ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

TP 3
Conception rapide d'une interface avec QtDesigner

Durée : 2 séances
Objectifs : Conception d'une application faisant intervenir du code généré par *QtDesigner*. Architecture document/vue. Dessin dans un widget personnalisé à l'aide d'un `QPainter`.

1 Présentation de QtDesigner

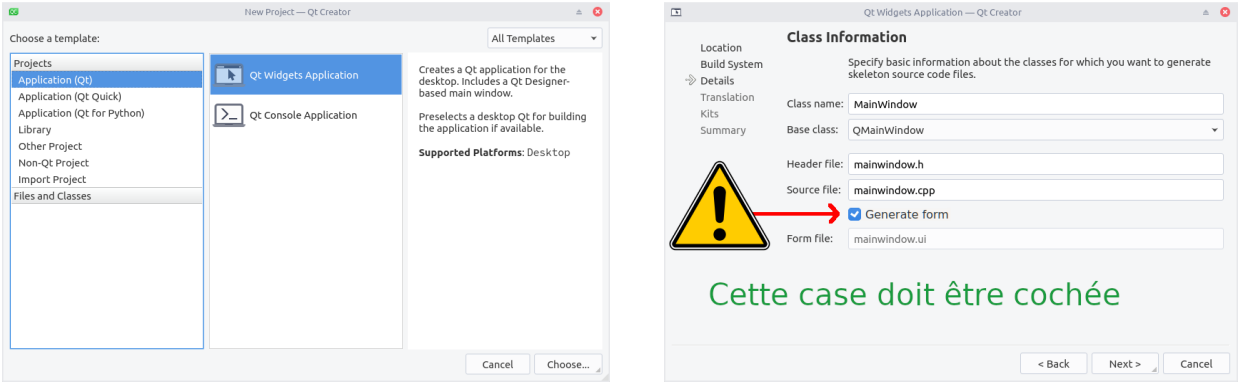
QtDesigner est un outil de développement visuel d'interfaces. C'est un programme autonome, mais qui peut aussi être exécuté directement au sein de l'IDE *QtCreator*. Il permet de réaliser rapidement des dialogues ou des fenêtres d'applications. Les interfaces sont sauvegardées dans des fichiers XML dont l'extension est « `.ui` ». Ces fichiers sont ensuite utilisés par le compilateur d'interfaces `uic` (*user interface compiler*) pour générer du code C++. La création de dialogues complexes, dont la construction ligne par ligne peut rapidement devenir laborieuse, est ainsi accélérée. Le fait que la conception soit visuelle offre une vision immédiate de l'interface pendant sa mise au point, sans qu'il soit nécessaire de compiler le programme pour vérifier la bonne organisation des widgets (un plus pour peaufiner l'ergonomie d'une interface).

L'ajout d'un dialogue à un projet est réalisé dans une ligne FORMS du fichier pro (*QtCreator* peut s'en charger, comme nous le verrons plus loin).

```
FORMS = mainwindow.ui
```

Le fichier `mainwindow.ui` ayant été créé à l'aide du programme *QtDesigner*. Bien sûr, tout cela ne se limite pas à la fenêtre principale. Des dialogues peuvent être construits de la même façon.

Dans le cadre de ce TP, vous pourrez encore plus simplement demander l'ajout et l'intégration de ce fichier lors de la création du projet (cf. figure 1).



(a) (b)

FIGURE 1 – Création d'un projet basé sur la classe `QMainWindow` et sur un formulaire *QtDesigner*.

Le programme `uic`, appelé par le Makefile généré par `qmake`, convertira le fichier `mainwindow.ui` en un

fichier `ui_mainwindow.h`. Dans ce fichier, une classe `Ui::MainWindow` est définie[1] qui contient comme attributs les widgets positionnés via *QtDesigner* et une méthode `setupUi()` qui crée et ordonne ces widgets. On peut dire, en résumé, que la classe `Ui::MainWindow` n'a d'autre utilité que de rassembler l'ensemble des pointeurs sur les widgets qui ont été placés via *QtDesigner*, et d'offrir une méthode `setupUi()` qui se charge de créer tous ces widgets, de les paramétrer et de les organiser à l'aide de gestionnaires de disposition.

L'utilisation du code produit par `uic` passe alors par la création d'une classe (p. ex. `MainWindow`) qui est associée par composition à la classe générée automatiquement (`Ui::MainWindow`). Les grandes lignes de ceci sont données dans les listings 1 et 2. Notez que cette seconde classe vous est aussi fournie par QtCreator lors de la création du projet. En résumé :

- La classe `Ui:MainWindow` est générée automatiquement à partir des modifications apportées au fichier `.ui` via *QtDesigner*.
- La classe `MainWindow` est la classe dans laquelle vous écrivez votre propre code.

Les noms des variables (pointeurs) correspondant aux widgets sont modifiables dans *QtDesigner* (cf. figure 2 (a)). Il est donc bien entendu possible de faire référence à ces widgets dans le code de la classe `MainWindow`. On pourra par exemple équiper la classe `MainWindow` de slots qui seront connectés à des signaux de ces widgets.

Notez enfin qu'il est possible avec *QtCreator* de créer, en une opération, un fichier de formulaire (`.ui`) ainsi que les fichiers sources minimaux (`.h` et `.cpp`) de la classe associée :

Clic droit sur le projet → « *Add New...* » → « *Qt* » → « *Qt Designer Form Class* ».

De même qu'il est possible de créer une classe C++ (sans formulaire, autrement dit sans fichier `.ui`) par :

Clic droit sur le projet → « *Add New...* » → « *C/C++* » → « *C++ Class* ».

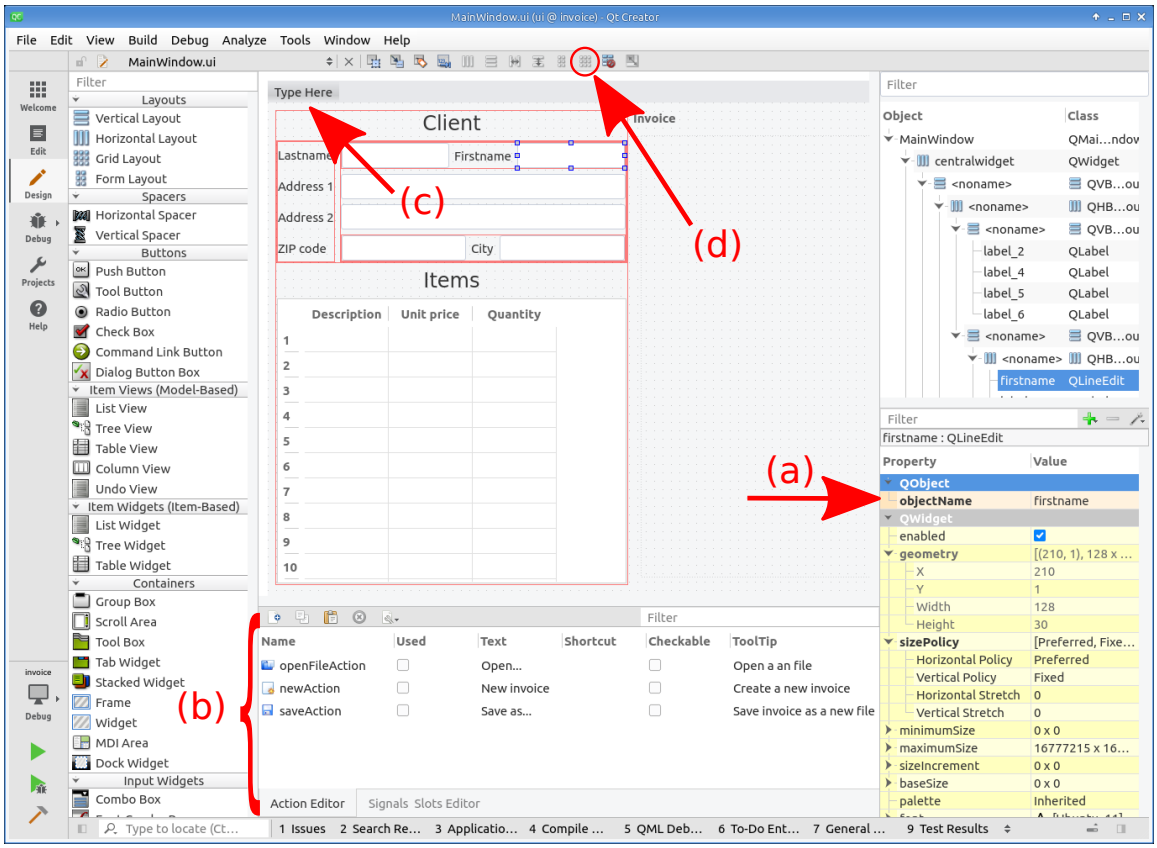


FIGURE 2 – Interface de QtDesigner (au sein de QtCreator).

1. Si MainWindow est le nom donné à la classe dans *QtDesigner*.

```
#ifnedf MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>

namespace Ui {
  class MainWindow;
}

class MainWindow : public QMainWindow {
  Q_OBJECT
 public:
  MainWindow(QWidget * parent=0);
  ~MainWindow();
  [...]                                  // éMthodes, signaux, slots
 private:
  Ui::MainWindow *ui;
};
#endif
```

Listing 1 – Fichier MainWindow.h

```
#include "MainWindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget * parent)
 : QMainWindow(parent),
   ui( new Ui::MainWindow )
{
  ui->setupUi(this);                     // èPremire instruction
  [...]
  connect( ui->pbClose, SIGNAL(clicked()),       // Exemple
           this, SLOT(close()) );                // d'utilisation
}

MainWindow::~MainWindow()
{
  delete ui;
}
```

Listing 2 – Fichier MainWindow.cpp

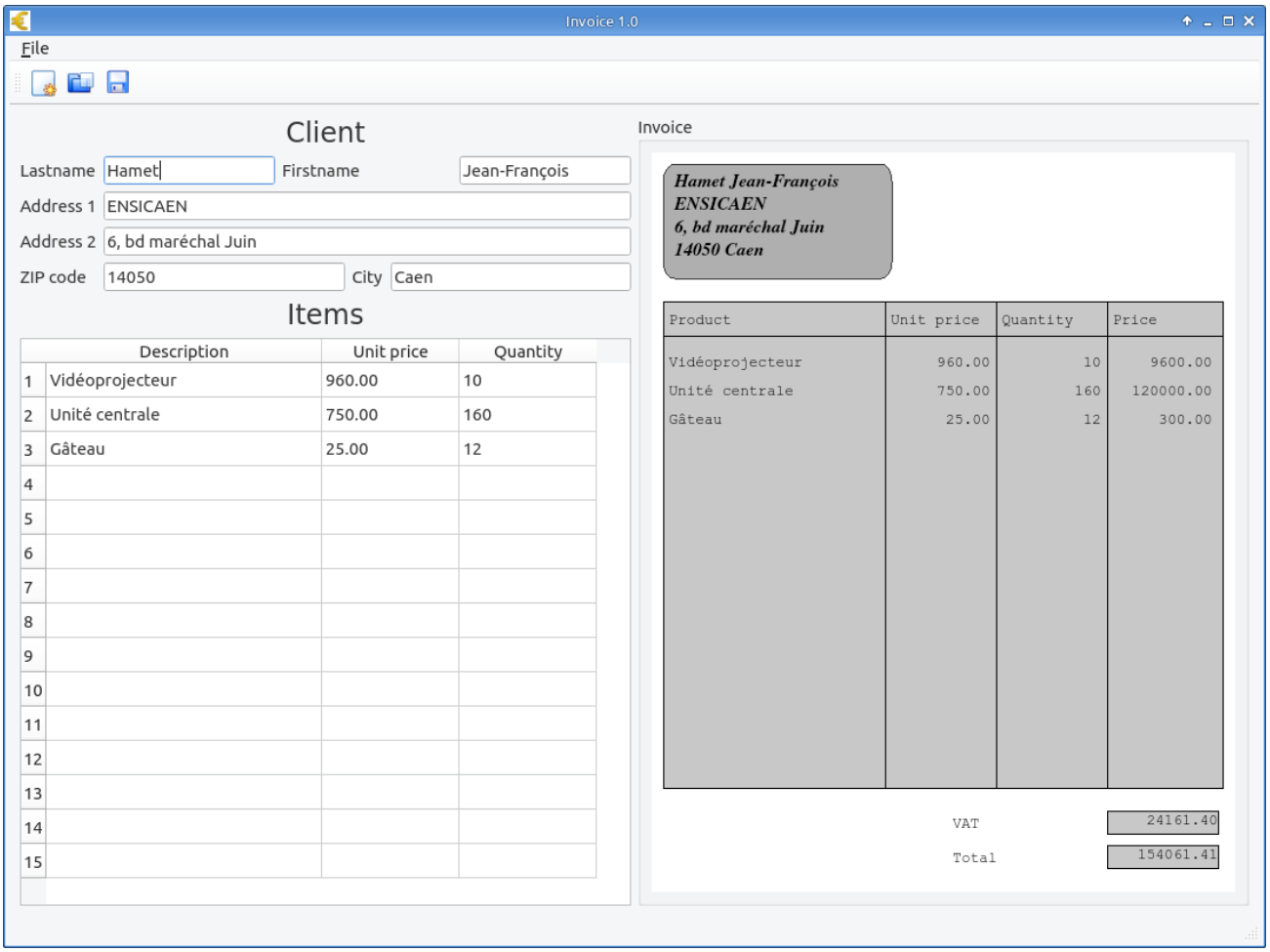


FIGURE 3 – Capture d'écran de la fenêtre principale à réaliser.

2 Travail à réaliser

A l'aide de *QtDesigner*, réalisez la fenêtre principale d'une application selon le modèle donné dans la figure 3 : un formulaire rudimentaire de saisie d'une facture. Pour commencer, vous pouvez créer dans *QtCreator* un nouveau projet « Applications (Qt) → Qt Widgets Application » (cf. figure 1). Comme proposé ensuite par l'assistant, votre classe sera basée sur la classe QMainWindow. Parcourez soigneusement le code créé pour vous par *QtCreator* pour en comprendre le fonctionnement. Après avoir lancé une première compilation, parcourez les deux fichiers `ui_mainwindow.h` et `ui_mainwindow.cpp` produits par `uic`. Vous pourrez alors apprécier cette partie du code dont vous n'avez pas à vous soucier autrement que de façon interactive et visuelle.

2.1 L'interface dans ses grandes lignes

Les données saisies dans les champs de gauche (figure 3) devront s'afficher aux endroits correspondants dans la partie droite qui est un aperçu du résultat de l'impression d'une facture. Cette interface est mono-document et utilise une architecture Document/Vue. Les données d'une facture constituent le document (ou modèle), la partie droite est une vue de ces données.

Une classe `InvoiceModel` pouvant servir de modèle de document vous est fournie sur Moodle. Cette classe se charge notamment des calculs de montants de chaque ligne ainsi que des totaux (hors taxes et TTC).

L'aperçu de la facture (figure 5) est un widget personnalisé (classe dérivée de `QWidget`) qui se comportera

comme une vue du modèle.

Les widgets de saisie (`QLineEdit`) permettent de modifier les données du modèle. Ils pourront être organisés à l'aide d'un `QGridLayout` (figure 2 (d)). Le tableau est un objet de la classe `QTableWidget` (à ne pas confondre avec `QTableView`, plus flexible mais trop long à mettre en œuvre pour notre usage dans le cadre de ce TP).

La collaboration entre tous les éléments est décrite dans la section 2.5.

2.2 Menu et actions

Les actions pourront être créées de différentes façons :

- Dans creator (en mode *QtDesigner*) via l'éditeur d'actions (figure 2 (b)).
- Dans *QtDesigner*, directement dans la barre de menu éditable (figure 2 (c)).
- Dans votre code source, comme pour le TP précédent.

Ces actions seront placées dans la barre d'outils et dans le menu « Fichier » (figure 4 ci-contre).

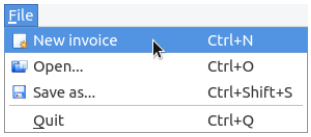


FIGURE 4 – Menu « Fichier ».

2.3 Icônes

Sous Linux, il est possible d'obtenir les icônes du thème courant de l'environement de bureau. Cela passe par l'utilisation de la méthode statique `QIcon::fromTheme()` qui repose sur un standard de dénomination de ces icônes.

Par exemple :

```
QIcon fileOpen = QIcon::fromTheme("document-open");
```

Si vous n'êtes pas sous Linux, quelques icônes vous sont fournies sur Moodle. Ces icônes pourront être intégrées à l'application sous forme de ressources (cf. support de cours) :

Clic droit sur le projet → « *Add New...* » → « *Qt* » → « *Qt Resource File* ».

2.4 Widget personnalisé de dessin de la facture

Pour afficher l'aperçu de la facture vous devez créer un widget personnalisé héritant de la classe `QWidget`. Dans cette classe, vous dessinerez à l'aide d'un objet `QPainter` lors des demandes de rafraîchissement (méthode virtuelle `paintEvent()`). Un rafraîchissement peut être provoqué grâce au slot `update()` de la classe `QWidget`.

Pour utiliser votre widget personnalisé avec *QtDesigner*, il suffit de placer un simple `QWidget` à l'endroit souhaité puis de le « promouvoir » (menu contextuel par clic droit sur le widget) pour que *QtDesigner* utilise votre propre classe dans le code qu'il produira.

Le widget personnalisé se comportera comme une *vue* du modèle de *document* (pattern Observateur et modèle

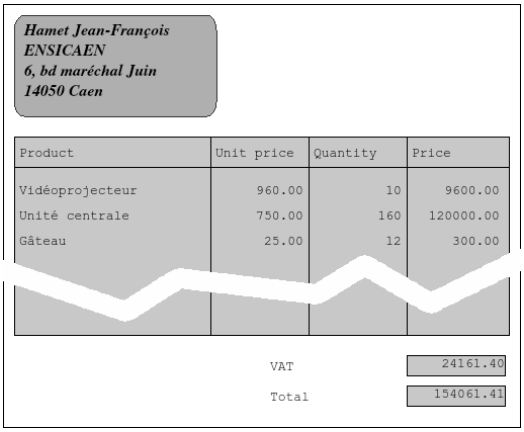


FIGURE 5 – Widget personnalisé de dessin d'une facture.

Document/Vue). À ce titre, ce widget doit être associé au modèle pour pouvoir accéder à ses données.

2.5 Collaboration entre les différentes classes

Le diagramme de collaboration de la figure 6 illustre le fonctionnement de l'application lors de la modification du champ « Prénom » par l'utilisateur. Autrement dit :

- L'utilisateur modifie un champ de la partie gauche.
- Le champ émet un signal (1.1) qui a été connecté à un slot du modèle.
- Le modèle est mis jour (1.2), il émet à son tour un signal de notification (2.1).
- Le signal `InvoiceModel::notify()` a été connecté au slot `update()` (2.2) du widget personnalisé (vue de droite). La méthode `QWidget::paintEvent()` est donc appelée.
- Le widget personnalisé, dans sa méthode `paintEvent()`, dessine l'aperçu à l'aide d'un objet `QPainter` en allant chercher les informations dans le modèle (3.1-3.3).

2.6 Internationalisation et localisation

En vous aidant du support de cours (Internationalisation), traduisez les chaînes de caractères de votre application dans une langue supplémentaire.

Pour tester le bon fonctionnement de votre traduction sans forcer l'utilisation d'une langue dans le code, vous pouvez (sous Linux) changer l'environnement de votre programme. Par exemple par la commande :

```
$ LANGUAGE=fr LC_ALL=fr_FR ./bin/invoice_editor
```

Adaptez à la langue souhaitée (p. ex. `en_US`).

2.7 Lecture / écriture (optionnel)

Afin de permettre la sauvegarde et le chargement d'une facture vers/depuis un fichier, et en vue de respecter le principe *Single Responsibility*, programmez deux classes : `InvoiceWriter` et `InvoiceReader`. Vous pourrez utiliser pour cela les classes `QFile` et `QDataStream` pour *sérialiser* très simplement des données (essentiellement des chaînes de caractères dans le cas présent). Équipez votre application des actions (si ce n'est pas déjà fait) et slots correspondants.

Notez que la classe `InvoiceModel` dispose de méthodes permettant de désactiver temporairement les notifications (signal `notify()`) pendant sa mise à jour lors de la lecture d'un fichier. Ceci permet d'éviter de multiples dessins complets de la vue à la modification de chacune des données.

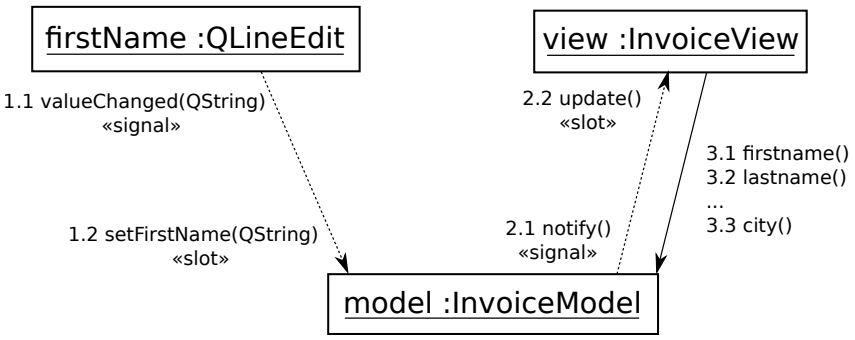


FIGURE 6 – Diagramme de collaboration (partiel) Document/Vue.