



SUPELEC - CAMPUS DE METZ

---

## PROJET DE CONCEPTION

Developpement d'une application Windows Phone 8  
pour générer les albums de voyage dynamiquement

---

Promo 2015

Hao XIONG      Min ZHAO

## Table des matières

<b>1 INTRODUCTION</b>	<b>5</b>
<b>2 ANALYSE ET MARKETING</b>	<b>6</b>
2.1 Analyse du besoin . . . . .	6
2.2 Analyse de l'existant . . . . .	6
2.3 Spécialité de notre application . . . . .	6
<b>3 GESTION DE PROJET (SPMP)</b>	<b>7</b>
3.1 Cycle de vie du projet (SDLC) . . . . .	7
3.2 Environnement de développement . . . . .	7
3.2.1 Développement . . . . .	7
3.2.2 Contrôle de versions . . . . .	8
3.2.3 Simulation, test et déploiement . . . . .	8
3.3 Spécification fonctionnelle . . . . .	9
3.4 Livrables . . . . .	9
<b>4 ARCHITECTURE</b>	<b>10</b>
4.1 L'architecture globale . . . . .	10
4.1.1 Architecture sur Windows Phone 8 . . . . .	10
4.1.2 Architecture de simulateur . . . . .	11
4.1.3 Architecture de communication . . . . .	11
4.2 L'architecture MVVM + PCL . . . . .	11
4.2.1 Model View ViewModel . . . . .	11
4.2.2 Portable Class Library . . . . .	11
4.3 Inversion de Contrôle (IoC) . . . . .	12
4.4 L'architecture MVVM + PCL . . . . .	13
<b>5 ALGORITHME</b>	<b>14</b>
5.1 Description générale . . . . .	14
5.2 Contexte de fonctionnement . . . . .	15
5.3 Fonctionnement en machine d'état . . . . .	15
<b>6 APPLICATION</b>	<b>17</b>
6.1 Description générale . . . . .	17
6.2 Navigation des photos . . . . .	17
6.2.1 Vue normale . . . . .	18
6.2.2 Vue par liste . . . . .	18
6.2.3 Vue par carte . . . . .	18
6.3 Visualisation . . . . .	19

<b>7 SIMULATEUR</b>	<b>20</b>
7.1 Description générale . . . . .	20
7.2 Interface graphique . . . . .	20
7.2.1 Contexte de fonctionnement . . . . .	21
7.2.2 Transmission des données . . . . .	21
7.2.3 Contrôle de la carte . . . . .	21
7.2.4 Révocation . . . . .	21
7.3 Simulateur . . . . .	21
7.3.1 Données de voyage . . . . .	21
7.3.2 Simulation . . . . .	22
7.4 Services WCF . . . . .	22
7.4.1 Classe service . . . . .	22
7.4.2 Environnement hôte et points finaux . . . . .	23
7.4.3 Environnement hôte et points finaux . . . . .	24
<b>8 TESTS ET SIMULATIONS</b>	<b>25</b>
8.1 Test unitaires . . . . .	25
8.2 Test de transmission . . . . .	26
8.2.1 Test de connectivité du service . . . . .	26
8.2.2 Test d'information de voyage . . . . .	27
8.3 Simulation . . . . .	27
<b>9 EVOLUTION FUTURES</b>	<b>29</b>
9.1 Classification des photos avec les critères différentes . . . . .	29
9.2 Amélioration d'interface graphique . . . . .	29
9.3 Réingénierie logicielle . . . . .	30
9.4 Optimisation de performance . . . . .	31
<b>10 CONCLUSION</b>	<b>32</b>
<b>11 ANNEXE</b>	<b>33</b>
11.1 Transition entre les états du traitement . . . . .	33
11.2 Traitements pour chaque état . . . . .	34

## Table des figures

1	Cycle de vie de développement - Stage Delivery . . . . .	7
2	Mode de travail en Git . . . . .	8
3	Spécification générale . . . . .	9
4	L'architecture générale du système . . . . .	10
5	Architecture MVVM . . . . .	11
6	Principe de l'IoC . . . . .	12
7	L'architecture générale de l'application . . . . .	13
8	Diagramme UML du fonctionnement du programme principal en architecture PCL . . . . .	15
9	Cycle de la machine d'état . . . . .	16
10	Cycle de vie générale de l'application . . . . .	17
11	Navigation entre les photos . . . . .	18
12	Éléments de visualisation . . . . .	19
13	Interface de simulateur . . . . .	20
14	Génération des données de voyage . . . . .	22
15	La mise à jour automatique de service côté client . . . . .	24
16	Test de connectivité . . . . .	26
17	Test d'information de voyage . . . . .	27
18	Simulation de voyage : état initial . . . . .	27
19	Simulation de voyage : état de traitement des photos . . . . .	28
20	Simulation de voyage : état de génération d'album . . . . .	28
21	Swiss Style visualisation . . . . .	29
22	Animation de voyage . . . . .	30
23	Analyse métrique de codes . . . . .	30
24	Analyse globale de l'application . . . . .	31
25	Analyse détaillée de l'exécution . . . . .	31

## Listings

1	Example d'une classe service WCF . . . . .	23
2	Configuration de hôte WCF . . . . .	23
3	Exemple de test unitaire . . . . .	25
4	Transition entre les états . . . . .	33
5	Traitements pour chaque état . . . . .	34

# 1 INTRODUCTION

Aujourd’hui c'est une ère de l'information, les gens possèdent des dispositifs de types différents plus abondants par rapport au précédent. L'émergence du terminal mobile nous apporte des diverses facilités et confort. Les applications mobiles jouent un rôle très important dans ce processus-là.

Apparemment, les caméras de mobile deviennent de plus évolués et pratiques. Il existe pas mal des applications de photo et de l'image, mais les manières qu'ils ont utilisés pour traiter ou organiser des photos ne sont pas assez riches et amusants. En effet, les applications de ce type n'ont pas été beaucoup évoluées depuis les années. A notre avis, nous pouvons développer une (ou plusieurs) visualisation plus dynamique et intéressante en bénéficiant des informations des photos, du temps, d'emplacement, etc.

C'est la raison pour laquelle nous avons décidé de réaliser une application pour visualiser les photos dans notre projet de cette année. Plus concrètement, c'est une application pour générer un album de voyage dynamiquement. Pour bien dérouler notre projet, nous avons bien réfléchi et analysé de l'environnement du marketing. De plus, nous allons appliquer une méthode professionnelle et adéquate de gestion du projet.

Nous allons aussi concevoir un simulateur fonctionnel et joli, ceci nous permet de réaliser visuellement des simulations et des tests de l'application. En considérant une combinaison extensible des applications sur différents plate-formes (WP8 pour l'instant, WSA8, voire iOS et Android pour le plus loin), nous allons utiliser le framework Portable Class Library. Alors, pour la séparation de l'interface d'utilisateur et la logique fonctionnelle, nous allons implémenter l'architecture Model-View-ViewModel.

La logique principale de l'application sera premièrement implémentée en PCL, plus les réalisations concrètes spécifiques à la plate-forme seront combinées à l'aide du récipient IoC. Pour la première version, l'algorithme peut-être assez simple mais quand même efficace au fonctionnement. Après la réalisation des fonctionnements essentiels, si nous avons encore de temps, nous essaierons d'ajouter les fonctionnements plus riches et d'améliorer la performance d'interface graphique.

## 2 ANALYSE ET MARKETING

### 2.1 Analyse du besoin

Des mobiles dotés d'une caméra sont utilisés manifestement très souvent pendant des voyages grâce à sa portabilité et sa simplicité. Cependant, comme il n'y a pas de l'application efficace pour organiser un grand nombre des photos, l'utilisateur doit soit organiser des photos lui-même, soit parcourir des photos sans règle. C'est la raison pour laquelle il y a vraiment la demande au marché d'une application dédié au besoin des utilisateurs.

D'autre part, en parcourant, partageant et exposant des ensembles des photos ou des albums plus réguliers et créatifs, l'utilisateur peut repasser et profiter d'une expérience visuelle dynamique de son voyage. De plus, les gens préfèrent les applications simples avec peu d'instructions et d'opérations pour obtenir un résultat à l'attente. De cette façon, ils ont plus de temps à jouir de leurs voyages.

### 2.2 Analyse de l'existant

Les applications concernant des photos qui existent :

- Traitement d'image/photo, ex. **Nokia Creative Studio, Snapseed, Photshop** ;
- Gestion des photos, ex. **iPhoto, Nokia Camera, Tidy** ;

Mais il n'y a pas beaucoup d'applications qui réalisent le fonctionnement permettant d'organiser automatiquement des photos de voyage et d'générer des albums avec des modes de visualisation innovante.

### 2.3 Spécialité de notre application

Créativité : Interaction créative entre l'utilisateur et des photos. C'est-à-dire que les actions d'utilisateur (ex. son trajet de voyage) peuvent apporter des variations intéressants de la visualisation des photos.

Simplicité d'utilisation : Afin de simplifier autant que possible des opérations d'utilisateur, notre application cible des fonctionnements plus originaux et plus efficaces. En effet, la classification des informations et l'organisation des photos peuvent être effectué avec automation en utilisant les algorithmes.

Pour une automation plus complète, nous décidons d'implémenter un agent du fond. Dès la fin de voyage, l'application va avertir l'utilisateur qu'un album a été créé. Ensuite, l'utilisateur peut le partager via les médias sociaux ou le visualiser dans notre application.

### 3 GESTION DE PROJET (SPMP)

#### 3.1 Cycle de vie du projet (SDLC)

Le modèle utilisée est le cycle **Staged Delivery**. Il permet d'assurer un candidat stable à la fin de chaque itération. Cela rende aussi la conception plus transparente et robuste contre des risques.

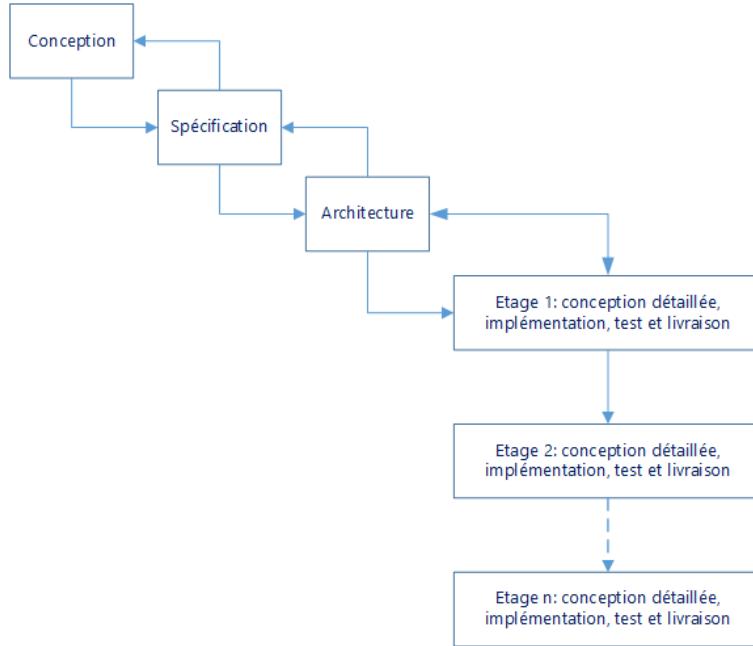


FIGURE 1 – Cycle de vie de développement - Stage Delivery

#### 3.2 Environnement de développement

##### 3.2.1 Développement

Le développement sera réalisé en .NET (C#) en utilisant **Visual Studio 2013 Ultimate** avec **Blend 4.0** pour améliorer les interfaces graphiques. Grâce à cette IDE, il est possible d'effectuer en même temps :

1. Architecture en UML
2. Implémentation des business logic
3. Implémentation de l'interface graphique
4. Tests unitaire et intégration
5. Test des performances

Les langages principales utilisés sont **C#5.0** et **XAML**.

### 3.2.2 Contrôle de versions

**Github** est utilisé pour gérer les versions de sources. Le mode de développement en adaptant la gestion des versions de Git est le suivant :

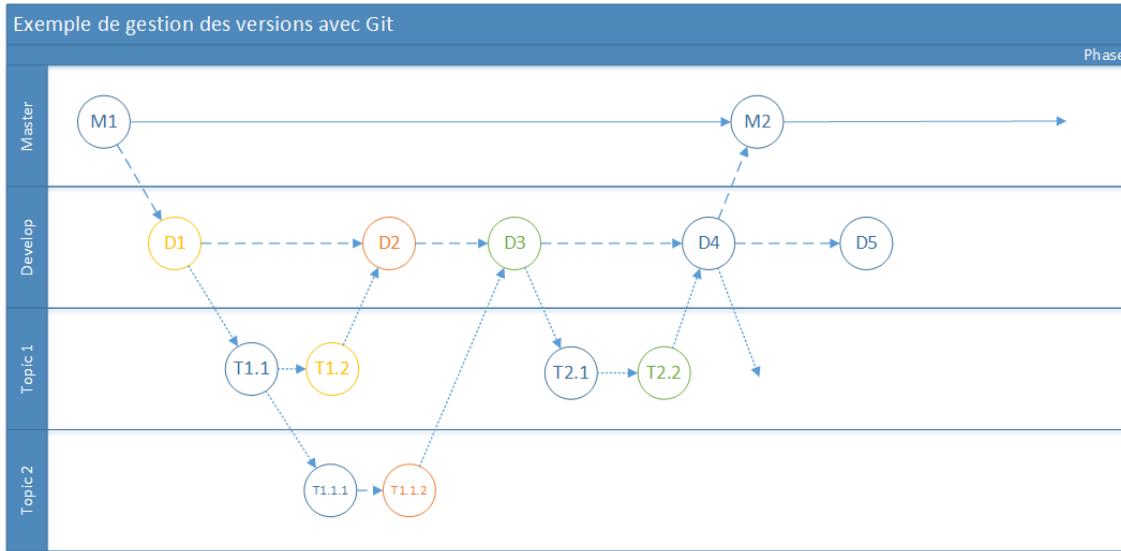


FIGURE 2 – Mode de travail en Git

Le branche **master** est maintenu pour les codes stables. Les branches **develop** est utilisés pour le développement (peut être instable partialement). Les branches **topic** est utilisés pour le développement des nouveaux fonctionnements.

### 3.2.3 Simulation, test et déploiement

Le test générale de l'application sera faite avec l'**émulateur Windows Phone** d'IDE ou un mobile **Lumia 1020**. L'application finale sera publiée sur le Windows Phone Store.

Car notre application fonctionne pendant tout le voyage, il faut créer un environnement assez complet pour pourvoir simuler toutes les possibilités. Une description complète de simulateur est mentionnée dans la section "Tests et simulation".

### 3.3 Spécification fonctionnelle

Après l'analyse du marché et du besoin, nous fixons une liste de spécification suivante. Cette liste n'est pas trop détaillée car nous allons l'adapter et la spécifier pendant chaque étage de développement (Stage Delivery).

Fonctionnement	Sous-fonction	Spécification
<b>Architecture</b>	UI et business logic Portabilité	Utilisation de <b>Model View - View - Model</b> . Utilisation de <b>Portable Class Library</b> .
<b>Navigation</b>	Générale Navigation des albums Navigation des photos	Simple et efficace. Plusieurs possibilités: par carte, par temps et par liste. Manuel (interactive) et automatique.
<b>Stockage</b>	Stockage cloud Stockage local	Interaction avec le service Skydrive. Stockage isolé de WP8.
<b>Visualisation</b>	Animation Design	Animation créative d'exposition Design créatif et efficace (SWISS Style)
<b>Algorithmes</b>	Analyse d'itinéraire Classification des photos	Détection de voyage. Robuste face à la qualité de connexion. Algorithme rapide et robuste. Sensible à l'utilisation des ressources.
<b>Test et simulation</b>	Voyage virtuel Communication Tests	Simulation d'itinéraire de voyage Génération des photos virtuelles de voyage. Interface interactive et complète. Synchronisation des données rapide en temps réels Serveur local mode Duplex entre équipements Test unitaire et intégration sur WP8

FIGURE 3 – Spécification générale

### 3.4 Livrables

A la fin du projet, les livrables attendus devant être produits par l'équipe :

- Bibliothèque portable concernant des business logic (API PCL)
- Application Windows Phone 8 concernant l'interface UI et le système d'informatique (XAP).
- Simulateur de voyage (Windows Form)
- Documentation SandCastle (HTML)

## 4 ARCHITECTURE

### 4.1 L'architecture globale

Le système global consiste en 4 parties : **application principale**, **application des test**, **l'agent du fond** et le **simulateur**. Leur relation est donnée par le schéma ci-dessous :

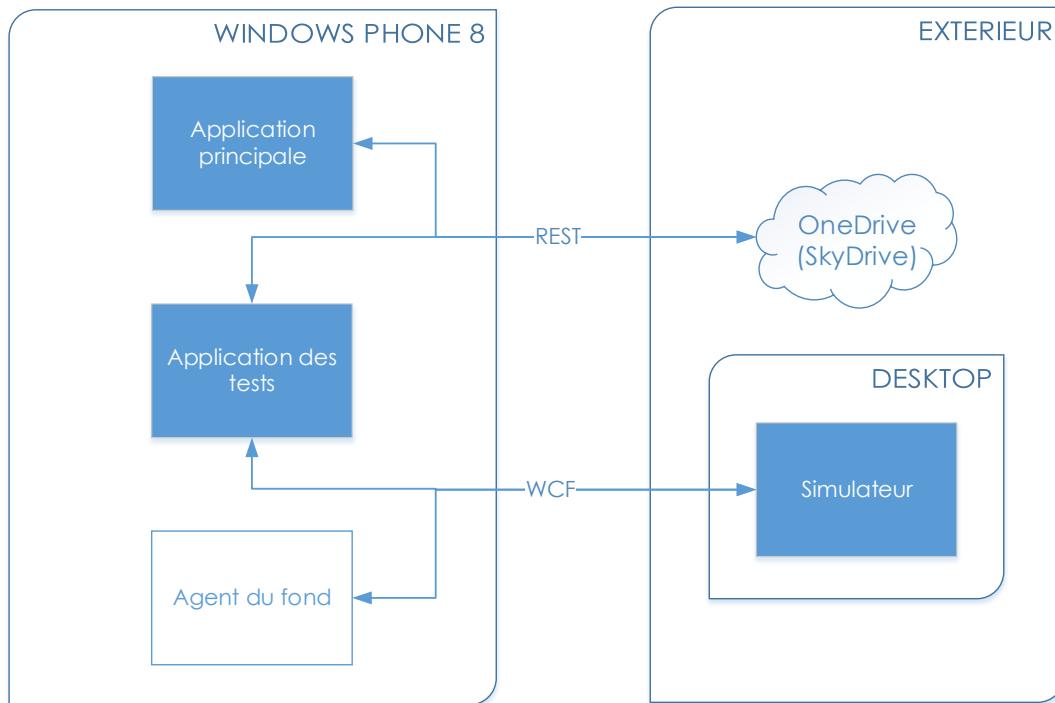


FIGURE 4 – L'architecture générale du système

- Application principale : visualisation des albums.
- Application des test : pilotage et communication avec le simulateur.
- Agent du fond : exécution de l'algorithme.
- Simulateur : simulation de voyage et tests de fonctionnement.

#### 4.1.1 Architecture sur Windows Phone 8

L'application principale, l'application des tests ainsi que l'agent du fond sont réalisées dans l'environnement Windows Phone 8. Pour les deux applications, l'**architecture MVVM + PCL** est utilisée pour une meilleure portabilité et séparation du design et codage. Nous montrons en détail cette architecture dans les sous-sections suivantes.

#### 4.1.2 Architecture de simulateur

Le simulateur consiste en deux parties : **interface graphique** et **service de simulation**. Nous l'aborderons en détail dans la section "Simulateur".

#### 4.1.3 Architecture de communication

Il existe deux types de communication dans le système : **REST** pour la partie synchronisation en Cloud et **WCF** pour l'échange des données avec le simulateur.

Grâce à l'API de SkyDrive, **LiveSDK**, nous pouvons facilement effectuer les enquêtes REST au service SkyDrive. L'authentification est gérée par l'API pour rassurer la sécurité de compte. Une fois authentifiée, l'application peut effectuer les échanges avec le service Cloud SkyDrive en fond.

### 4.2 L'architecture MVVM + PCL

#### 4.2.1 Model View ViewModel

**Model View ViewModel** est une dérivé de **Model View Controller** pour générer la partie interface et la partie logique. Comme MVC, MVVM offre un couplage tellement faible qu'une modification de la interface graphique n'aura aucune impacte sur la partie logique et vice versa. Autrement dire, le Design et le Développement sont complètement isolés.

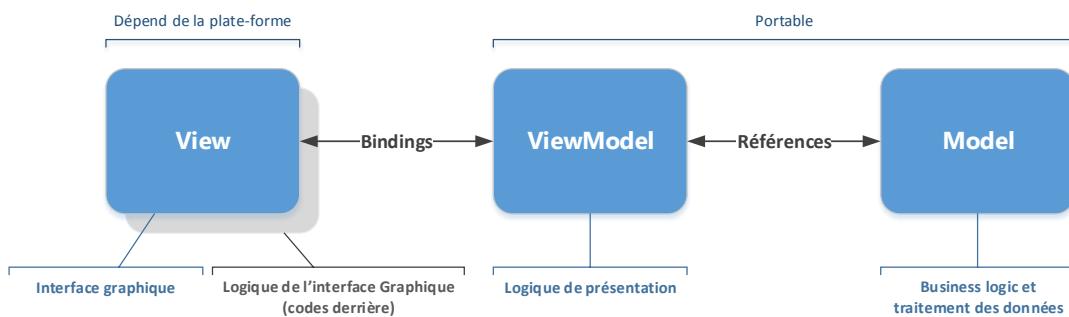


FIGURE 5 – Architecture MVVM

Cependant MVVM profite beaucoup du framework .NET. Inspiré de la conception **Binding** et **Event** de .NET, MVVM donne une architecture plus sensible au couplage des ressources interface graphique et les données cachées derrières.

#### 4.2.2 Portable Class Library

Le **Portable Class Library** est une API .NET qui permet de créer des applications portables. En utilisant le projet type PCL dans Visual Studio, une sous-partie portable des APIs .NET sont référencées. Elles sont portables en plusieurs plate-formes comme Windows Phone, Windows 8, voire Andriod, Linux et IOS. En effet, référencées par un autre projet de type spécifique, un compilateur va transformer les codes PCL en code spécifique du projet.

### 4.3 Inversion de Contrôle (IoC)

Avec les deux architectures mentionnées précédentes, nous arrivons à créer un système structuré et portable. En revanche, il reste de faire la connexion entre les environnements différents (PCL et les plate-forme spécifique).

L'**inversion de contrôle (IoC)** est un patron d'architecture commun à tous les frameworks. Il fonctionne selon le principe que le flot d'exécution d'un logiciel n'est plus sous le contrôle direct de l'application elle-même mais du framework ou de la couche logicielle sous-jacente.

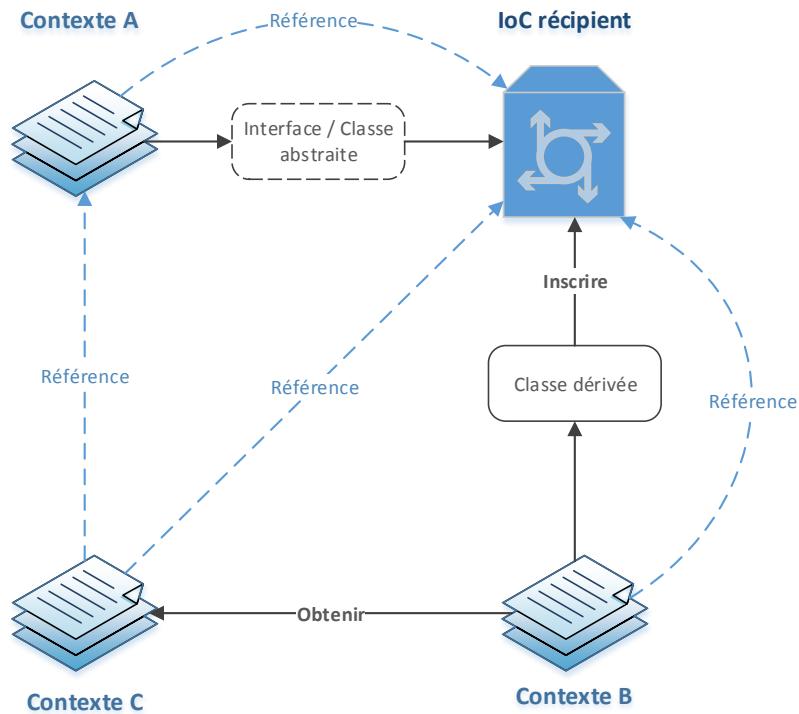


FIGURE 6 – Principe de l'IoC

Prenons un exemple suivant, le programme A a déclaré une interface qui est référencée dans programme B. Pourtant la réalisation est effectuée dans le programme B. Or le programme C n'intéresse pas comment est réalisée l'interface. Par l'inscription (injection du programme B au récipient IoC), l'IoC conserve une instance de classe concrète. En utilisant le **Service Locateur**, le programme C peut facilement retirer l'interface sans connaître l'instance réelle. Comme ça, la contexte B et C sont isolées.

## 4.4 L'architecture MVVM + PCL

Nous allons implémenter le logique générale en PCL. Les implémentations concrètes spécifiques au plate-forme vont être injectées à l'aide du récipient IoC.

Pour ce projet, seulement les parties PCL et WP8 seront réalisées. Pourtant, l'extension à Windows Store Application est assez simple grâce à l'architecture PCL + MVVM. Il faut juste implémenter les interfaces selon les contraintes spécifique du plate-forme WP8 ainsi que les composants graphiques.

Nous modélisons l'architecture générale de l'application à l'aide du diagramme des couches dans Visual Studio.

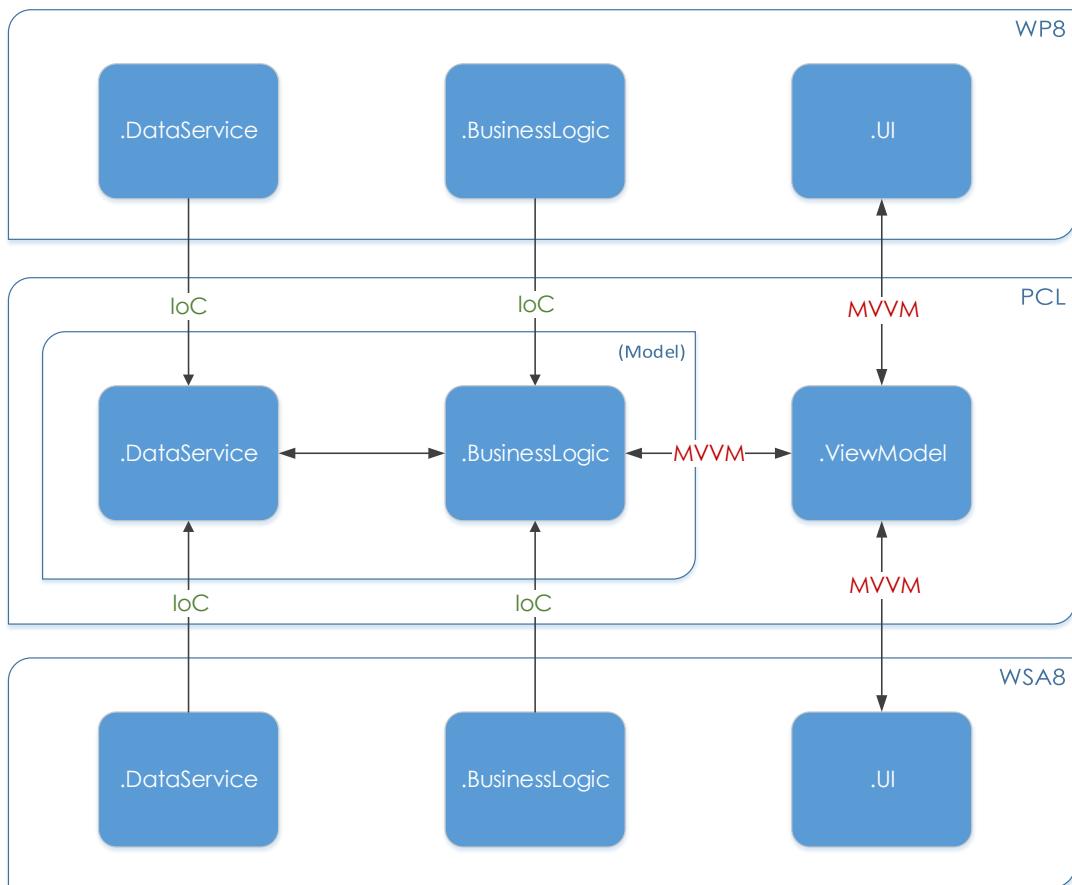


FIGURE 7 – L'architecture générale de l'application

## 5 ALGORITHME

### 5.1 Description générale

L'algorithme principal de l'application est développé en architecture PCL. Afin de bien structurer nos programmes et afin que l'implémentation spécifique à la plate-forme WP8 ou l'extension d'application à Windows Store 8 seront facilement réalisable, nous rangeons des différents comportements d'objet à des différent classes ces qui sont bien abstraits. Les étapes du fonctionnement et les différent tâches des classes ou interfaces sont expliqués comme le suivant :

Le processeur arrière-plan (**BackgroundProcessor**) joue un rôle le plus important dans le processus d'exécution car il contrôle tous les comportements entre les autres classes(ou interfaces) et lui-même.

Le service web (**IWebService**) peut obtenir des informations de l'emplacement de l'utilisateur et les transmet à Le processeur. Il contient aussi des méthodes pouvant obtenir des informations de GPS pointé par **Bing Maps REST Service**.

**DataManager** est une classe dont la tâche consiste à sérialiser des données locales, plus les enregistrer en nuage (OneDrive pour Windows) et vice versa. D'ailleurs, elle contient des méthodes qui peuvent être appelés pour chargement et déchargement des données entre le processeur et lui-même.

Après la réception des données actuels de l'utilisateur et des anciens données provenant de base de données, le processeur va déterminer à quelle état il est se situé. Si cette fois, le processeur est dans l'état **PhotoHandlerState**, il effectuera des opérations suivantes pour traiter les informations des photos. (Le fonctionnement en machine d'état est expliqué par la suite)

**IPhotoManager** se charge de tester l'existence des nouveaux photos qui sont pas encore traités, et si oui, il procède au traitement ces nouveaux photos. Nous utilisons une délégation ici afin de séparer la logique d'traitement de la méthode appelé. De cette façon, **IPhotoManager** peut réaliser sa tâche sans connaître de précisément de autres objets et sans accéder des ressources supplémentaires.

Une fois que le processeur trouve du nouveau photo, il appelle la méthode d'**IExifExtractor** pour tirer les données intéressants de GPS et les repasser à **IWebService** pour obtenir finalement des informations d'emplacement.

Attention, toutes les opérations ci-dessous ne sont pas sûrement terminées dans une fois d'exécution. Heureusement, les opérations qui ne sont pas terminées sont continuées à être traitées à la prochaine fois d'exécution. En effet, après avoir testé l'efficacité d'exécution, le résultat prouve que notre application se déroule sans problème.

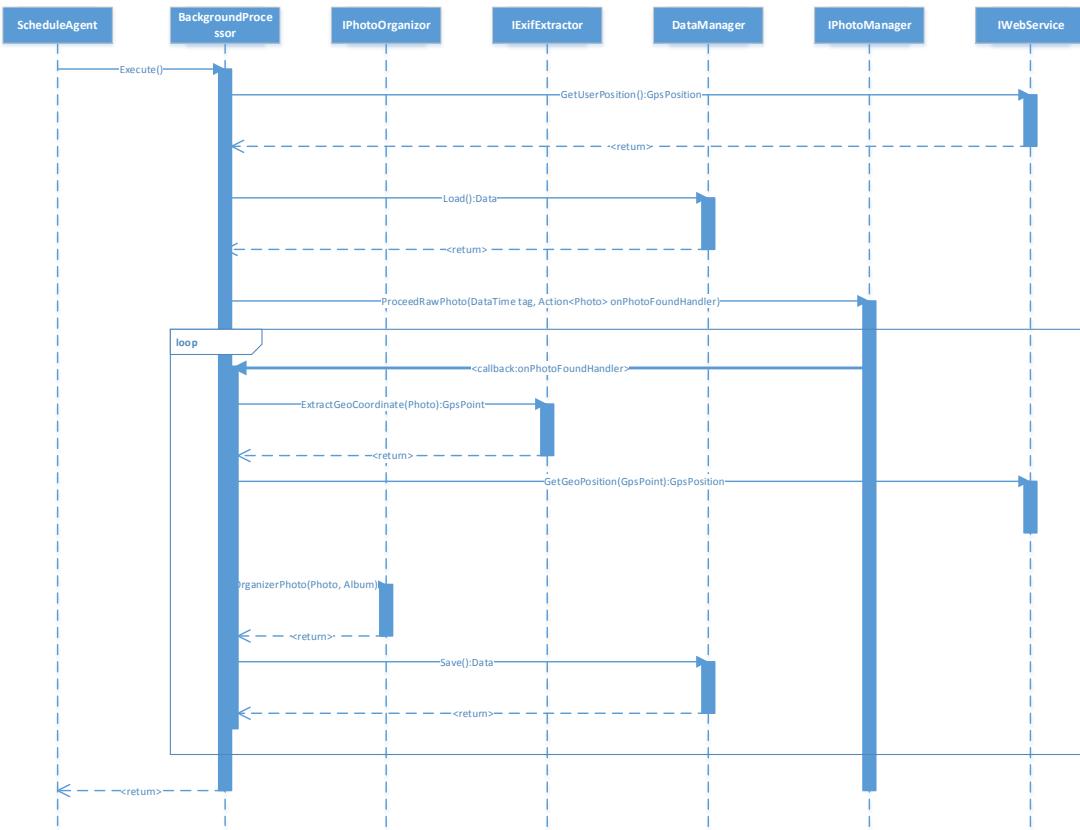


FIGURE 8 – Diagramme UML du fonctionnement du programme principal en architecture PCL

## 5.2 Contexte de fonctionnement

Nous avons déjà vu le déroulement principal du programme, mais toutes les tâches ci-dessous doivent être exécutées sous une condition très importante. Windows Phone propose un système différent de gestion du multitâches. Ce n'est pas du multitâches à proprement parler, mais la possibilité de faire des choses de manière périodique en tâche de fond lorsque l'application est désactivée. Il s'agit des **Background Agent**, qui sont de tâches qui s'exécutent en arrière-plan. Les tâches que nous utilisons pour notre application sont les tâches périodiques. Une tâche périodique doit être exécutée assez rapidement et doit faire quelque chose de simple, comme mettre à jour de nouvelles photos et tirer des informations de GPS dans notre cas. Ces tâches sont exécutées toutes les 30 minutes, sont limitées en nombre par téléphone (cela dépend de la configuration du téléphone) et ne dépassent pas 25 secondes d'exécution.

Nous utilisons ce système pour notre application non seulement pour la besoin de fonctionnement périodique, mais aussi pour la simplicité des tâches étant souhaitées à effectuer. Pendant l'implémentation, nous avons essayé d'utiliser des tâches asynchrones mais il y avait pleine de bugs et quand nous les modifiions en manière synchrone, nous avons trouvé ceux-ci sont plus faisable et assez efficace.

## 5.3 Fonctionnement en machine d'état

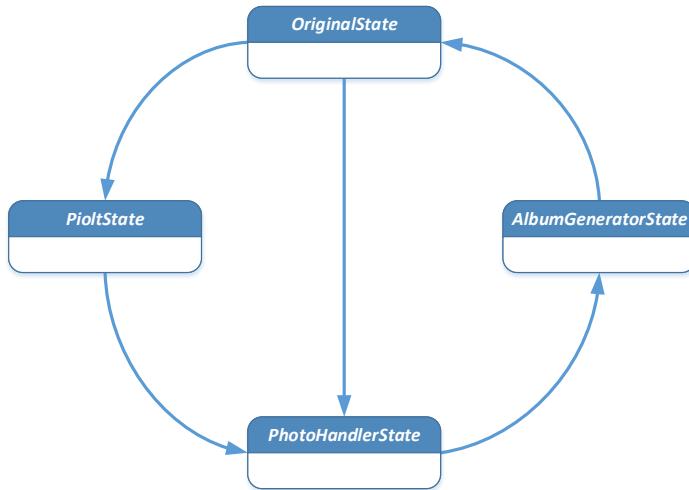


FIGURE 9 – Cycle de la machine d'état

Etant donné qu'il existe plusieurs états possibles pour l'utilisateur et pour des photos, par exemple, l'emplacement de l'utilisateur est toujours variable et des photos sont faites arbitrairement pendant du voyage, c'est mieux que partager des tâches d'opération en partie différentes selon des états différentes afin que chaque état s'occupe des tâches nécessaires sans des opérations superflues.

Nous modélisons quatre états pour une cycle du fonctionnement, soit le premier état : **OriginalState**; le deuxième état : **PioltState**; le troisième état : **PhotoHandleState**; le quatrième état : **AlbumGeneratorState**;

Tout d'abord, le programme tourne toujours dans l'état **OriginalState** si l'utilisateur ne sort pas de sa ville originale. Dans cet état-là, il n'y a aucune opération.

Une fois que l'utilisateur n'est plus à sa ville originale et il n'y a pas encore des nouveaux photos, on passe au état suivant-**Pioltstate**. Pour cet état, il n'y a pas d'opération sauf que l'enregistrement des informations de GPS de l'utilisateur.

Si maintenant quelques photos sont générées, l'état est mise à jour en même temps. On arrive à l'état **PhotoHandlerState**, ce qui sert à traiter des informations des photos. On note des GPS points des nouveaux photos et plus obtient des informations de l'emplacement (pays, province, ville etc.) sur la base des GPS points. De même, on enregistre aussi des informations de GPS de l'utilisateur comme l'état précédent. Evidemment, l'état **OriginalState** peut traverser directement à l'état **PhotoHandlerState** sans passer l'état **PioltState** en certains cas.

Quand l'utilisateur retourne dans sa ville originale et toutes les photos sont bien traitées, on passe à l'état final- l'état **AlbumGeneratorState**. Le but de cet état est l'organisation des albums des photos. On obtient un ensemble des noms des villes visités en parcourant toutes les photos existant et enlève des GPS points de l'utilisateur dont la ville n'a aucune photo. Lorsque ces tâches sont tous finis, on repasse à l'état original.

# 6 APPLICATION

## 6.1 Description générale

Comme nous avons mentionné dans la section précédente, l'application va fonctionner principalement en fond. Nous présentons ci-dessous la cycle de vie générale de l'application avec un **diagramme BPMN 2.0**.

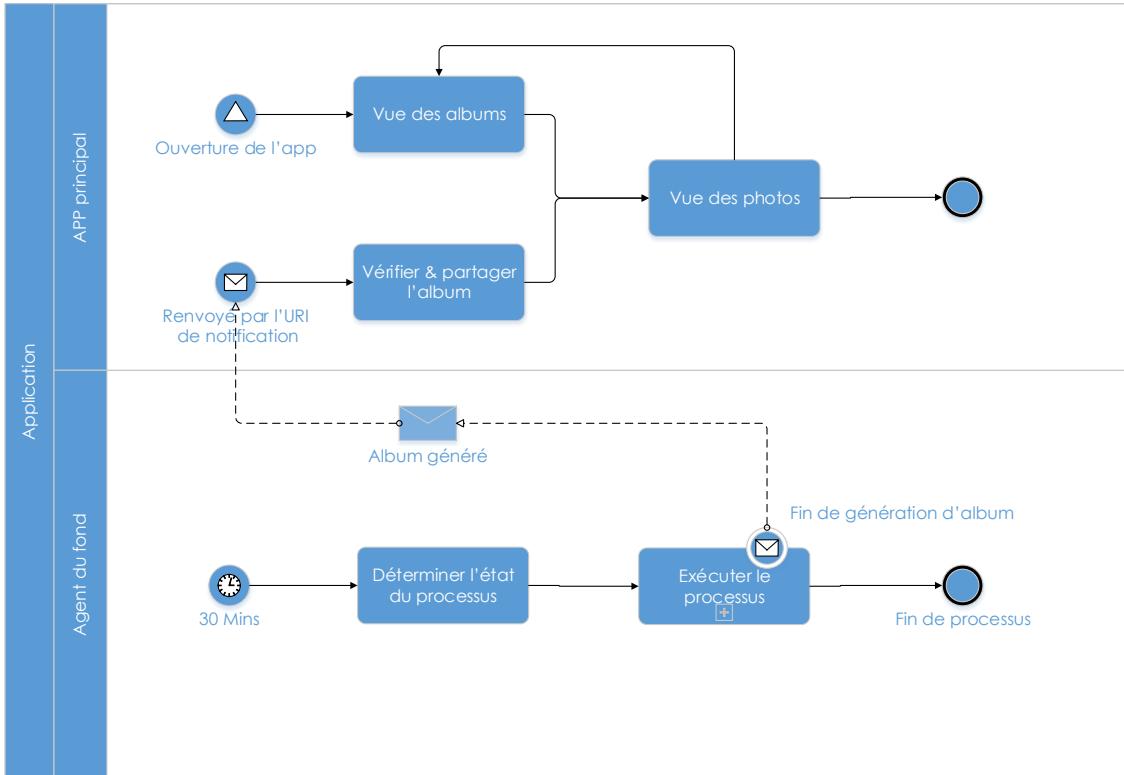


FIGURE 10 – Cycle de vie générale de l'application

L'application se divise en deux parties séparées : **application principale** et **l'agent du fond**. Dans l'application principale, l'utilisateur peut visualiser les photos et albums pris pendant un voyage. En même temps, l'agent du fond travaille sans cesse pour détecter un voyage et générer les albums.

## 6.2 Navigation des photos

Nous envisageons trois modes pour naviguer entre les photos :

- la vue normal (navigation simple par les gestes simples)
- la vue par la liste des photos
- la vue par la carte

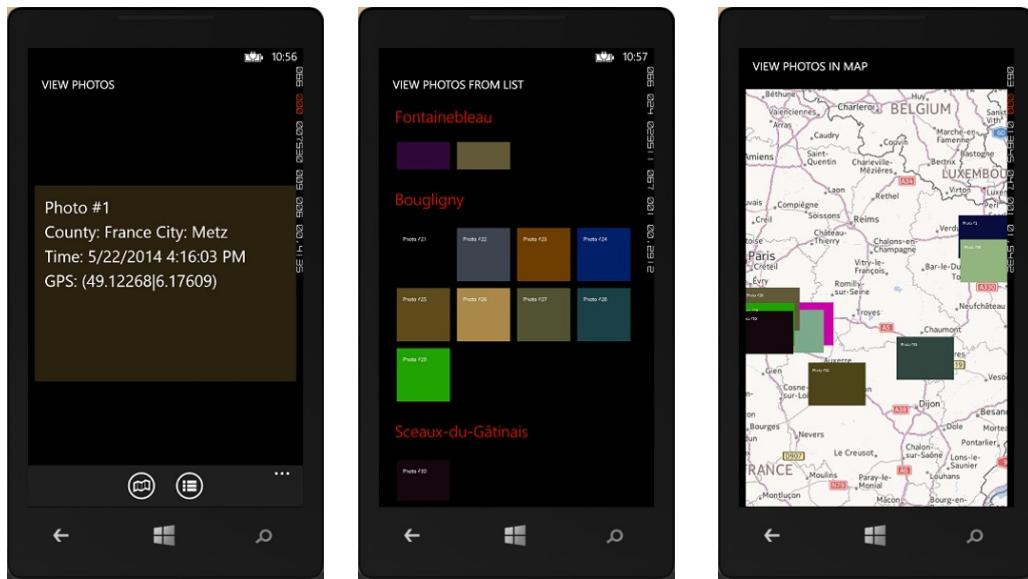


FIGURE 11 – Navigation entre les photos

### 6.2.1 Vue normale

Dans la vue normale, utilisateur peut naviguer en gauche et à droite les photos par la vue classique d'un album. Aussi, l'utilisateur peut zoomer et dézoomer sur un photo avec les gestes correspondantes.

### 6.2.2 Vue par liste

Nous réalisons un vue par "jump-list" pour rendre plus facile la navigation non-linéaire des photos. L'utilisateur peut entrer dans cette vue en cliquant sur le bouton en bas du page. Les photos sont regroupé en fonction des position (ville et pays). Grâce au jump-list, utilisateur peut facilement naviguer au groupe désiré en cliquant sur l'entête des groupes puis choisir le groupe correspondante.

### 6.2.3 Vue par carte

Nous réalisons un vue par **carte** pour mieux présenter les photos selon leur position ainsi que l'itinéraire de voyage. Cette vue est basée sur le contrôle **carte** de Windows Phone 8. Les photos sont placés dans la carte en tant que les étiquettes relatives géographiques situées dans une couche séparée. Grâce à cette configuration, la détection des gestes différentes comme le déplacement de la carte et la sélection du photo peuvent être facilement réalisée. Une animation de localisation est réalisée pour rendre plus dynamique la visualisation initiale des photos sur la carte.

### 6.3 Visualisation

Comme la visualisation joue un rôle très important dans l'application principale, nous réalisons plusieurs implémentations visuelles dans le prototype. Les éléments marqué avec un astérisque seront implémentés dans les versions futures.

coggle

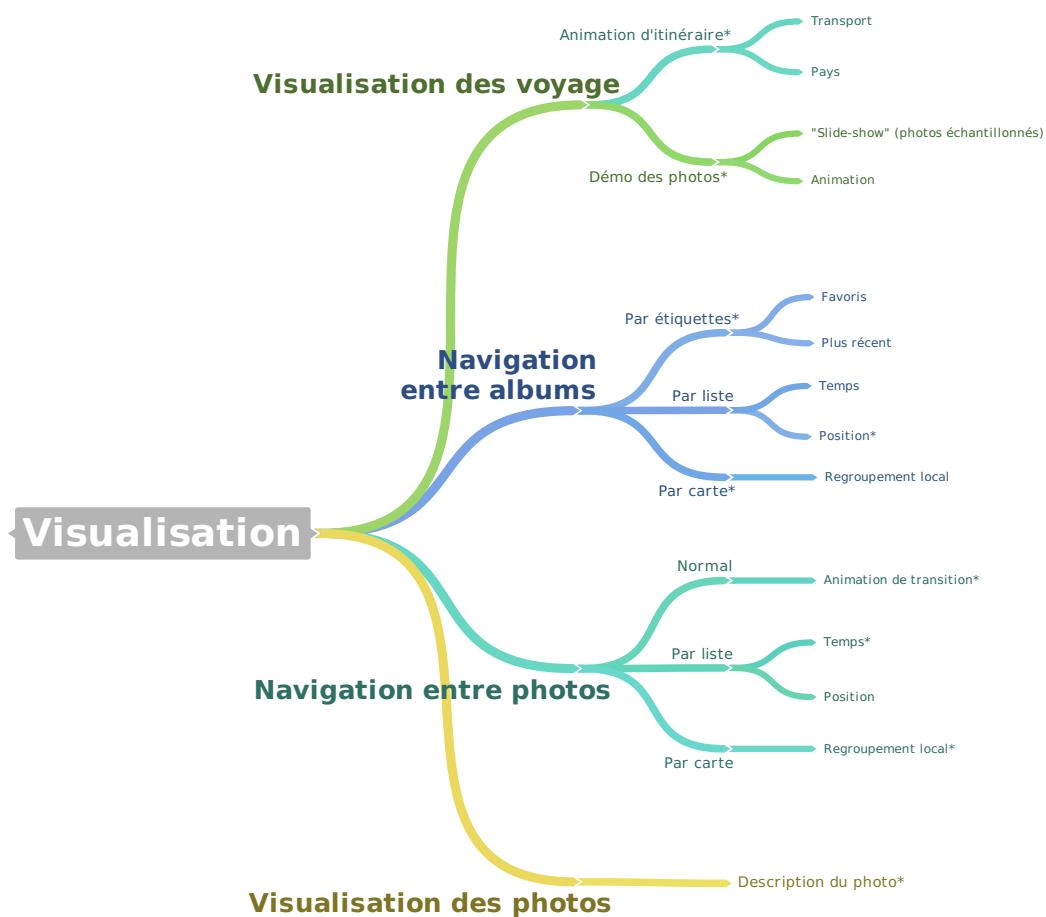


FIGURE 12 – Éléments de visualisation

# 7 SIMULATEUR

## 7.1 Description générale

Le simulateur consiste en deux parties générales : interface d'utilisateur et une service établie localement. Il permet la planification d'itinéraire de voyage, la génération des photos virtuels, la simulation en temps réel, etc.

La technologie utilisée pour communiquer entre le simulateur et l'application mobile est **Windows Communication Foundation (WCF)**. Elle est une framework introduite en C# 3.0 pour unifier les méthodes différentes de communication en Windows. Les détails sont données dans les sous-sections suivantes.

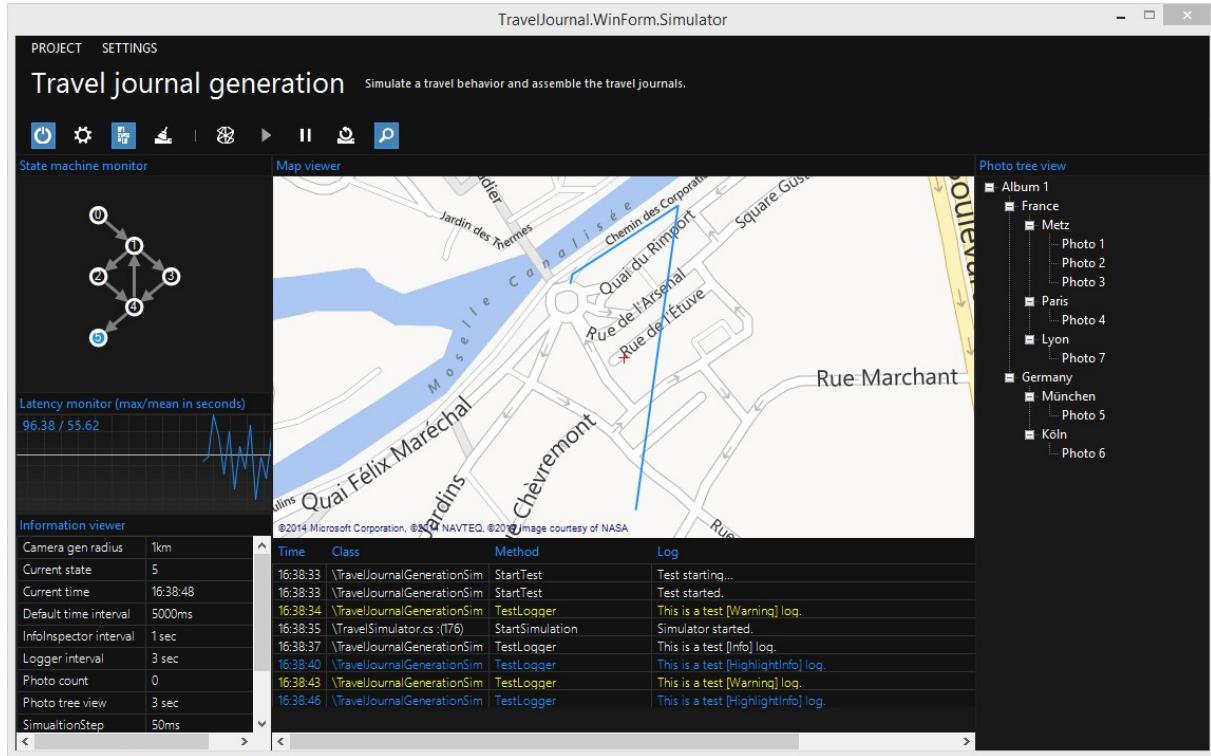


FIGURE 13 – Interface de simulateur

## 7.2 Interface graphique

La partie interface du simulateur consiste en plusieurs modules :

- console de contrôle pour gérer la simulation et la configuration
- moniteur de la **carte** pour visualiser le voyage
- moniteur d'état du programme pour visualiser l'état du processeur (c.f. section 5.3)
- moniteur d'**information** pour visualiser les données intermédiaires

- moniteur des **albums** générés
- moniteur de **connexion entre serveur (Simulateur) et client (Windows phone)**
- moniteur des **logs** pour le debug

### 7.2.1 Contexte de fonctionnement

Chaque moniteur travaille dans son propre **Thread**, c'est-à-dire tout est doublement amorti. Cela conduit à donner une animation plus lisse. Mais cette configuration demande une ressource allouée plus importante.

### 7.2.2 Transmission des données

Pour la transmission des données entre modules, nous utilisons la **Sérialisation**. Cela permet aussi de vérifier la donnée en tout moment par la visualisation d'XML. Le mécanisme utilisé est la sérialisation WCF qui s'appelle **Data Contract Serialization**.

Héritées de la classe ConfigDataBase, toutes les classes de données utilisent le **patron d'Observateur**. Une fois une donnée est modifiée, tous modules la observant seront mis à jour. La réalisation profite de les notions d'**événement** et **propriété** de .NET : deux événements ( **OnDataChanging** et **DataChanged**) seront invoqués lors la donnée est modifiée par le Setter.

De plus, la classe ConfigDataBase expose une méthode abstraite **Display()** qui permet au moniteur d'information d'inspecter ces données.

### 7.2.3 Contrôle de la carte

Nous utilisons la contrôlé **Great Map Control** pour visualiser et inter-opérer avec la carte. Une documentation complète est donnée sur son site officiel.

### 7.2.4 Révocation

Le **patron de Memento** est utilisé pour rendre possible l'annulation de toute opération effectuée. Ceci est important pendant la construction d'itinéraire de voyage.

## 7.3 Simulateur

La simulateur est responsable de deux missions : la génération des données et la simulation en temps réel.

### 7.3.1 Données de voyage

Les données de voyage initiale sont présentées par les "ancres" de voyage. Il s'agit des **GPS** avec le **nombre de photos** seront pris dans le proche (N). Le testeur va les créer par un designer de la carte. Une fois enregistrées, ces données seront interprétées par un **compilateur** de simulation. Un **générateur aléatoire gaussienne** va générer N nouveaux GPS avec 1 photos pour chaque GPS initiale. Ensuite, ces données compilées seront traitées par le simulateur pour simuler un voyage.

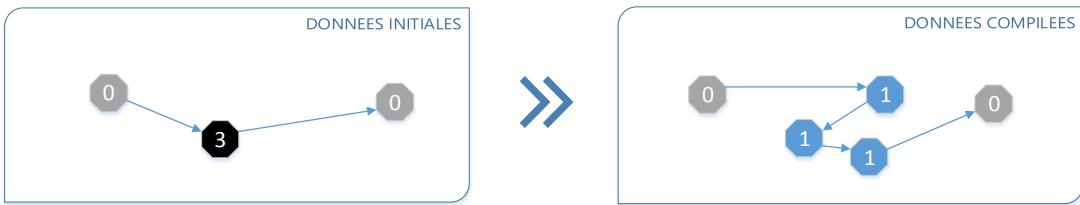


FIGURE 14 – Génération des données de voyage

Cette configuration modélise effectivement la prise des photos comme car elle dépend normalement de la location géographique.

La durée de voyage entre deux "ancres" sont la même. C'est-à-dire pour modifier la vitesse, il suffit de modifier la distance entre des "ancres". Pour une distance identique, plus il y a des "ancres", moins elle est la vitesse.

De plus, la ville d'habitation est indiquée lors la création de voyage.

### 7.3.2 Simulation

La simulation de voyage est conduite par un **Timer**. Une fois les données de voyage sont générées et compilées, elles sont chargées dans le simulateur. Un **curseur (GPS)** va parcourir toute la route et retourner à la fin à l'origine. Tous les photos "prises dans le voyage" (un photo est prise quand le GPS avec ce photo est parcouru par le curseur) sont enregistrés dans une liste pour donner ultérieurement à l'application mobile via les services WCF.

## 7.4 Services WCF

Les services **Windows Communication Foundation (WCF)** jouent un rôle très important dans la simulation. Elles permet la communication entre le simulateur et le mobile.

WCF est un modèle de programmation introduite en C# 3.0 pour unifier les méthodes différentes de communication en Windows. **Inter-opérabilité** est la caractéristique fondamentale de WCF. Elle est composée de plusieurs éléments de Web Service, Remoting, MSMQ et COM+. Dans un mot, elle donne un plate-forme commune pour toutes les moyennes de communication dans le monde .NET.

Particulièrement, la programmation est simple avec WCF. L'utilisation des attributs facilite beaucoup l'implémentation des services du serveur. Un service WCF est composé de trois parties :

- Une classe service
- Un environnement hôte
- Un ou plusieurs points finaux

### 7.4.1 Classe service

La classe service n'est rien d'autre qu'une interface/classe décorée de quelques attributs WCF.

Listing 1– Example d'une classe service WCF

```
[ServiceContract]
public interface ISimulationServices
{
    #region Connection services

    [OperationContract]
    bool Connect(string deviceName);

    [OperationContract]
    bool Disconnect(string deviceName);

    #endregion
}

[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
public class SimulationServices : ISimulationServices
{
    #region Connection services

    public bool Connect(string deviceName)
    {
        ...
    }

    public bool Disconnect(string deviceName)
    {
        ...
    }

    #endregion
}
```

Les attributs utilisés permet à WCF de repérer les classes services lors la compilation.

A l'exécution, WCF va gérer la transmission des données via la **Sérialisation WCF**, c'est-à-dire tous les augment et la valeur retournée vont être sérialisés et déserialisés pour être utilisés à distance.

#### 7.4.2 Environnement hôte et points finaux

Dans le contexte de notre simulateur, le service est "hosté" localement dans le simulateur (Application Windows Forme). Nous avons choisi cette configuration car le service ne sert qu'à la simulation de l'application — le simulateur sera forcément ouvert. Pour configurer le hôte, il suffit de modifier le fichier de configuration "app.config" :

Listing 2– Configuration de hôte WCF

```
<system.serviceModel>
    <behaviors>
        <serviceBehaviors>
            <behavior name="">
                <serviceMetadata httpGetEnabled="true" httpsGetEnabled="true" />
                <serviceDebug includeExceptionDetailInFaults="false" />
            </behavior>
        </serviceBehaviors>
    </behaviors>
    <services>
        <service name="TravelJournal.WinForm.Simulator.SimulationServices">
```

```

<endpoint address="" binding="basicHttpBinding" contract="TravelJournal.
    WinForm.Simulator.ISimulationServices">
    <identity>
        <dns value="localhost" />
    </identity>
</endpoint>
<endpoint address="mex" binding="mexHttpBinding" contract="
    IMetadataExchange" />
</service>
</services>
<bindings />
<client />
</system.serviceModel>

```

---

Un point final est un portail pour la communication. Il est composé d'une **adresse**, un **binding** et un **contrat**.

L'adresse de service est locale selon l'adresse IP de l'ordinateur. Le binding est en fait le protocole utilisé pour communiquer avec le service WCF. Nous utilisons le **BasicHttpBinding** qui utilise HTTP comme le protocole de transmission et XML comme l'encodage de message. Il est non sécurisé et faible au niveau de inter-opérabilité, cependant il est facile d'implémenter et est suffisant pour la simulation. Le contrat est l'interface **ISimulationServices** déclarée dans le codes.

#### 7.4.3 Environnement hôte et points finaux

Comme le service est "hosté" dans le simulateur, il faut ouvrir le simulateur pour mettre à jours le service. La mise à jours est occupée automatiquement par l'IDE.

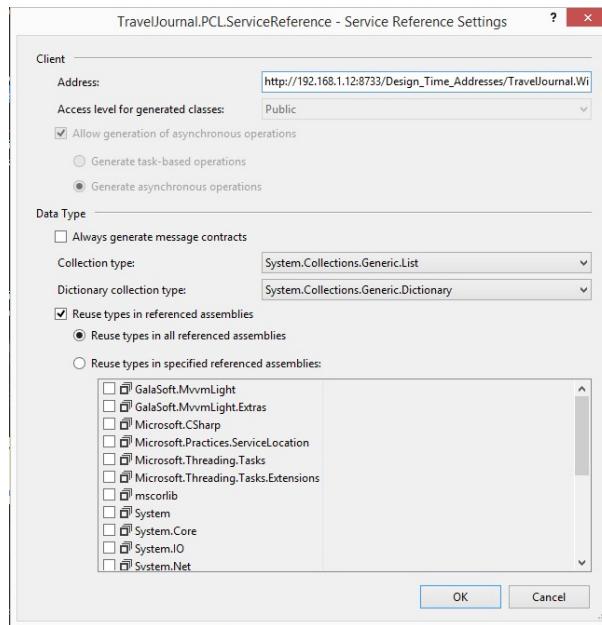


FIGURE 15 – La mise à jour automatique de service côté client

## 8 TESTS ET SIMULATIONS

Pour travailler efficacement et agilement, nous faisons les test unitaires régulièrement dans l'implémentation de business logic. Ensuite, les test de transmission sont faits pour pouvoir lancer une simulation complète. A la fin, une simulation complète de voyage est effectuée.

### 8.1 Test unitaires

Un test unitaire consiste en 3 parties : **arrangement**, **action** et **assertion**. Grâce à l'IDE, les tests sont automatisés et l'implémentation peut facilement basée en test (Test Driven Développement). Nous allons modifier l'implémentation jusqu'à la "passe" de tous les tests. Voici un exemple de test unitaire sur l'extraction des données EXIF de photo.

Listing 3– Exemple de test unitaire

```
[TestClass()]
public class ExifLibExtractorTests
{
    [TestMethod()]
    public void ExtractGeoCoordinateTest()
    {
        // Arrange
        ExifLibExtractor extractor = new ExifLibExtractor();
        MediaSource mediaSource = MediaSource.GetAvailableMediaSources().First(
            source => source.MediaSourceType == MediaSourceType.LocalDevice);
        Picture samplePicture;
        using (MediaLibrary mediaLibrary = new MediaLibrary(mediaSource))
        {
            PictureAlbum cameraRollAlbum = mediaLibrary.RootPictureAlbum.Albums.
                First((album) => album.Name == "Camera Roll");
            samplePicture = cameraRollAlbum.Pictures.First();
        }
        Photo photo = new Photo()
        {
            Name = "WP_20131201_13_37_04_Pro.jpg"
        };
        // Act
        GpsPoint point = extractor.ExtractGeoCoordinate(photo);
        // Assert
        Assert.AreNotEqual (default(double), point.Latitude);
        Assert.AreNotEqual (default(double), point.Longitude);
        Assert.AreNotEqual (default(DateTime), point.Timestamp);
    }
}
```

---

## 8.2 Test de transmission

Pour le test d'intégration, nous créons un client mobile pour lancer l'agent du fond. Le client dispose de 3 tests : **test de connectivité du service** et **test d'information de voyage**.

### 8.2.1 Test de connectivité du service

Ce test destine à tester la connectivité du serveur et client. Il est composé de deux tests : échange d'un paquet entre le simulateur et le client et un processus périodique de même opération. Dans ces tests, une liste des entières est utilisée. Testé avec la connexion locale (simulateur et émulateur WP),

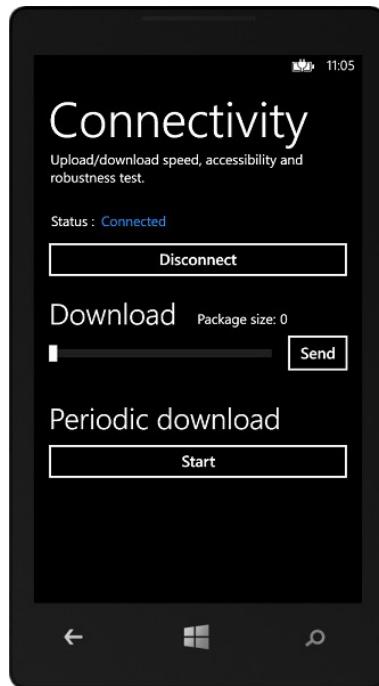


FIGURE 16 – Test de connectivité

la délai d'échange d'un paquet de **500000/1000000** éléments vaut **0.89/1.23 seconds**. Le dernier paquet utilisé est supérieur à la taille de donnée réelle de simulation. Donc la vitesse de communication est assurée, nous pouvons négliger la délai de transmission.

### 8.2.2 Test d'information de voyage

Ce test destine à tester le téléchargement des photos générés par le simulateur. Après ce test, nous pouvons considérer que l'échange est bien définie alors la partie de transmission est assurée pour une simulation complète.

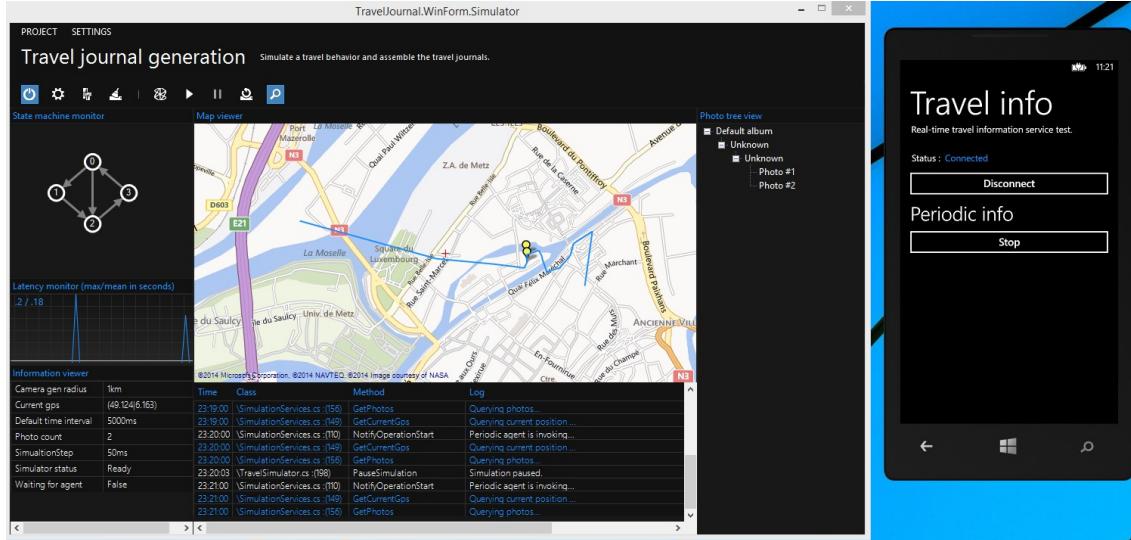


FIGURE 17 – Test d'information de voyage

### 8.3 Simulation

La simulation complète est basée sur un voyage à partir de Metz avec 36 photos. Au début, le curseur est à Metz, alors le programme reste à l'état initial.

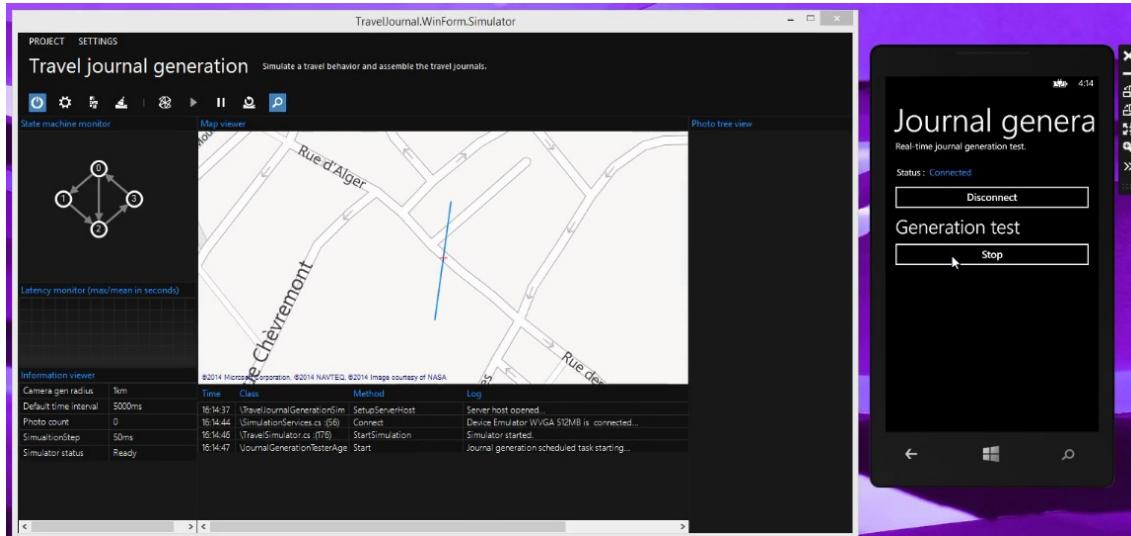


FIGURE 18 – Simulation de voyage : état initial

Lors les photos sont générées, le programme tourne à l'état 2 : traitement de photo. Nous pouvons

observer les itinéraire de voyage ainsi que la route parcourue. Les procédures (obtention de la position d'utilisateur, enquête de position géographique... ) de traitement sont affichées dans le moniteur des logs.

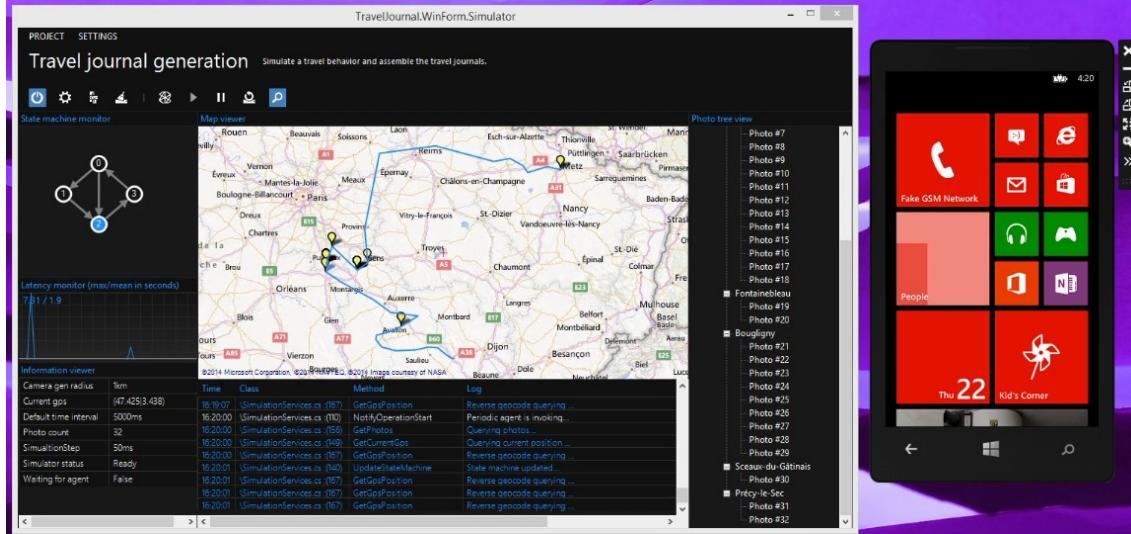


FIGURE 19 – Simulation de voyage : état de traitement des photos

A la fin, dès que le curseur est retourné à Metz, l'album est complété. Un notification est envoyé à l'utilisateur pour indiquer la génération d'album. En cliquant sur le toast de notification, l'application est ouverte et l'utilisateur peut soit partager les photos de voyage en réseau sociaux, soit les visualiser dans l'application. L'effet visuel d'interface est donné dans la section 7.

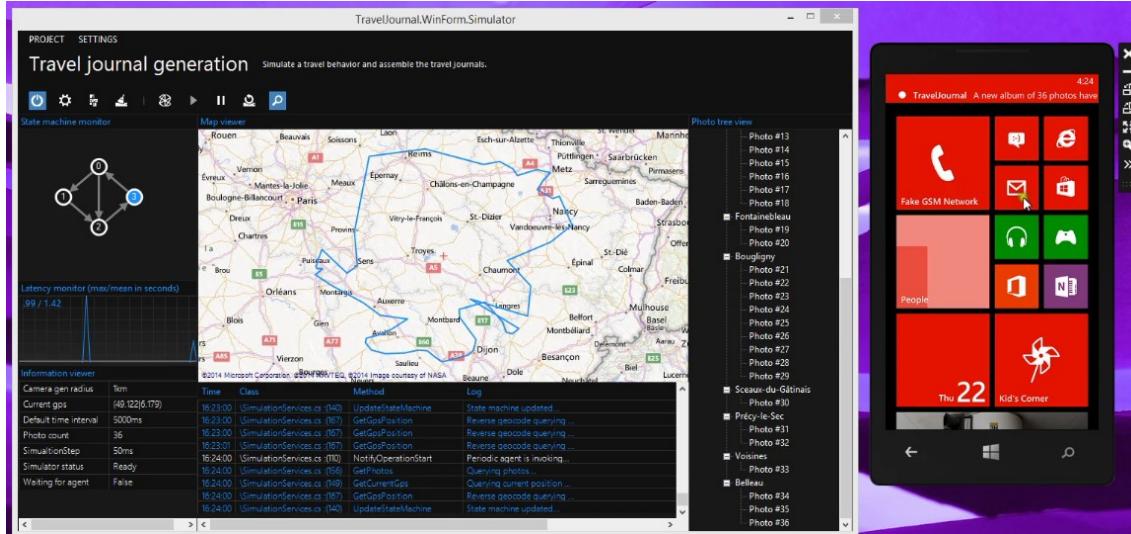


FIGURE 20 – Simulation de voyage : état de génération d'album

## 9 EVOLUTION FUTURES

A cause de la durée courte de projet, il existe encore plusieurs fonctionnements et optimisations que nous avons pas pu réalisés. Nous citons ici quelques évolutions possibles pour la commercialisation finale de notre application.

### 9.1 Classification des photos avec les critères différentes

Dans certaines conditions la classification des photos par l'enquête géocode n'est plus valable. Par exemple, lorsque l'utilisateur n'a pas de connexion d'internet, le lieu de voyage n'est pas reconnu (ou imprécis) par le service (forêt, désert, etc.). Dans ce cas, une classification par les coordonnées GPS devient importante et efficace. Pour le faire, nous pouvons utiliser les algorithmes de classification comme K-moyens.

### 9.2 Amélioration d'interface graphique

Car la visualisation est très importante dans l'application, nous pensons à améliorer l'interface par les animations et les effets visuels. En inspirant de la style Suisse (**Swiss Style**), nous proposons quelques designs suivants :



FIGURE 21 – Swiss Style visualisation

Quand un changement de location de ville ou de pays a lieu, un pop-up peut être affiché pour indiquer la location de ville ou de pays.

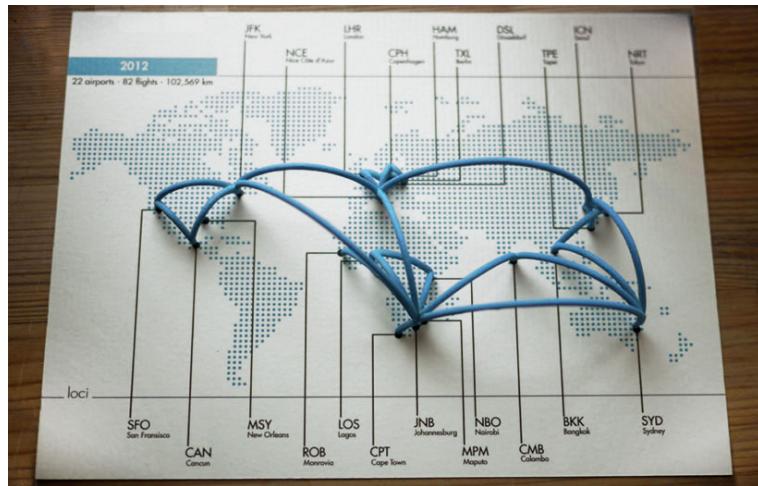


FIGURE 22 – Animation de voyage

Lors la visualisation de voyage sur la carte, une animation peut créer pour donner une dynamique sur la présentation. La moyenne de transport peut être interpréter par un calcul de vitesse (avec les données GPS), qui est ensuite visualisé dans l'animation.

### 9.3 Réingénierie logicielle

En analysant la qualité métrique de codes avec l'IDE, nous apercevons que la maintenabilité peut encore être modifiée avec la réingénierie logicielle (**Refactoring**). Par exemple, la **complexité cyclomatique** et le **couplage des classes** peuvent être améliorés avec les patrons de conception et les réingénierie d'architecture.

Code Metrics Results						
Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code	
⚠ One or more projects were skipped. Code metri						
↳ TravelJournal.WP8 (Debug)	88	458	5	147	655	
↳ TravelJournal.PCL (Debug)	89	223	4	73	335	
↳ TravelJournal.WP8.UI (Debug)	76	105	9	106	258	
↳ TravelJournal.PCL.ViewModel (Debug)	89	69	4	36	101	
↳ TravelJournal.WP8.UnitTest (Debug)	77	48	8	60	94	

FIGURE 23 – Analyse métrique de codes

## 9.4 Optimisation de performance

Une analyse de performance est faite pour les améliorations ultérieures. L'analyse est faite par l'IDE avec le fonctionnement **Windows Phone Application Analysis**. Globalement la défaut plus important de l'application est le délai de l'interface utilisateur. En analysant les détaillés de l'exécution,

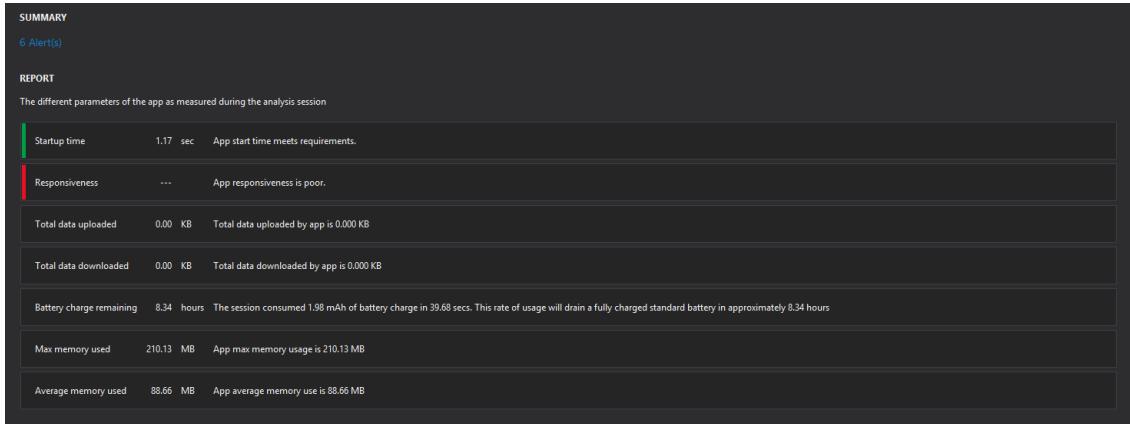


FIGURE 24 – Analyse globale de l'application

nous apercevons que ce problème devient particulièrement gênant pendant la visualisation des photos en liste ou dans la carte. Nous pensons à faire les caches de toutes les images dans le fond quand l'utilisateur ouvrir un album (**Prefetching**). Cette cache ne sera pas être très grande car seulement les imagettes seront utilisées dans les contrôles.



FIGURE 25 – Analyse détaillée de l'exécution

## 10 CONCLUSION

Grâce à ce projet, nous avons pu expérimenter la cycle de développement complet : l'analyse de besoin et de marketing, la construction de cahier des charges, le design de l'architecture, l'implémentation, les tests et simulations.

De plus, nous avons pu nous familiariser avec le plate-forme Windows Phone 8. Nous avons appris plusieurs contrôles importants comme le contrôle de la carte, le contrôle Panorama, le système de navigation, etc.

Nous avons aussi utilisé l'agent du fond (Background agent). Cette conception est fondamentale de notre système — rien ne marche si nous ne pouvons pas encadrer les processus dans cet agent. Comme l'agent ne fonctionne que pour 25 seconds et les ressources allouées sont assez limitée, nous avons modifié plusieurs fois l'algorithme pour pouvoir obtenir un résultat satisfaisant.

Le simulateur joue un rôle important dans le projet. Nous analysons la condition de simulation ainsi que la faisabilité de chaque design. Puis nous utilisons la méthodologie de développement rapide (Rapid Application Development) pour créer le simulateur. Nous partageons les tâches alors le projet avance bien pendant et après la réalisation de simulateur.

Pendant le design et la réalisation, nous avons rencontré plusieurs difficultés concernant les choix de technologies, l'implémentation et debug, ainsi que les test et simulation. Par exemple, la connexion entre le simulateur et l'application. A cause du "sand-box" des applications, nous avons eu une difficulté pour partager les données entre l'application de Test et l'application principale. Enfin, nous avons la contourné en réalisant la synchronisation des données sur le Cloud. Ceci est aussi important pour pourvoir utiliser l'application en plusieurs équipements. Grâce à ce projet, nous avons pu nous entraîner à faire les choix techniques et décisifs en face des problèmes et difficultés.

Nous bénéficions beaucoup des conseils de notre tuteur dans les aspects d'utilisateur et algorithmiques. Par exemple, l'idée de "prefetching" des imagettes pour l'amélioration de performance dans l'interface d'utilisateur. Malheureusement, limité en temps, nous n'avons pas pu réaliser toutes les idées intéressantes comme le regroupement par les critères différents. Cependant elles sont intéressantes pour les versions futures.

La recherche d'information est aussi une démarche très important pendant la réalisation. Dans le plupart du temps, nous nous référençons vers les sites informatiques comme StackOverflow et MSDN. Une recherche précise et efficace est indispensable pour la résolution des problèmes. Par ailleurs, nous lirons aussi plusieurs œuvres classiques comme "Windows Phone 8 Recipes" pour se familiariser avec Windows phone.

A la fin, nous pouvons dire que ce projet a été effectué avec succès : nous avons réalisé un prototype stable ; nous avons construit un infrastructure efficace pour les simulations ultérieures ; nous avons respecté le cahiers des charges ; et enfin, nous sommes pas loin à donner une version officielle pour la commercialisation.

# 11 ANNEXE

## 11.1 Transition entre les états du traitement

Listing 4– Transition entre les états

```
public class Transition
{
    public void Transform(Processor processor)
    {
        if (processor.State == null)
        {
            throw new MissingMemberException("State not loaded");
        }
        string stateType = processor.State.GetType().Name;
        processor.UserPosition = processor.WebService.GetUserPosition().Result;
        bool haveNewPhoto = processor.PhotoManager.CheckRawPhoto(processor.Album.
            TimeTag);
        if (processor.UserPosition.City == processor.DataManager.Data.UserInfo.
            OriginalPosition.City)
        {
            if (stateType.ToUpper() == "PILOTSTATE")
            {
                if (haveNewPhoto == false)
                {
                    processor.State = new OriginalState();
                }
                else
                {
                    processor.State = new PhotoHandlerState();
                }
            }
            if (stateType.ToUpper() == "PHOTOHANDLERSTATE" && (!haveNewPhoto))
            {
                processor.State = new AlbumGeneratorState();
            }
            if (stateType.ToUpper() == "ALBUMGENERATORSTATE" && processor.
                AlbumCompleted)
            {
                processor.State = new OriginalState();
            }
        }
        else
        {
            if (stateType.ToUpper() == "ORIGINALSTATE")
            {
                if (haveNewPhoto)
                {
                    processor.State = new PhotoHandlerState();
                }
                else
                {
                    processor.State = new PilotState();
                }
            }
            if (stateType.ToUpper() == "PILOTSTATE")
            {
                if (haveNewPhoto)
                {
                    processor.State = new PhotoHandlerState();
                }
            }
        }
    }
}
```

```
        }
    }
}
```

---

## 11.2 Traitements pour chaque état

Listing 5– Traitements pour chaque état

```
[DataContract]
[KnownType(typeof(OriginalState))]
[KnownType(typeof(PilotState))]
[KnownType(typeof(PhotoHandlerState))]
[KnownType(typeof(AlbumGeneratorState))]
public abstract class State
{
    public abstract void Execute(Processor processor);
}
public class OriginalState : State
{
    public override void Execute(Processor processor)
    {

    }
}
public class PilotState : State
{
    public override void Execute(Processor processor)
    {
        processor.TourRoutePoints.Add(processor.UserPosition);
    }
}
public class PhotoHandlerState : State
{
    public override void Execute(Processor processor)
    {
        processor.TourRoutePoints.Add(processor.UserPosition);
        if(processor.PhotoManager.CheckRawPhoto(processor.Album.TimeTag))
            processor.PhotoManager.ProceedRawPhoto(processor.Album.TimeTag,
                processor.PhotoHandler);
    }
}
public class AlbumGeneratorState : State
{
    public override async void Execute(Processor processor)
    {
        foreach (Photo p in processor.Album.PhotoList)
        {
            if (!processor.TouristCity.Contains(p.Position.City))
                processor.TouristCity.Add(p.Position.City);
        }
        foreach (GpsPosition p in new List<GpsPosition>(processor.TourRoutePoints))
        {
            if (!processor.TouristCity.Contains(p.City))
                processor.TourRoutePoints.Remove(p);
        }
        // Notify album completes
        processor.AlbumCompleted = true;
        processor.AlbumCompletedCallback.Invoke(processor);
    }
}
```

---