Ce document sert de documentation technique à l'application LogoGo que j'ai réalisé dans le cadre de mon Travail Pratique individuel.

Documentation Technique

Rapport de TPI

Walter Jauch

1. Table des matières

3. Résumé du rapport TPI

2.1. Situation de départ

Les TPI 2020 ont eu lieu de façon particulière. À cause le la pandémie (COVID-19) qui nous a pris par surprise, nous avons réalisé nos TPI à la maison. Nous avons disposé (comme pour les élèves ayant passé le TPI en 2019) de 11 jours au total. Malgré les conditions particulières, aucune modification n'a été apportée sur la durée ou sur la difficulté de ce travail de fin de formation. Mise à part une date reporté d'environ un mois, le TPI a eu lieu "normalement".

Pour mon TPI j'ai réalisé une application WindowsForm en C# orienté objet permettant à l'utilisateur de créer des logos. Il est possible d'ajouter des formes et du texte sur une certaine zone et d'en modifier les propriétés. L'utilisateur peut changer la taille, la couleur, l'épaisseur, le remplissage, la position, et d'autres propriétés permettant de faire ce qu'il souhaite des formes mises à disposition. Il y a également un système de calques permettant de choisir l'ordre dans lequel les formes apparaissent. L'utilisateur peut choisir sur quel calque créer les formes et peut aussi les changer de calque après leur création. Il est possible d'exporter le logo en image (plusieurs formats sont disponibles). Si l'utilisateur souhaite continuer la création de son logo plus tard, il peut sauvegarder le logo puis le rouvrir. La sauvegarde se fait en créant un fichier XML.

2.2. Mise en œuvre

Durant le premier jour de TPI, j'ai pris du temps pour analyser le cahier des charges en le lisant plusieurs fois pour être sûr des fonctionnalités qui devaient être disponibles à la fin des 11 jours. Je n'ai pas eu besoin d'estimer les tâches que j'avais à faire puisqu'un planning prévisionnel m'a été fourni. Ce planning m'a été d'une grande aide car je n'ai eu qu'à reprendre la même structure et la remplir au fur et à mesure que j'avançais pour obtenir mon planning effectif.

Après un certain temps de réflexion sur la façon dont j'allais m'y prendre, j'ai réalisé un diagramme de classe qui semblait correct. J'ai pris un long moment pour créer l'interface principale et pour faire en sorte qu'elle tienne la route. L'image d'exemple donnée dans le cahier des charges m'a servi d'inspiration.

En parallèle de la réalisation de l'application, je me suis arrangé pour maintenir un certain rythme dans la rédaction des différentes documentations pour ne pas me retrouver submergé par la documentation en fin de projet. J'ai également sauvegardé mon projet entre une et trois fois par jour pour garder une trace des modifications ainsi que pour pouvoir, en cas de problème, revenir à une version précédente. Pour sauvegarder mon projet, j'ai utilisé git qui, étant la manière que j'ai le plus utilisé pour sauvegarder efficacement des projets au sein du CFPT, m'a semblé être le meilleur choix.

2.3. Conclusion

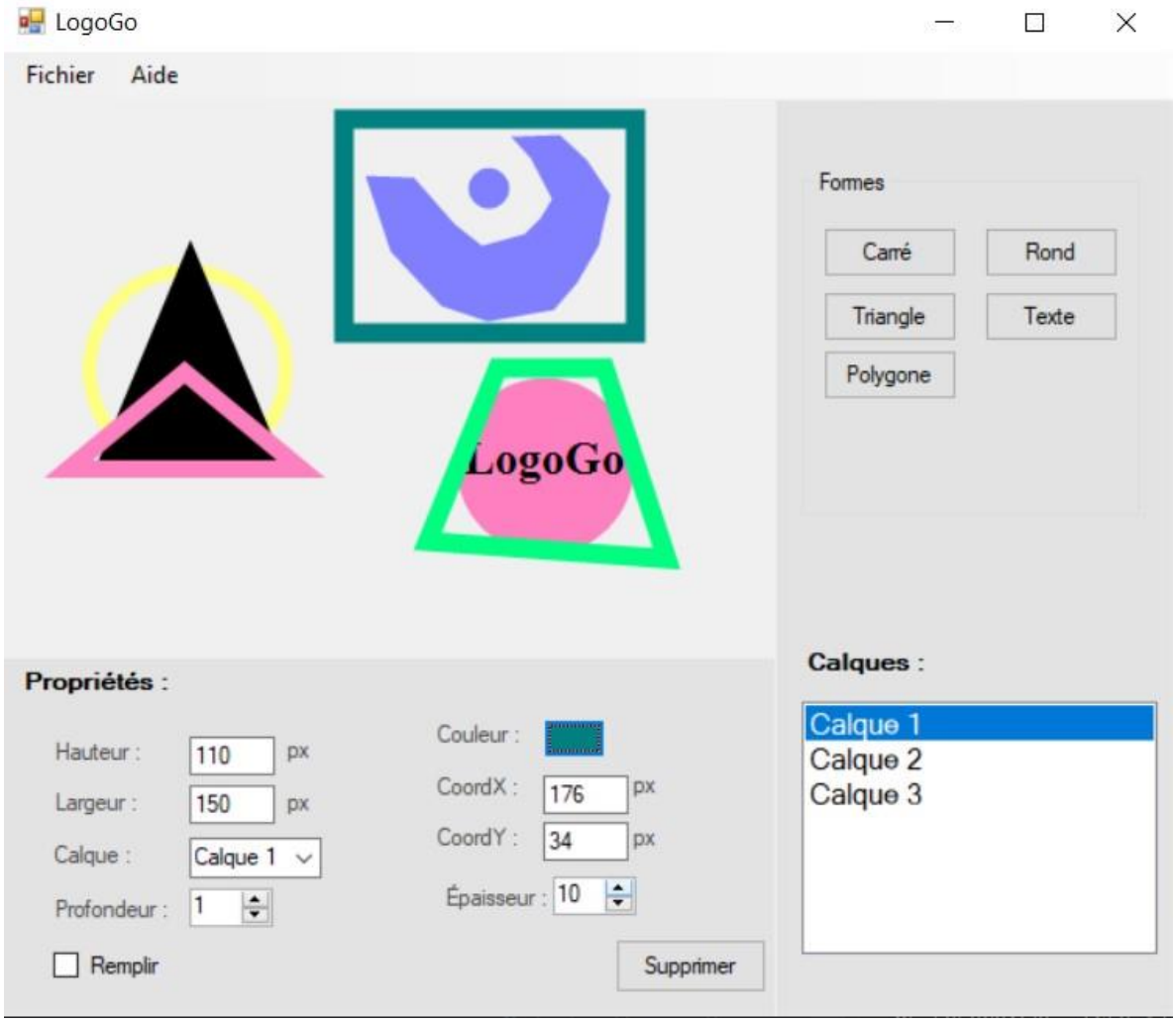
Pendant ces 11 jours de travail, j'ai réussi à réaliser tous les points demandés dans le cahier des charges. Toutes les fonctionnalités sont utilisables et je n'ai pas eu de retard particulier. Durant la première moitié du travail, le planning effectif était assez différent du planning prévisionnel mais cette différence s'est estompée au fil des jours et les deux plannings se sont équilibrés.

3. Introduction

Cette documentation contient les détails de la réalisation de mon TPI (Travail Pratique Individuel). Elle a pour but d'expliquer précisément les étapes de ce travail.

Dans le cadre de ce travail de fin de formation, j'ai dû réaliser une application nommée LogoGo.

C'est une application Windows Forms réalisée entièrement en C# orienté objet. Elle permet de créer un logo de A à Z en utilisant des formes géométriques ainsi que du texte que l'utilisateur peut modifier. Un certain nombre de propriétés est modifiable pour chaque forme et ces formes se trouvent sur des calques. Les calques servent évidemment à gérer, dans le cas présent, au moins trois plans différents. L'une des fonctionnalités clé de ce projet est bien entendu la possibilité d'exporter le logo en image. Le logo peut également être enregistré dans le but de, par exemple, modifier le logo plus tard.



4. Méthodologie

Pour tout projet conséquent, il faut avoir une méthodologie de travail qui nous aide à raisonner et à organiser le travail. J'ai donc choisi d'appliquer une méthodologie qui nous a été enseignée en cours au CFPT : la Méthode en 6 Étapes.

Comme son nom laisse imaginer, elle contient, en tout, 6 étapes :

1. S'informer

   Ici, la source principale d'information étant l'énoncé, une lecture complète et détaillée du document est nécessaire pour pouvoir cerner le travail à réaliser et savoir où aller.

2. Planifier

   Puisque le planning prévisionnel m'a été fourni, j'ai décidé de garder la structure de ce dernier et de la remplir avec l'avancement réel du projet pour obtenir un planning effectif.

3. Décider

   Il faut souvent choisir une méthode à une autre au cours d'un projet. Figurent dans le journal de bord les décisions importantes que j'ai décidé de prendre.

4. Réaliser

   Faisant suite à la phase de décision, la réalisation est là pour que les décisions prises soient appliquées.

5. Contrôler

   Afin d'éviter toute régression, il est important de contrôler les fonctionnalités qui sont ajoutées à l'application.

6. Évaluer

   L'évaluation est une sorte d'auto-critique qui permet de savoir ce qui aurait pu être amélioré. La conclusion de **ce document** sert d'évaluation. En plus de la conclusion, j'ai réalisé un petit bilan quotidien après chacun des 11 jours.

5. Résumé du cahier des charges

5.1. Organisation

Nom Complet	Adresse e-mail	Domaine
Monet Stéphane	stephane.monet@gmail.com	Expert
Fontanini Alain	alain.fontanini@outlook.com	Expert
Bonvin Pascal	edu-bonvinp@eduge.ch	Responsable
Jauch Walter	walter.jch@eduge.ch	Candidat

5.2. Objectifs

Le but est de concevoir une application C# permettant de créer des logos. Il doit être possible de modifier des formes dans plusieurs calques ainsi que de sauvegarder et d'exporter le logo en plusieurs formats d'images.

Le projet doit être terminé dans les 11 jours durant lesquels se passe le TPI, documentation comprise.

5.3. Matériel

Pour ce travail, j'utilise le matériel suivant :

- Mon ordinateur personnel
- Windows 10 Édition Famille
- Microsoft Visual Studio 2019
- Typora
- Sous-système Debian GitHub
-

5.4. Livrables

À la fin des 11 jours de travail, les livrables du projet doivent être les suivants :

- Planning
- Rapport de projet
- Manuel utilisateur Journal
- de travail

5.5. Points techniques

Comme chaque projet, celui-ci comporte sept points techniques obligatoires qui sont spécifiques au projet. Ces points correspondent aux points A14 à A20 de la grille d'évaluation du TPI.

Voici les points techniques obligatoires à ce projet :

- Le système de sauvegarde/chargement d'un logo fonctionne
- Un diagramme de classe documente le projet
- Les objets graphiques ont une classe mère commune (héritage)
- Le logiciel de logo permet de composer un logo avec au moins trois formes et du texte
- Le planning réel est comparé au planning prescrit
- Le logiciel comporte au minimum trois plans graphiques
- Le logiciel permet d'exporter un logo au format jpeg ou png

La réalisation de ces 7 points est détaillée dans la suite de ce document.

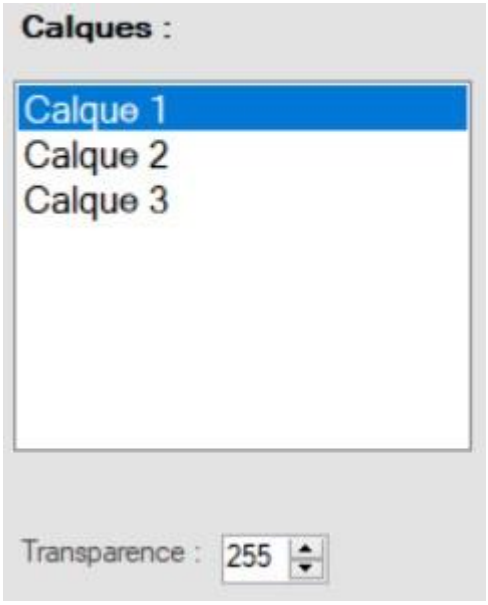
6. Analyse fonctionnelle

Une application se doit d'avoir un certain nombre de fonctionnalités que l'utilisateur peut exploiter pour, ici, créer ce qu'il désire. Ce chapitre de la documentation explique les fonctionnalités disponibles ainsi que les détails graphiques de LogoGo.

6.1. Liste des fonctionnalités

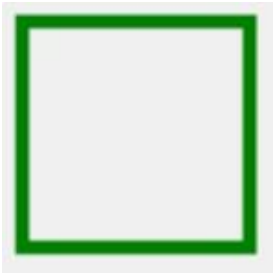
6.1.1. Sélection du plan de travail

L'utilisateur doit pouvoir choisir sur quel plan les formes sont créées. Les formes doivent donc se positionner sur le calque que l'utilisateur a choisi. Par défaut, le calque "Calque 1" est sélectionné. En ce qui concerne l'ordre d'apparition des calques, j'ai décidé de que le calque 1 serait le premier à apparaître, le calque 2 apparaît ensuite et ainsi de suite. Ainsi, le dernier calque apparaît au-dessus des autres.



6.1.2. Création d'une forme

L'utilisateur peut créer une forme qui, selon le bouton choisit, varie. La forme apparaît alors sur le calque que l'utilisateur à précédemment eu l'occasion de sélectionner. Toutes les formes ont des propriétés de base qui sont (pour la majorité des formes actuellement disponibles), 100 pixels de hauteur ainsi que 100 pixels de largeur, par exemple.



6.1.3. Sélection d'une forme

L'utilisateur a la possibilité de sélectionner une forme en cliquant dessus avec la souris. Il est désormais possible de voir les propriétés de cette forme. Les formes n'ont pas toutes les mêmes propriétés : un texte n'a pas les mêmes caractéristiques qu'un rond. Pour cela, deux interfaces différentes sont prévues avec les propriétés respectives de chaque objet.

Nous pouvons, ici, cliquer avec la souris sur le rond ou sur le texte.

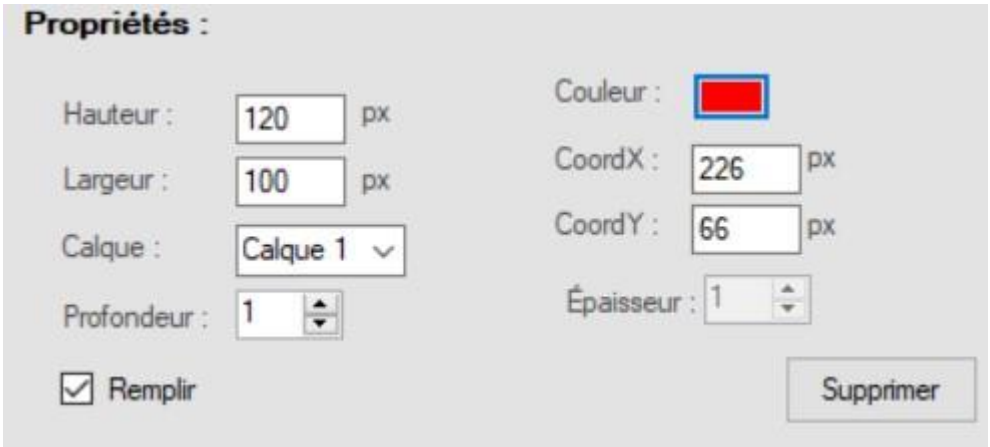
6.1.4. Voir les propriétés d'une forme

Si l'utilisateur choisit une forme, les propriétés de celle-ci deviennent visibles à partir de la section "Propriétés" de la fiche principale. Les propriétés visibles dépendent de la forme sélectionnée. Une forme standard a des propriétés du genre : Hauteur, Largeur, Épaisseur, etc. Si c'est un texte que l'utilisateur sélectionne, d'autres propriétés sont disponibles. Des propriétés telles que la taille de la police ou le texte à afficher sont affichées.

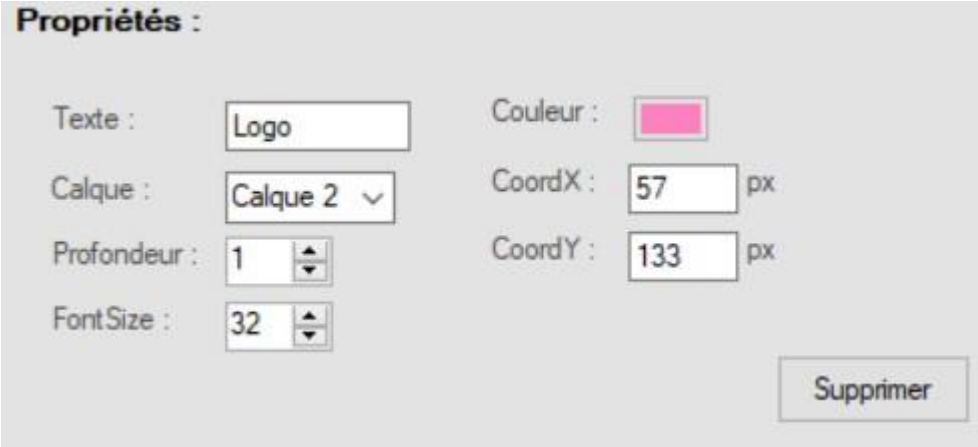
Cependant, certaines propriétés sont communes. La position (en axe X comme en axe Y) ainsi que la couleur, par exemple, ne dépendent pas du type d'objet choisit.

Dans tous les cas, les propriétés se mettent à jour dès que l'utilisateur sélectionne un autre objet. Elles peuvent également se mettre à jour si l'objet n'existe plus (en cas de suppression par l'utilisateur).

Pour reprendre l'exemple du rond et du texte, voilà ce que l'on devrait voir lorsque nous cliquons sur le rond rouge :



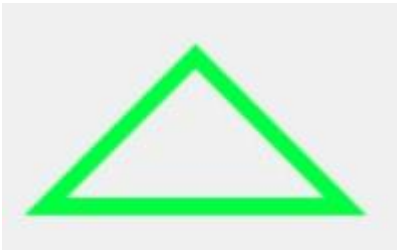
Voici l'affichage si nous cliquons sur le texte rose :



6.1.5. Modification des propriétés d'une forme

L'utilisateur peut paramétrer une forme comme il le souhaite. Les propriétés visibles sont modifiables et les modifications sont automatiquement appliquées et visibles sur la forme concernée.

Pour illustrer cette fonctionnalité, voilà deux images, avant et après avoir modifié les propriétés d'un triangle :



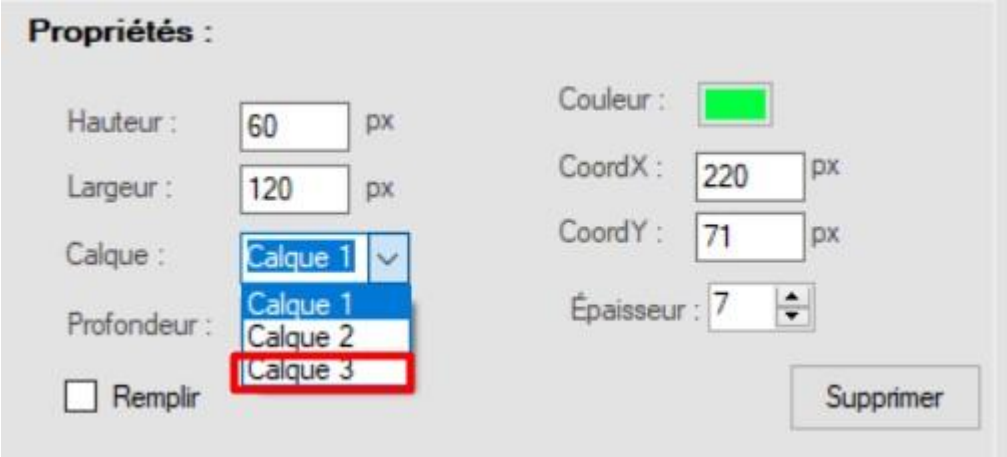
Les propriétés modifiées ici sont celle marquées en rouge sur l'image qui suit :

6.1.5.1. Modification du calque d'une forme

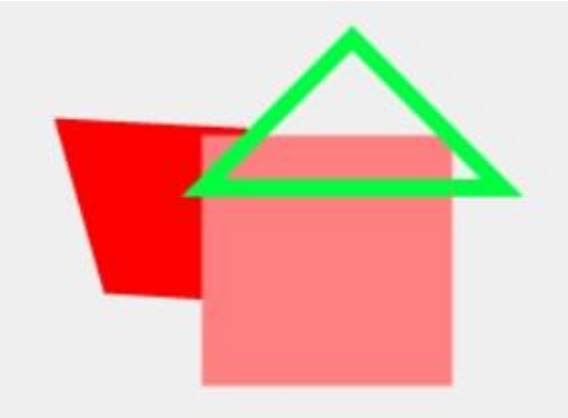
L'utilisateur peut aisément changer une forme de calque, même après la création de la forme. Il peut le faire en modifiant la propriété correspondante et en choisissant le calque souhaité. On voit alors la forme changer de calque. En passant du calque 1 au calque 3, par exemple, on voit la forme se placer au-dessus de toutes les formes du calque 1 et du calque 2. Étant donné qu'elle serait la dernière ajoutée au calque 3, elle serait également par-dessus des formes qui faisaient déjà partie de ce calque.

Voici un exemple concret :

Sur l'image ci-dessus, on distingue trois formes : un triangle vert, un polygone rouge, ainsi qu'un carré saumon. Le carré est sur le calque 2 et les deux autres sont sur le calque 1. En cliquant sur le triangle, nous pouvons voir sur quel calque il est et modifier la propriété calque. Comme l'image le montre, il suffit, ici, de cliquer sur "Calque 3" pour changer le triangle de calque :



Comme expliqué plus tôt dans le chapitre, le triangle passe au-dessus des formes des calques 1 et 2. Voilà à quoi ressemblera la disposition des formes :



6.1.5.2. Modification de profondeur

Deux formes dans un même calques apparaissent dans un certain ordre (ordre de création). Cependant, l'utilisateur peut décider de cet ordre en modifiant la propriété correspondante. Tout comme pour les calques, on voit la forme se mettre au-dessus des formes ayant une profondeur moins élevée.

Voici une démonstration :

Ces deux formes se trouvent toutes deux sur le même calque. Il suffit d'augmenter la propriété Profondeur du carré comme suit pour changer l'ordre des formes.

Voilà le résultat :

6.1.6. Sauvegarde d'un logo

Le projet peut être sauvegardé. En sauvegardant, l'utilisateur doit choisir le chemin (destination) de la sauvegarde. Il peut ensuite choisir le nom du fichier qui sera sauvegardé. L'extension du fichier est fixe (.xml).

Si l'utilisateur ne choisit aucun nom, le fichier prend automatiquement le nom "Logo.xml". En ce qui concerne la destination du fichier, il est impossible de ne choisir aucune destination puisque le SaveFileDialog s'ouvre obligatoirement avec un chemin existant.

6.1.7. Chargement d'un logo

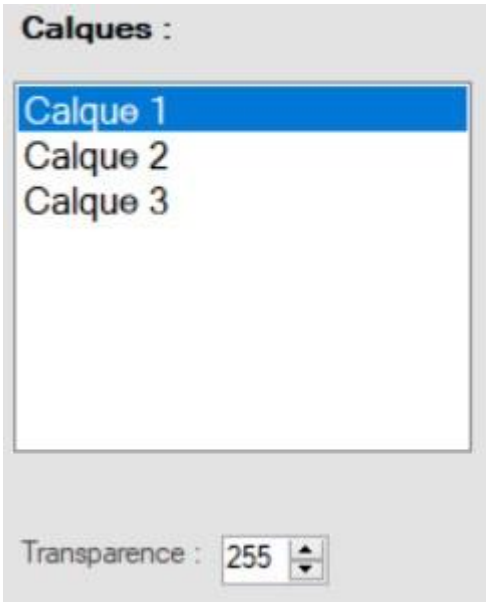
Les fichiers sauvegardés peuvent être à nouveau ouverts et chargés dans LogoGo. Les formes précédemment sauvegardées se retrouvent alors de nouveau dans l'application avec les bonnes propriétés et sur les bons calques. L'utilisateur choisit donc un fichier à ouvrir (obligatoirement un fichier .xml). Si le fichier ouvert par l'utilisateur ne correspond pas à ce que l'application attend (une structure XML correcte comprenant des objets qui correspondent au projet), un message d'erreur apparaît.

6.1.8. Exportation d'un logo

Une fois que l'utilisateur estime que son logo est terminé, il peut l'exporter en image. Il est bien entendu possible de choisir où le fichier sera enregistrer. Plusieurs choix d'extensions sont également disponibles.

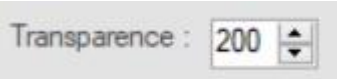
Le logo doit, avant tout, avoir la bonne taille. Pour cela, j'ai élaboré une méthode permettant de recadrer l'image finale là où se trouvent les formes créées par l'utilisateur afin de pouvoir couper l'image au bon endroit.

6.1.9. Modification de le Transparence



Comme vous pouvez le voir, vous pouvez voir et modifier la transparence du calque sélectionné. Voici un exemple de transparence :

Sur cette image, on peut observer que le rond noir (qui est sur le calque 1) est visible à travers le triangle rouge (qui est sur le calque 2). La transparence du calque 1 n'a pas été changée. Le calque 2, lui, a vu sa transparence passer de 255 à 200 comme vous pouvez le voir sur l'image cidessous :



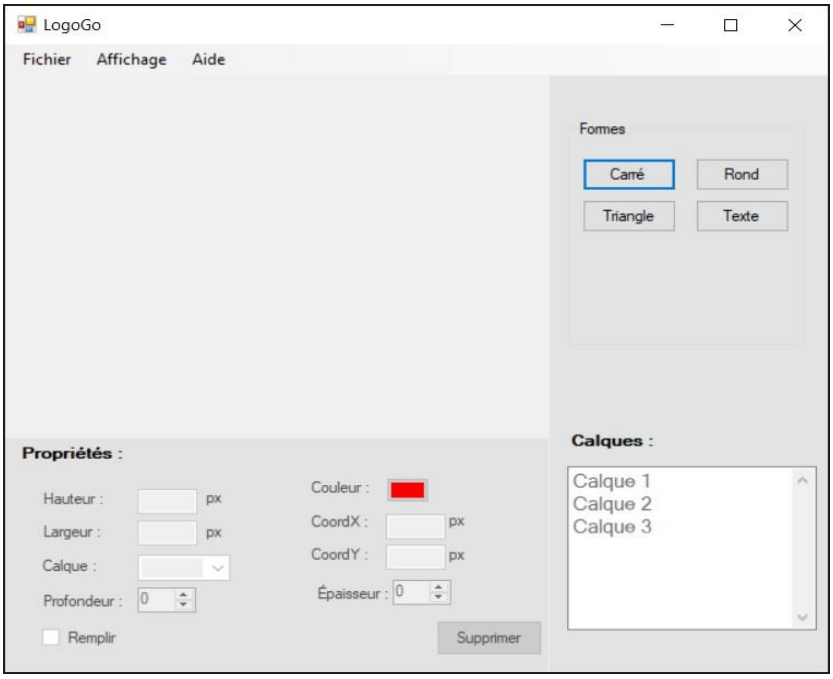
La valeur de cet objet NumericUpDown s'affichant dynamiquement, on voit la valeur correspondant au calque actuellement sélectionné. Voici ce que le NumericUpDown affiche lorsque l'on clique sur Calque 1 :

Le fonctionnement exact de la transparence d'un plan est expliqué en détails dans le Chapitre **8.6. Plans graphiques**.

6.2. Interfaces

6.2.1. Fiche principale

La fiche principale est la première fiche qu'on voit en arrivant sur l'application.



C'est à partir de cette fenêtre qu'on a accès à toutes les fonctionnalités de LogoGo. Elle contient trois parties et chaque partie est documentée ci-dessous :

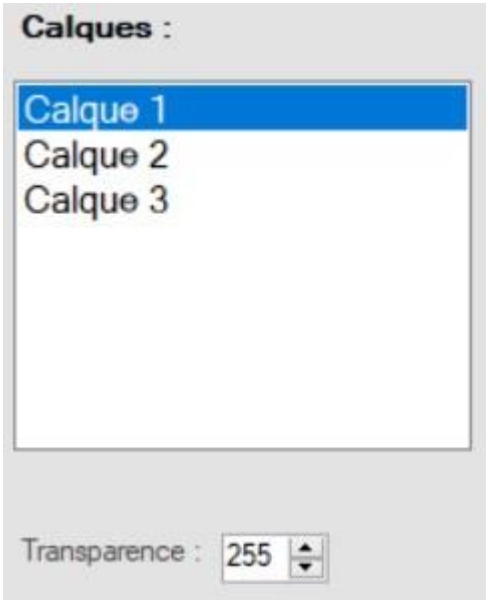
1. Formes

Dans cette partie de la fenêtre, il y a le choix entre plusieurs types de formes. Chaque bouton correspond à une forme différente qui, à l'appui d'un bouton, apparaîtra sur le calque actif. Le fonctionnement est identique pour chaque forme, sauf une, le Polygone.

Lorsque l'utilisateur souhaite créer un polygone quelconque, une fenêtre (voir **6.2.3 Fiche de création de Polygone**) s'ouvre et permet à l'utilisateur de placer des points dans une PictureBox. Le Polygone s'affiche sur la fiche principale uniquement lorsque l'utilisateur finit de placer les points.

2. Calques

Chaque calque est présent dans une ListBox (élément WindowsForm). Lorsqu'on sélectionne un calque, celui-ci devient actif et, pour cela, il faut simplement mémoriser le numéro du calque sélectionné. Le calque actif est celui sur lequel les formes apparaissent.



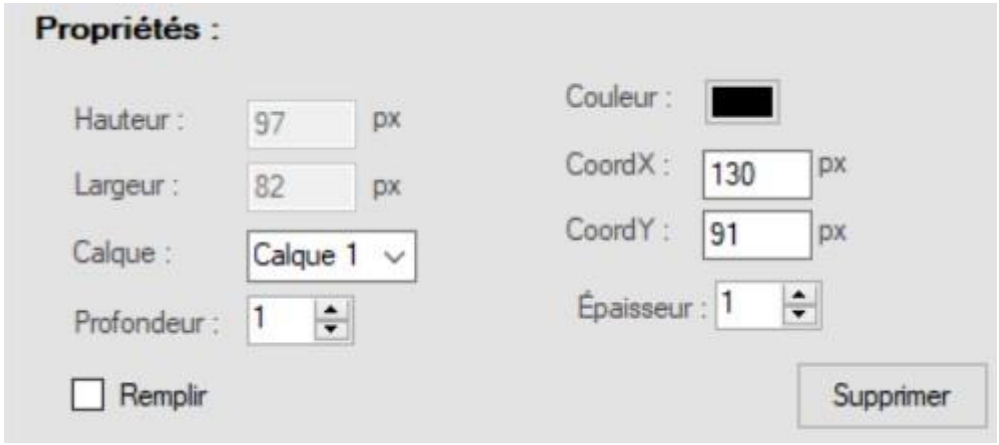
Comme vous pouvez le deviner, c'est également ici qu'on modifie la transparence d'un calque. Il faut le sélectionner puis changer la valeur "Transparence".

3. Propriétés

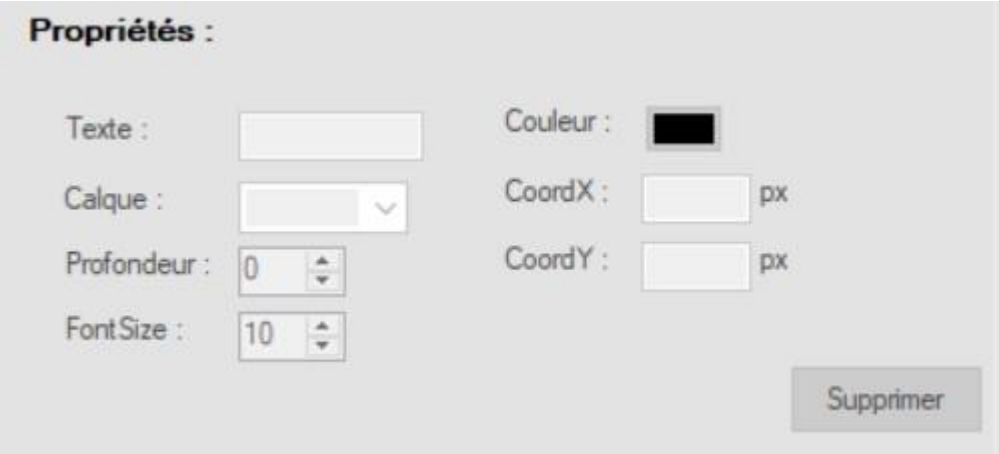
Ce sont les caractéristiques d'une forme (Sprite). Elles sont modifiables via les champs disponibles pour chacune d'elles.

Les propriétés ci-dessus sont celles d'une forme standard (Rond, Carré, Triangle). Si la forme actuellement sélectionnée est un Polygone dessiné par l'utilisateur, les propriétés "Hauteur" et "Largeur" ne sont pas modifiables car cela déformerait la forme que l'utilisateur à créer. La hauteur et la largeur de la PictureBox sont calculées automatiquement dans le code.

Voici à quoi ressemble les propriétés d'un polygone créé par un utilisateur :



Pour le texte aussi, la taille n'est pas modifiable car elle dépend complétement de la longueur du texte ainsi que de la taille de la police utilisée.
Voilà à quoi ressemble le panel des propriétés d'un objet Texte :



6.2.2. Barre de menu

La barre de menu est accessible à tout moment sur le haut de la fiche principale et comporte deux options :

- Fichier Aide
-

Voyons à quoi correspondent chacune de ces options.

Fichier

Voilà ce que l'on retrouve sous le menu Fichier :

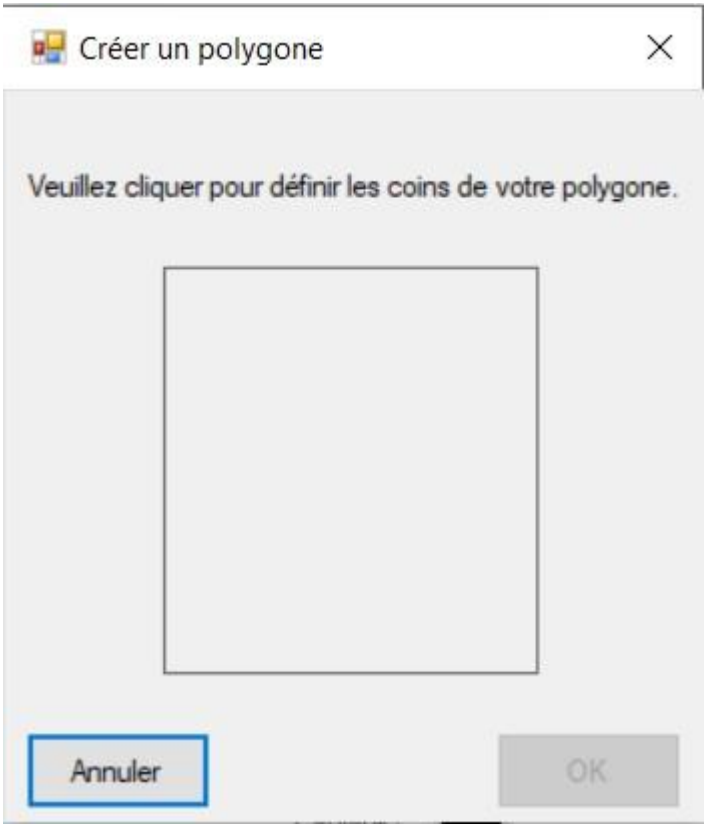
Ce sont trois fonctionnalités essentielles à l'application : l'enregistrement et l'ouverture ainsi que l'exportation en image. C'est là que nous avons accès à ces fonctionnalités. Un clic sur chacune de ces options déclenche la fonctionnalité respective mais, avant toute chose, les trois déclenchent l'ouverture d'une boîte de dialogue.

Enregistrer et Exporter ouvrent une boîte de dialogue de type SaveFileDialog puisque ces deux fonctionnalités correspondent à un enregistrement de fichier. Pour "Ouvrir", par contre, c'est un OpenFileDialog qui est ouvert, vu que l'utilisateur doit chercher un fichier à ouvrir. **Aide**

Un appui sur le bouton "Aide" ouvre une page internet avec toute la documentation du projet. L'utilisateur est redirigé sur le lien GitHub du projet pour qu'il puisse regarder la documentation qui lui est destinée.

6.2.3. Fiche de création de Polygone

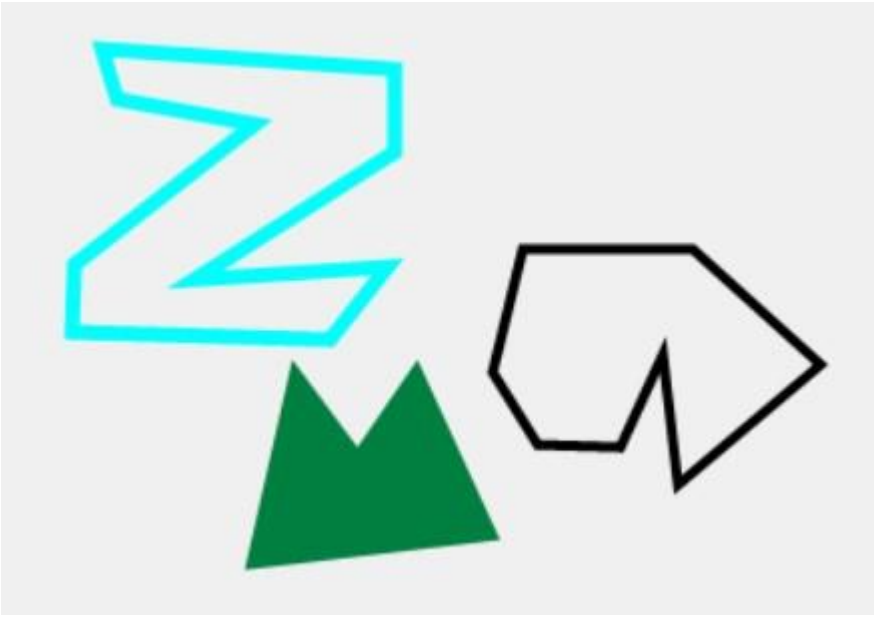
Cette fenêtre s'affiche lorsque l'utilisateur clique sur le bouton "Polygone" de la fiche principale. Elle permet de placer les points d'un polygone comme on le souhaite et, ainsi, créer une forme particulière.



C'est uniquement à l'intérieur de la PictureBox (délimitée par un rectangle) que l'utilisateur peut placer ses points. La forme n'est créée que si l'utilisateur le décide en validant sa saisie.

Cette forme possède deux boutons : "Annuler" et "OK".
Ces deux boutons ont leur DialogResult configuré sur "Cancel" et "OK", respectivement. À l'appui du bouton "OK", les points placés par l'utilisateur sont validés et la forme apparaît sur la fiche principale. Le bouton "Annuler", lui, annule l'action, ferme la fenêtre et la forme ne se crée pas.

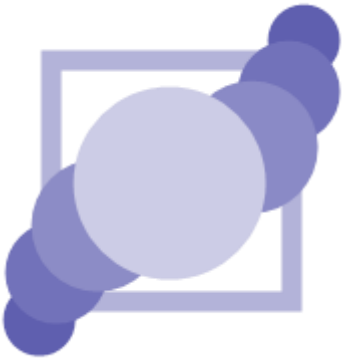
Voici quelques exemples de ce que l'on peut créer :



6.3. Le logo

Le logo de l'application a été créé sur LogoGo. Cela m'a permis de vérifier la facilité ou les difficultés que l'on pouvait rencontrer durant la création d'un logo. J'ai utilisé les trois claques pour créer ce logo.

Il n'est composé que de ronds (7) et de carrés (1). J'ai modifié la bordure du carré et je l'ai placé sur le Calque 1. Ensuite, l'ai mis un rond au centre du carré mais, cette fois, sur le Calque 3. Le calque 2 m'a permis de placer après coup les 6 petits ronds entre celui qui est sur le premier plan et le carré à l'arrière.



Voilà à quoi ressemble de fichier .xml de ce logo :

```
<?xml version="1.0"?>
<Logo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <SpritesSerializables>
  <ListeDeSpriteSerializable>
   <SpriteSerializable>
    <Couleur>-5000231</Couleur>
    <Location>
     <X>157</X>
     <Y>114</Y>
    </Location>
    <EpaisseurPen>10</EpaisseurPen>
    <Remplir>false</Remplir>
    <NumeroCalque>1</NumeroCalque>
    <ProfondeurParCalque>1</ProfondeurParCalque>
    <Nom>Carré_1</Nom>
    <IdType>1</IdType>
    <TaillePolice>0</TaillePolice>
    <Size>
     <Width>120</Width>
     <Height>120</Height>
    </Size>
   </SpriteSerializable>
   <SpriteSerializable>
    <Couleur>-10526801</Couleur>
    <Location>
     <X>265</X>
     <Y>86</Y>
    </Location>
    <EpaisseurPen>1</EpaisseurPen>
    <Remplir>true</Remplir>
    <NumeroCalque>2</NumeroCalque>
    <ProfondeurParCalque>1</ProfondeurParCalque>
    <Nom>Rond_2</Nom>
    <IdType>2</IdType>
    <TaillePolice>0</TaillePolice>
    <Size>
     <Width>35</Width>
     <Height>35</Height>
    </Size>
```

```
    </SpriteSerializable>
    <SpriteSerializable>
      <Couleur>-10526801</Couleur>
      <Location>
        <X>133</X>
        <Y>225</Y>
      </Location>
      <EpaisseurPen>1</EpaisseurPen>
      <Remplir>true</Remplir>
      <NumeroCalque>2</NumeroCalque>
      <ProfondeurParCalque>1</ProfondeurParCalque>
      <Nom>Rond_3</Nom>
      <IdType>2</IdType>
      <TaillePolice>0</TaillePolice>
      <Size>
        <Width>35</Width>
        <Height>35</Height>
      </Size>
    </SpriteSerializable>
    <SpriteSerializable>
      <Couleur>-9276742</Couleur>
      <Location>
        <X>134</X>
        <Y>194</Y>
      </Location>
      <EpaisseurPen>1</EpaisseurPen>
      <Remplir>true</Remplir>
      <NumeroCalque>2</NumeroCalque>
      <ProfondeurParCalque>1</ProfondeurParCalque>
      <Nom>Rond_4</Nom>
      <IdType>2</IdType>
      <TaillePolice>0</TaillePolice>
      <Size>
        <Width>50</Width>
        <Height>50</Height>
      </Size>
    </SpriteSerializable>
    <SpriteSerializable>
      <Couleur>-9276742</Couleur>
      <Location>
```

```
      <X>250</X>
      <Y>104</Y>
    </Location>
    <EpaisseurPen>1</EpaisseurPen>
    <Remplir>true</Remplir>
    <NumeroCalque>2</NumeroCalque>
    <ProfondeurParCalque>1</ProfondeurParCalque>
    <Nom>Rond_5</Nom>
    <IdType>2</IdType>
    <TaillePolice>0</TaillePolice>
    <Size>
      <Width>50</Width>
      <Height>50</Height>
    </Size>
  </SpriteSerializable>
  <SpriteSerializable>
    <Couleur>-7566138</Couleur>
    <Location>
      <X>147</X>
      <Y>163</Y>
    </Location>
    <EpaisseurPen>1</EpaisseurPen>
    <Remplir>true</Remplir>
    <NumeroCalque>2</NumeroCalque>
    <ProfondeurParCalque>1</ProfondeurParCalque>
    <Nom>Rond_6</Nom>
    <IdType>2</IdType>
    <TaillePolice>0</TaillePolice>
    <Size>
      <Width>65</Width>
      <Height>65</Height>
    </Size>
  </SpriteSerializable>
  <SpriteSerializable>
    <Couleur>-7697722</Couleur>
    <Location>
      <X>224</X>
      <Y>124</Y>
    </Location>
    <EpaisseurPen>1</EpaisseurPen>
```

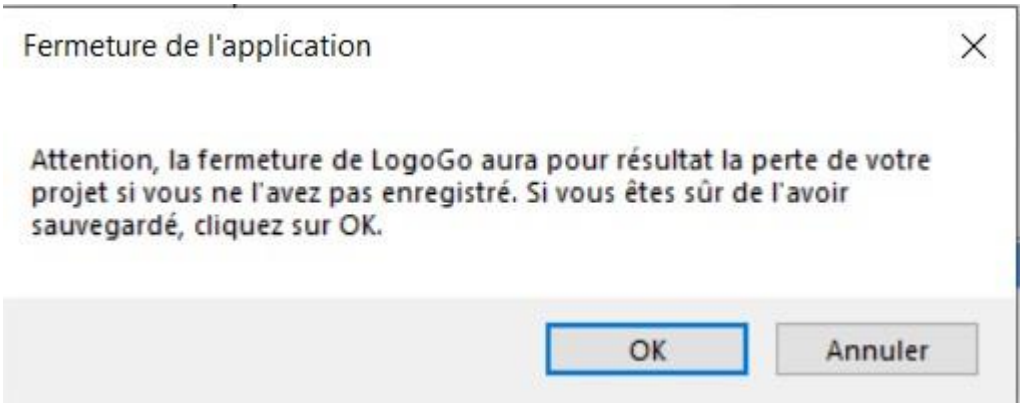
```
      <Remplir>true</Remplir>
      <NumeroCalque>2</NumeroCalque>
      <ProfondeurParCalque>1</ProfondeurParCalque>
      <Nom>Rond_7</Nom>
      <IdType>2</IdType>
      <TaillePolice>0</TaillePolice>
      <Size>
       <Width>65</Width>
       <Height>65</Height>
      </Size>
     </SpriteSerializable>
     <SpriteSerializable>
      <Couleur>-3355418</Couleur>
      <Location>
       <X>168</X>
       <Y>127</Y>
      </Location>
      <EpaisseurPen>1</EpaisseurPen>
      <Remplir>true</Remplir>
      <NumeroCalque>3</NumeroCalque>
      <ProfondeurParCalque>1</ProfondeurParCalque>
      <Nom>Rond_1</Nom>
      <IdType>2</IdType>
      <TaillePolice>0</TaillePolice>
      <Size>
       <Width>95</Width>
       <Height>95</Height>
      </Size>
     </SpriteSerializable>
    </ListeDeSpriteSerializable>
  </SpritesSerializables>
</Logo>
```

6.4. Les messages

Pendant l'utilisation de l'application LogoGo, il est possible de rencontrer certains messages...
Voyons dans quelles circonstances ces messages peuvent apparaître et à quoi ils ressemblent.

6.4.1. Fermeture de l'application

Avez-vous bien enregistré votre logo avant de quitter l'application ?
Lorsque vous tentez de quitter LogoGo, une fenêtre vous avertit que, si vous n'avez pas sauvegardé votre logo, vous perdrez toute progression dans la création de votre image. Voici à quoi ressemble cette fenêtre d'avertissement :



Ce message est là pour vous aider. Il vous suffit de cliquer sur OK si vous avez bien enregistré votre projet ou si vous ne souhaitez pas sauvegarder les dernières modifications. Si vous voulez retourner sur LogoGo pour continuer ou pour sauvegarder votre logo, appuyez sur Annuler, cela annulera la fermeture de LogoGo.

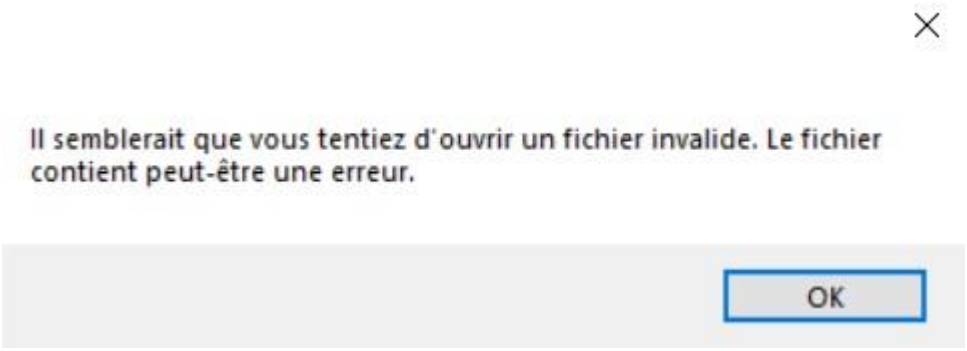
6.4.2. Enregistrement

Comme vous venez de le voir, si vous tentez de quitter l'application, un message vous avertira que vous devez enregistrer votre projet à part si votre projet est vide. À l'inverse, si vous tentez d'enregistrer un projet qui ne contient aucune forme, cette fenêtre s'ouvrira :

Vous devez ajouter au moins une forme pour pouvoir enregistrer le logo.

6.4.3. Ouverture

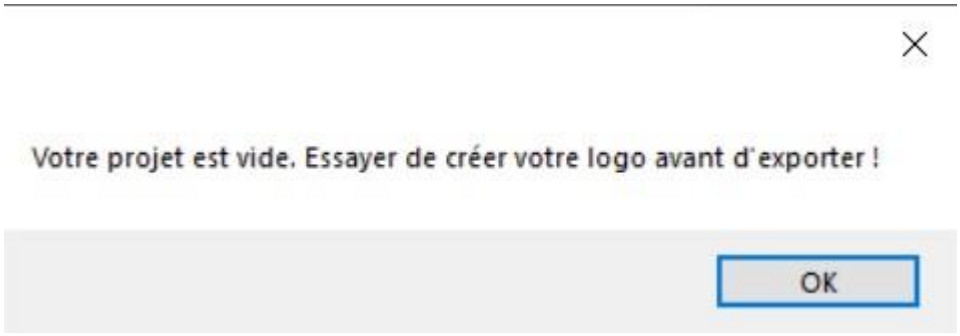
Si le fichier ouvert est invalide, ce message apparaîtra :



Un fichier est invalide lorsqu'il est modifié de manière incorrecte ou s'il est corrompu.

6.4.4. Exportation

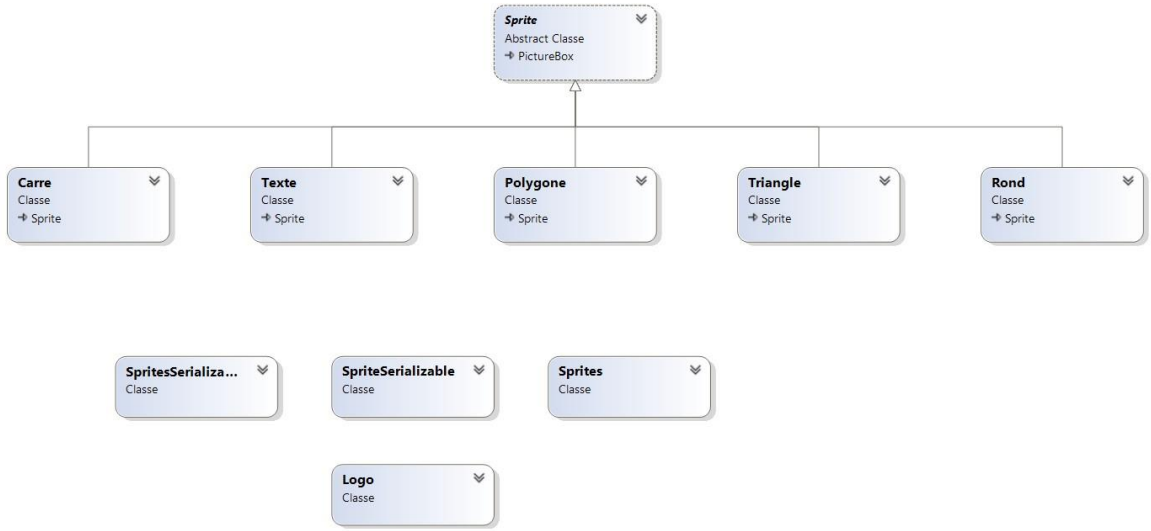
Il est possible de vous retrouver face à ce message si vous tentez d'exporter un projet vide. Vous ne pouvez pas exporter en image un projet qui ne contient rien du tout.



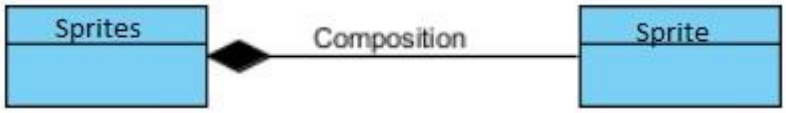
Vous devez ajouter au moins une forme pour pouvoir exporter le logo.

7. Analyse organique

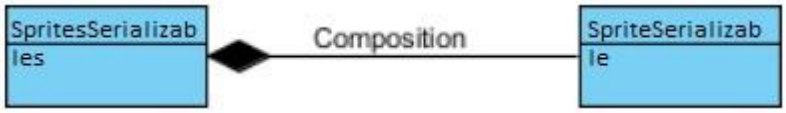
7.1. Classes



Vous pouvez voir, sur l'image ci-dessus, mon diagramme de classe. On peut y observer la structure de mes classes. Pour une version plus détaillée, voir **8.2. Diagramme de classe** On voit donc que la classe Sprite hérite de PictureBox et que toutes les formes sont des classes filles héritant de Sprite. Cependant, il est bien de corriger une erreur figurant sur cette image : le lien entre Logo et les autres classes. En effet, les relations entre les différentes classes doivent être précisées.



La classe Sprites contenant une liste d'objets de type Sprite, il serait juste de noter une composition entre les deux classes. Il en va de même pour les classes SpritesSerializables et SpriteSerializable :



Concernant la classe Logo, c'est elle qui contient les instances des classes Sprites et SpritesSerializables. Il y a donc une composition entre la classe Logo et ces deux dernières.

7.2. Description des principales méthodes

Ce chapitre regroupe les explications des méthodes les plus importantes de chaque classe du projet. Certains points sont mieux expliqués dans **8. Réalisation**.

7.2.1. Classe Logo

La classe Logo est comme une sorte de container. Un objet Logo contient tous les sprites créés et peut accéder à chacun d'eux.

7.2.1.1. Actions sur les sprites

La classe logo contient trois petites méthodes :

- AjouterSprite
- SupprimerSprite
- TrierSprites

AjouterSprite fait appel à la méthode "Ajouter" de la classe Sprites. **SupprimerSprite** fait, comme la précédente, appel à la méthode "Supprimer" de la classe Sprites. Pareil pour **TrierSprites**, elle appelle la méthode "Trier" de la même classe que les autres.

7.2.1.2. XMLSerialize / XMLDeserialize

Ces deux méthodes travaillent de pair pour les fonctionnalités de sauvegarde et de chargement. Un utilisateur doit être en mesure de sauvegarder un fichier mais aussi de le rouvrir. Pour la sauvegarde, c'est XMLSerialize qui s'en occupe. Avant de sérialiser, une conversion de la liste "Sprites" est nécessaire (voir **8.1 Sauvegarde et chargement**). Le code concernant directement la sérialisation est tirée d'une méthode vue en classe avec M. Bonvin dans le cadre d'un atelier C#.

Au moment du chargement de fichier, c'est XMLDeserialize qui est concernée. Tout comme pour XMLSerialize, une conversion est nécessaire mais, cette fois-ci, c'est après la désérialisation que l'on doit faire cette conversion. Il faut passer d'une liste de type SpriteSerializable à une liste de type Sprite. La désérialisation a également été abordée avec M. Bonvin lors des ateliers.

7.2.2. Classe Sprite

La classe Sprite est essentielle à ce type d'application car c'est une classe mère. Toutes les formes de l'application sont des classes filles qui héritent de Sprite. Les classes filles en question sont :

- Rond
- Carre
- Triangle
- Polygone Texte
-

Cette classe contient plusieurs méthodes...

7.2.2.1. SpritePaintAvecGraphics

Cette méthode est abstraite car elle doit être différente pour chaque type de sprite (un carré n'est pas dessiné de la même façon qu'un triangle). Chaque classe fille de Sprite a donc une surcharge (override) de cette méthode, ce qui permet de coder quelque chose de différent pour chaque cas.

- Dans la classe **Carre** :

  Dessine le carré avec FillRectangle ou DrawRectangle selon la valeur de la propriété Remplir.

- Dans la classe **Rond** :

  Dessine le rond avec FillEllipse ou DrawEllipse selon la valeur de la propriété Remplir. • Dans les

classes **Polygone** et **Triangle** :

  Dessine le triangle/polygone avec FillPolygone ou DrawPolygone selon la valeur de la propriété Remplir.

- Dans la classe **Texte** :

  Dessine le texte avec DrawString avec une certaine police d'écriture.

Cette méthode accepte un objet de type Graphics en paramètre car cela facilite certaines choses. Par exemple, lors de l'exportation, je dois pouvoir dessiner les formes avec un Graphics créé à partir d'une image (Graphics.FromImage(...)). C'est de cette façon qu'il est possible de récupérer l'image de la PictureBox contenant tous les sprites. Dans le Paint de cette PictureBox, j'appelle tout simplement la méthode SpritePaintAvecGraphics et, en paramètre, je passe e.Graphics.

7.2.2.2. EnSpriteSerializable

Cette méthode, bien qu'importante, est très simple. Elle ne fait que créer un nouvel objet SpriteSerializable puis appeler la méthode "AttribuerValeursProprietes" de la classe SpriteSerializable avant de retourner l'objet créé.

7.2.2.3. Le déplacement de sprite

Une des fonctionnalités qui rend l'application agréable à l'utilisateur est celle qui permet de déplacer une forme avec la souris (comme un drag n drop). Cette fonctionnalité se compose de trois événements :

SpriteMouseDown sert surtout à s'assurer que l'utilisateur a bien appuyé sur la forme avec le clic gauche de la souris. C'est au moment de l'appui sur une forme que l'événement est appelé. **SpriteMouseMove** se charge de déplacer la forme en fonction de la position de la souris pour que la forme se déplace comme le souhaite l'utilisateur. Cet événement est déclenché lorsque l'utilisateur déplace la souris. L'événement nommé **SpriteMouseUp** sert à marquer la fin du déplacement, étant donné qu'il est déclenché lorsque l'utilisateur lâche le bouton de sa souris.

7.2.3. Classe Sprites

7.2.3.1. Action sur les sprites

Comme la Classe Sprite, Sprites contient trois méthodes simples qui agissent sur la liste de type Sprite. La méthode **Ajouter** ne fait qu'ajouter le sprite spécifié à la liste et, respectivement, **Supprimer** supprime l'objet Sprite spécifié. Finalement, la méthode **Trier** trie la liste selon deux propriétés : NumeroCalque et Profondeur. Pour plus d'informations sur le tri de la liste, voir **8.6. Plans Graphiques**.

7.2.3.2. EnListeSerializable

La méthode EnListeSerializable est complémentaire à la méthode EnSpriteSerializable de la classe Sprite. Par le biais d'une boucle foreach, elle parcourt toutes les formes présentes dans la liste de sprites. Pour chaque objet Sprite, elle créé un objet de type SpriteSerializable et l'ajoute à une liste. La méthode retourne la liste de type SpriteSerializable qui a été créée.

7.2.4. Classe SpriteSerializable

7.2.4.1. EnSprite

Méthode simple qui contient un Switch case permettant de savoir quel type de forme créer selon la propriété IdType. Chaque forme ayant un identifiant (Carré = 1, Rond = 2, ...), il est possible de créer le bon type de Sprite à chaque fois. La méthode retourne toujours un objet. Si l'identifiant est inconnu, la méthode retourne un carré par défaut. Cela arrive grâce à la possibilité de créer un cas par défaut dans un Switch case. Voilà un code d'exemple de cas par défaut :

```
default:
    return new Carre(this, parent);
```

7.2.4.2. AttribuerValeursProprietes

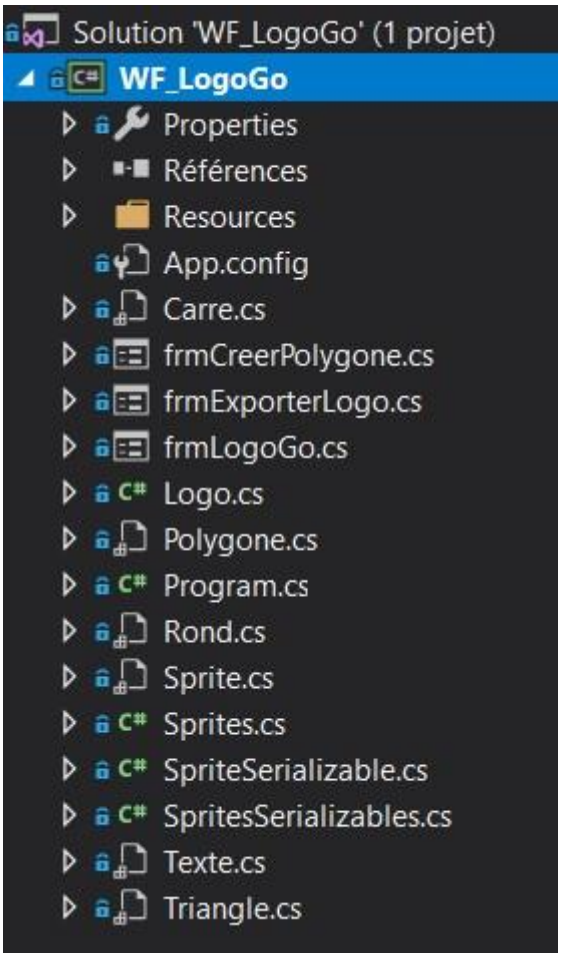
Cette méthode est essentielle car, pour sérialiser un Sprite, il est indispensable de mémoriser ses caractéristiques si on veut pouvoir les retrouver plus tard. La méthode AttribuerValeursProprietes prends un Sprite un paramètre et l'utilise pour attribuer les valeurs contenues dans ce dernier à l'objet SpriteSerializable. À la fin de cette opération, un sprite aura été, en quelque sorte, converti en SpriteSerializable.

7.2.5. Classe SpritesSerializables

7.2.5.1. EnSprites

Si la classe Sprites a une méthode EnListeSerializable, la classe SpritesSerializables a une méthode qui inverse l'opération. La méthode EnSprites est utile au moment de la désérialisation. C'est elle qui va utiliser la méthode EnSprite de la classe SpriteSerializable pour chaque élément. Elle parcourt la liste d'objets SpriteSerializable et en créé une liste d'objets Sprite. La liste est retournée et peut ensuite être utilisée.

7.3. Arborescence des fichiers



8. Réalisation

8.1. Sauvegarde et chargement

La sérialisation/désérialisation est une chose que j'avais bien exercé avant le TPI. Nous avions, en classe, expérimenté la sérialisation d'objets simples ainsi que de listes d'objets, ce qui était le cas pour mon projet. La subtilité de ce travail a été de trouver la bonne manière de sérialiser puisqu'un objet Sprite n'est pas sérialisable (il hérite de PictureBox et il est impossible de sérialiser un objet WindowsForm). Lors de la première réflexion, après avoir lu l'énoncé, c'est un problème que j'ai envisagé. J'avais prévu de faire une deuxième classe Sprite (SpriteSerializable) qui, elle, n'hériterait pas de PictureBox. Ceci servirait à mémoriser les propriétés d'une forme.

Il fallait donc trouver un moyen de convertir ces objets dans les deux sens : passer d'un Sprite à un SpriteSerializable au moment de la sérialisation, puis l'inverse lors de la désérialisation. Pour ce faire, j'ai créé une méthode (dans la classe SpriteSerializable) pouvant prendre un Sprite en paramètre. Cette méthode attribue la valeur des propriétés du Sprite passé en paramètre aux propriétés d'un SpriteSerializable. Ensuite, la méthode EnListeSerializable (dans la classe Sprites) permet de convertir toute la liste de formes en une liste de formes sérialisables. De cette manière, ce n'est pas la liste d'objets Sprite que l'on essaie de sérialiser, mais celle contenant les objets de type SpriteSerializable.

C'est en suivant la même logique que j'ai réussi la désérialisation. La classe SpriteSerializable contient une méthode nommée EnSprite et la classe SpritesSerializables contient une méthode EnSprites. Pour savoir quel type de sprite il faut recréer, j'ai fait un Switch case en utilisant le IdType.

Pour les méthodes de sérialisation et désérialisation, j'ai utilisé les fonctions que nous avons vu avec M. Bonvin en classe. Voici la méthode (incomplète) permettant la sérialisation :

```
private void XMLSerialize()
{
   Stream stream = File.Open(NomFichier, FileMode.Create);
   XmlSerializer formatter = new XmlSerializer(typeof(Logo));
   formatter.Serialize(stream, this);
   stream.Close();
}
```

Voici la méthode (incomplète) pour la désérialisation :

```
public void XMLDeserialize()
{
   Stream stream = File.Open(NomFichier, FileMode.Open);
   XmlSerializer formatter = new XmlSerializer(typeof(Logo));
   Logo obj = (Logo)formatter.Deserialize(stream);
   stream.Close();
}
```

8.1.1. Pourquoi le XML ?

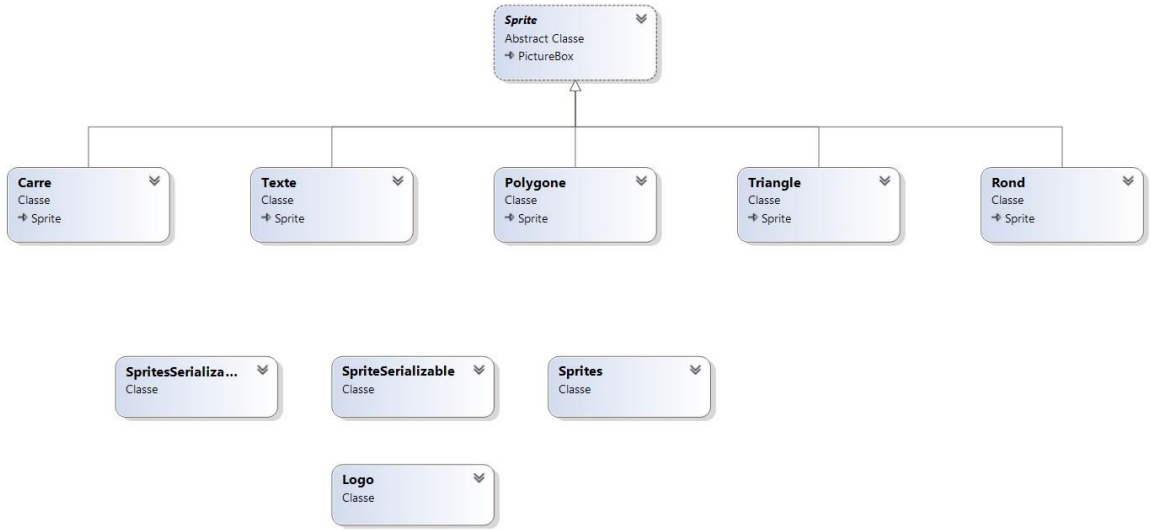
Il est possible de sérialiser en plusieurs formats, alors pourquoi le XML ? Tout simplement car c'est le format le plus adapté à LogoGo. La syntaxe de XML étant composée de balises, il est facile de comprendre l'architecture d'un fichier de ce type. Voici à quoi le XML ressemble :

```
<?xml version="1.0"?>
<Logo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SpritesSerializables>
    <ListeDeSpriteSerializable>
      <SpriteSerializable>
        <Couleur>-16777216</Couleur>
        <Location>
          <X>105</X>
          <Y>70</Y>
        </Location>
        <EpaisseurPen>1</EpaisseurPen>
        <Remplir>true</Remplir>
        <NumeroCalque>1</NumeroCalque>
        <ProfondeurParCalque>1</ProfondeurParCalque>
        <Nom>Carré_1</Nom>
        <IdType>1</IdType>
        <Size>
          <Width>100</Width>
          <Height>100</Height>
        </Size>
      </SpriteSerializable>
      <SpriteSerializable>
        <Couleur>-16711872</Couleur>
        <Location>
          <X>153</X>
          <Y>119</Y>
        </Location>
        <EpaisseurPen>5</EpaisseurPen>
        <Remplir>false</Remplir>
        <NumeroCalque>1</NumeroCalque>
        <ProfondeurParCalque>1</ProfondeurParCalque>
        <Nom>Rond_1</Nom>
        <IdType>2</IdType>
        <Size>
          <Width>100</Width>
          <Height>100</Height>
        </Size>
      </SpriteSerializable>
    </ListeDeSpriteSerializable>
  </SpritesSerializables>
</Logo>
```

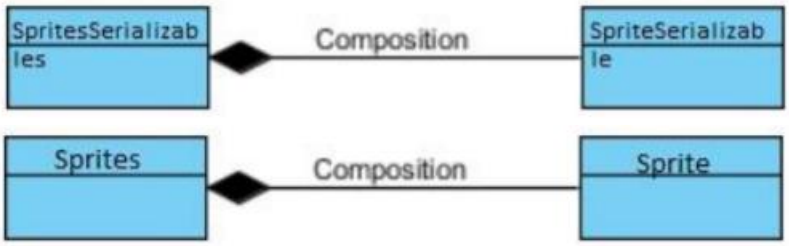
L'avantage des fichiers .xml (par rapport aux fichiers .bin, par exemple), c'est qu'on a la possibilité de l'éditer aisément. Je peux sans soucis ouvrir le fichier de mon logo et rajouter/supprimer/modifier une forme. C'est une valeur ajoutée comparée à plusieurs formats.

8.2. Diagramme de classe

Comme vous avez pu le voir plus tôt, voilà à quoi ressemble le diagramme de classe de l'application :

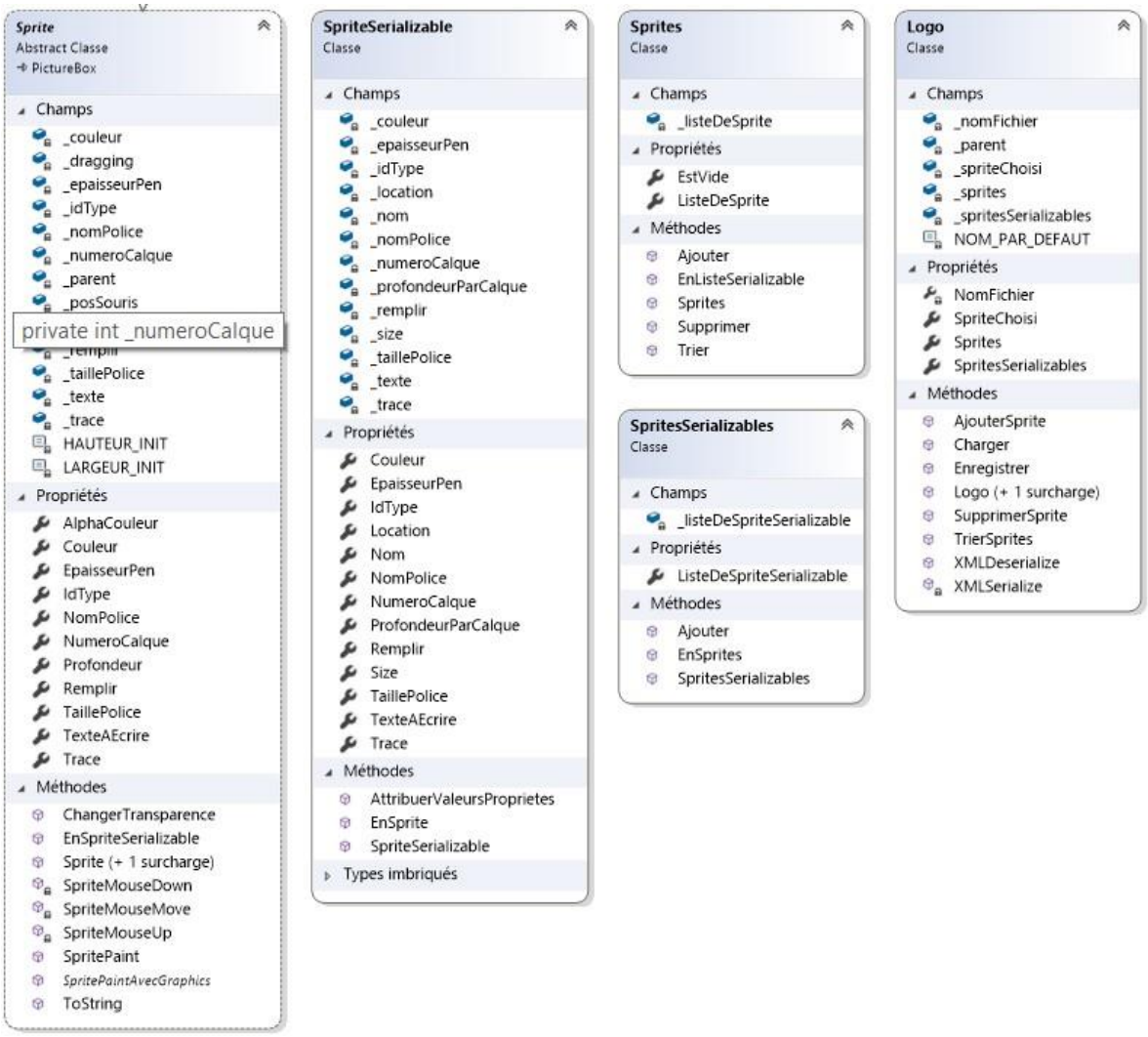


Et, comme corrigé dans le chapitre **Analyse organique**, voilà les liaisons appropriées entre les paires de classes Sprite/Sprites et SpriteSerializable/SpritesSerializables :



Ces classes sont reliées par une composition car les classes au pluriel (Sprites et SpritesSerializables) contienne des listes d'objets Sprite et SpriteSerializable. Si un objet Sprites est supprimé, les objets Sprite seront également supprimés et il en va de même pour les classes d'objets sérialisables. De plus, la classe Logo contient un objet Sprites ainsi qu'un objet SpritesSerializables. On peut dont également dire qu'il y a une composition entre ces deux classes et Logo.

Voici à quoi ressemble chaque classe en détails :



8.3. Classe mère commune (héritage)

La création d'une classe mère commune à plusieurs autres classes est, même si elle paraît évidente pour ce projet, nécessaire pour la réalisation de ce projet.

En effet, l'utilisateur doit pouvoir créer plusieurs types d'objet qui ont tous les propriétés communes. N'importe lequel de mes objets ont besoin, par exemple, d'une hauteur, d'une largeur, d'une position (X, Y), d'une couleur, etc.
J'ai donc procédé de la façon suivante : une classe Sprite sert de classe mère et est une classe abstraite. Toutes les autres formes (Carre.cs, Triangle.cs, Texte.cs, etc.) héritent de cette classe et donc, de ses propriétés. Si une nouvelle classe doit être créée (car nous voulons ajouter un nouveau type de forme), il suffit de créer une nouvelle classe qui hérite de Sprite.

Il suffit de surcharger les méthodes de la classe Sprite si les classes filles en ont besoin. C'est notamment le cas pour la méthode SpritePaintAvecGraphics qui doit dessiner quelque chose de différent pour chaque type de forme (FillEllipse pour un rond, FillRectangle pour un carré, etc).

Il est bon de noter que la classe mère (Sprite.cs) hérite elle-même de la classe PictureBox car, dans le cadre de ce projet, plusieurs aspects de la Picturebox sont nécessaires. De ce fait, je n'ai pas eu besoin de créer de

propriétés pour la taille ou pour la position des sprites puisqu'ils utilisent les propriétés "Location" et "Size" de PictureBox.

8.4. Formes disponibles

L'énoncé dit que l'utilisateur doit pouvoir créer un logo avec "au moins trois formes et du texte". Dans la dernière version de LogoGo, il y a la possibilité d'ajouter quatre types de formes ainsi que du texte.

On peut ajouter des carrés, des ronds, des triangles, des polygones quelconques, et du texte. Les carrés et les ronds sont les moins complexes puisque l'objet Graphics contient déjà des méthodes pour ces deux formes (DrawEllipse/DrawRectangle et FillEllipse/FillRectangle). Pour ce qui est du texte, la méthode DrawString est idéale.
Pour les triangles et autres types de polygone, par contre, il a fallu utiliser les méthodes DrawPolygon et FillPolygon qui nous donnent la possibilité de dessiner des formes plus complexes en donnant des points à la méthode. Pour faire simple, ces méthodes relient les points qu'on leur donne en paramètres. Il m'a donc fallu, pour ce qui est du triangle, calculer la position des sommets d'un triangles à partir de sa taille puisque la position des sommets dépend directement de la taille que l'utilisateur choisit.

Pour les polygones plus complexes, c'est l'utilisateur qui place les points. Ces points sont récoltés dans un tableau qui est donné aux méthodes FillPolygon et DrawPolygon.

8.5. Planning réel / Planning prescrit

Le planning prévisionnel m'a été donné dès le début, ce qui m'a facilité la tâche. Cela ma permit de reprendre la structure du planning prescrit et, ainsi, de le remplir en fonction de mon avancement.

Voici le planning que l'on m'a fourni au début du travail :

Jour	1		2		3		4		5		6		7		8		9		10		11	
Demi-Journée	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Etude du sujet. Planification																						
Installation, mise en place																						
Conception interface																						
Classes de base, OA																						
Conception des plans, formes																						
Gestion fichiers / export																						
Finalisation / Corrections																						
Tests																						
Documentation																						
Résumé																						
Finalisation / Impressions																						
Journal																						

En remplissant une copie de ce planning pendant mon travail, voilà ce que j'ai obtenu :

Jour	1		2		3		4		5		6		7		8		9		10		11	
Demi-Journée	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Etude du sujet. Planification																						
Installation, mise en place																						
Conception interface																						
Classes de base, OA																						
Conception des plans, formes																						
Gestion fichiers / export																						
Finalisation / Corrections																						
Tests																						
Documentation																						
Résumé																						
Finalisation / Impressions																						
Journal																						

Il y a beaucoup de différences mais on observe également quelques similitudes. J'ai, par exemple, une certaine continuité au niveau de la documentation, des tests et du journal de bord qui correspond au planning prévu. J'ai par ailleurs écrit dans le journal un jour de plus que ce qui était prévu.

En ce qui concerne les différences, il a des choses qui m'ont pris plus longtemps que prévu et, pour d'autres, moins que prévu. La conception de l'interface, par exemple, m'a pris quelques heures en plus ce que le planning prévisionnel indiquait alors que la mise en place n'a duré que quelques minutes. Le fait d'être chez moi m'a aidé au niveau de la tâche "Installation, mise en place" car je n'ai pas eu besoin de préparer grand-chose, à part une mise à jour de VisualStudio qui fût très rapide.

La réalisation des classes de base m'ont pris nettement moins de temps que les classes filles de Sprite. Les classes telles que Texte.cs ainsi que Polygone.cs m'ont pris plus de temps que les autres classes et plus de temps que ce qui était prévu. C'est clairement les réflexions qui m'ont pris le plus de temps.

La partie concernant la gestion de fichiers (export, enregistrement et chargement) m'a pris un peu plus de temps que sur le planning prévisionnel mais, comme j'ai eu un petit peu d'avant sur le reste, j'ai commencé la gestion de fichier une demie journée plus tôt. On voit d'ailleurs qu'au début, le planning effectif était plutôt différent de celui qui m'a été fourni. Au fil des jours, le planning prévisionnel s'est avéré plus ou moins exact, puisque tout a été terminé plus ou moins au même moment que les prévisions.

Pour résumer, je pense qu'on peut affirmer que le planning prévisionnel était assez réaliste.

8.6. Plans graphiques

La conception des différents plans de travail, ou calques, m'a demandé un moment de réflexion. J'ai commencé par imaginer une classe Calque qui aurait une propriété ListeSprites (List) qui nous aiderait à savoir quels sprites sont contenus dans chaque calques.

Cependant, en partant sur cette idée, je me suis rendu compte que ce ne serait pas pratique. À chaque fois que je voudrais rafraîchir l'ordre d'affichage (lorsqu'un Sprite passe d'un calque à l'autre, par exemple), il aurait alors fallu parcourir la liste de sprites contenue dans chaque calque pour, ainsi, les afficher dans le bon ordre.

En comprenant que cela ne serait pas la manière la plus optimisée, j'ai remarqué que meilleure possibilité que j'avais était simplement de ne pas faire de classe calque.

Voilà comment j'ai procédé : dans la classe Sprite, il existe une propriété nommée "NumeroCalque". Ce numéro de calque sert tout simplement à identifier la priorité d'affichage d'un sprite. Lors de la création d'un sprite, cette propriété (NumeroCalque) prend la valeur du numéro de calque actuellement sélectionné par l'utilisateur. Au moment de l'affichage, je trie simplement la liste de tous les objets Sprite (liste qui est contenue dans la classe Sprites) par la propriété "NumeroCalque". Le code parcourt ensuite la liste triée et affiche les formes une par une.

Dans l'application, il est également possible de modifier la propriété "Profondeur" d'une forme. Cela correspond à un niveau de profondeur dans un même calque et, ainsi, un carré ayant une profondeur plus élevée qu'un autre carré appartenant au même calque apparaîtra au-dessus de ce dernier. Pour ce faire, j'ai fait en sorte que la liste d'objet de type Sprite soit triée par, non pas une, mais deux propriétés : d'abord par le numéro de calque, puis par la profondeur.

```
ListeDeSprite = ListeDeSprite.OrderBy(s => s.NumeroCalque).ThenBy(s =>
s.Profondeur).ToList<Sprite>();
```

De cette façon, tous les sprites sont toujours dessinés dans le bon ordre.

Dans l'énoncé, il était également question de transparence des plans. Pour mener à bien cette tâche, j'ai changé, pour être précis, la transparence de tous les sprites qui sont sur le calque concerné. En fait, un calque n'a pas vraiment de transparence, ce sont les formes sur ce calque qui doivent se dessiner d'une couleur plus ou moins transparente. La transparence d'une couleur est déterminée par la valeur de son alpha. Dans l'acronyme RGBA, le A correspond à la transparence (alpha) de la couleur. Il suffit donc de dessiner les sprites en utilisant une couleur avec la transparence du calque choisie par l'utilisateur.

8.7. Exportation en image

L'exportation d'image est, globalement, assez simple. Étant donné que nous pouvons facilement récupérer l'image d'une PictureBox, il est possible sauvegarder cette image de la méthode suivante :

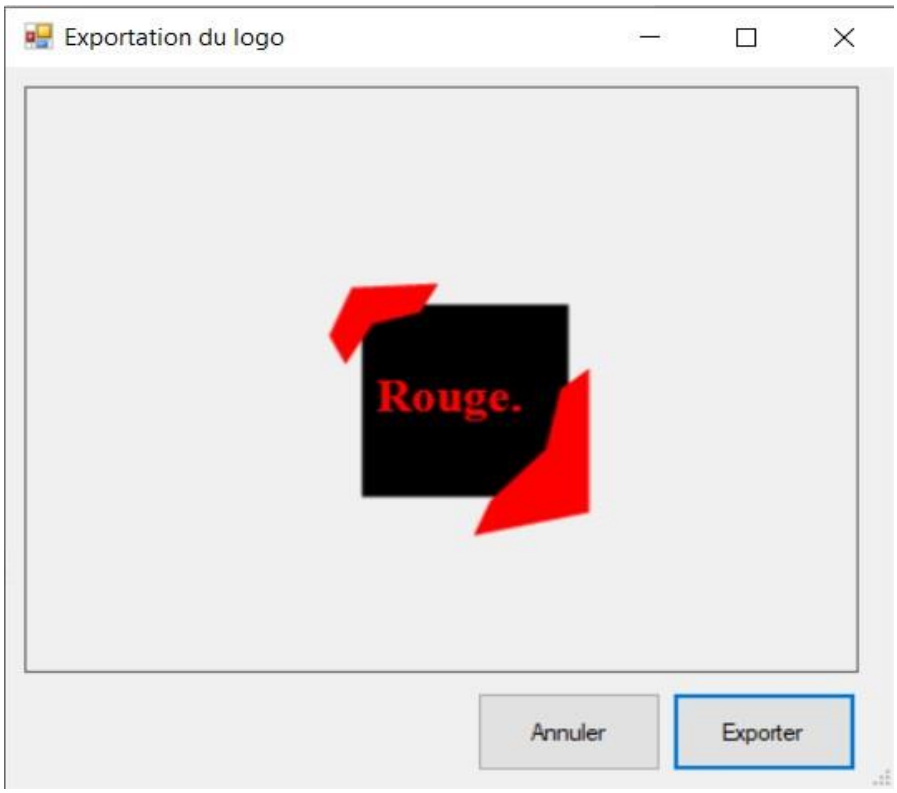
```
maPictureBox.Image.Save(@"C:\Images\monImage",ImageFormat.Jpeg);
```

Cependant, le projet LogoGo a une subtilité à ce niveau : nous n'avons pas qu'une Picturebox, mais plusieurs. Tous les objets PictureBox (tous les sprites) ne doivent former qu'une seule image. Ce point m'a donc demandé une certaine réflexion mais, après quelques heures, j'ai réussi à faire en sorte que cette fonctionnalité fonctionne correctement.

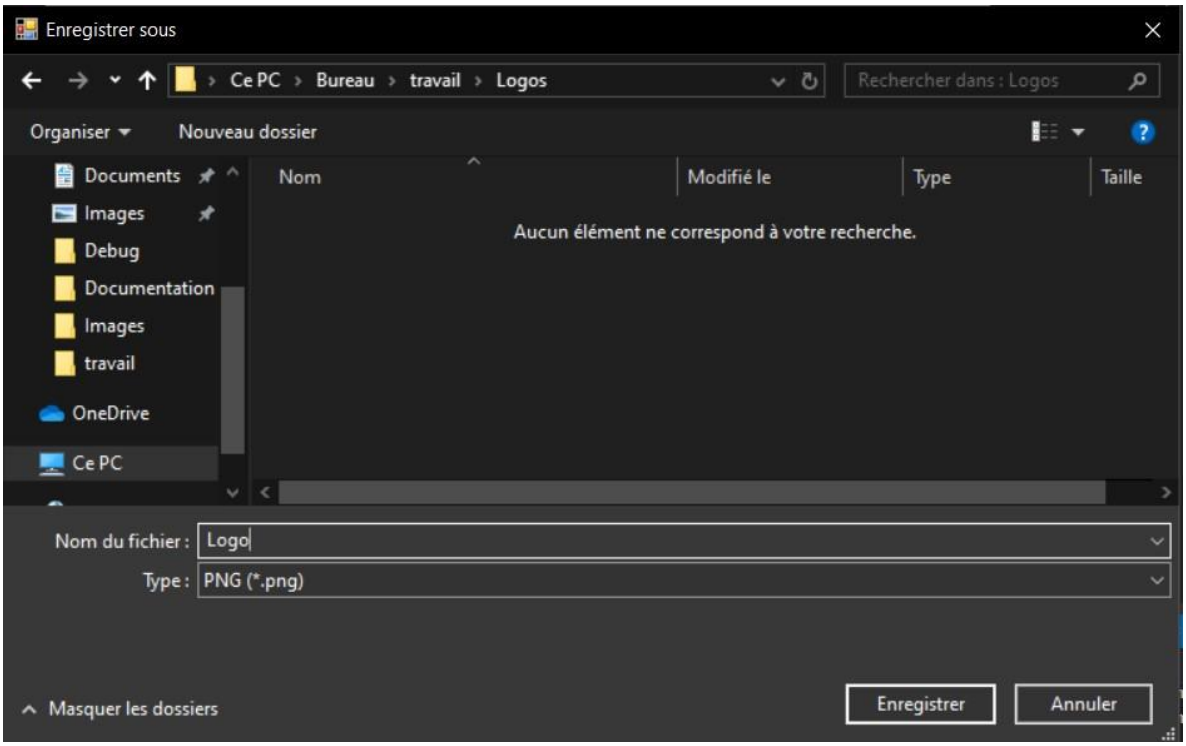
Tout d'abord, quand l'utilisateur clique sur "Fichier" dans la barre de menu, il doit cliquer sur "Exporter", comme le montre l'image ci-dessous :

Suite à cela, la fenêtre d'aperçu s'ouvre et montre à l'utilisateur un aperçu de son logo.

Pour ce faire, j'ai dessiné tous les sprites précédemment créés par l'utilisateur sur une même PictureBox : celle au centre de la fiche ExporterLogo (fiche montrant l'aperçu). En parcourant la liste des sprites, j'ajoute le dessin de chaque sprite ("SpritePaintAvecGraphics" dans le code) au Paint de la PictureBox d'aperçu.



Dès que l'utilisateur clique sur "Exporter", le programme recadre l'image de la PictureBox d'aperçu, la fenêtre d'aperçu se ferme et un SaveFileDialog apparaît et permet à l'utilisateur de choisir la destination, le nom, ainsi que l'extension de l'image finale. Voilà à quoi ressemble ce SaveFileDialog :



Une fois que l'utilisateur clique sur "Enregistrer", se trouvant en bas à droite du SaveFileDialog, l'image est correctement enregistrée au bon endroit et recadrée de la bonne façon. En ce qui concerne justement le code du recadrage, je m'y suis pris de manière à ce que le code trouve l'endroit auquel l'image doit être découpée. Pour cela, je parcours la liste des formes et compare leur position afin de trouver les coins du logo final. En connaissant les coordonnées des coins, il est possible de trouver la largeur ainsi que la hauteur du logo. Il suffit ensuite d'appeler la méthode RedimensionnerLogo qui accepte en paramètre une image (image que l'on veut recadrer), des coordonnées X et Y (position à laquelle on veut recadrer), une largeur ainsi qu'une hauteur (taille de l'image finale). Voici le code de la méthode RedimensionnerLogo :

```
public Bitmap RedimensionnerLogo(Bitmap original_image, int x, int y, int width, int height)

{
  Rectangle crop = new Rectangle(x - BORDURE_OFFSET, y - BORDURE_OFFSET, width + 2 * BORDURE_OFFSET,
height + 2 * BORDURE_OFFSET);

  var bmp = new Bitmap(crop.Width, crop.Height);
  using (var gr = Graphics.FromImage(bmp))
  {
    gr.DrawImage(original_image, new Rectangle(0, 0, bmp.Width, bmp.Height), crop, GraphicsUnit.Pixel);
  }
  return bmp;
}
```

Ce code a été pris d'internet (https://stackoverflow.com/questions/13217156/cropping-picture-getting-rectangle-from-x-y) et il me permet de couper l'image comme je veux. J'y ai apporté quelques modifications pour que l'image ait, par exemple, une certaine marge de chaque côté.

De cette manière, on obtient une image finale qui ressemble à celle-ci :

9. Plan de tests

Afin de vérifier qu'aucune régression et aucune erreur ne survienne au cours du développement, il est pratique d'avoir un certain scénario qui, étant testé chaque jour, permet de s'assurer que tout fonctionne.

Voilà le plan de test que j'ai utilisé :

Id	Description	Résultat attendu
1	Lancement de l'application.	Impossible de modifier les propriétés. Calque 1 choisi par défaut.
2	Création d'une forme de chaque type.	Chaque forme se crée correctement et apparaît avec les propriétés par défaut. On peut modifier les propriétés.
3	Modification des propriétés de chaque forme.	Chaque forme est modifiée comme prévu et les modifications sont bien visibles.
4	Changement de la propriété "Calque" pour avoir au moins une forme sur chaque calque.	Les formes se mettent sur le bon calque et on voit les formes se mettre dans le bon ordre.
5	Enregistrement du logo.	L'explorateur de fichier s'ouvre et on peut enregistrer le logo sans problème. Le nom par défaut est "Logo".
6	Suppression de toutes les formes.	Les formes se suppriment les unes après les autres comme on le souhaite. La partie "Propriétés" se met à jour.
7	Ouverture d'un fichier.	Charge le logo correctement. Les formes ont les bonnes propriétés et sont sur les bons calques.
8	Modification des formes.	Les modifications fonctionnent et aucune erreur ne survient.
9	Exportation du logo.	L'explorateur de fichier s'ouvre et nous permet de sauvegarder le logo en divers formats d'image. Le nom par défaut est "Logo".
10	Vérification de l'image.	L'image enregistrée contient bien le logo et fait la bonne taille.
11	Fermeture de l'application.	La MessageBox demande si on est sûr de vouloir fermer.
12	Appui sur OK de la MessageBox.	La MessageBox se ferme. LogoGo se ferme.

10. Rapports de tests

Dans ce chapitre, j'ai répertorié le résultat des tests de j'ai réalisé en suivant mon plan de test (voir **9. Plan de tests**). Voici trois rapports de test qui montrent l'évolution des fonctionnalités avec le temps.

10.1. Rapport de tests du 28 Mai :

Id	Date	Réussi	Description	Résultat
1	28.05.2020	OUI	Lancement de l'application.	Impossible de modifier les propriétés. Calque 1 choisi par défaut.
2	28.05.2020	NON	Création d'une forme de chaque type.	Manque le texte
3	28.05.2020	OUI	Modification des propriétés de chaque forme.	Impossible pour texte.
4	28.05.2020	OUI	Changement de la propriété "Calque" pour avoir au moins une forme sur chaque calque.	Fonctionne pour toutes les formes disponibles.
5	28.05.2020	NON	Enregistrement du logo.	Pas implémenté
6	28.05.2020	NON	Suppression de toutes les formes.	Pas implémenté.
7	28.05.2020	NON	Ouverture d'un fichier.	Pas implémenté.
8	28.05.2020	NON	Modification des formes.	Pas implémenté.
9	28.05.2020	NON	Exportation du logo.	Pas implémenté.
10	28.05.2020	NON	Vérification de l'image.	Ne peut pas encore être crée.
11	28.05.2020	NON	Fermeture de l'application.	Pas implémenté.
12	28.05.2020	NON	Appui sur OK de la MessageBox.	Pas implémenté.

10.2. Rapport de tests du 2 juin :

Id	Date	Réussi	Description	Résultat
1	02.06.2020	OUI	Lancement de l'application.	Impossible de modifier les propriétés. Calque 1 choisi par défaut.
2	02.06.2020	OUI	Création d'une forme de chaque type.	Les formes se créent et les propriétés se mettent à jour comme prévu.
3	02.06.2020	OUI	Modification des propriétés de chaque forme.	La modification marche pour toutes les formes et on le voit.
4	02.06.2020	OUI	Changement de la propriété "Calque" pour avoir au moins une forme sur chaque calque.	Fonctionne pour toutes les formes disponibles.
5	02.06.2020	NON	Enregistrement du logo.	Pas implémenté
6	02.06.2020	OUI	Suppression de toutes les formes.	Il est possible de supprimer les formes que l'on a créé. Les propriétés s'actualisent correctement.
7	02.06.2020	NON	Ouverture d'un fichier.	Pas implémenté.
8	02.06.2020	NON	Modification des formes.	Ne peut pas essayer tant que le chargement de fichier n'est pas implémenté.
9	02.06.2020	NON	Exportation du logo.	Pas implémenté.
10	02.06.2020	NON	Vérification de l'image.	Ne peut pas encore être crée.
11	02.06.2020	NON	Fermeture de l'application.	Pas implémenté.
12	02.06.2020	NON	Appui sur OK de la MessageBox.	Pas implémenté.

10.3. Rapport de tests du 4 juin :

Id	Date	Réussi	Description	Résultat

1	04.06.2020	OUI	Lancement de l'application.	Impossible de modifier les propriétés. Calque 1 choisi par défaut.
2	04.06.2020	OUI	Création d'une forme de chaque type.	Les formes se créent et les propriétés se mettent à jour comme prévu.
3	04.06.2020	OUI	Modification des propriétés de chaque forme.	La modification marche pour toutes les formes et on le voit.
4	04.06.2020	OUI	Changement de la propriété "Calque" pour avoir au moins une forme sur chaque calque.	Fonctionne pour toutes les formes disponibles.
5	04.06.2020	OUI	Enregistrement du logo.	Enregistre le fichier correctement. Le nom par défaut est Logo.xml
6	04.06.2020	OUI	Suppression de toutes les formes.	Il est possible de supprimer les formes que l'on a créé. Les propriétés s'actualisent correctement.
7	04.06.2020	OUI	Ouverture d'un fichier.	Crée les formes correctement avec les bonnes propriétés et sur les bons calques.
8	04.06.2020	OUI	Modification des formes.	Les modifications fonctionnent normalement.
	04.06.2020	OUI	Exportation du logo.	Nom par défaut "Logo.bmp". Tout est correct.
10	04.06.2020	OUI	Vérification de l'image.	L'image se créer parfaitement avec les formes correctes. Bonne taille.
11	04.06.2020	OUI	Fermeture de l'application.	La MessageBox s'affiche correctement.
12	04.06.2020	OUI	Appui sur OK de la MessageBox.	L’application se ferme comme prévu après l’appui du bouton OK.

11. Fonctionnalités à ajouter

Arrivant à la fin du projet, je me suis rendu compte qu'il existe beaucoup de fonctionnalités intéressantes qui seraient très utiles à l'application. Cet avant-dernier chapitre liste certaines fonctions qui pourraient être ajoutées dans le futur.

Masquer un calque

Pour mieux travailler sur chaque calque, la possibilité d'afficher ou non les calques de notre choix peut être une idée très intéressante. Cette fonctionnalité est disponible dans de grands logiciels comme, par exemple, Photoshop.

Ajout / Suppression de calque

La création et suppression de calque sont deux fonctionnalités qui ne seraient pas difficiles à implémenter et qui ajouteraient un plus à la version actuelle de LogoGo. Certains logos n'ont pas forcément besoin de trois calques et, à l'inverse, certains en demandent bien plus,

La rotation de sprite

La rotation de sprite est une fonctionnalité qui permettrait de créer des logos bien plus complexes. Pour être honnête, c'est une fonctionnalité que je voulais implémenter dès le début du projet. Les 11 jours sont malheureusement passés trop vite et je n'ai pas pu ajouter la rotation aux fonctionnalités de l'application car j'ai préféré consacrer mon temps aux fonctionnalités essentielles.

Fusion de calques

C'est une autre fonctionnalité facile à implémenter. Il suffirait d'ajouter toutes les formes de deux calques sur un seul et même calque dans le bon ordre. Cette fonctionnalité rendrait, selon moi, la création de logo plus organisée.

Plus d'options

Pour les textes par exemple, il serait intéressant de pouvoir souligner, surligner ou encore changer la police d'un texte. Même si les modifications que LogoGo permet de faire aux textes fonctionnent, il serait bien d'avoir plus de choix.

Plus de formes !

Plus il y a de choix, mieux c'est. Créer des étoiles, flèches, traits, octogones, hexagones et j'en passe. Une fois qu'on connait le chemin à tracer, la création de n'importe quelle forme devient très simple. Il serait judicieux d'ajouter plus de formes à celle disponibles actuellement. L'utilisation de courbes de Bézier serait également très intéressante pour créer des formes plus raffinées.

12. La documentation

Toute la documentation de mon projet a été générée. Pour la documentation principale (Documentation Technique et Manuel Utilisateur), j'ai utilisé le logiciel Typora pour tout écrire en Markdown avant de convertir les documents en PDF.

Le MarkDown m'a permis d'écrire la documentation bien plus rapidement puisque je ne me suis pas occupé de la mise en forme. J'ai directement écrit toute la documentation en y ajoutant les images sur Typora. Pour ce qui est du document du Code source, je l'ai réalisé en utilisant LaTex. Le site « Overleaf » m'a permis d'éditer le fichier LaTex et d'importer les différents fichiers de mon code. Il m'a ensuite suffi d'exporter le document en PDF.

J'ai également utilisé Doxygen pour avoir une documentation détaillée de tout le code. Cela permet d'extraire les commentaires que j'ai attribué aux différentes méthodes et de les affichés bien mis en forme sur un site en local qui documente l'intégralité du code. Vous pouvez trouver cette documentation dans docHTML.zip qui se trouve dans le dossier Documentation. Il vous suffit ensuite d'ouvrir index.html dans un navigateur.

La documentation a été réalisée de cette manière car M. Bonvin me l'a conseillé. Nous n'avons effectivement pas le temps, en 11 jours, de fournir une documentation ainsi que du code d'une qualité parfaite. Nous avons donc opté pour une documentation simple qui se génère elle-même. Pour réaliser une documentation beaucoup plus complète en plus d'un code de qualité, il faudrait bien plus que 11 jours.

13. Conclusion

C'est grâce aux exercices faits en classe tout au long de ma formation que j'ai pu réaliser ce projet. Certaines connaissances ayant été acquises en première année (DrawString, DrawRectangle, ...) et d'autres en quatrième (Sérialisation, désérialisation, ...), j'ai été obligé d'utiliser de nombreuses notions pour être en mesure d'arriver au bout de ce travail. Même si le travail est globalement réussi, beaucoup de fonctionnalités peuvent encore être ajoutées ou optimisée...

Je suis satisfait du travail que j'ai fourni durant ces 11 jours de travail car, même si on fait beaucoup d'exercices et de travaux en groupe durant les cours, je n'avais jamais fourni cette quantité de travail seul pour une période de temps si courte. Cependant, j'ai remarqué que j'ai encore beaucoup de choses à améliorer. J'ai, par exemple, remarqué qu'il m'arrivait de perdre la notion du temps et des tâches qu'il me restait à accomplir. Même si cela ne m'a pas spécialement porté préjudice durant le TPI, c'est un défaut sur lequel je dois travailler pour éviter d'avoir du retard dans de futurs projets d'une certaine envergure.

Les points techniques demandé ayant tous été remplis, je peux dire que je suis content du résultat du résultat de ce travail. Les conditions de ce travail ayant été particulières (TPI à la maison à cause du COVID-19), il fallait s'attendre à ce que certaines choses ne se passent pas comme je le voulais. Le fait de travailler à la maison m'a fait remarquer des pertes de motivation à certains moments. Le simple trajet de la maison au lieu de travail donne plus envie de travailler que le travail à domicile, selon moi. Même en considérant ces éléments inattendus, j'estime que, lors de mes prochains grands projets, je pourrais mieux m'organiser afin d'être satisfait d'avantage sur certains points.