Une démonstration des manipulations de base de l'environnement et de quelques classes de base sera faite et projetée dans la salle de TP. Soyez attentif !

Dans QtCreator, fermer tous les projets et éditeurs, puis, recréez un nouveau projet de type « **Application Qt avec Widgets** ».

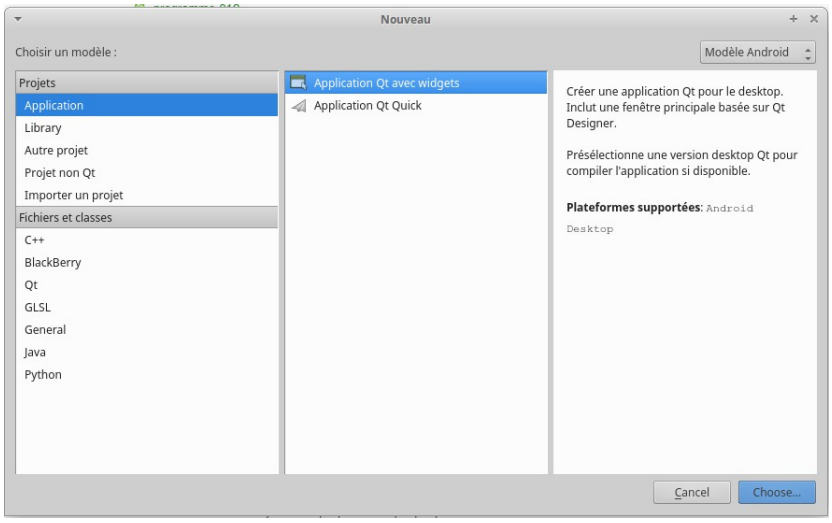


Figure 1 - Nouveau projet graphique basé sur un Widget

- Donner un nom et un emplacement au projet comme précédemment.
- Sélectionner le kit « Desktop » comme précédemment.
- Choisissez la classe représentant la fenêtre de l'application en **QWidget**.
  - **QWidget** → fenêtre vide toute simple
  - **QMainWindow** → fenêtre avec menu principale
  - **QDialog** → fenêtre avec boutons « ok » et « annuler »

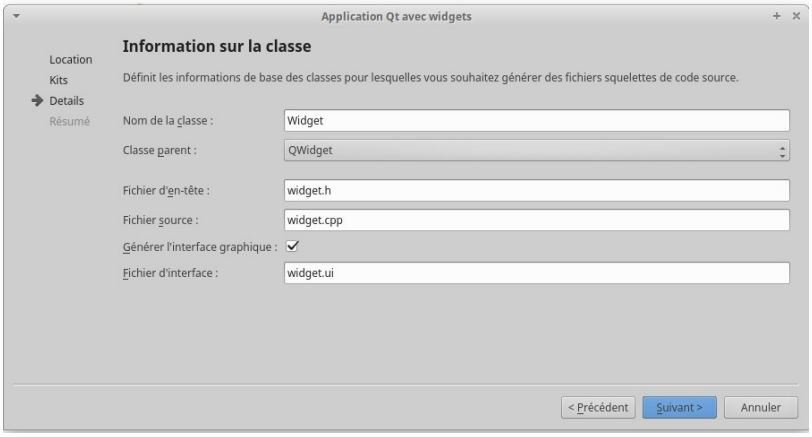


Figure 2 - Choix de la classe mère qui représentera la fenêtre de l'application.

- Terminer la création du projet, avec les paramètres par défaut

Vous noterez la création de plusieurs fichiers dans le répertoire du projet :

- le fichier de projet (**.pro**)
- le programme principal **main.cpp**
- la déclaration de la classe représentant la fenêtre de l'application, **widget.h**

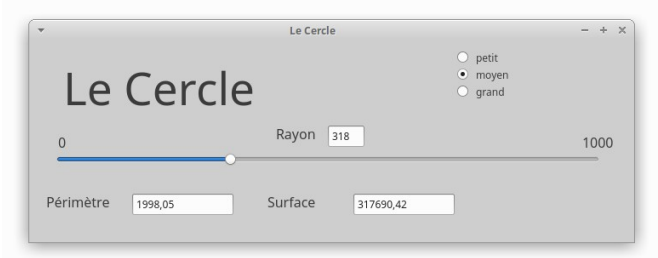
- l'implémentation des fonctions de la classe représentant la fenêtre de l'application, **widget.cpp**
- un fichier contenant le ***design*** de l'interface (*User Interface*), **widget.ui**

Travail demandé[1]

- Application graphique qui calcule le périmètre et la surface d'un cercle à partir de son rayon entier compris entre 0 et 1000.

- autre variante mettant en œuvre un Slider. La valeur du rayon ne peut pas être modifiée autrement que par le slider

- autre variante mettant en œuvre des RadioButton quand on agit sur le slider, un RadioButton se coche automatiquement (petit : r<300, moyen : 300<r<600, grand sinon)



- autre variante mettant en œuvre un MessageBox "êtes vous sûr ?" quand on quitte l'application. Pour cela, il faut ajouter un slot void *closeEvent*(QCloseEvent* event);

et y mettre le code :

```
void Widget::closeEvent(QCloseEvent* event){
    int reponse = QMessageBox::question(this, "Vérification","Etes vous sûr ?");
    if (reponse==QMessageBox::Yes) event->accept();
    else event->ignore();
}
```

- autre variante en remplaçant le *slider* par un bouton rotatif (QDial).

1 les interfaces doivent être **exactement identiques** à celles demandées

Annexes

Commencer avec les classes Qt les plus utilisées

Nous allons présenter ici les quelques composants visuels de Qt qui sont **incontournables** afin de pouvoir rapidement démarrer. Il vous faudra ensuite aller lire la documentation, puis lire la documentation et enfin lire la documentation, pour enrichir vos applications.

Quand vous créez une application graphique Qt, basée sur un QWidget (ou sur un QMainWindow) un objet nommé ***ui***, attribut de votre classe principale, est créé.

Cet objet ***ui*** représente l'interface utilisateur (*user interface* ou IHM, interface homme machine) de l'application. On peut directement décrire (*designer*) graphiquement cette IHM avec l'outil intégré à QtCreator, **Qt Designer**, en ouvrant le fichier avec l'extension **.ui** du projet et en faisant glisser les composants (à gauche de la fenêtre) sur l'interface.

Chaque composant ainsi positionné sera représenté dans le programme par un objet, attribut de ***ui***, attribut lui même de votre classe principale. Cet objet, comme tous les objets C++, comporte des attributs qui représentent ses caractéristiques (nom, taille, couleur, police, *.etc*.) mais également des méthodes qui représentent son comportement. Il est possible d'initialiser certains attributs (taille, police, couleur, *.etc*.) directement dans **QtDesigner** (en bas à doite de la fenêtre). Cela peut également se faire par programme au début de l'application (dans le constructeur de votre classe principale par exemple).

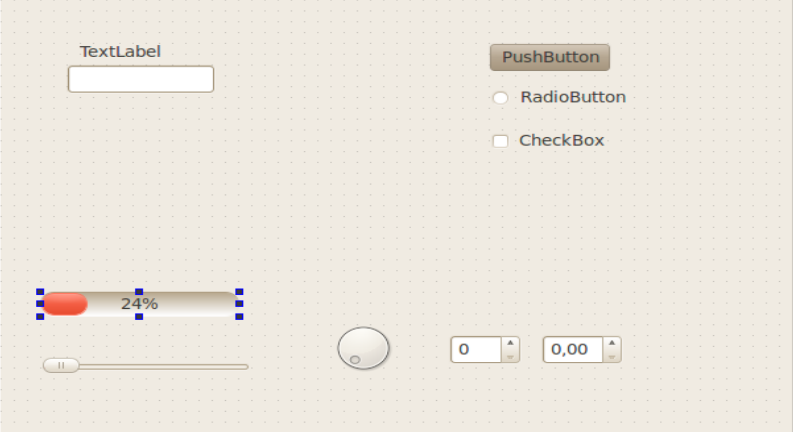


Figure 3 - exemple de *design* d'interface avec QtDesigner.

Par le seul fait de faire glisser les composants sur l'interface, vous allez créé des objets (instances) des classes Qt correspondantes à ces composants : (Les déclarations sont dans un fichier généré automatiquement **ui_widget.h**, qu'on a pas à toucher *a priori*).

Le tableau ci-dessous montre les objets correspondant à l'exemple de la figure 3 ci-dessus. Des objets seront instanciés automatiquement (dynamique) par votre programme au démarrage.

Déclaration	Signification	Type
`Ui::Widget *ui;`	L'interface utilisateur complète, membre de notre application (Widget ou MainWindow)	ptr
`QLabel *label;`	Un texte fixe, étiquette, membre de l'objet **`*ui`** ci-	ptr

Déclaration	Signification	Type
	dessus	
`QLineEdit *lineEdit;`	Une ligne de texte modifiable, membre de l'objet **`*ui`** ci-dessus	ptr
`QPushButton *pushButton;`	Un bouton, membre de l'objet **`*ui`** ci-dessus	ptr
`QRadioButton *radioButton;`	Un bouton radio (que l'on enfonce), membre de l'objet **`*ui`** ci-dessus	ptr
`QCheckBox *checkBox;`	Un bouton à cocher (que l'on coche), membre de l'objet **`*ui`** ci-dessus	ptr
`QProgressBar *progressBar;`	Pour afficher un progression (modifiée par le programme), membre de l'objet **`*ui`** ci-dessus	ptr
`QSlider *horizontalSlider;`	Un composant de réglage d'une valeur entière (que l'on ajuste), membre de l'objet **`*ui`** ci-dessus	ptr
`QDial *dial;`	Un bouton rotatif, membre de l'objet **`*ui`** ci-dessus	ptr
`QSpinBox *spinBox;`	Une boite contenant une valeur entière réglable, membre de l'objet **`*ui`** ci-dessus	ptr
`QDoubleSpinBox *doubleSpinBox;`	Une boite contenant une valeur réelle réglable, membre de l'objet **`*ui`** ci-dessus	ptr

Les classes Qt qui représentent ces composants ayant toutes des ancêtres communs (au sens de l'héritage), vous remarquerez que l'on retrouve souvent chez elles les mêmes méthodes. Les quelques lignes de programme suivantes représentent des actions courantes que l'on peut faire sur ces composants :

```
// quelques variables et objets
    int i;          // un entier
    QString chaine; // une chaine de caractère Qt
    double x;       // un double

    // le Qlabel qui représente un texte fixe
    ui->label->setText("entrez une valeur");   // modifier le texte du QLabel
    chaine = ui->label->text();                // récupérer le texte du QLabel
    ui->label->setVisible(false);              // rendre le QLabel invisible
    ui->label->setVisible(true);               // rendre le QLabel visible
    if (ui->label->isVisible()){               // est il visible?
                                }
    QRect g ;                                  // un objet rectangle
    g = ui->label->geometry() ;                // le rectangle g représente l'occupation du label
                                               // sur la fenêtre : position, largeur, hauteur
    // le QlineEdit une boite de saisie de texte
    ui->lineEdit->setText("ABC0");          // mettre ABC0 dans le texte du QLineEdit
    chaine = ui->lineEdit->text();          // récupérer le contenu du QlineEdit
                                            // (Attention c'est du Texte, QString!)
    bool ok;                                // une variable booléene
    x = chaine.toDouble(&ok);               // conversion chaine → double. Ok indique
                                            // si la conversion a été possible
    if (ok) {                               // la conversion est possible
            x++;                            // on incrémente x pour faire qq chose
            chaine.setNum(x);               // conversion double → chaine
            ui->lineEdit->setText(chaine);  // écriture du résultat (texte) sur l'IHM
```

```
        }
    else {  // la chaine ne contient pas un nombre → un message d'erreur critique[2]
         QMessageBox::critical(this, "Erreur de saisie",
                                  "la chaine de caractères à convertir n'est pas valide");
       }

//le QPushButton, un bouton à cliquer
ui->pushButton->setText("Démarrer");           // changer le texte du QPushButton
ui->pushButton->setVisible(false);             // le rend invisible
ui->pushButton->setVisible(true);              // le rend visible
ui->pushButton->setGeometry(100,20, 60, 20);   // modifier la position et
                                               // la taille du QpushButton

// le QradioButton, un seul enfoncé à la fois
if (ui->radioButton->isChecked() ) {                  // récupérer si le QRadioButton
                                                      // est enfoncé
                                    }
 ui->radioButton->setChecked(true);                   // le cocher par programme
 ui->radioButton->setChecked(false);                  // le décocher par programme

// le QcheckBox, possibilité d'en cocher plusieurs à la fois
ui->checkBox->setText("correction angle");            // modifier le texte du QCheckBox
if (ui->checkBox->isChecked()){                       // est il coché ?
                               }

// le QProgressBar et le QHorizontalSlider
ui->progressBar->setMaximum(340);                     // définir l'intervalle du QProgressBar
ui→progressBar→setMinimum(-10);                       //   idem
i = ui->progressBar->value();                         // récupérer sa valeur (position)
i=i * 1.10;                                           // 10% de plus
ui->progressBar->setValue(i);                         // modifier sa position

ui->progressBar->setValue(ui->horizontalSlider->value()); // la valeur du slider copiée dans
                                                          // le progresseBar

// le QdoubleSpinBox, une boite de saisie d'un nombre réel (double)
ui->doubleSpinBox->setMaximum(76.6);                  // définir l'intervalle du QDoubleSpinBox
ui->doubleSpinBox->setMinimum(1.5);                   // définir  ""    ""     ""       ""   ""
x = ui->doubleSpinBox->value();                       // récupérer sa valeur
ui->doubleSpinBox->setValue(x + 3.14159);             // modifier sa valeur

//le QspinBox, le même mais en entier (int)
ui->spinBox->setValue(10);                     // idem ci-dessus mais en entier
                                               // au lieu de double
// le Qdial, un bouton rotatif
ui->dial->setMinimum(0);                       // modifier l'intervalle du QDial
ui->dial->setMaximum(600);                     //   idem
i = ui->dial->value();                         // récupérer sa valeur
i++;                                           // la modifier dans les variables
ui->dial->setValue(i);                         // la remettre dans le composant
i = ui->dial->value();                         // récupérer sa valeur

// on revient au QLineEdit avec la valeur de i
chaine.sprintf("valeur: %d", i);               // formater une chaine de caractères
ui->lineEdit->setText(chaine);                 // écrire la chaine dans le QLineEdit
```

Les signaux et les slots

Le *framework* Qt, comme avec la plupart des *frameworks*, utilise un mode de programmation dit **« événementiel ».** En effet, votre programme n'aura pas à venir tester l'état d'un bouton dans une boucle sans fin pour savoir s'il est enfoncé ou non, comme on le ferait en *Arduino* par exemple.

2 Il existe plusieurs types de messageBox en fonction de leur usage (informer, signaler une erreur, *.etc.*) – consulter l'aide Qt sur la classe QMessageBox.

La classe Qt qui représente le bouton, **QPushButton**, est suffisamment « intelligente » pour s'occuper de ce test toute seule et envoyer un message, ***signal***, décrivant le clic. Pour comprendre ***intuitivement*** le processus de Signal, on pourrait dire que le bouton « *crie* » à Qt quelque chose comme : « on m'a cliqué, avec le bouton droit de la souris, à la position (x,y), en appuyant sur Maj, .*etc.*). On dit que le bouton envoie un signal à **Qt**.

Pour pouvoir faire une action lors de cet événement, il va falloir que l'on crée une méthode (***SLOT***), dans une classe Qt du programme (souvent dans la classe principale – **Widget** ou **MainWindow**), puis qu'on dise à **Qt** de connecter cette méthode, *slot*, au *signal* correspondant à notre bouton.

Dans **QtDesigner**, il est très facile de faire ce type de connexion en faisant un clic droit sur un composant puis « *aller au slot* », et en choisissant le type de *signal*[3]. Un nouveau *slot* est automatiquement créé dans la classe principale et connecté au *signal* choisi – déclaration dans la classe (fichier header .h) et implémentation dans le programme (fichier .cpp)[4].

Pour ceux qui iront plus loin avec Qt, il est également possible de faire/défaire cette connexion dans votre programme, avec les méthodes *connect/disconnect* communes à tous les objets Qt.

3 Chaque composant Qt peut envoyer des signaux différents en fonction de ce qu'il sait faire : « on m'a cliqué », « ma taille a changée », « on a modifié ma valeur », .etc.

4 **Attention** si vous avez créé un *slot* par erreur, il faut bien penser à l'enlever du fichier d'implémentation (.cpp), mais également du fichier *header* de déclaration de la classe (.h) – *Erreur courante du débutant qui fait perdre beaucoup de temps !*