TP1 : Environnement Gtkmm et Glade

L'objectif de ce TP est triple :

- configurer un environnement Eclipse pour le développement d'application avec Gtkmm,
- développer une première application simple,
- présentation de l'utilisation de Glade.

Ce TP est donc assez long …

Pour les deux derniers points, nous allons développer un module indépendant de l'application requise pour le TP, à savoir le sélecteur de couleurs. Nous le ferons une première fois en « pur code », puis nous utiliserons Glade.

1 *Configuration de l'environnement Eclipse/Gtkmm*

Solution 1 : Référez vous au tutoriel pour configurer un nouveau projet Eclipse disponible sur le site du cours,

Solution 2 : Reprendre le projet vierge configuré (disponible dans votre espace partagé),

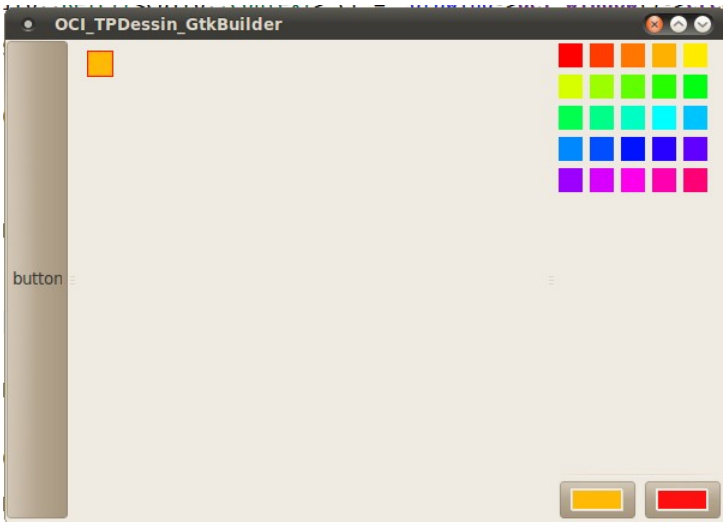
Solution 3 : Créer un projet CMake inspiré des exemples du couors et générer les fichiers de projet Eclipse avec CMake (personnellement, la génération d'un projet Eclipse ne marche pas, mais le Makefile peut être utilisé !)

2 *Développement d'un sélecteur de couleur*

A *Contextualisation : une version très simple du projet final*

L'objectif est de développer la petite interface ci-dessous. À gauche, un bouton, au centre, un DrawingArea dans lequel on peut dessiner des formes, et à droite le sélecteur de couleur, un ensemble de widget un peu plus complexe sur lequel on va travailler.

Dans l'idée, lorsqu'on clique sur le bouton, un carré est dessiné dans la zone de dessin avec les couleurs du sélecteur de couleur.



Pour le sélecteur de couleur que nous allons développé comme un widget indépendant hérité d'un `Gtk::VBox`, il y a :

- en bas : des `Gtk::ColorButton` qui permettent de d'accéder facilement à une boîte de dialogue pour la sélection de couleur,
- en haut : un ensemble de couleurs prédéfinies est disponible pour l'utilisateur pressé. (la palette de couleurs)

Lorsqu'on utilise un clic-droit sur l'une des couleurs prédéfinies, elle est attribué au `ColorButton` du *foreground* (couleur de premier plan). Lorsqu'on utilise le clic-gauche, la couleur est attribuée au `ColorButton` du *background* (couleur de fond).

B Création de la fenêtre générale ManualWindow

1. Créer une classe `ColorSelector` qui hérite de `Gtk::VBox` dans laquelle vous ajouterez deux fonctions publiques :
   - `Gdk::Color get_fg()` : getter de la couleur *foreground*,
   - `Gdk::Color get_bg()` : getter de la couleur *background*.

Dans l'immédiat, ces deux fonctions se contenteront de retourner une couleur prédéfinie dans le code. Le reste de la classe sera complété dans la section suivante

2. Créer une classe `ManualWindow` qui hérite de `Gtk::Window` dans laquelle vous placerez un bouton, un `DrawingArea` et un `ColorSelector` de sorte à ressembler aux spécifications précédentes,
3. Créer un fichier `main.cpp` pour lancer votre fenêtre.

Vous connecterez le clic du bouton à une fonction dédiée au dessin d'un carré. Le contenu de cette fonction est donné ci-dessous. Les noms des variables peuvent bien évidemment être différents pour vous.

```
//On récupère la couleur depuis le manager :
Gdk::Color colorfg=_colorselector->get_fg();
Gdk::Color colorbg=_colorselector->get_bg();

//Récupération du contexte Cairo pour dessiner dans le DrawingArea
Cairo::RefPtr<Cairo::Context> cr = _drawing->get_window()->create_cairo_context();

//Sauve le contexte
cr->save();

//Changement des options de dessin
cr->set_source_rgb(colorfg.get_red_p(), colorfg.get_green_p(), colorfg.get_blue_p());
cr->set_line_width(2.0);

//Définition du chemin à dessiner
cr->move_to(10, 10);
cr->line_to(10, 30);
cr->line_to(30, 30);
cr->line_to(30, 10);
cr->close_path();

//Réalisation effective du dessin
// stroke => tracé de la ligne, preserve => on conserve la forme en mémoire
cr->stroke_preserve();
```

```
//Changement de couleur de dessin
cr->set_source_rgb(colorbg.get_red_p(), colorbg.get_green_p(), colorbg.get_blue_p());
//Réalisation effective du dessin, mais en utilisant un remplissage
cr->fill();

//Restaure le contexte
cr->restore();
```

C Création d'une classe ColorViewer

Pour répondre aux spécifications, je propose de créer une classe spécifique qui héritera d'un widget et aura comme membre la couleur qu'elle affiche. Sur le principe, on veut mettre un label sans texte, mais comme on veut d'une part le coloré et, d'autre part, récupérer les évènements qui peuvent survenir dans cette zone, il est nécessaire d'avoir un `Gtk::EventBox` pour cela.

La classe `ColorViewer` va ainsi hériter d'un `Gtk::EventBox` et aura deux membres :

- `_color` : la couleur,
- `_label` : le label que contiendra l'`EventBox` (conteneur simple).

Le constructeur prendra comme argument un `Gdk::Color`. il devra :

- attribuer la couleur `_color` à ce Widget en utilisant la fonction `modify_bg`,
- définir la taille du label à 20x20 en utilisant la fonction `set_size_request`,
- déclarer le captage des clics de boutons (droit et gauche) en utilisant la fonction `set_event`

D Implémentation de la classe ColorSelector

La création d'un widget et des connexions se fait dans le constructeur, soit directement dans le constructeur soit au travers de fonctions du type `createWidget()` et `createConnexions()` qui sont appelées dans le(s) constructeur(s). Ici, à cause du tableau de `ColorViewer`, il est plus pratique de construire les widgets en même temps que les connexions. On fait donc tout une même fonction `createWidget()` appelée par le constructeur.

D.1 Création de la structure

Dans la fonction `createWidget()` (appelée par le constructeur), mettre en place les widgets dans le `ColorSelector` (un `Gtk::VBox`). Pour cela, vous aurez besoin :

- `Gtk::Table` : un conteneur multiple sous la forme d'une matrice (attention : utilisation différentes des boxes, *voir la documentation à son sujet*),
- `Gtk::ColorButton` : une classe de bouton dédiée au choix des couleurs (rien à faire de plus !!),
- 25 instances de `ColorViewer`, pour mettre en place ces widgets, il peut être intéressant de systématiser l'ajout des widgets quelque soit le nombre de colonne et de ligne,
- autres widgets usuels : `Gtk::Box`, `Gtk::Separator`, ...

Pour attribuer des couleurs initiales, vous pourrez utiliser `Gdk::Color::set_hsv(...)`. La teinte est alors données par un *double* entre 0 et 360, la saturation et luminance sont données par des *doubles* entre 0 et 1.

D.2 Connexion des évènements

Modifier le constructeur pour ajouter des connexions qui vont permettre de récupérer les évènements de clic sur les `ColorViewer` vers une fonction `set_color` (membre de la classe `ColorSelector`).

Le signal à utiliser sur le ColorViewer est `signal_button_press_event` (*cf.* documentation en ligne pour les contraintes de connexion).

Cette fonction doit modifier la couleur de l'un ou l'autre des `ColorButton` en fonction de la nature du clic (droit ou gauche). Pour distinguer les clics-droit des clics-gauche à partir d'un `GdkEventButton* event`, il faut regarder si le membre `button` vaut 3 ou non (si oui, c'est un clic droit !).

3 Utilisation de Glade pour la création de l'interface générale

Dans cette partie, nous allons nous aider de Glade pour construire l'interface générale de la fenêtre, mais nous allons réutilisé la classe `ColorSelector` qui répond déjà bien à notre besoin.

Glade est un outil interactif pour la création d'interfaces graphiques chargeables par les API de Gtk+. Un même projet d'interface graphique peut servir à la fois pour un projet en Java, en python ou en C++ (*i.e.* Gtkmm). Dans la suite, je reste bien évidemment sur ce dernier binding.

L'outil Glade aide le développeur à construire graphiquement son interface. Le projet Glade est ensuite enregistré sous la forme d'un document XML qui décrit l'interface. Ce fichier XML peut ensuite être chargé dynamiquement par une application grâce à la classe `Gtk::Builder`.

Dans l'absolue, Glade est un outil qui permet de construire l'interface graphique **et** de définir *a priori* les connexions. Néanmoins, la seconde partie qui existe pour Gtk+ n'a pas été étendue à Gtkmm.

Historiquement, il existe deux formats de description des interfaces : les fichiers des projets LibGlade (correspondant à la version 2 de Glade) et les fichiers des projets GtkBuilder (correspondant à la version 3 de Glade). Dans les deux cas, l'extension est `.glade`, mais ils ne sont pas compatibles (deux formats XML mais avec des DTD légèrement différents). Il est préférable d'utiliser les fichiers générés par des projets GtkBuilder. Ces fichiers sont utilisables directement depuis l'interface `Gtk::Builder` alors que les fichiers générés par un projet LibGlade devront subir une transformation préalable à l'aide de l'outil `gtk-builder-convert` pour les transformer en fichier `.ui`.

A Création d'une interface graphique depuis Glade

A.1 Lancement et description de l'interface de Glade

1. Lancer Glade et créer un nouveau projet GtkBuilder (ne changer rien aux autres options …). Pour revenir à cette boîte de dialogue, il faut aller dans le menu **`Edition>Préférences.`**

Vous voilà maintenant dans l'interface de Glade. Dans la suite, je décris brièvement cette interface et je donne quelques informations sur son utilisation.

La fenêtre de Glade est structurée en trois parties (configuration par défaut qui peut être modifiée) :

- le panneau de gauche liste les widgets qui peuvent être utilisés dans une IHM,
- le panneau central est la zone de création d'un widget,
- le panneau de droite contient l'inspecteur qui décrit l'organisation hiérarchique des widgets et la fenêtre des propriétés pour éditer les propriétés du widget sélectionné.

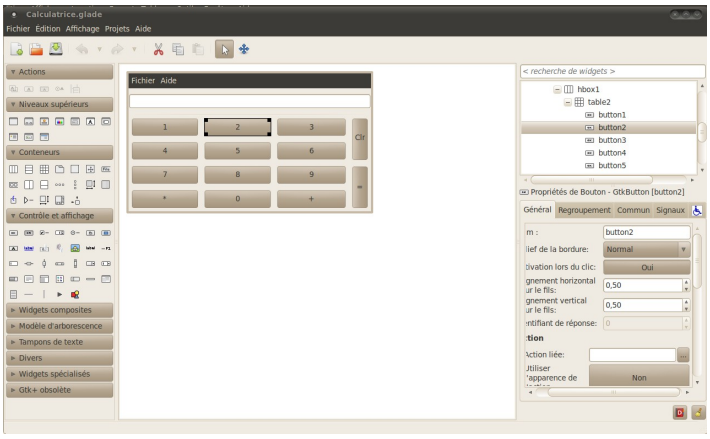
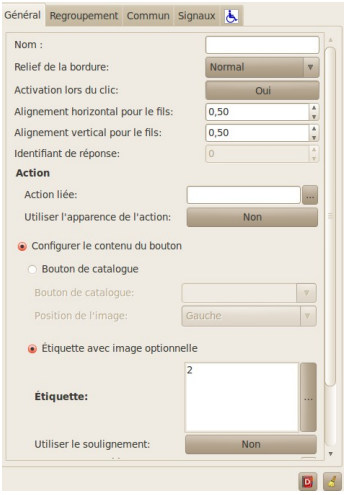
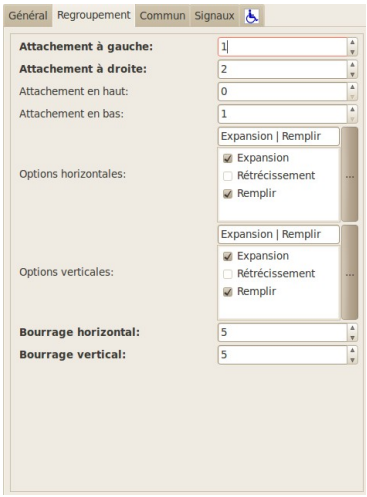
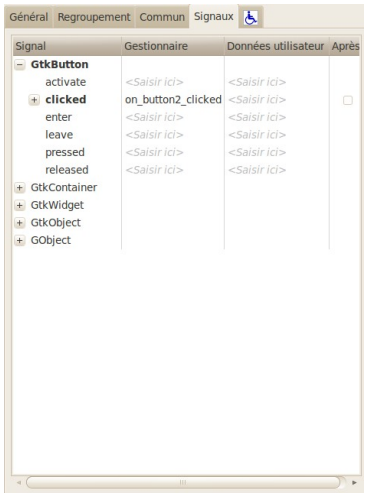


Illustration 1: Interface de Glade

Pour sélectionner un widget (afin d'en définir les propriétés), vous pouvez cliquer dessus dans la zone centrale **ou** dans la hiérarchie de widgets. L'utilisation de la hiérarchie est souvent nécessaire (et plus efficace) que la sélection depuis la visualisation de l'interface. Certains widgets comme les Boxes ou bien les `EventBox` sont très difficiles à sélectionner par la vue graphique (car sans ou quasiment-sans surface).







La fenêtre de propriétés comporte trois onglets principaux illustrées ci-dessus.

L'onglet **`Général`** décrit les propriétés générales du widget, en particulier son nom et les étiquettes (affichage des boutons et des labels). La liste des options générales dépend du type de widget. Pour les widgets de type Box, on y retrouvera les propriétés liées à l'homogénéité et au *padding* (traduit en Français par « Bourrage »).

L'onglet **`Regroupement`** décrit les propriétés de *packing*. Dans la traduction française, `EXPAND`

correspond à développer (si ce n'est pas développé, c'est `SHRINK`) et remplir est vrai lorsque vous faite de l'`EXPAND_WIDGET` (sinon c'est de l'`EXPAND_PADDING`).

L'onglet **`Commun`** décrit les propriétés générales partagées par l'ensemble des widgets. On y retrouve, par exemple, les infobulles ou l'activation des signaux pour les widgets le nécessitant.

L'onglet **`Signal`** permet de pré-déclarer des connexions entre des évènements boutons et des fonctions. Ces fonctions devront être créées par l'utilisateur, mais dans le cas de Gtkmm (et pour un premier et seul TP sur Glade), il vaut mieux faire ses connexions proprement « à la main ».

Une petite précision sur la sémantique des boutons de propriétés : lorsque le choix d'une valeur pour une propriété utilise un bouton, ce qui est affiché par le bouton correspond à la valeur actuelle de la propriété et non la valeur que prendra la propriété si vous cliquez sur le bouton. Ceci n'est pas nécessairement très intuitif (!).

Notez que pour plus de lisibilité, il est possible d'éditer les propriétés d'un widget dans une fenêtre séparée en utilisant un clic-droit sur le widget (dans la hiérarchie ou sur la visualisation) et en sélectionnant le menu **`Editer séparement`**.

A.2 Création de l'interface initiale de l'application de dessin

2. Commencer par créer une fenêtre vide (disponible dans la liste des widget « Niveaux supérieurs »).
3. En utilisant les différents widget de Glade, créer l'interface suivante :

   - la partie gauche contient le bouton (nommé `button1`)
   - la partie centrale contient un drawing area (nommé `drawingarea1`)
   - la partie droite contient simplement un `VBox` (nommé `colorselectorbox`) : ce widget de la même classe que la classe dont hérite la classe `ColorSelector` devra contenir plus tard une instance de cette dernière classe (créé par vos soins plus haut). Il n'est donc pas nécessaire de la remplir plus.

4. Enregistrer le projet (sous le nom `ProjetOCI.glade` par exemple). Il est alors intéressant de consulter le fichier généré dans un éditeur de texte.

A.3 Intégration d'un fichier Glade à une interface

Nous allons maintenant intégrer la fenêtre construite avec Glade dans le code C++. Pour cela, il faut passer par un objet `Gtk::Builder` capable d'interpréter les fichiers `.glade`. Le code ci-dessous illustre le chargement de notre fichier Glade.

```
Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
try {
    refBuilder->add_from_file("ProjetOCI.glade");
} catch(const Glib::FileError& ex) {
    std::cerr << "File Error: " << ex.what() << std::endl;
    return EXIT_FAILURE;
} catch(const Gtk::BuilderError& ex) {
    std::cerr << "GtkBuilder Error: " << ex.what() << std::endl;
    return EXIT_FAILURE;
}
```

Il y a deux solutions pour charger des fenêtres définies dans un fichier Glade :

- `get_widget` : le chargement d'un widget standard composé dans Glade,
- `get_widget_derived` : le chargement d'un widget standard qui sera « peuplé » ou enrichi par un objet d'une classe qui hérite du même widget. Dans un tel cas, la classe doit fournir un constructeur avec des paramètres précis (*cf.* ci-dessous).

1. Commencez par construire une classe `BuilderWindow` très similaire à la classe `ManualWindow`, plus précisément,
   - nous aurons besoin de pointeurs sur le bouton (pour le connecter), sur la zone de dessin (pour la modifier dans la fonction à connecter au bouton) et finalement un pointeur sur un `ColorSelector`,
   - recopier la fonction de dessin (identique),
   - le constructeur doit avoir l'allure suivante pour être utilisée par la fonction `get_widget_derived` :

```
BuilderWindow::BuilderWindow( BaseObjectType* cobject, const Glib::RefPtr<Gtk::Builder>& builder) :
Gtk::Window(cobject)
{
}
```

2. Dans le constructeur, il faut faire le lien entre les noms de widget du fichier Glade et les les pointeurs définis dans la classe. Deux cas se présentent alors : le bouton et la zone de dessin sont des widgets « purs », on peut donc utiliser la fonction `get_widget` et pour le `ColorSelector`, il est nécessaire d'utiliser `get_widget_derived`. On a ainsi les instructions suivantes :

```
builder->get_widget("drawingarea1", _drawing);
builder->get_widget("button1", _button);
builder->get_widget_derived("colorselectorbox", _colorselector);
```

Ensuite, compléter le constructeur par l'ajout de la connexion du bouton.

3. Pour utiliser `get_widget_derived`. pour un `ColorSelector`, il faut lui ajouter un nouveau constructeur avec un profil similaire au constructeur d'un `BuilderWindow` (mêmes paramètres imposés par la fonction `get_widget_derived`).
4. Modifier le `main` pour charger le fichier Glade et peupler une instance de `BuilderWindow` (classe héritée d'un `Gtk::Window`)

Finalement, lorsque votre travail fonctionne, essayer de modifier l'interface avec Glade (en déplaçant les widgets ou en en ajoutant de nouveaux, mais en laissant les trois widget principaux pour l'application). Et relancer votre application (sans compiler) … constatez, admirez, … je ne sais plus comment dire tellement nous sommes ébahis !