Feuille de TD n°8 :

Exercice de synthèse : modélisation UML / Programmation événementielle / C++ / modèle MVC / bibliothèque wxWidgets

Objectif pédagogique

Cette feuille de TD a pour but proposer un exercice récapitulatif :

- Mettant en œuvre les enseignements de ce module en lien avec la modélisation objet : cas d'utilisation => diagramme de classes => diagramme états-transitions => conception d'une interface graphique => Traduction de l'analyse sous la forme d'un programme événementiel à interface graphique avec C++ et wxWidgets, en respectant une organisation du code inspirée du modèle MVC.
- Permettant aussi de se préparer au travail de conception et de développement en équipe

Ressources nécessaires - Consignes

- Le cours d'IHM, + documentation de wxWidgets (http://docs.wxwidgets.org/3.0/)
- Le cours d'UML
- Cette feuille de TD n°8 et ses annexes

SUJET :

Il s'agit de réaliser un jeu de Chifoumi **simple,** précédé d'une **rapide** analyse UML.

SPECIFICATIONS EXTERNES DU JEU DE CHIFOUMI :

- L'utilisateur joue seul contre la machine
- Lorsque l'utilisateur lance le programme, le plateau se remplit avec les éléments du jeu (cf. Figure 1) :

Figure 1 : Etat initial

Figure 2: Partie en cours

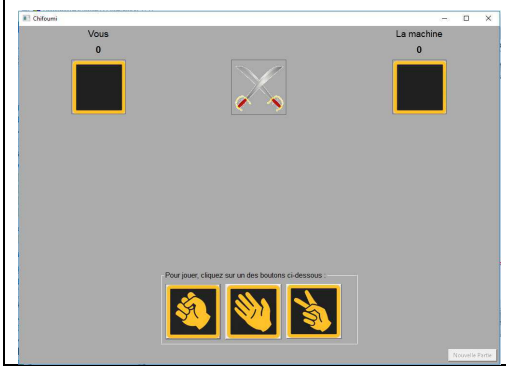


Figure 3: Pas de redimensionnement des objets, mais un repositionnement lors de l'agrandissement de la fenêtre

- Des zones de visualisation, pour consulter son score, le dernier coup qu'il a joué, le score de la machine, le dernier coup que la machine a joué.
- Des zones d'interaction, où le joueur peut jouer un nouveau coup, ou bien commencer une nouvelle partie.

- Le joueur joue en sélectionnant la figure qu'il souhaite tirer (pierre, papier, ciseau). La machine joue également. Les coups respectifs sont affichés, et les scores mis à jour (+ 1 point) en fonction du coup de chacun (pierre gagne à ciseau, ciseau gagne à papier, papier gagne à pierre). Les scores restent inchangés en cas de coup identique.

TRAVAIL À FAIRE :

- Vous travaillerez par groupes de 2 à l'intérieur de votre groupe de TP.
- Le travail s'organisera autour des étapes décrites ci-dessous. Le livrable final à rendre sera constitué des éléments d'analyse et de conception (documentation décrite ci-dessous) ainsi que des sources du programme.

0. **Spécifications externes du produit réalisé.**
   Préciser **uniquement** les points qui vous ont semblés flous ou bien incomplets.

1. **Scénario nominal**
   Décrire sous forme textuelle le scénario nominal d'utilisation de ce jeu correspondant au cas d'utilisation "JOUER" donné en Annexe.
   Les **messages** à destination du système seront bien mis en évidence (par exemple soulignés) de sorte à faciliter l'identification des **méthodes** (préparation pour la question 2.-)

2. **Diagramme de classe (UML) – v1 - aspects métier**
   (a) Élaborer ***manuellement*** *(*sans atelier !*)* ***une première version*** du diagramme de classes UML du jeu `Chifoumi`, en vous focalisant sur les classes 'métier', c'est à dire celles décrivant les éléments du jeu indépendamment des éléments d'interface.
   (b) Établir *succinctement* le dictionnaire des éléments et des méthodes :
   - nom, type et signification pour les éléments
   - nom, but, nom-type des paramètres et type de la valeur résultante pour les méthodes

   (c) Créer un projet CodeBlocks *en mode Console* pour implémenter les classes définies précédemment.
   - Créer les fichiers avec la/les classes définies précédemment
   - *Veiller à ce que le code source intègre toujours une version à jour et complète des dictionnaires des éléments et des méthodes*
   - Laisser le fichier `main.cpp` vide pour l'instant
   - Compiler / corriger jusqu'à éliminer les fautes de frappe
   - Tester le code produit avec le fichier de test (main.cpp) qui vous sera fourni

   (d) *-> <u>dossier</u> : Une fois l'analyse validée par votre enseignant et le code stabilisé*, reporter sur votre dossier le schéma de classes UML (a) ainsi que le dictionnaire des éléments et des méthodes (b-c)

3. **Diagramme états-transitions**
   (a) Lister les différentes valeurs d'**états du jeu**.
   (b) À partir de cette liste, établir le **diagramme états-transitions** du jeu
   (c) Établir le dictionnaire des *états* (nom état, signification), des *événements* (nom événement, signification) et des *actions* (nom, but) correspondant
   (d) Élaborer la table **`T_EtatsEvenementsJeu`** correspondant à la version matricielle du diagramme états-transitions :
   - en *ligne : les* ***événements*** faisant changer le jeu d'état
   - en *colonne : les* ***états*** du jeu

   Un exemple est fourni en Annexe 2.
   (e) *-> <u>dossier</u> : Une fois ces éléments validés par votre enseignant*, reporter dans votre dossier le diagramme (b), les dictionnaires (c) et la table (d).

4. **Interface**
   (a) Compléter la table **`T_EtatsEvenementsJeu`** avec les noms des variables correspondant aux objets graphiques qui généreront les événements
   (b) Créer *succinctement sur papier* une maquette de votre future interface.
   - Vous pouvez sauter cette étape si vous suivez le modèle d'interface proposé dans le sujet.

   (c) Identifier sur cette maquette les éléments graphiques cités dans la table **`T_EtatsEvenementsJeu`**

(d) Créer un projet CodeBlock utilisant le générateur d'interfaces wxSmith,
- Vous trouverez en annexe 3 la marche à suivre pour créer un projet CodeBlocks correspondant.

(e) Créer l'interface de l'application conformément à votre maquette, **sans** les sizers.
- *Veiller à ce que le code source de vos classes graphiques intègre toujours une version à jour et complète des dictionnaires des éléments et des méthodes.*

(f) -> *dossier : Une fois ces éléments validés par votre enseignant*, reporter dans votre dossier la table complétée (a), une copie d'écran de votre maquette (ou bien celle du sujet), puis décrire l'état des éléments de l'interface pour chaque état de l'application (du jeu) :
- nom de l'état
- état/propriétés de chacun des objets d'interface pour cet état de l'application (du jeu) : actif/inactif, visible/non visible, focus,…

5. Organisation du code - MVC

(a) Faites un schéma pour préciser la manière dont sera organisé le code, sachant qu'il doit respecter le modèle MVC comme dans l'exercice précédent (Additionneur).
- Pour ce faire, vous pouvez vous inspirer du corrigé de l'exercice précédent

(b) -> *dossier : Une fois ces éléments validés par votre enseignant*, reporter ce schéma dans votre dossier

6. Mise en conformité du code : compléter les fichiers de spécification des classes (fichiers .h)

(a) Rapatrier dans le projet graphique les classes métier développées au point 2.-

(b) Compléter les fichiers de spécification (fichiers .h) de vos classes métier et graphique pour qu'ils soient conformes au modèle MVC
- La qualité de documentation de votre code fera apparaître les parties correspondant au Modèle, au Contrôleur, à la Vue

(c) -> *dossier : Une fois ces éléments validés par votre enseignant*, reporter les contenus de ces fichiers dans votre dossier (cf. exemple de l'Additionneur en annexe 2)

7. Finalisation de la programmation

(a) Mettre en conformité les fichiers de définition (fichiers .cpp) associés aux précédents fichiers de spécification (fichiers .h)

(b) -> *dossier :* Insérer dans le dossier (dans le paragraphe dédié à l'interface) une copie d'écran de l'interface sur laquelle vous aurez reporté les sizers permettant le redimensionnement/repositionnement des éléments graphique en fonction du changement de taille de la fenêtre principale.

(c) Coder la mise en œuvre des sizers.

8. Spécifications internes du programme réalisé

(a) *Si cela est nécessaire*, lister dans le dossier des éléments du programme autres que ceux déjà décrits dans les questions précédentes :
- définition de types
- variables / constantes : nom, type, signification

(b) *Si cela est nécessaire*, décrire succinctement dans le dossier les sous-programmes utilisés autres que ceux déjà décrits dans les questions précédentes
- nom, éventuellement type de la valeur retournée, liste des paramètres et but

9. Bibliographie / webographie . Ajouter à votre dossier les ressources utilisées.

10. Bilan des activités

Préciser dans votre dossier le temps passé (en heures/groupe, y compris le temps prévu à l'edt), apprentissages/pratiques à retenir pour de futurs projets.

A déposer sur eLearn : 1 fichier .pdf (dossier) + code source du projet dûment documenté + les fichiers des ressources (images .jpg ou .gif) utilisées pour réaliser l'interface

---

Pour celles et ceux qui veulent aller plus loin :

(a) Sauvegarder votre précédent travail → *version Simple*

(b) Faites évoluer votre document d'analyse et votre projet Codeblocks vers une *version avecParamétrage* qui permettra à l'utilisateur de personnaliser son prénom à tout moment du programme (cf. figures suivantes) :

Figure 4 : Etat initial

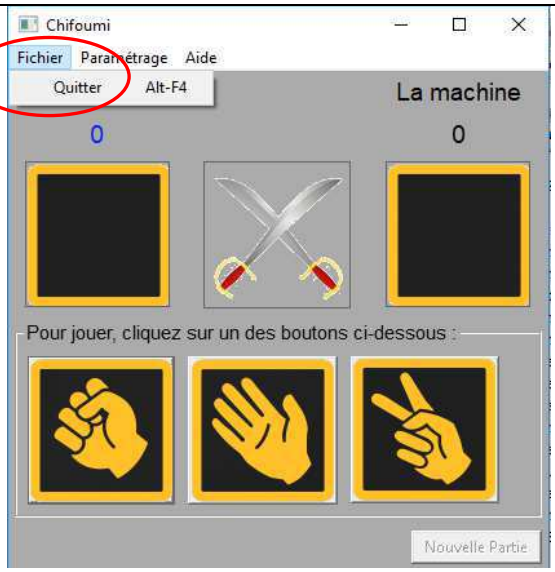


Figure 5: Le menu 'Fichier' contient 1 unique item 'Quitter' qui permet de quitter l'application

Figure 6: Le menu 'Aide' contient 1 unique item 'Au sujet de…' qui …..

Figure 7: …..ouvre une boîte de Message

Figure 8: Le menu 'Paramétrage' contient 1 unique item 'Joueur' qui permet de modifier le prénom du joueur,…

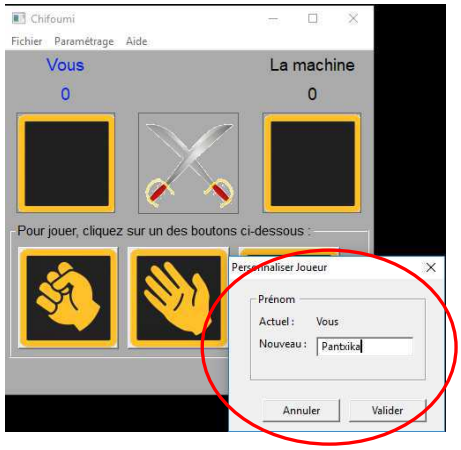


Figure 9: … à n'importe quel moment du jeu. Ici, le paramétrage est fait dans une fenêtre modale

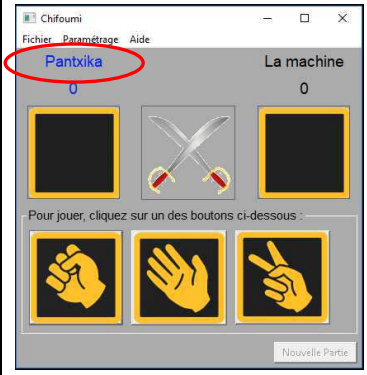
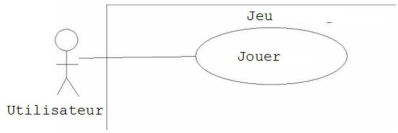


Figure 10: Le prénom du joueur a été modifié

ANNEXE 1 - DOCUMENTS DISPONIBLES

1. Cas d'Utilisation du système



ANNEXE 2 - FORMULAIRES DISPONIBLES

Scénario

Titre :
Résumé : . **Acteur** : Utilisateur (acteur principal)
Pré-condition :
Post-condition :
Date de création : **Date de mise à jour** : **Version** : .
Créateur :

Utilisateur	Système

Dictionnaire des Classes : attributs et méthodes

Classe XXX

Attributs

Type	Nom attribut	Signification	Exemple (si nnécessaire)
`int`	`opG`	`opérande Gauche de l'opération opG + opD`	`12`

Méthodes

Déclaration **// But**
`void demandeAddition(string p_opG, string p_opD);` Traite la demande d'addition pour les paramètres p_opG et p_opD textuels. Le type string des paramètres a été préféré pour son côté *universel* Ce que fait le contrôleur : • si les paramètres p_opG et p_opD correspondent à des entiers, • calcule la somme, fait éventellement changer le système d'état et ordonne à la vue de se mettre à jour • sinon, fait changer le système d'état et ordonne à la vue de se mettre à jour

Diagramme états-transitions

Diagramme du système (et non de chacun de ses composants)

Ici le diagramme états-transition

Dictionnaire du Diagramme états-transitions

États du jeu (=du système)

Nom état	Signification
saisiesEnCours	L'application **attend** que l'utilisateur lance le calcul. En principe, l'utilisateur doit terminer de saisir les valeurs dans les zones de texte disponibles. Mais il peut lancer le calcul à tout moment, que les zones de texte soient vides ou pas, correctement renseignées ou pas. Ecran associés : écran 1 (2 zones de texte vides), écran 3 (2 zones de texte saisies).

Événements faisant changer le jeu (=système) d'état

Nom événement	Signification
calculDemandé	L'utilisateur lance le calcul à partir des valeurs de X et Y qu'il a saisies. Cette demande de calcul peut aboutir (si les valeurs saisies sont correctes) ou pas. Cette condition sur l'état des zones de texte s'exprime par une **garde**. En fonction de la garde, fait passer le système à l'état saisieEnCours ou bien à l'état saisieEnCours

Actions qui accompagnent chaque transition

Nom action	But
Action 1	Le système efface le contenu des champs de saisie et la zone réservée à afficher le résultat de l'addition, et autorise de nouvelles saisies. Rien n'est précisé (et ne sera fait) quant au placement du curseur sur un quelconque élément de l'interface.

Version matricielle du Diagramme états-transitions

Evénement → *↓ nomEtat*	calculDemandé	notificationAcceptée	effacementDemandé
saisieEnCours	[expCalculable=vrai] : **saisiesEnCours** **----------** [expCalculable=faux] : **erreur**	----	**saisiesEnCours**
erreur	----	**saisiesEnCours**	----

Description de l'interface de l'application – état par état

Copie d'écran de l'interface

Ici une première copie d'écran de l'interface, avec en légende les noms des éléments graphiques qui la composent

Ici une autre copie d'écran de l'interface, avec les sizers, dessinés et légendés

*Etats des éléments graphiques lorsque le système est dans l'état **XX***

Elément d'interface	Description (visible / invisible ; activé / inactif ; focus ; …)

*Etats des éléments graphiques lorsque le système est dans l'état **XX***

Elément d'interface	Description (visible / invisible ; activé / inactif ; focus ; …)

Organisation des spécifications des classes – inspirée de MVC

Modèle

```
class Additionneur
{
public:
    ///* méthodes métier
    Additionneur();
    virtual ~Additionneur();

    int getOpG();
    int getOpD();
    void setOpG(int p_op);
    void setOpD(int p_op);
    int getSomme();
    // retourne le résultat de l'opération opG+opD

protected:
    ///* attributs métier
    int opG; // opérande Gauche de l'opération opG + opD
    int opD; // opérande Droit de l'opération opG + opD
       // pour simplifier la classe, j'ai décidé de ne pas stocker la somme
```

Contrôleur

```
public :
    ///* états du système
    enum UnEtat {saisiesEnCours, erreur};

    ///* Méthodes du Contrôleur
        // Pour gérer la logique de fonctionnement du système
    UnEtat getEtat();
    void setEtat(UnEtat p_etat);

       // Pour gérer les événements = actions de l'Utilisateur sur la Vue
    void demandeAddition(string p_opG, string p_opD);
    /* Traite la demande d'addition des contenus des paramètres p_opG et
       p_opD de type string */

    void accepteNotification();
    // Traite l'acceptation, par l'utilisateur, du message d'erreur

    void demandeEffacement();
    // Traite la demande d'effacement faite par l'utilisateur

       // Pour gérer le lien entre le Modèle et la Vue
    Principale* getVue();       // retourne pointeur sur vue

    void setVue(Principale* p_vue);
    // Permet à la vue de s'enregistrer auprès du modèle

protected:
    ///* Attributs liés au Contrôleur
    Principale* laVue;     // mémorise la Vue pour s'adresser à elle
    UnEtat etat;           // mémorise l'état du système
};
```


changement

```
class Principale : public wxFrame
{
    ///* Méthodes et attributs 'classiques' de la Vue
    public:
        // Le constructeur
        Principale(const wxString& title);

        // Le destructeur
        virtual ~Principale();

    private:
           ///* Gestionnaires d'événements
        void OnClicBtonAddition (wxCommandEvent& event);
        /* Maintenant, notifie le Contrôleur que l'utilisateur a demandé le
              calcul */

        void OnClicBtonEffacer (wxCommandEvent& event);
        /* Maintenant, notifie le Contrôleur que l'utilisateur a demandé
           l'effacement */

           ///* Attributs membres
        // pas de changement par rapport à la version antérieure
```

en plus

```
    ///* Méthodes de mise à jour de la Vue
    public:
        void effacerZonesSaisie();
        // efface le contenu des 2 champs de saisie

        void afficherResultat(int p_res);
        // afficher l'entier dans la zone label_res

        void afficherEtatErreur();
        // affiche la vue de l'application lorsqu'elle est en état d'erreur


    ///* Méthodes et attributs pour faire le lien avec le Contrôleur
    public:
        Additionneur* getAdditionneur();
        void setAdditionneur (Additionneur* p_add);

    private:
        Additionneur* additionneur;

};
```

ANNEXE 3 –

CRÉATION D'UN PROJET CODEBLOCKS AVEC GÉNÉRATEUR D'INTERFACES GRAPHIQUES

Reportez-vous sur eLearn – rubrique TD n8 - merci

ANNEXE 3 –
CRÉATION D'UN PROJET CODEBLOCKS
AVEC GÉNÉRATEUR D'INTERFACES GRAPHIQUES

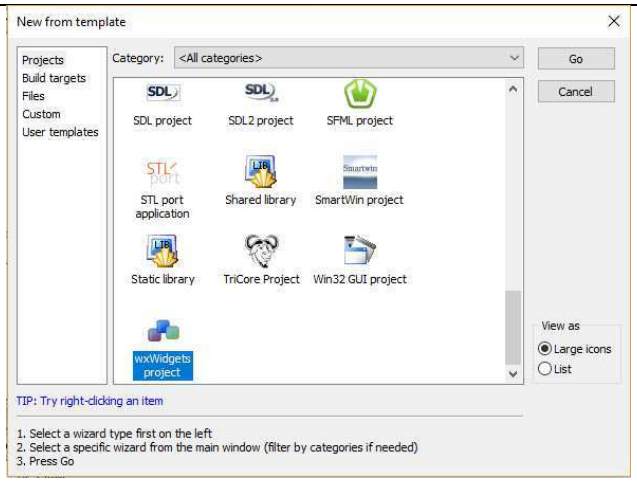


Figure 11 : Création d'un projet CodeBlocks- wxWidgets

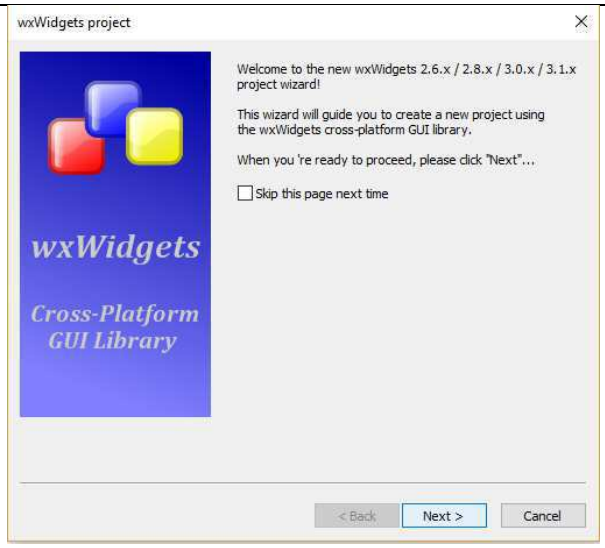


Figure 12 : Accueil

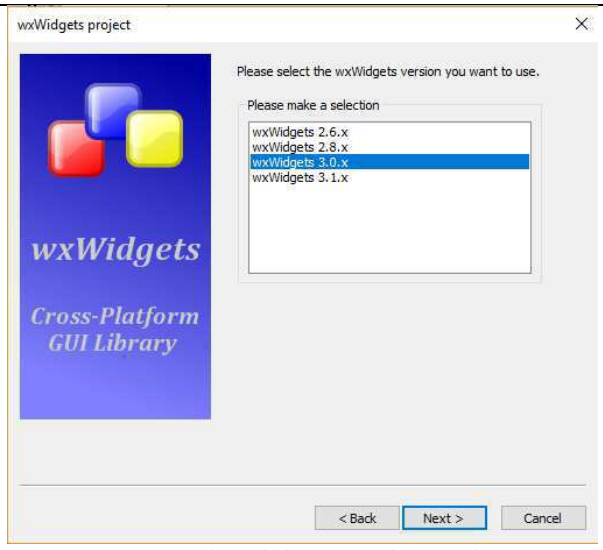


Figure 13 : Choix de la version de wxWidgets

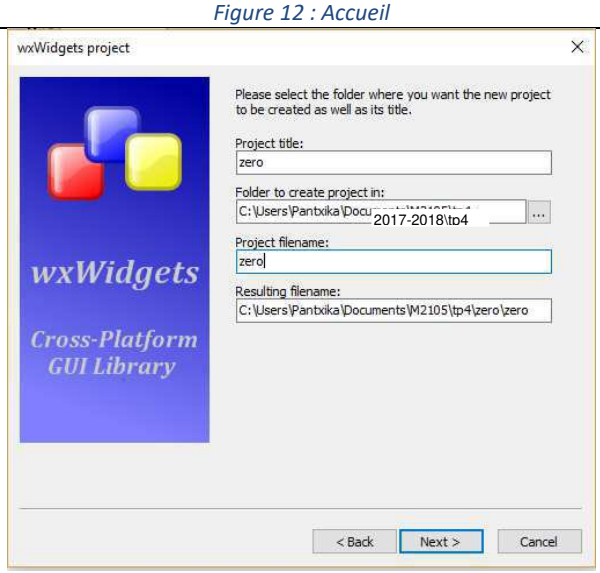


Figure 14 : Création d'un projet zero dans répertoire tp4

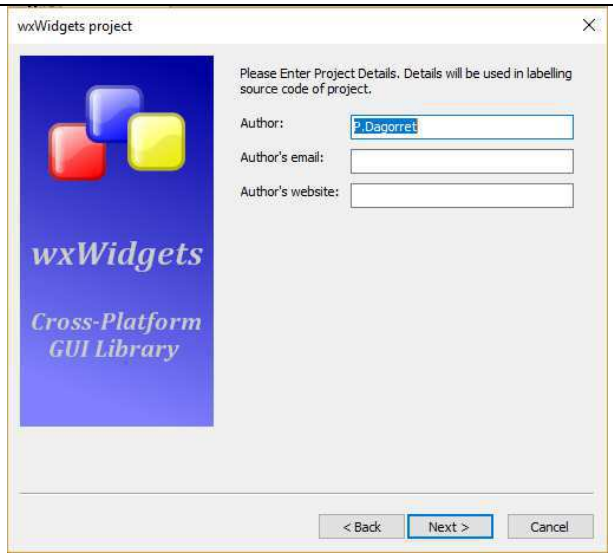


*Figure 15 : **Détails du projet***

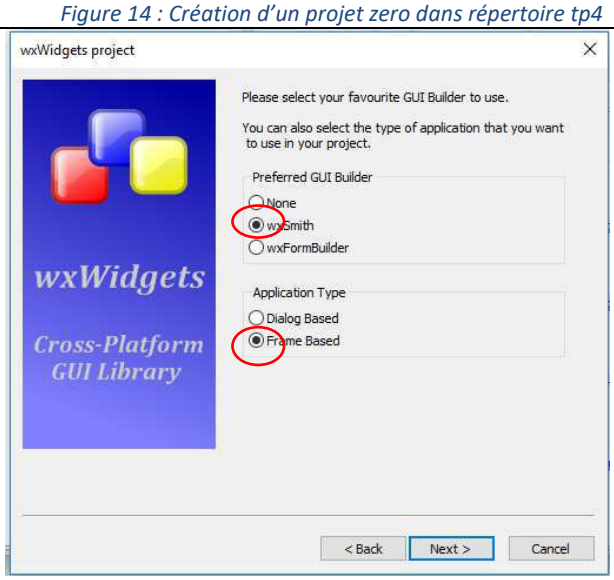


Figure 16 : Utilisation du générateur d'interfaces graphique wxSmith – la fenêtre du programme sera de type Frame

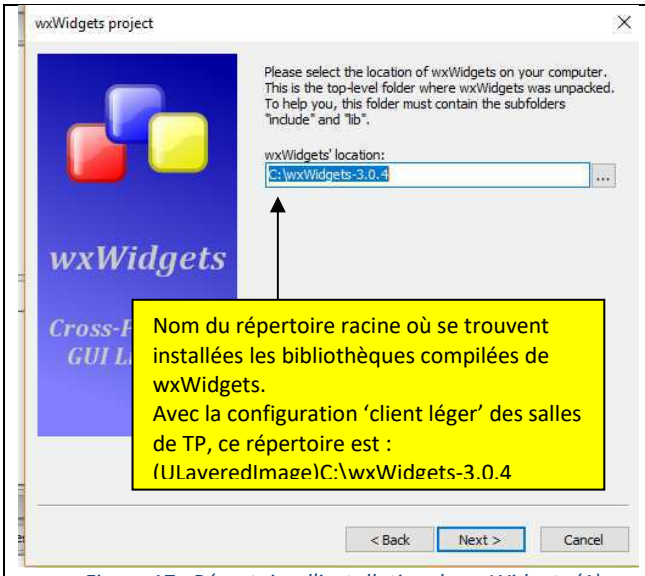


Figure 17 : Répertoire d'installation de wxWidgets (1)

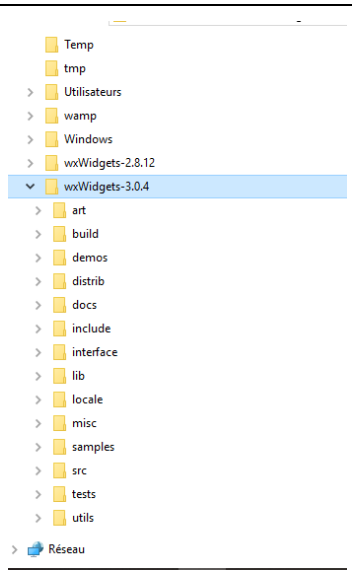


Figure 18 : Répertoire d'installation de wxWidgets

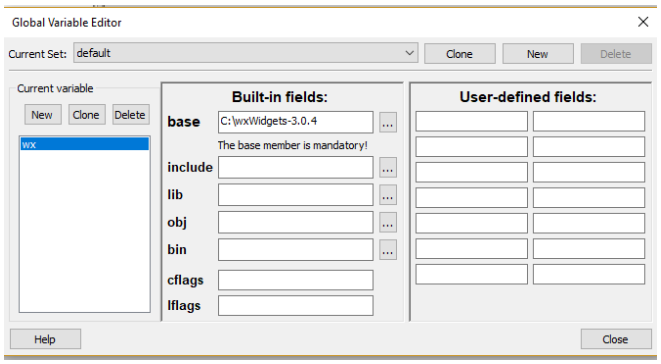


Figure 19 : Répertoire base de localisation des objets graphiques wxWidgets // également accessible via le menu Settings/Global variables… de CodeBlocks

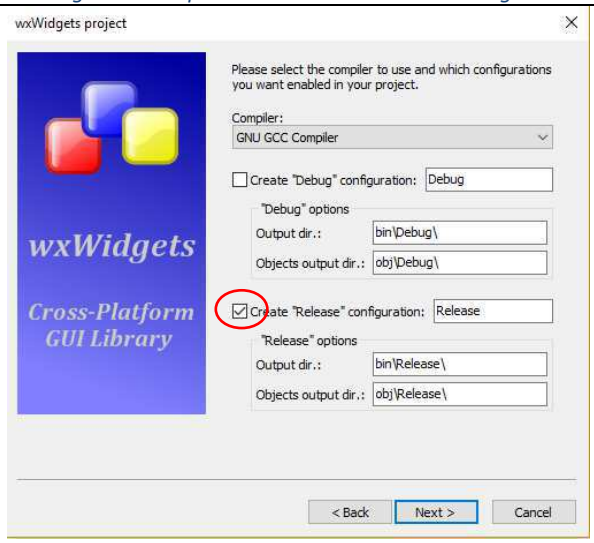


Figure 20 : Choix du compilateur et du mode RELEASE (2)

Figure 21 ***: Paramètres de compilation et du projet (1) : Caractères UNICODE, la bibliothèque wxWidgets sera générée sous la forme d'un unique fichier***

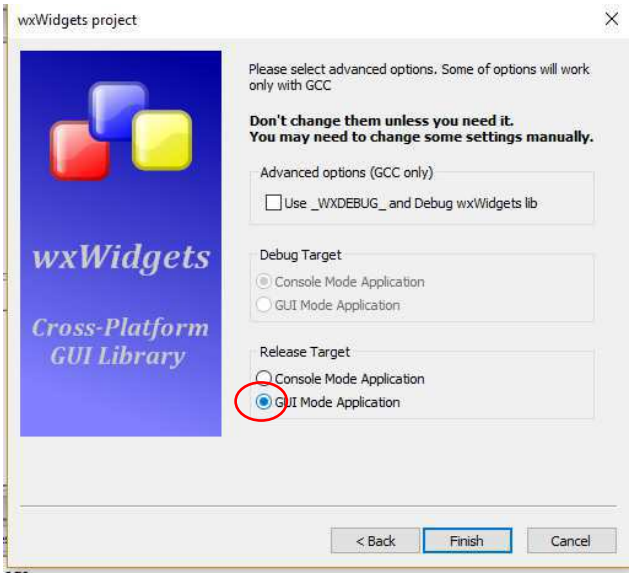


Figure 22 ***: Paramètres de compilation et du projet (2) : création d'une application à Interface Graphique (GUI)***

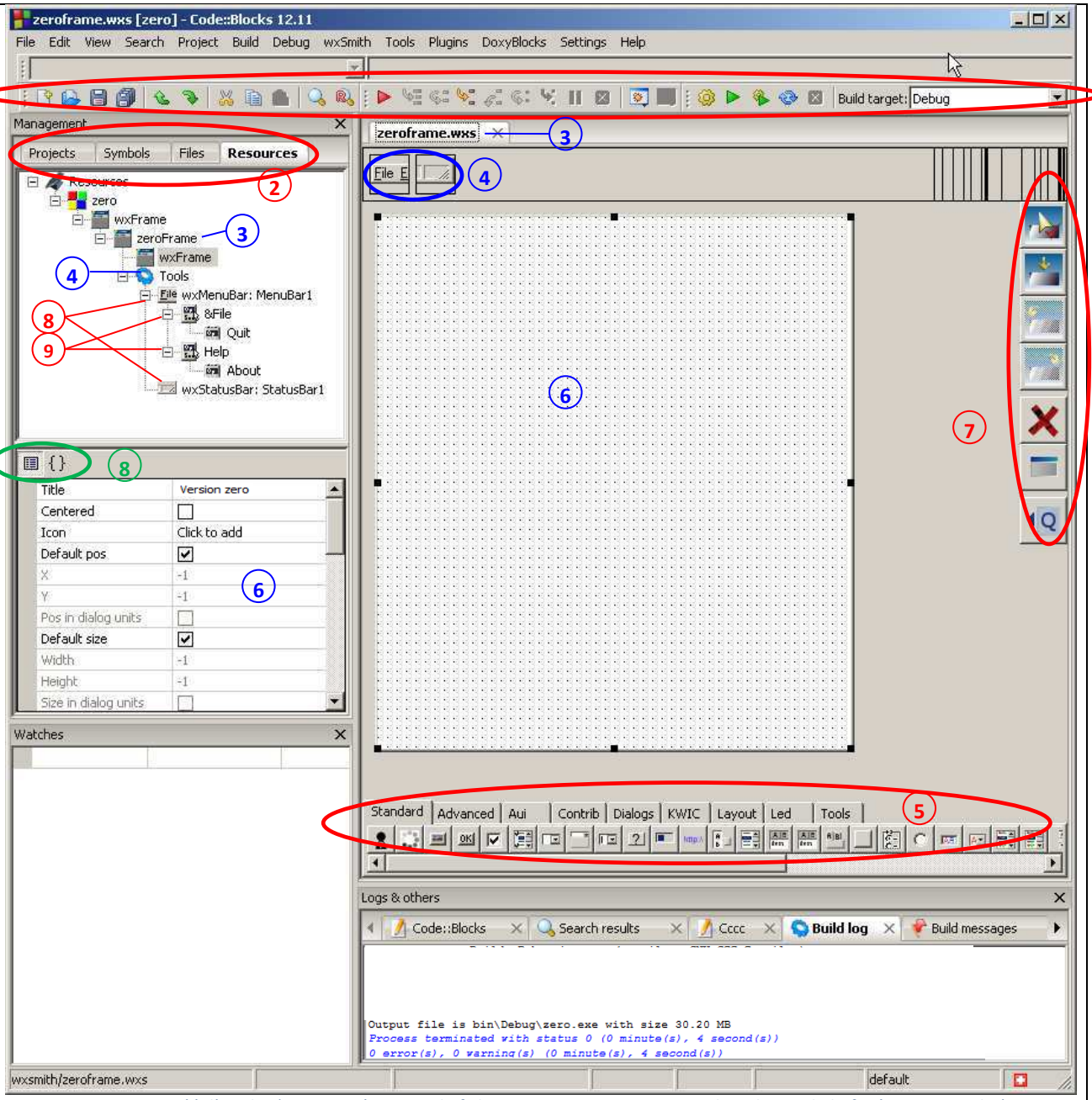


Figure 23 : Projet créé, l'onglet 'Resources' recense le fichier wxFrame, qui correspond au dessin de la fenêtre principale (encore vide) de l'application)

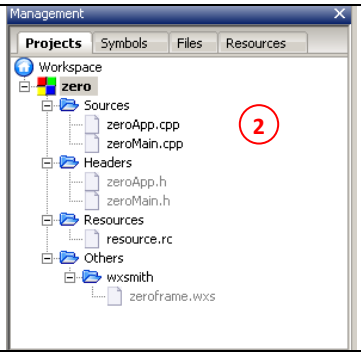


Figure 24 :

L'onglet 'Project' de la fenêtre 'Manager' recense les fichiers de l'application générés par wxSmith : fichiers de l'application minimale + le fichier contenant le dessin de le fenêtre principale

Découvrez l'interface du générateur d'interfaces graphiques wxSmith

1. Nommer et expliquer le rôle des éléments numérotés
2. Demander à votre enseignant ce que vous ne comprenez pas