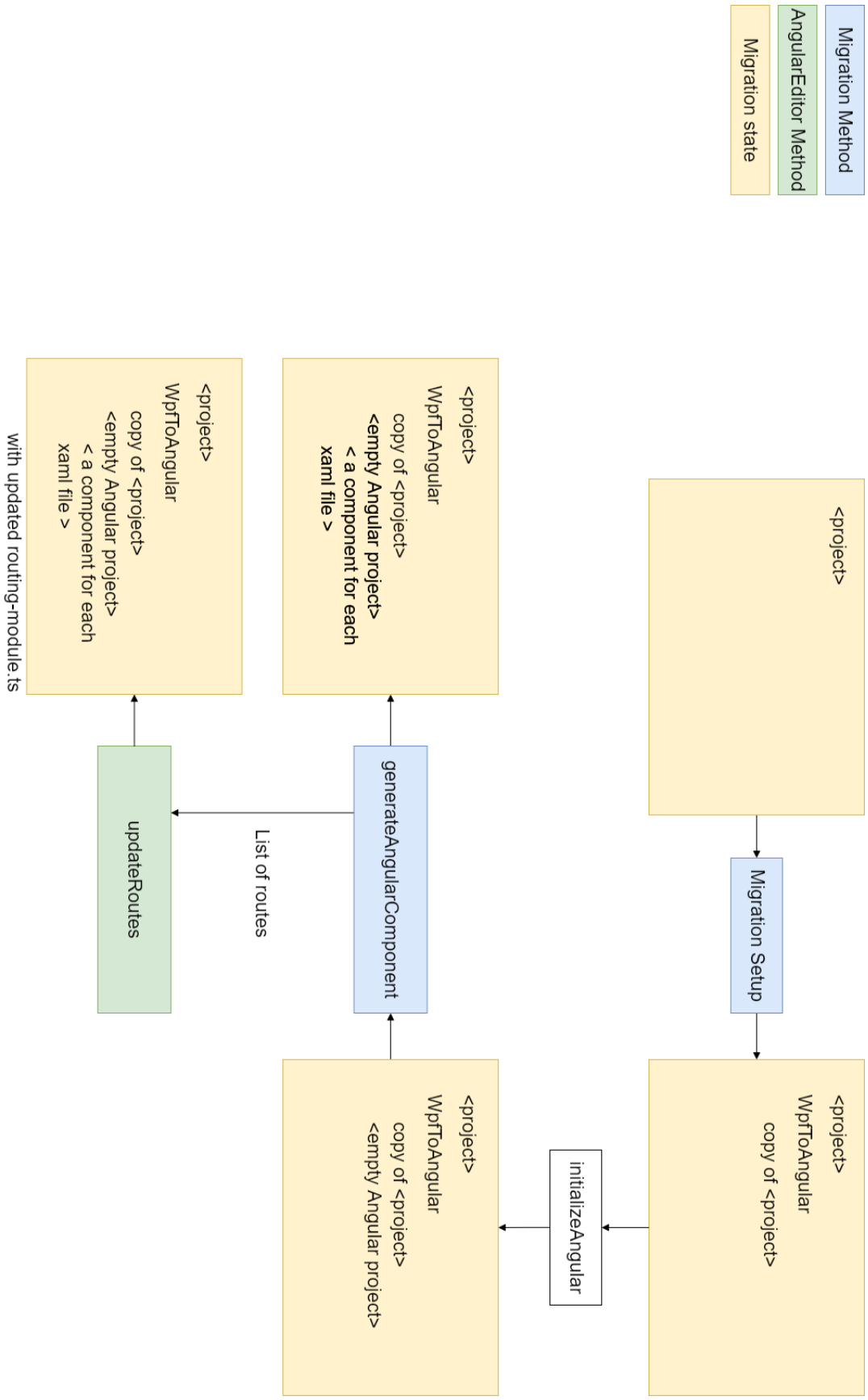


FIGURE 4.14 – Workflow du programme de Migration

5 Expérimentation

Pour juger de la qualité d'une migration, des tests sont forcément obligatoire pour obtenir des statistiques sur les résultats obtenus. Ces test servent à mesurer la qualité de notre outil et à determiner s'il serait plus efficace sur un certain type d'application.

Nos tests serons réalisés sur le jeu d'applications que nous avons trouvé sur le web. Nous y ajouterons les applications que nous avons faites nous-même et une application fournie par nos encadrants.

5.1 Protocole de test

Notre protocole de test sera plutôt simple. Nous avons un jeu d'applications variés et de tailles différentes, sur lequel nous allons exécuter notre script de migration automatique. Nous obtiendrons alors une application Angular correspondante à l'application source en WPF. Nous comparerons ces deux applications selon des critères établis au préalable.

5.2 Critères d'évaluation

Existance des composants

Pour préserver l'utilisabilité de l'application, il est nécessaire de juger si les composants présents dans Angular correspondent effectivement à une fenêtre dans WPF, mais aussi si toutes les fenêtres de WPF sont représentées par un composant Angular.

Positionnement des composants

Un autre point important est de respecter le positionnement de départ des éléments graphiques. Pour cela, nous devons vérifier s'ils occupent la même place dans les deux applications, et si les proportions sont respectées.

Fonctionnement de la logique métier

Pour juger de la qualité de la migration, nous devons respecter le fonctionnement de l'application, c'est-à-dire la correspondance entre les actions effectuées sur l'application source et sur l'application migrée. Les boutons doivent réaliser les mêmes actions, les accès aux données doivent rester identiques... Cela permet d'obtenir une application avec un fonctionnement équivalent.

Apparence de l'interface

Ce point est plutôt équivalent au précédent, mais il concerne l'esthétique. Du point de vue de l'utilisateur, tout ce qui est affiché doit rester identique, donc un utilisateur lambda ne doit pas pouvoir différencier les deux applications de part leur interface, ni leur fonctionnement. Donc les couleurs, les styles... permettent de conserver l'utilisabilité de l'application. En revanche, ce point reste moins important que les précédents, car ce n'est que l'aspect esthétique qui est affecté.

5.3 Tests

Dans ce qui suit, nous allons noter les tests de migration réalisés sur les applications que nous avions choisies. Il y avait donc :

— TestTER
— Tic-Tac-Toe-main
— WPFTemplate-master
— Gantt
— Barcode (l'application de génération de code QR)

Tout les codes sources obtenus lors de nos tests, sont disponibles sur notre dépôt Github.

5.3.1 TestTER

Voici ce que les tests ont donné pour cette application :

— Migration des .XAML : 3 / 3 (correspond aux nombres de fichiers .xaml migrés par rapport à ceux à migrer)
— Migration des .CS : 3 / 3
— Existance des composants : 2 / 2 (Sans compter le composant "racine" App)
— Positionement des composant : 0 / 2 (donc ici, pas de positionnement sur les deux composants)
— Esthétique des composant : 0 / 2
— Logique métier des composant : 0 / 2

Remarques

Sur cette migration, nous avons remarqué que l'outil de migration ne prennait pas en compte les images contenues dans le projet WPF. Cela nous fait un point a améliorer pour la suite : prendre en compte les assets.

5.3.2 Tic-Tac-Toe-main

Voici ce que les tests ont donné pour cette application :

— Migration des .XAML : 5 / 5
— Migration des .CS : 5 / 22
— Existance des composants : 4 / 4 (Sans compter le composant "racine" App)
— Positionement des composant : 0 / 4
— Esthétique des composant : 0 / 4
— Logique métier des composant : 0 / 4

Remarques

Sur cette migration, nous avons remarqué que l'outils de migration ne conserve pas toute la logique métier. En effet, sur cette application, il y a beaucoup de code C#. Ce code n'a pas été migré, car il n'appartient pas à un "module" graphique, il s'agit de code externe qui n'est pas lié à une interface graphique. Cela nous fait un point a améliorer pour la suite : considérer tous les fichiers C#.

5.3.3 WPFTemplate-master

Voici ce que les tests ont donné pour cette application :

— Migration des .XAML : 4 / 4
— Migration des .CS : 4 / 14
— Existance des composants : 3 / 3 (Sans compter le composant "racine" App)
— Positionement des composant : 0 / 4
— Esthétique des composant : 0 / 4
— Logique métier des composant : 0 / 4

Remarques

Encore une fois, l'entièreté des fichiers C# n'a pas été migrée, cependant, un en particulier pose problème car il s'agit d'un fichier d'interface écrit en C#.

5.3.4 Gantt

Voici ce que les tests ont donné pour cette application :

— Migration des .XAML : 30 / 30
— Migration des .CS : 30 / 107
— Existance des composants : 29 / 29 (Sans compter le composant "racine" App)
— Positionement des composant : 0 / 4
— Esthétique des composant : 0 / 4
— Logique métier des composant : 0 / 4

Remarques

Sur cette migration, nous avons vu que notre application résiste aux projets de grande taille. Cependant, encore une fois, les fichiers C# n'ont pas tous été migrés. Les images et les assets en général, comme des fichiers json, xml, n'ont pas été migrés.

5.3.5 Barcode

Voici ce que les tests ont donné pour cette application :
— Migration des .XAML : 2 / 2
— Migration des .CS : 1 / 3
— Existance des composants : 1 / 1 (Sans compter le composant "racine" App)
— Positionement des composant : 0 / 4
— Esthétique des composant : 0 / 4
— Logique métier des composant : 0 / 4

5.4 Etude des résultats

5.4.1 Applications obtenues

Les applications que nous avons obtenues ne sont pas encore conformes à l'application source. Nous devons encore implémenter quelques fonctionalités, notamment le positionnement et l'esthétique qui ne sont pas encore terminés ainsi que la logique métier.

Nous avons relevé quelques disfonctionnements dans notre migration. Par exemple, elle ne prend pas en compte l'entièreté des fichiers C#, ni les assets (Images, Json, Xml). Cependant, les composants sont bien présents, donc, sur ce point, la migration se déroule plutôt correctement.

5.4.2 Durée de la migration

Nous pouvons aussi nous intéresser à la durée de la migration, car en effet, en fonction de la durée, cela peut devenir un problème. Voici la durée des migrations pour les différents tests :

— TestTER : 1 minutes 48 secondes.
— Tic-Tac-Toe-main : 2 minutes 39 secondes.
— WPFTemplate-master : 1 minutes 52 secondes.
— gantt : 4 minutes 34 secondes.
— barcode : 1 minutes 41 secondes.

Dans l'ensemble, ces durées sont raisonnables. Nous pouvons remarquer que plus le nombre de fichiers ".xaml" augmente, plus l'exécution est longue. Cependant, la partie la plus longue de l'exécution est la création du projet Angular avec la commande *ng new* et la commande *npm install*. En effet, ces commandes sont plutôt longue à exécuter, car elles initialisent beaucoup de fichiers et modules.

6 Gestion du projet

6.1 Organisation

Durant notre TER, nous avons appliqué la méthode agile tout au long du projet avec des sprints réguliers (environs toutes les semaines), avec nos tuteurs de projet. Durant chaque sprint, nous montrions à nos tuteurs les avancées faites durant la semaine, ainsi, nos tuteurs nous faisaient des retours sur nos travaux pour que nous puissions les corrigés assez vite. Pendant nos sprints, nous discutions aussi des travaux à faire pour la semaine suivante. Ces sprints permettaient de faire le point avec les encadrants et de nous guider dans la bonne direction pour la continuité du projet. Pour chaque sprint, nous devions en amont faire une synthèse de ce que nous avions fait la semaine et envoyer cette synthèse à nos tuteurs. Elle permettait de gagner du temps durant nos réunions car nos tuteurs savaient à l'avance les travaux faits durant la semaine. Cela nous a aussi servi à gagner du temps dans la rédaction du rapport, car il suffisait juste de synthétiser l'ensemble de ces synthèses déjà écrites. Après chaque sprint, nous écrivions un compte rendu de la réunion afin de bien saisir les étapes à réaliser lors de la semaine suivante, mais aussi pour garder une trace des solutions apportées par nos tuteurs pour les problèmes que nous rencontrions durant notre TER.

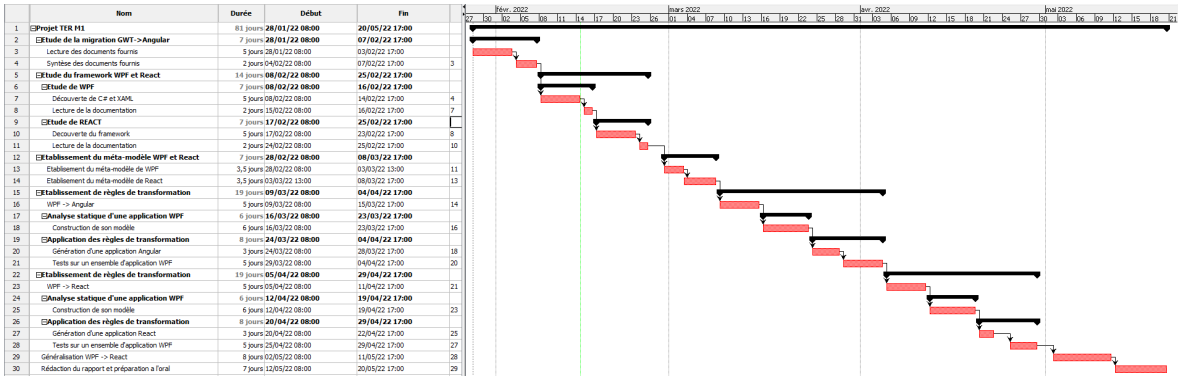
Pour notre TER nous avions choisi Gaetan ROMERO comme chef de projet. Son rôle était de faire la liaison entre nos tuteurs et nous-mêmes, en envoyant chacune de nos synthèses et compte rendus. Il se chargeait aussi de rédiger les mails quand nous devions dialoguer avec nos tuteurs.

6.2 Organisation des tâches

Pour chaque sprint, nous avions une liste de travaux à réaliser pour la prochaine fois. Pour une meilleure productivité, nous nous séparions chaque tâche. Certain membre du groupe se chargeait de la partie Angular, d'autre de la partie WPF ou des scripts de migration.

Au début du projet, nos choix d'organisation étaient de répartir le travail de façon que chaque tâche individuelle soit réalisée par une partie du groupe. Or, cette optique nous permettait seulement de terminer une partie des tâches que nous nous étions fixées de faire, mais pas la totalité car certains membres du groupe se retrouvaient à faire les mêmes parties. Pour remédier à cela, nous avons choisi de changer notre organisation pour que chacun choisisse une tâche, pour laquelle il a une affinité, et soit seul à la faire. De cette façon, notre efficacité fut bien plus importante.

Gantt



	Nom	Durée	Début	Fin	
1	Projet TER M1	81 jours	28/01/22 08:00	20/05/22 17:00	
2	Etude de la migration GWT->Angular	7 jours	28/01/22 08:00	07/02/22 17:00	
3	Lecture des documents fournis	5 jours	28/01/22 08:00	03/02/22 17:00	
4	Syntèse des documents fournis	2 jours	04/02/22 08:00	07/02/22 17:00	3
5	Etude du framework WPF et React	14 jours	08/02/22 08:00	25/02/22 17:00	
6	Etude de WPF	7 jours	08/02/22 08:00	16/02/22 17:00	
7	Découverte de C# et XAML	5 jours	08/02/22 08:00	14/02/22 17:00	4
8	Lecture de la documentation	2 jours	15/02/22 08:00	16/02/22 17:00	7
9	Etude de REACT	7 jours	17/02/22 08:00	25/02/22 17:00	
10	Decouverte du framework	5 jours	17/02/22 08:00	23/02/22 17:00	8
11	Lecture de la documentation	2 jours	24/02/22 08:00	25/02/22 17:00	10
12	Etablissement du méta-modèle WPF et React	7 jours	28/02/22 08:00	08/03/22 17:00	
13	Etablissement du méta-modèle de WPF	3,5 jours	28/02/22 08:00	03/03/22 13:00	11
14	Etablissement du méta-modèle de React	3,5 jours	03/03/22 13:00	08/03/22 17:00	13
15	Etablissement de règles de transformation	19 jours	09/03/22 08:00	04/04/22 17:00	
16	WPF -> Angular	5 jours	09/03/22 08:00	15/03/22 17:00	14
17	Analyse statique d'une application WPF	6 jours	16/03/22 08:00	23/03/22 17:00	
18	Construction de son modèle	6 jours	16/03/22 08:00	23/03/22 17:00	16
19	Application des règles de transformation	8 jours	24/03/22 08:00	04/04/22 17:00	
20	Génération d'une application Angular	3 jours	24/03/22 08:00	28/03/22 17:00	18
21	Tests sur un ensemble d'application WPF	5 jours	29/03/22 08:00	04/04/22 17:00	20
22	Etablissement de règles de transformation	19 jours	05/04/22 08:00	29/04/22 17:00	
23	WPF -> React	5 jours	05/04/22 08:00	11/04/22 17:00	21
24	Analyse statique d'une application WPF	6 jours	12/04/22 08:00	19/04/22 17:00	
25	Construction de son modèle	6 jours	12/04/22 08:00	19/04/22 17:00	23
26	Application des règles de transformation	8 jours	20/04/22 08:00	29/04/22 17:00	
27	Génération d'une application React	3 jours	20/04/22 08:00	22/04/22 17:00	25
28	Tests sur un ensemble d'application WPF	5 jours	25/04/22 08:00	29/04/22 17:00	27
29	Généralisation WPF -> React	8 jours	02/05/22 08:00	11/05/22 17:00	28
30	Rédaction du rapport et préparation a l'oral	7 jours	12/05/22 08:00	20/05/22 17:00	29

FIGURE 6.1 – Architecture Github

6.3 Outils utilisés

6.3.1 Google Drive

Nous utilisions un Google Drive dans notre TER pour nous partager des documents et les stocker. Sur celui-ci, nous déposions chaque compte rendu et synthèse que nous écrivions. Ceux-ci étaient mis à disposition de tous les membres, afin que chacun puisse y ajouter ce qu'il avait réalisé. Lors de chaque réunion, un membre était désigné pour prendre des notes pendant que les autres écoutaient et participaient. De plus, nous stockions tous les fichiers et dossiers utiles au développement de notre projet et qui n'étaient pas des fragments de code, dans ce Drive.

Nous stockions aussi des documents "Draw.io" qui nous permettait de faire tous les diagrammes de classe et les workflows que nous avions besoin. Nous stockions d'autres documents important telles que des mémoires ou d'autres documents qui nous aider pour notre migration.

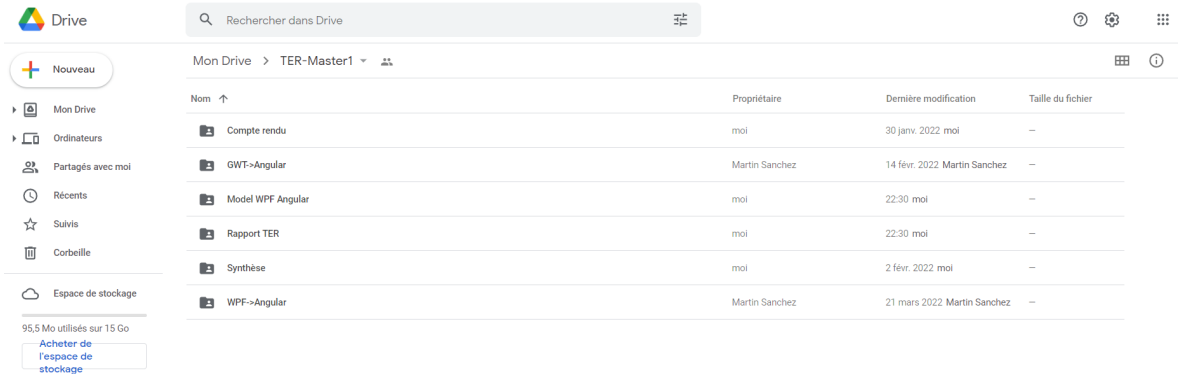


FIGURE 6.2 – Architecture Google Drive

6.3.2 Trello

Pour une bonne organisation de notre projet, nous utilisions trello afin de voir parfaitement où nous en étions dans notre avancée.
Nous avons organisé le trello avec plusieurs rubriques :

— Une rubrique "travail à réaliser chaque semaine".
Cette rubrique contient tous les travaux identiques à chaque semaine comme les comptes rendus, les synthèses ou les dates de réunion.
— Une rubrique "next steps" :
Cette rubrique contient les étapes à réalisées à court terme.
— La rubrique "A faire plus tard".
Cette rubrique contient tous les travaux à faire pour la suite du projet. Chaque semaine, nous glissions, de cette catégorie, les nouvelles tâches vers la rubrique "Next Steps".
— Une rubrique "Terminé"
Cette rubrique contient toutes les étapes déjà réalisées dans notre projet. A chaque étape réalisée, on la glisse de "next steps" vers "Terminé".

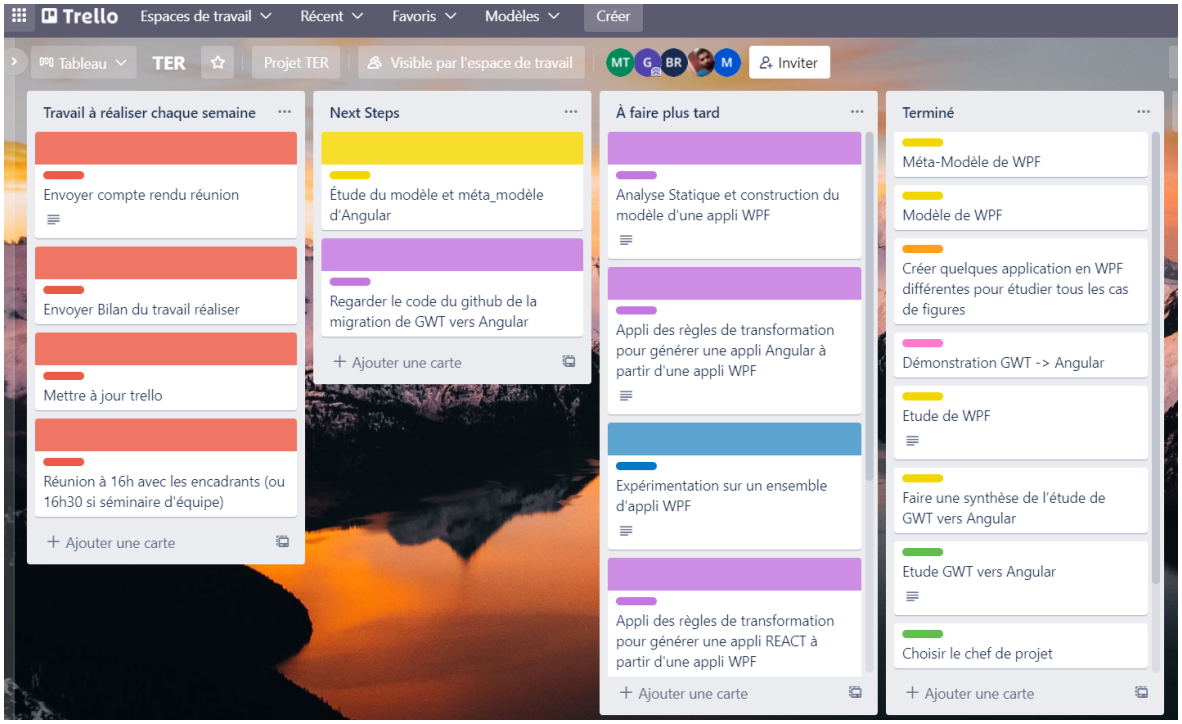


FIGURE 6.3 – Architecture du trello

6.3.3 GitHub

Pour nous partager du code nous utilisions Github. Nous avons créé plusieurs dépôts dans notre organisation.

Un repository(dépôt) AngularApps et un WPFapps qui contiennent respectivement les applications Angular et WPF que nous avions créées, pour prendre en main chacun des Frameworks ainsi que les applications WPF pour tester notre migration.

Le repository WPFParser contient tous les scripts de parser XAML et C# que nous devions implémenter dans notre migration.

Le repository Migration qui lui contient tous les scripts permettant de faire la migration (génération du projet angular, connexion des composant au Route du projet angular etc..)

Et enfin, le repository AngularTemplate qui contient simplement un projet Angular avec le Framework CSS Materialize déjà implémenté dans le projet, ce qui est plus simple pour la génération du projet Angular.

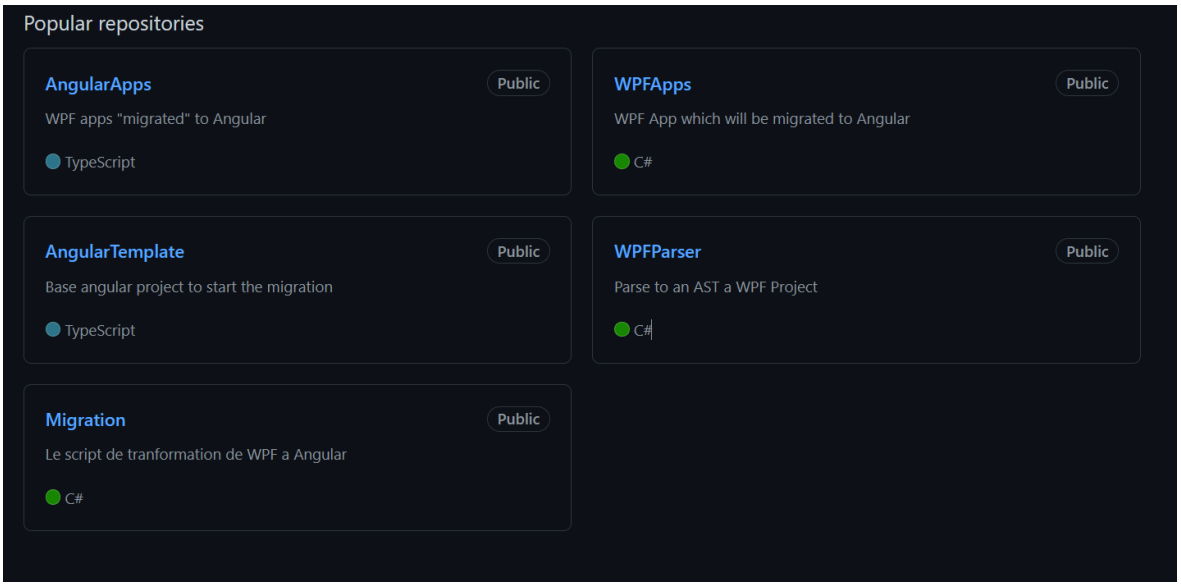


FIGURE 6.4 – Architecture Github

6.3.4 Overleaf

Cet outil nous a servi pour rédiger les synthèses, les comptes rendus et le rapport en LateX. Son utilisation est très utile, car elle nous permet de pouvoir écrire sur différentes parties en parallèle, sans empiéter sur la partie d'un autre.

7 Conclusion

7.1 Conclusion générale

Pour conclure ce rapport, nous tenons à avouer que nous sommes plutôt satisfaits de l'état de notre rapport et de l'avancée générale de notre projet. Lors de celui-ci, nous avons élargi notre compréhension des technologies mais aussi notre façon de gérer un projet collectivement. Nous avons aussi beaucoup appris, ce qui nous sera très utile pour de futurs projets. En effet, comprendre des Frameworks comme WPF et Angular, nous a permis de franchir une étape dans notre façon de penser et de réfléchir lors de l'utilisation de nouvelles technologies, qui étaient jusque-là inconnue pour nous.

Enfin, nous ne prétendons pas avoir terminé le projet, qui aurait mérité quelques semaines, voir mois en plus, mais notre avancement actuel nous montre que ce type de projet qui nous était encore inconnu il y a de cela quelques mois, est en fait réalisable. Nous pensons d'ailleurs le continuer même après la fin de l'année, afin de renforcer nos connaissances sur le sujet.

7.2 Apports sur le plan personnel

Lors de ce projet, nous avons surtout approfondi nos connaissances sur les Frameworks et l'ingénierie des modèles. Ces sujets étaient des sujets que nous n'avions, jusque-là, encore jamais ou très peu rencontrés dans notre formation. De ce fait, cela nous a grandement apporté sur le plan personnel. Nous avons aussi beaucoup appris sur la prise de décision et la gestion de groupe lors de projet réalisé en équipe. En prenant du recul, cela nous a permis d'avoir une autre vision de l'informatique, c'est-à-dire autre que directement développé des programmes ou réalisé des interfaces graphiques. Ici, une grande partie du travail consistait à de la réflexion et de l'apprentissage plutôt que le reste. C'est donc en cela que ce projet diffère des précédents.

7.3 Perspectives

Concernant les perspectives, nous avons encore beaucoup de travail à réaliser, étant donné que notre migration n'est pas encore aboutie. Dans un premier temps, nous aurons à améliorer et rendre opérationnel nos parsers. Ensuite, nous aurons aussi à peaufiner nos règles de mapping pour couvrir encore plus de cas et à rendre la transformation de l'AST en code opérationnelle et aussi automatique.

Une autre étape qu'il manque à notre migration est la possibilité de migrer les assets (donc les image, les fichiers Json...). Il faudrait aussi pouvoir migrer les fichiers .cs qui ne sont que des classes externes et non des fichiers correspondants à une fenêtre.

Bibliographie

[1] M. Begoug Mahi *Approche générique pour la migration de la partie client d'application web en utilisant l'ingénierie dirigée par les modèles : Application à la migration de GWT vers Angular* 2020/2021

[2] Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai,Laurent Deruelle, Stéphane Ducasse, Mustapha Derras *GUI Migration using MDE from GWT to Angular 6 : An Industrial Case* 2019

[3] Benoît Verhaeghe, Nicolas Anquetil, Stéphane Ducasse, Abderrahmane Seriai,Laurent Deruelle, Mustapha Derras *Migrating GWT to Angular 6 using MDE*

[4] Benoit Verhaeghe *Incremental Approach for Application GUI Migration using Metamodel*

[5] *Dépôt GitHub de notre migration*