Dossier n°4

Rapport de conception logicielle

vendredi 13 février 2009

Projet ***Monofin***

4ième année Informatique

INSA Rennes

YOANN CHAUDET
PAUL GARCIA
QUENTIN GAUTIER
NICOLAS LE SQUER
NICOLAS MUSSET
XAVIER VILLOING

Encadreurs

PATRICE LEGUESDRON
LAURENT MONIER
FULGENCE RAZAFIMAHERY

Sommaire

Table des figures

I. Introduction

Ce quatrième dossier fait suite au dossier de planification. Le travail étant planifié, les spécifications détaillées rédigées, nous avons pu commencer à réfléchir à la conception logicielle.

Depuis le début du projet nous hésitions entres deux langages de programmation : C++ et Java. De même, nous n'étions pas certains de la manière et de l'algorithme à adopter pour effectuer de l'extraction de contours. La première partie de ce dossier se consacre donc à répondre à ces questions et plus généralement à répondre à toutes les questions d'ordre technique qui étaient restées en suspend jusqu'à maintenant.

Dans un second temps, le dossier présentera l'analyse statique qui a été effectuée. Celle-ci exposera les différents schémas UML modélisant : l'interface graphique, les structures de données manipulées et la partie se chargeant de générer dynamiquement le script COMSOL. Nous n'avons modélisé que les grandes lignes de l'application, celles ayant une grande importance. L'application toute entière que nous allons créer est une interface graphique, les structures de données sur lequel nous travaillerons devront nous permettre de modéliser une monopalme mais également de générer dynamiquement un script COMSOL. Les schémas UML modélisent donc ces trois aspects de l'application.

Dans un dernier temps, le dossier présente l'analyse dynamique effectuée sur l'interface graphique. Cela se traduit par un schéma d'enchainement des pages, un diagramme d'activité ainsi que des diagrammes de séquence.

II. Choix techniques

A. Choix des outils

Un premier choix qui nous restait à prendre était de choisir un langage de programmation pour l'implémentation de notre application. Nous hésitions entre le JAVA et le C++ car tous deux permettent d'utiliser la librairie Qt. Je rappelle que cette librairie nous permettra de créer plus facilement l'interface graphique de notre application grâce à de nombreux outils de développement. Le choix s'est finalement porté sur le C++. La décision s'est notamment faite car certains des membres de l'équipe ont commencé à étudier la librairie Qt via C++, nous maîtrisons donc mieux Qt C++. Le langage C++ est également un langage de programmation que nous venons d'apprendre et maîtriser cette année, il nous est paru donc intéressant d'utiliser ce langage pour ce projet. Enfin un programme écrit en C++ a certaine qualité comme sa rapidité d'exécution. Ce dernier point n'est pas à négliger lorsqu'on manipule des objets graphiques animés (courbes de Bézier).

Nous avions également décidé de travailler sur l'environnement de développement Eclipse qui intègre parfaitement Qt via un *plugin* ainsi que le Qt-Designer. Qt-Designer est une aide dans la création de fenêtre. Nous avons par ailleurs découvert que la compagnie Trolltech, qui a développé Qt, a créé son propre environnement de développement qui intègre Qt. Cette application, le Qt Creator, est encore en version bêta mais il n'est pas impossible que l'on teste cet environnement pour notre projet.

Enfin il nous restait à choisir un algorithme pour l'extraction de contour qui permettrait d'intégrer une forme vectorisée directement dans l'éditeur de dessin.

B. Extraction de contour

Deux algorithmes avaient été étudiés. Le premier est l'algorithme de Sobel qui se révèle être très simple à implémenter. Cependant il s'est avéré que cet algorithme est trop peu puissant pour pouvoir être utilisé dans notre projet. Il sert principalement à la détection de contour. Cependant pour extraire une seule et unique forme d'une image, trop d'opérations fastidieuses devraient être implémentées.

Le second algorithme, basé sur les contours actifs ou *snakes* est celui qui a attiré notre attention car il semblait efficace pour détecter une seule forme dans une image. Nous voulions cependant avoir l'avis de personnes travaillant sur la détection de contour.

Après discussion avec Carole Le Guyader (enseignante-chercheur en Mathématiques) et Yann Ricquebourt (enseignant-chercheur au département Informatique), nous avons définitivement abandonné l'algorithme de Sobel. Un autre algorithme nous a également été présenté. Celui-ci est basé sur des notions mathématiques, les *level-set* (lignes de niveau). Sans rentrer dans les détails, les *level-set* permettent de représenter des courbes mathématiques sans les paramétrer. Cependant, dans notre application, nos courbes se basent sur des courbes de Bézier qui sont des courbes paramétrées. Ainsi cet algorithme ne correspondait pas à nos attentes. *A contrario* l'algorithme des contours actifs est basé sur des courbes paramétrées. C'est donc l'algorithme des contours actifs que nous allons utiliser dans notre projet.

Il nous reste à savoir comment implémenter cet algorithme. L'algorithme des contours actifs est basé sur les travaux de Kass, Witkin et Terzopoulos (Kass, Witkin, & Terzopoulos, 1987). Il nous

est possible de trouver certaines implémentations de l'algorithme, notamment auprès de Carole Le Guyader. Il devra cependant être adapté à notre projet car il ne suffit pas de détecter un contour mais de l'extraire pour le vectoriser. Vectoriser le contour ne devrait pas être un problème car le contour que nous détecterons sera une courbe paramétré. Cependant il nous est nécessaire de trouver des points caractéristiques de cette courbe pour les transformer en points de contrôle. Cette dernière tâche sera certainement la plus compliquée à implémenter.

C. Fichier de sauvegarde

Pour pouvoir sauvegarder notre structure interne de monopalme (dont la décomposition en classes sera détaillé en partie III.A page 9) sur laquelle va s'appuyer l'essentiel de notre application nous avons décidé d'utiliser le format XML, et ce pour plusieurs raisons que nous allons détailler. Après quoi nous décrirons comment notre document de sauvegarde XML se décompose.

1. Choix du format XML

Nous avons choisi le format XML pour nos fichiers de sauvegarde car c'est un fichier au format texte et qui est interopérable. Le format XML peut ainsi être partagé entre plusieurs machines ayant des architectures différentes. Enfin, le format XML est spécifiable à l'aide de schémas ce qui le rend très simple à faire évoluer avec le temps.

Le principal défaut reconnu du format XML est qu'il est très verbeux et par conséquent volumineux en mémoire. Dans notre cas ce ne sera pas vraiment un problème puisqu'*a priori* le volume de données à sauvegarder sera raisonnable (avec le format que nous avons choisi).

2. Description du fichier de sauvegarde

a. La racine

La racine du document sera une balise :

Code XML :

```xml
<project name="test" version="1.0">
<!-- reste du projet -->
</project>
```

Elle possède deux attributs, un attribut `name` qui laisse la possibilité de nommer le projet et un attribut `version` qui contient quand à lui la version du logiciel qui a procédé à la sauvegarde. Ce champ est particulièrement utile pour gérer finement les différentes versions et la rétrocompatibilité (si on devait avoir des évolutions majeures dans la structure de données par exemple).

Le projet contient deux sous parties distinctes : une sous partie chargée de sauvegarder la géométrie de la monopalme, et une autre chargée de sauvegarder les paramètres de configuration de la simulation.

b. *La géométrie de la monopalme*

Nous allons à l'aide de l'exemple ci-dessous expliquer comment est sauvegardée la géométrie de la monopalme.

Code XML :

```
<monofin>
 <segments>
  <segment numero="1">
   <intersectionpoint number="0"/>
   <intersectionpoint number="1"/>
   <controlpoint number="0"/>
  </segment>
 </segments>
 <points>
  <intersection>
   <point number="0" X="10" Y="10"/>
   <point number="1" X="20" Y="10"/>
  </intersection>
  <control>
   <point number="0" X="15" Y="15"/>
  </control>
 </points>
 <strates>
  <strate position="0" height="10" width="10"/>
 </strates>
</monofin>
```

La géométrie d'une monopalme, telle que nous l'avons définie, se décompose en deux parties distinctes, une surface définie par des segments (eux-mêmes définis par des points) et l'épaisseur gérée sous forme de strates d'une certaine longueur et d'une certaine épaisseur.

Un segment est identifié par un numéro unique (pour l'ensemble des segments) il contient des références vers deux points intersections (obligatoire car ils définissent les extrémités du segment) et une référence vers un point de contrôle (car il définit éventuellement la courbure du segment grâce aux courbes de Bézier)

Un point est identifié par un numéro unique (pour l'ensemble des points) et possède en outre deux attributs supplémentaires, `X` et `Y` qui permettent de définir ses coordonnées en abscisse et en ordonnée.

Une strate est définie par sa position (0 étant la strate la plus basse) et par deux attributs `height`, et `width` qui permettent de définir sa hauteur et sa longueur.

c. *La configuration de la simulation*

Au travers de l'exemple ci-dessous nous allons détailler comment est sauvegardée la configuration de la simulation.

Code XML :

```
<config>
 <stratesConfig>
  <strate position="0" young="123" poisson="456" ro="1.1"/>
 </stratesConfig>
 <resultats>
  <video check="true"/>
 </resultats>
 <parameters>
  <equation formula="sin(x)+cos(y)"/>
  <timestep t="0.1"/>
  <length l="20"/>
 </parameters>
</config>
```

La configuration d'une simulation contient des informations sur les matériaux constituants les différentes strates de la palme, les types de résultats attendus, et enfin les paramètres de calcul.

La configuration des strates (identifiées grâce à position) donne, donc, les informations sur les matériaux des strates synthétisées par le module de Young, le coefficient de Poisson, et par la masse volumique.

La section concernant les résultats attendus liste l'ensemble des options de rendus et leur état (sélectionné ou non) grâce à l'attribut `check`.

Les paramètres de calculs correspondent à l'équation du modèle à appliquer à la monopalme, le pas de temps, la longueur de la simulation, les informations sur le milieu, etc.

III. Analyse statique

La structure des données concernant une simulation doit pouvoir recouvrir à la fois les données géométriques, et les paramètres physiques et dynamiques. D'autre part, cette structure doit permettre d'accéder à toutes ces informations lors d'une demande de sauvegarde, et de les écraser en cas de chargement. Enfin, il est nécessaire que cette zone soit transparente et ne serve que de banque de données.

A. Structure de données

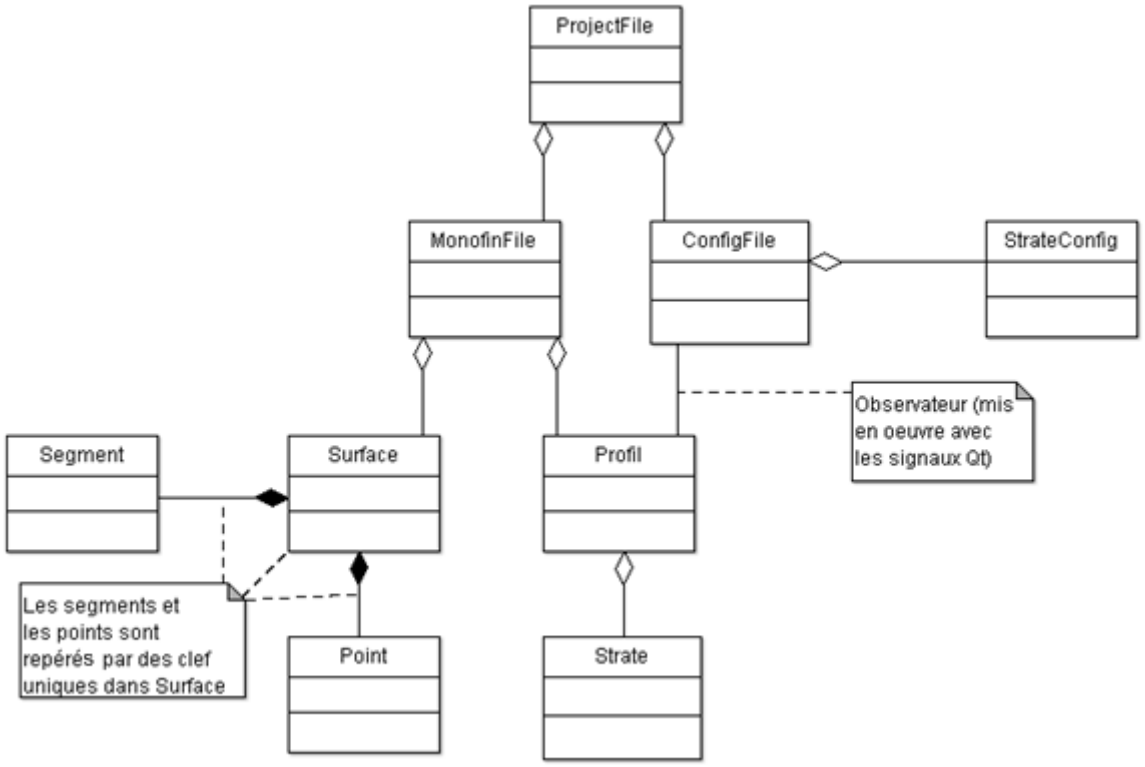

Figure 1 Schéma UML, structure de données principale

1. ProjectFile

Il s'agit de la classe principale. Elle contient toutes les informations relatives à un projet : la géométrie de la palme (`MonofinFile`) et les paramètres de la simulation (`ConfigFile`). Son rôle consiste à gérer les requêtes de modification en provenance de l'interface graphique et à les aiguiller selon qu'elles interviennent sur le `ConfigFile` ou le `MonofinFile`. D'autre part, elle permet la réalisation du fichier de sauvegarde.

2. MonofinFile

La classe `MonofinFile` est composée d'une `Surface` et d'un `Profil`. Elle contient donc toutes les informations relatives à la géométrie de la palme. C'est une classe amie de la classe `ProjectFile`, afin de récupérer les informations concernant la configuration des strates, qui sont référencées dans `ConfigFile`. Il est ainsi possible d'implémenter un *pattern* observateur entre le `Profil` et le `ConfigFile`. La classe `MonofinFile` possède donc plusieurs accesseurs sur les éléments `Surface` et `Profil`, afin de relayer les fonctions de modification transmises par l'interface graphique.

Elle dispose de plus d'une fonction `setReferenceConfigFile(*ConfigFile)` qui va chercher le `ConfigFile` créé et le donner au profil, sous forme de pointeur, afin de réaliser l'observateur.

3. ConfigFile

La classe `ConfigFile` contient les informations nécessaires au paramétrage de la simulation. Elle contient également les informations concernant la composition des strates dans le profil. Cependant, ces informations, par exemple le nombre de strates, et leur position doivent être mises à jour depuis le dessin du profil, car c'est à cet endroit que l'utilisateur les modifie (au niveau de `ConfigFile`, on ne fait que recevoir leur paramètres physiques). La classe `ConfigFile` contient donc un tableau des strates, qui est mis à jour via un observateur depuis la classe `Profil`, grâce aux méthodes suivantes :

- `updateCreate(int i)` Appelée dans Profil, cette fonction crée simplement une nouvelle strate dans le tableau, au rang `i`, et modifie le tableau si besoin est (décalage, changement d'indices).
- `updateRemove(int i)` Réalise l'opération inverse de `updateCreate`.
- `StrateConfig getStrateConfig(int i)` Récupère une strate d'indice `i` pour lui donner des paramètres ou en récupérer.
- `addStrateConfig(Strateconfig s, int i)` Ajoute une strate dans le tableau à l'indice `i` et décale les strates suivantes. Cette méthode (privée) est appelée par `updateCreate`.
- `removeStrateConfig(int i)` Retire la strate dans le tableau à l'indice `i` et décale les strates suivantes. Cette méthode (privée) est appelée par `updateRemove`.

4. Profil

La classe `Profil` est fortement liée à la classe `ConfigFile`, via un observateur. En effet lors de la construction de `Profil`, on donnera en attribut un pointeur vers un objet `ConfigFile` créé au préalable. Ce pointeur servira à effectuer la mise à jour de `ConfigFile`, lorsque des strates seront ajoutées ou retirées dans `Profil`. Le rang de la strate dans le tableau indique sa position dans l'empilement des strates sur le dessin du profil. Les méthodes suivantes seront implémentées :

- `Strate getStrate(int i)` Récupère la strate d'indice `i` dans le tableau des strates afin de modifier ou de récupérer ses informations.
- `addStrate (int i, Strate s)` Place la strate `s` dans le tableau à l'indice `i`, tout en décalant les strates supérieures. Elle indique également à `ConfigFile` d'ajouter une `StrateConfig` qui sera paramétrée ensuite.
- `remStrate (int i)` Retire la Strate d'indice `i` du tableau, tout en décalant les strates supérieures. Elle indique également à `ConfigFile` de retirer une strate et de faire les mises à jour qui s'imposent.
- `Link (*ConfigFile)` Donne l'objet `ConfigFile` à `Profil` afin d'appeler correctement les fonctions de mise à jour de `ConfigFile`.

- `notifyCreate(int i)` Indique à `ConfigFile` qu'une strate à été créée à l'indice `i`. Pour cela, elle appelle sur le pointeur `ConfigFile` sa propre fonction `updateCreate`, avec l'indice `i` en paramètre. Cette fonction (privée) est appelée par `addStrate`.
- `notifyDelete(int i)` Réalise le même travail que `notifyCreate`, mais vis-à-vis de la fonction `updateRemove`.

5. StrateConfig

La classe `StrateConfig` correspond aux éléments de la liste de `ConfigFile`. Elle contient plus précisément le module de Young, le coefficient de Poisson et La masse volumique du matériau associé à la palme et des valeurs par défaut. Elle contient les fonctions *getters* (fonctions d'accès) et *setters* (fonctions de modification) habituelles, accessibles depuis le niveau `ConfigFile` par `getStrateConfig`.

6. Surface

La classe `Surface` permet de gérer les points et les segments, que ce soit lors de leur création comme de leur destruction. Celle-ci contient trois tables de hachage, une pour les points d'intersection qui délimitent les segments, une autre pour les points de contrôle qui gèrent la courbure du segment, et une dernière pour les segments. Ces tables servent de banque de données pour le dessin. Lorsqu'un nouveau point est créé, il est ajouté à sa table des points et les modifications nécessaires sont appliquées dans la table des segments. Ces trois tables sont paramétrées par une clé entière qui permet de sauvegarder un dessin et de le charger, en se souvenant des indices de chaque point. Cela permet de communiquer avec l'éditeur graphique pour qu'il sache que tel point possède tel indice.

- `int addSegment(int cleIP1, int cleIP2, int cleCP)` Cette fonction construit un nouveau segment dans la table des segments en utilisant les deux premiers paramètres pour définir quels sont ses points d'intersection (`cleIP1`, `cleIP2`), et le dernier pour son point de contrôle (`cleCP`). un indice spécial est utilisé s'il n'y a pas de point de contrôle à associer. La fonction rend d'autre part la clé du segment créé.
- `addSegment(Segment s, int cle)` Cette fonction permet de charger le segment `s` construit auparavant depuis le fichier de sauvegarde XML. la clé est elle aussi répercutée depuis le fichier de sauvegarde et il suffit de l'ajouter à la table, en bonne place.
- `(int, int) addIPoint(int x, int y)` Il s'agit de construire un nouveau point. Cette fonction n'est appelée que pour la construction de la première série de points, et prend en arguments les coordonnées du point à ajouter (`x`, `y`). L'ajout d'un point implique donc l'ajout d'un segment. Pour cela il est nécessaire de se souvenir du dernier point créé (tous ces points sont créés à la chaine). La fonction renvoie la clé du nouveau point, et celle du segment créé.
- `addIPoint(Point p, int cle)` Le même fonctionnement que `addSegment` pour le chargement a partir de la lecture d'un fichier XML. Cette fonction doit être appelée avant `addSegment` pour que les clés passées en paramètres de la reconstruction d'un segment soit les bonnes.

- `int addCPoint(int x, int y)` Ajoute un point de contrôle dans un segment sélectionné, possédant les coordonnées `x` et `y`. Le segment en question contient désormais les informations nécessaires à son paramétrage et à son affichage. La fonction rend la clé du point créé.
- `addCPoint(Point p, int cle)` Ajoute un point de contrôle depuis le fichier XML. Le fonctionnement est le même que les autres fonctions d'ajout via un fichier de sauvegarde XML.
- `(int, int) subdiviseSegment(int cleSegment, int x, int y)` Décompose un segment désigné par `cleSegment` en deux avec comme délimiteur le nouveau point de coordonnées `x`, `y`. Rend la clé du nouveau point créé, ainsi que celle du nouveau segment créé.
- `int removeIntersectionPoint(int cle)` Détruit un point d'intersection désigné par sa clé et fusionne les deux segments qui l'avaient pour extrémité. Rend la clé du segment détruit.
- `removeControlPoint(int cle)` Détruit un point de contrôle et remplace la valeur de sa clé par la clé spéciale dans le champ du segment qui l'avait pour point de contrôle.

On dispose de plus des fonctions d'accès (*getters*) et de modification (*setters*) pour les différents attributs.

7. Segment

La classe `Segment` est une structure simple permettant de gérer les points composant chaque segment. Les attributs de cette classe sont les clés des deux points d'intersection et du point de contrôle dans la table des points de la surface. On dispose pour cette classe des fonctions *getters* et des *setters* habituelles.

8. Point

En ce qui concerne la classe `Point`, celle-ci ne contient que les coordonnées du point, et les *getters* et *setters* classiques. On ne distingue pas les points d'intersection des points de contrôle du point de vue objet, mais ils sont référencés dans deux tables différentes dans `Surface`. Cependant, ce choix pourrait être amené à évoluer selon les vérifications que l'on souhaitera effectuer lors de l'implémentation de la classe `Surface`.

B. Interface graphique (Qt)

Les classes dont le nom commence par la lettre *Q* sont des classes de Qt.

1. Fenêtre d'accueil

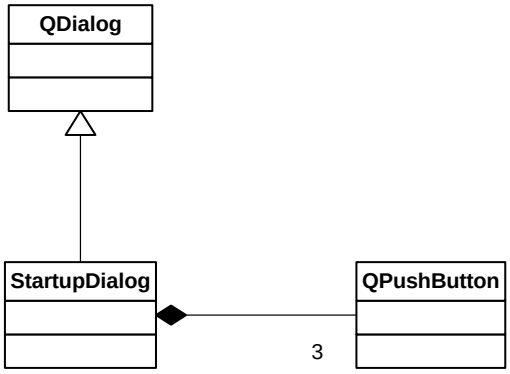

Figure 2 Schéma UML, Fenêtre d'accueil

La fenêtre d'accueil (`StartupDialog`) dérive directement de `QDialog`. Elle possède trois boutons-poussoirs permet de choisir le mode (« nouveau projet », « ouvrir un projet » ou « détection de contour »).

2. Fenêtre principale

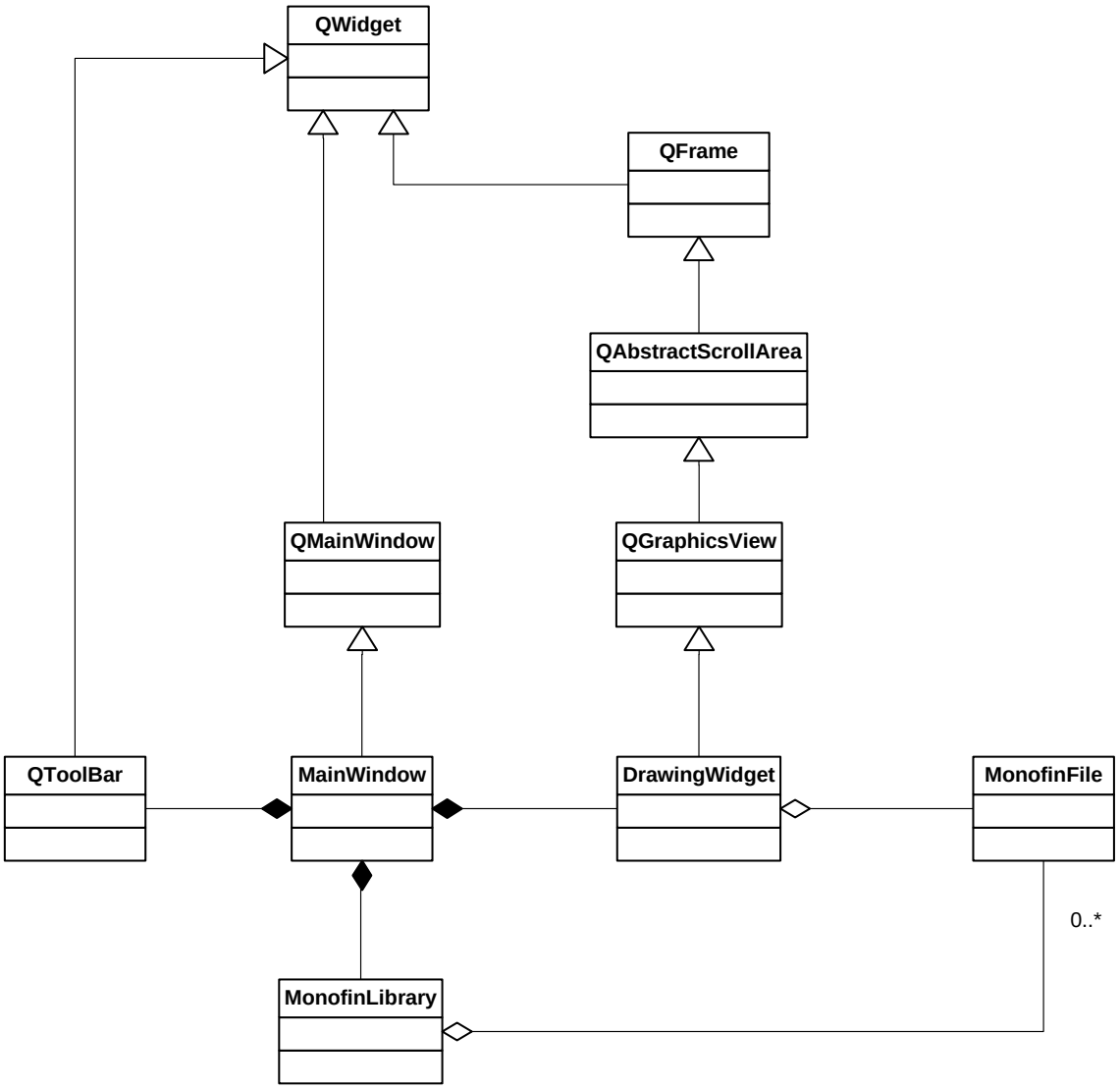

Figure 3 Schéma UML, fenêtre principale

La fenêtre principale dérive de `QMainWindow`. Elle possède une barre d'outils (`QToolBar`), un *widget* de dessin et une bibliothèque de monoplames prédéfinies (`MonofinLibrary`).

Il est intéressant de noter que la plupart des objets graphique de Qt dérivent de `QWidget`. Cela présente de multiples avantages. Tout d'abord les fonctions nécessaires à l'affichage et à la disposition sont communes. Ensuite cela nous permettra d'être plus souples si on veut changer la présentation du logiciel et proposer une interface multi-fenêtrée puisqu'un `QWidget` qui n'appartient à aucune fenêtre devient lui-même une fenêtre.

3. Autre fenêtre ou widget

En ce qui concerne les autres *widgets*, ils dérivent tous de `QWidget` et utilisent simplement des objets Qt pour la saisie de données par l'utilisateur. Comme ils seront générés automatiquement grâce au *designer* de Qt, l'étude leur diagramme de classe n'est pas d'un grand intérêt.

C. Génération du script COMSOL

La génération du script COMSOL est une partie sensible de l'application dans la mesure où celle-ci doit rester le plus extensible possible. De cette manière, les futures évolutions de l'application pourront être facilement accueillies. Ainsi, nous proposons les classes suivantes pour permettre la génération du script COMSOL responsable du lancement de la simulation.

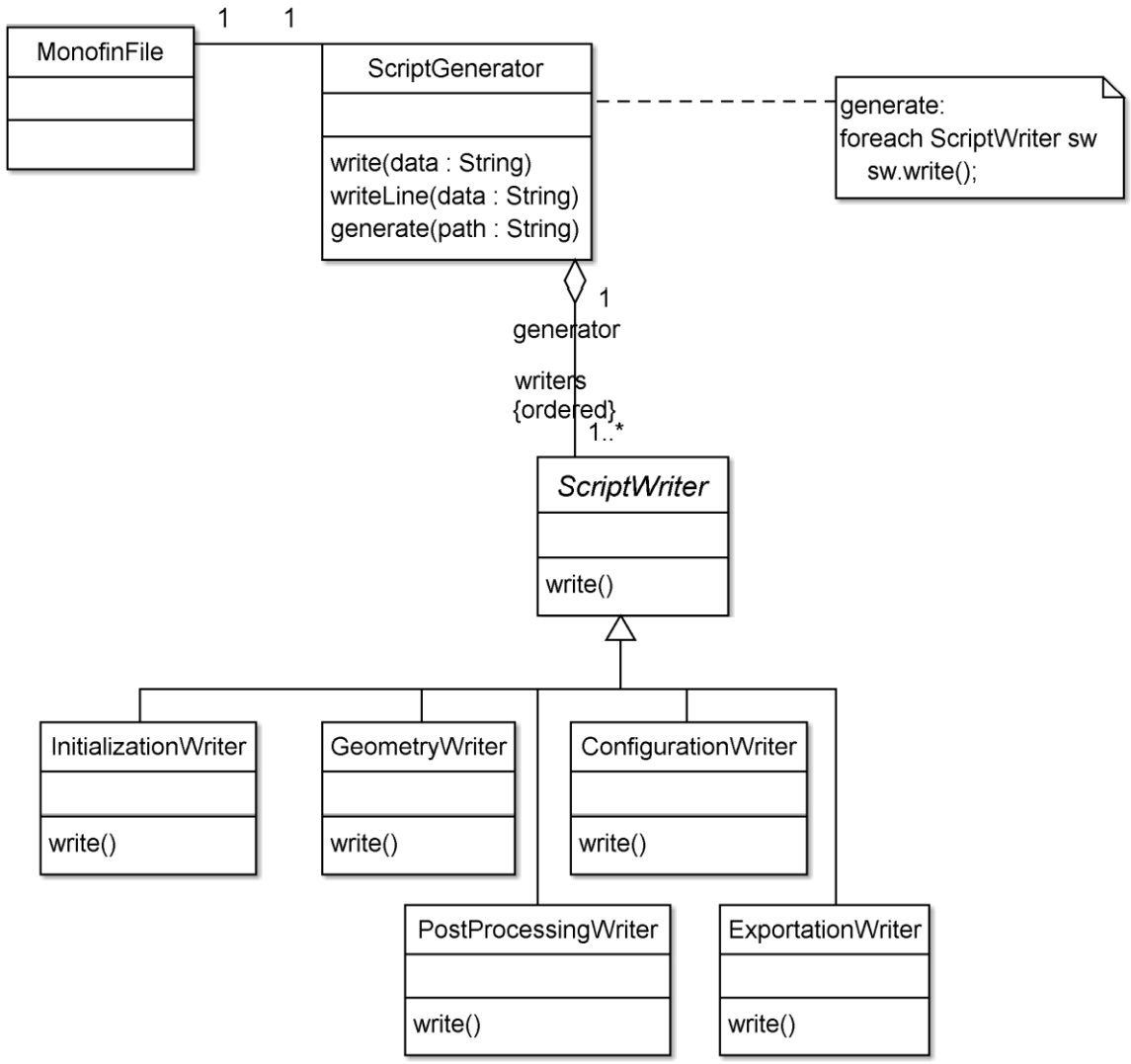

Figure 4 Schéma UML, génération du script COMSOL

Chaque classe héritant de `ScriptWriter` sera chargée de l'écriture d'une portion du script. La classe `ScriptGenerator` contiendra un tableau ordonné de `ScriptWriter`, l'ordre est ici important car un script COMSOL s'exécute séquentiellement. Classiquement, un générateur de script (instance de `ScriptGenerator`) sera composé (dans l'ordre) des classes suivantes :

- `InitializationWriter` qui se chargera de faire les initialisations nécessaires au script
- `GeometryWriter` qui transformera la géométrie de la monopalme provenant de la classe `MonofinFile` en une suite de primitives graphique connues de COMSOL. Les primitives

permettront de reconstruire notre géométrie de monopalme dans un format que COMSOL comprend.

- `ConfigurationWriter` qui se chargera de paramétrer la simulation. C'est par exemple ici que l'on précisera le matériau physique constituant chaque strate de la monopalme.
- `PostProcessingWriter` qui se chargera de paramétrer les données de post-traitement que devra nous retourner le script COMSOL. C'est ici que l'on pourra par exemple régler la longueur de la vidéo montrant le mouvement de la monopalme, ou encore spécifier les échelles de couleur à utiliser sur la vidéo.
- `ExportationWriter` se chargera de préciser où les données de la simulation pourront être récupérées. Ce point reste à définir mais classiquement il s'agira de créer un dossier temporaire dans lequel on laissera COMSOL écrire les fichiers résultants de la simulation (vidéo, rapport de simulation, *dump* des variables de la simulation, images de rendu…).

IV. Analyse dynamique

A. Schéma d'enchainement des fenêtres

Le schéma ci-après présente l'ensemble des différentes fenêtres qui seront réalisées pour l'application. Chaque état représente ici l'ouverture d'une fenêtre, mais il ne présente pas les interactions entre celles-ci. Dans le cas général, lors d'une annulation ou d'une non validation, on retourne à la fenêtre précédente (la fenêtre de démarrage ou la fenêtre principale de dessin). Les deux fenêtres « chargement d'une image » et « chargement d'un projet » seront très similaires, mais elles serviront à deux cas d'utilisation différents. Le diagramme d'activité dans la partie suivante présente plus en détail les interactions entre les fenêtres.

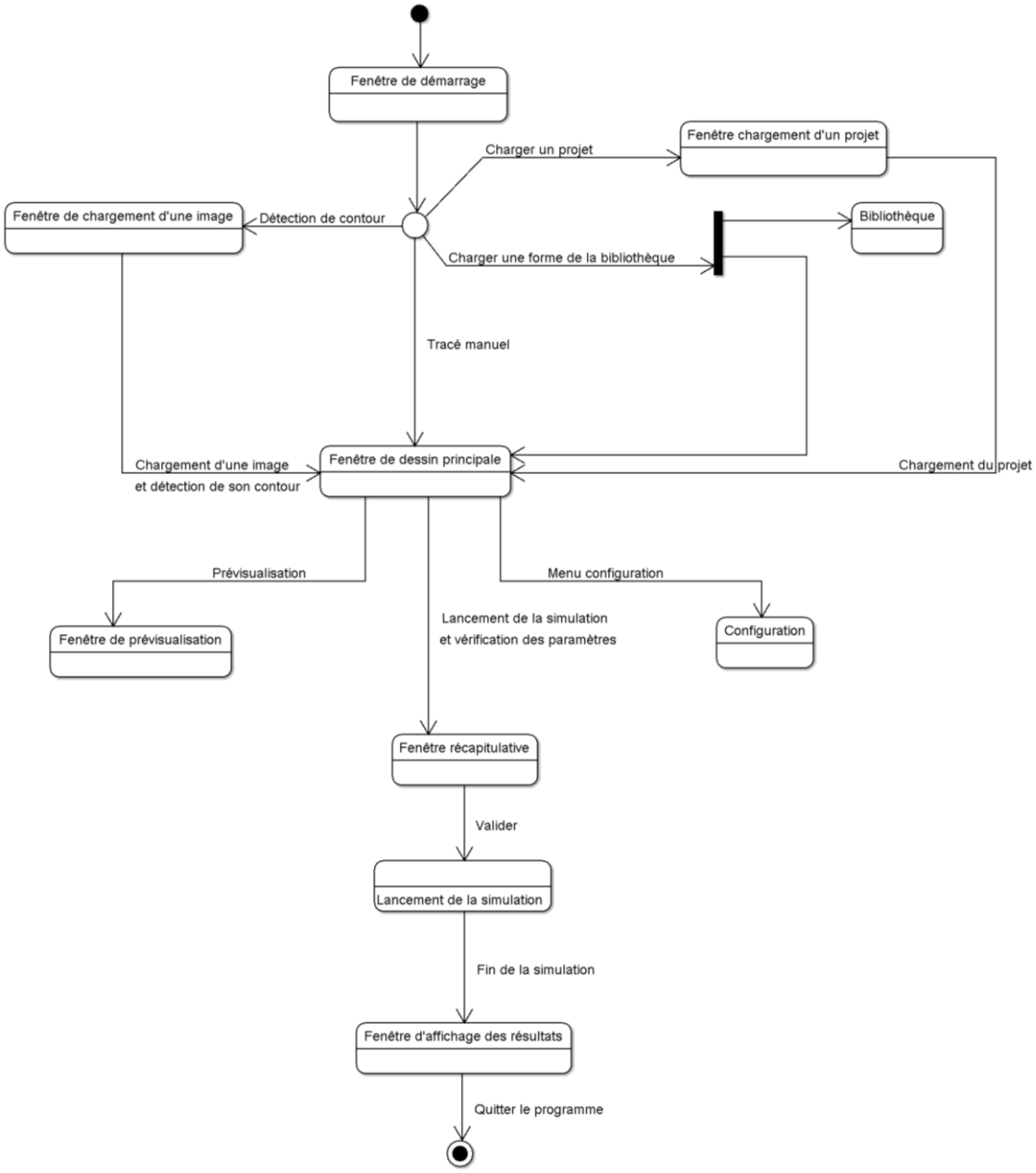

Figure 5 Schéma d'enchainement des fenêtres

B. Diagramme d'activité général

Ce diagramme est assez général et présente les actions générées par les différentes fenêtres accessibles à l'utilisateur. La partie dessin de la palme n'est pas traitée dans ce diagramme (voir Diagramme d'activité du tracé manuel page 19). Chaque action est détaillée plus précisément dans la suite du rapport.

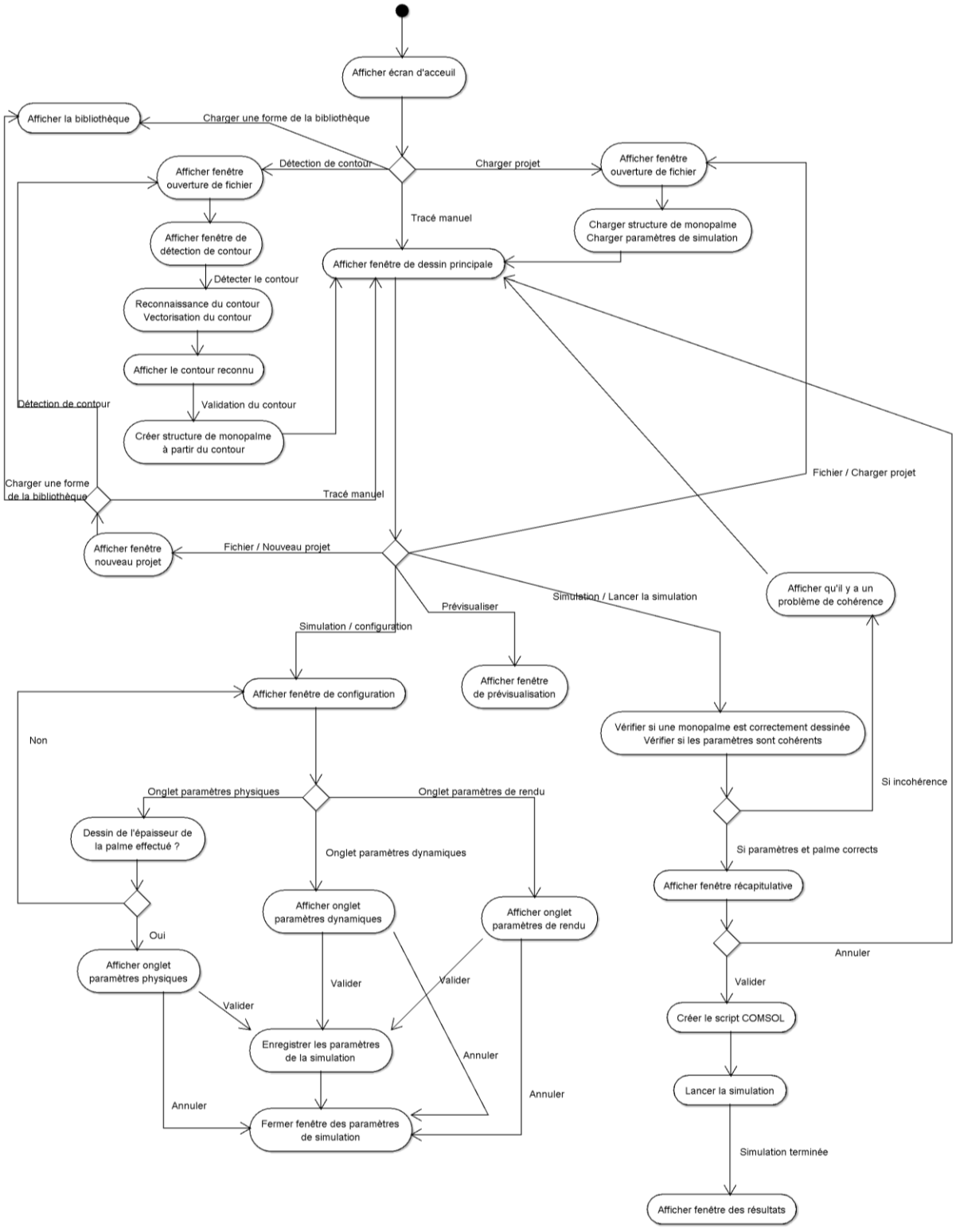

Figure 6 Diagramme d'activité général

C. Diagramme d'activité du tracé manuel

Ce diagramme présente les différentes étapes à contrôler lorsque l'utilisateur commence le tracé manuel d'une monopalme dans la zone de dessin. La partie en rouge est à ajouter si on contraint l'utilisateur à travailler toujours dans une zone de travail prédéfinie (c'est-à-dire qu'on lui impose de travailler toujours au-dessus de l'axe de symétrie par exemple), ou à enlever si l'utilisateur peut travailler aussi bien en dessous qu'au-dessus de l'axe de symétrie. Ce choix ne change *a priori* pas beaucoup l'implémentation, et si l'implémentation laissant le choix à l'utilisateur est facile à coder, ce choix sera sûrement validé (la connaissance en librairie Qt est limitée et ne permet pas pour le moment d'effectuer ce choix de manière sûre). Voir le diagramme à la page suivante.

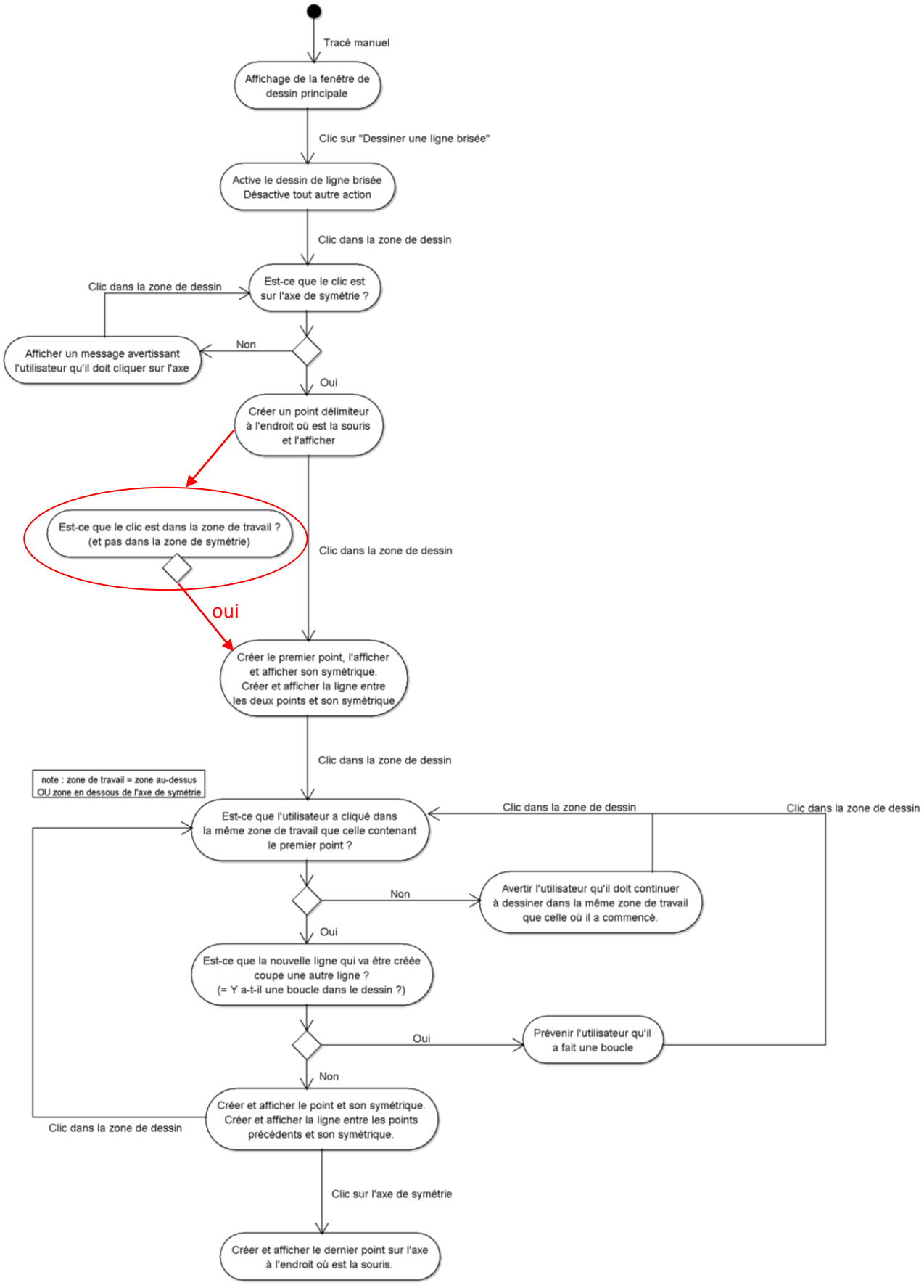

Figure 7 Diagramme d'activité du tracé manuel

D. Diagrammes de séquence

Les diagrammes de séquences suivants analysent les différents scénarios correspondants essentiellement aux interactions de l'utilisateur avec l'interface graphique. Ils permettent notamment de mettre en évidence les connexions entre les différents objets. Ces connexions seront assurées pour la plupart par le mécanisme de « signal/slot » de la librairie Qt. Afin d'adapter les diagrammes habituels à ce nouveau mécanisme, le formalisme suivant a été utilisé :

function()	un appel de fonction sur un autre objet		un retour explicite
function()	un appel de fonction sur le même objet	message	un message, généralement le résultat d'une fonction
signal	un signal Qt		une attente, ou une période de calcul

1. Nouveau projet

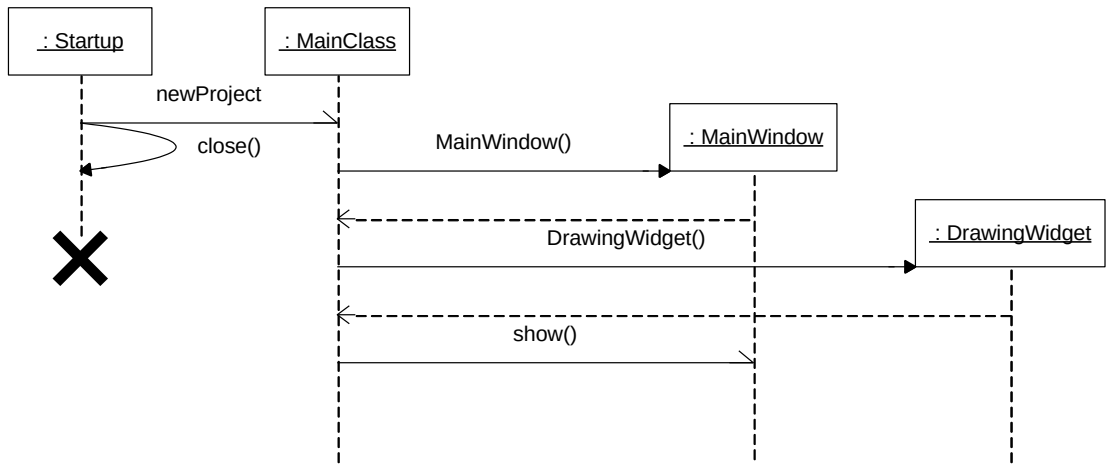

Figure 8 Diagramme de séquence, Nouveau projet

Scénario :

L'utilisateur choisit l'option « nouveau projet ». L'information est envoyée à la classe principale (nommée ici `MainClass`) qui se charge de créer et d'afficher la fenêtre principale et le *Widget* de dessin.

Remarque :

`DrawingWidget` est contenu dans la fenêtre `MainWindow`, sa création aurait pu être effectuée par `MainWindow`. Pour des raisons de simplicité de dialogue lors des traitements ultérieurs, il est plus simple de laisser `MainClass` s'en occuper. L'appartenance graphique de `DrawingWidget` à `MainWindow` sera réalisée par le mécanisme de « parent/enfant » de Qt.

2. Ouvrir un projet

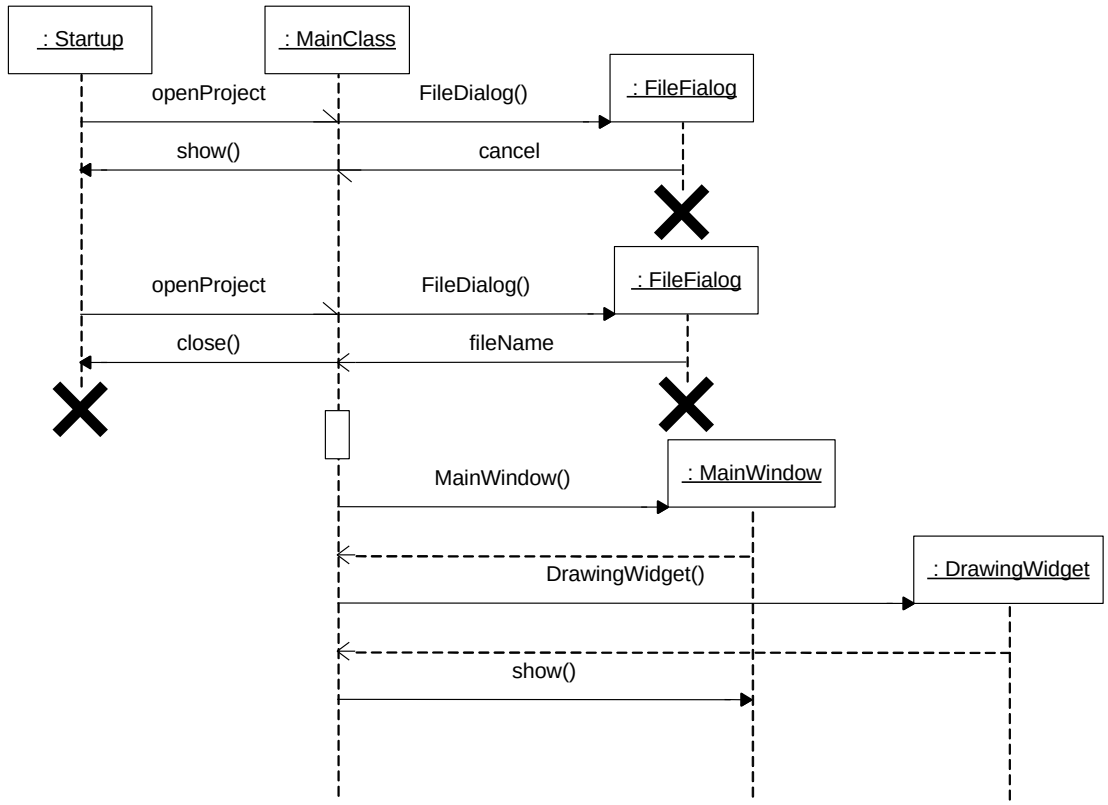

Figure 9 Diagramme de séquence, ouvrir un projet

Scénario :

L'utilisateur choisit l'option « ouvrir un projet ». L'information est envoyée à la classe principale qui crée une boîte de dialogue (`FileDialog`) d'ouverture de fichier. Si l'utilisateur annule l'opération. La fenêtre d'accueil est de nouveau affichée au premier-plan.

Lorsqu'un fichier est sélectionné, la boîte de dialogue renvoie le nom du fichier qui peut-être traité. Si le fichier est valide, l'étape de création de la fenêtre principale débute comme précédemment.

Remarque :

Lorsque le fichier n'est pas valide, on peut envisager deux cas : le programme se termine, ou on affiche à nouveau la fenêtre d'accueil. A ce stade, aucune des deux solutions n'est privilégiée. Le choix se fera par retour d'expérience des utilisateurs.

3. Extraction de contour

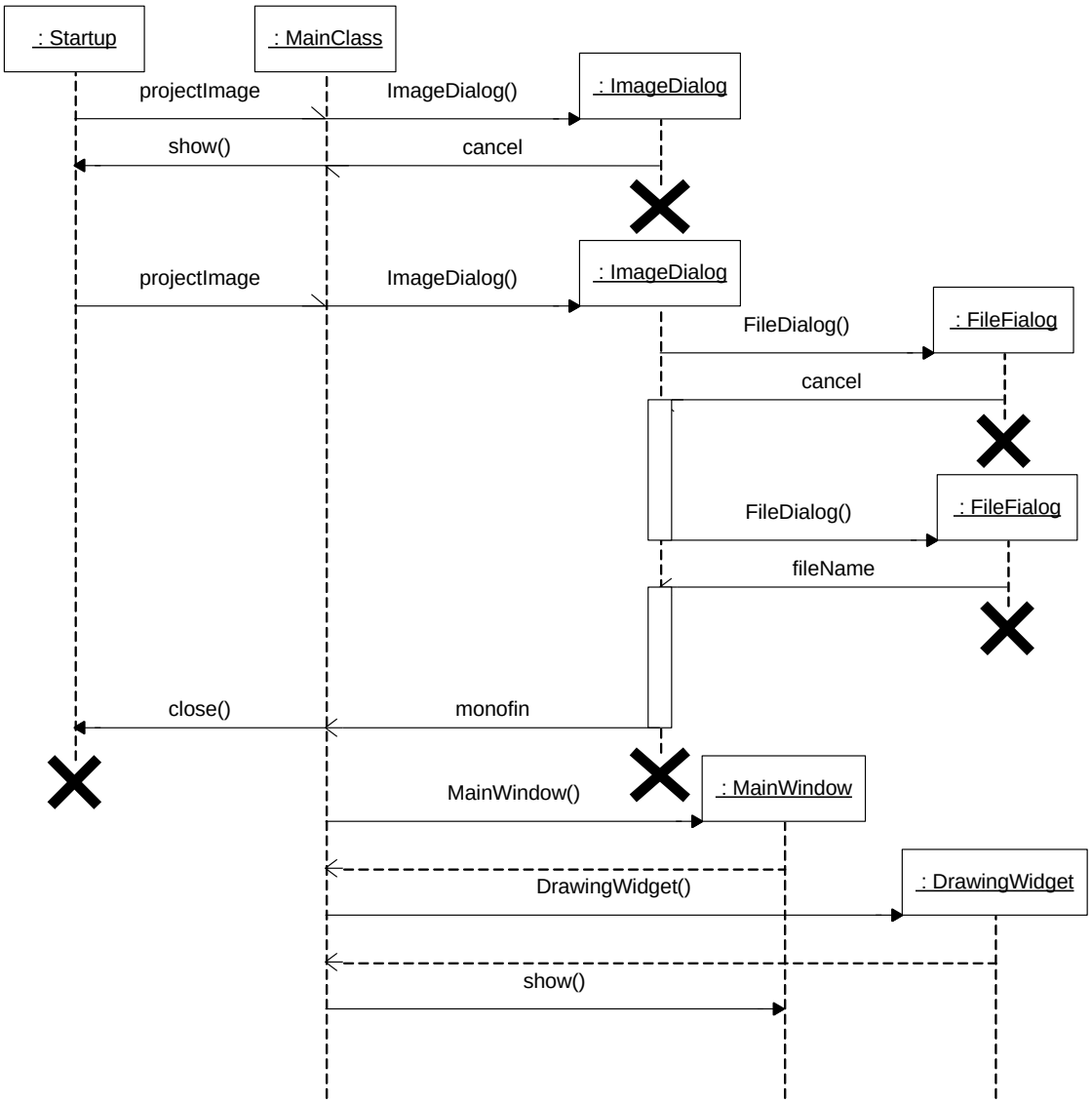

Figure 10 Diagramme de séquence, Extraction de contour

Scénario :

L'utilisateur choisit l'option « extraction de contour ». La classe principale crée et affiche une boîte de dialogue `ImageDialog`. En cas d'annulation, on retourne à la fenêtre d'accueil.

L'utilisateur peut sélectionner une image grâce à la boîte de dialogue de sélection de fichier. Il a la possibilité d'effectuer certains traitements sur l'image avant de lancer l'extraction de contour.

Lorsque la détection est terminée, on crée la fenêtre principale.

4. Configuration de la simulation

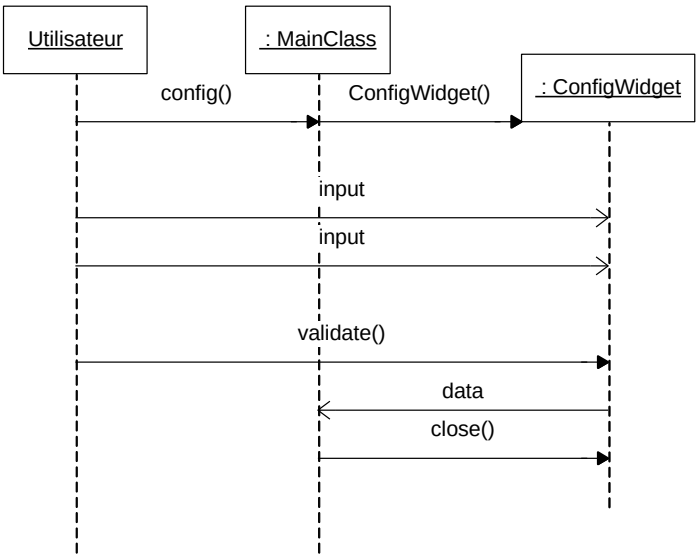

Figure 11 Diagramme de séquence, configuration de la simulation

Scénario :

L'utilisateur se contente de saisir des informations qui sont ensuite récupérées et retransmises à la classe principale. A la validation du *widget*, La classe principale récupère les données puis ferme la fenêtre. Cette dernière n'est pas détruite mais est simplement masquée, puisqu'une utilisation ultérieure est envisageable.

Remarque :

Ce scénario est commun à tous les *widgets* de configuration.

V. Conclusion

Les choix techniques sont à présent clairs tant au niveau des langages (C++, Qt), que des algorithmes à employer pour des opérations comme l'extraction de contour. Le format de sauvegarde est également mis au point, et répond à toutes les exigences imposées par le chargement et la sauvegarde d'un projet. Celui-ci peut compter sur une structure de données hiérarchisée, qui est elle-même transparente vis-à-vis de l'interface.

La génération du script est à présent précisée, et l'enchainement séquentiel proposé ici devra être respecté lors de l'implémentation. Les résultats produits par le script devront ensuite être traités pour apparaitre sous une forme exploitable par l'utilisateur.

D'autre part, les différents aspects de l'interface graphique sont à présent précisés, ainsi que leur fonctionnement. L'enchainement des étapes n'est désormais plus soumis à un fil d'Ariane (cf. Dossier n°2 : Spécifications (05/12/08)) précis mais est laissé à la discrétion de l'utilisateur. Enfin, les interactions entres les différentes fonctionnalités de la fenêtre de dessin ont été passées en revue.

Les différents points clé de la réalisation de l'application sont donc clairement établis. L'implémentation devra respecter ces consignes, mais il ce peut que celle-ci dévoile des failles dans la conception. Les écarts entre ce qui est présenté aujourd'hui et le résultat final seront donc notés et compilés dans le rapport final, pour évaluer la justesse de cette conception.

VI. Rapports précédents

- *Dossier n°1 : Rapport de pré-étude (24/10/08)*
- *Dossier n°2 : Spécifications (05/12/08)*
- *Dossier n°3 : Planification (17/12/08)*

VII. Travaux cités

Kass, M., Witkin, A., & Terzopoulos, D. (1987). Snakes: active contour models. *International Journal of Computer Vision* , 259–268.