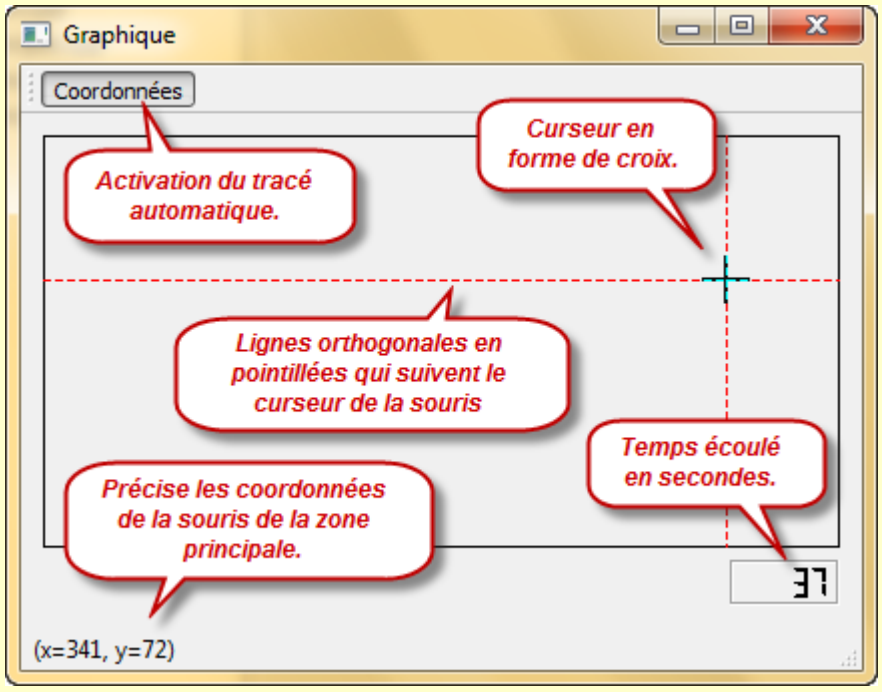
Ce projet va nous permettre de créer de nouveaux composants graphiques à partir de ceux existants. Nous en profiterons pour apprendre comment faire du tracé personnalisé, en temps réel, sur ces composants là.

Lorsque nous fabriquons de nouveaux composants, nous pouvons redéfinir quelques méthodes issues des classes héritées, notamment les méthodes de gestion événementielle.

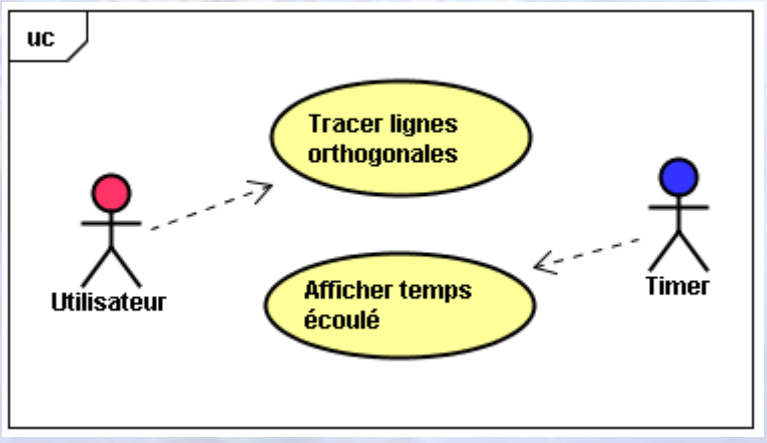
A propos des événements, nous verrons aussi comment créer un signal de toute pièce pour pouvoir interagir à distance avec un autre composant qui à priori n'est pas accessible.

Toutes ces nouveautés seront expliquées au travers d'un projet de tracé graphique :

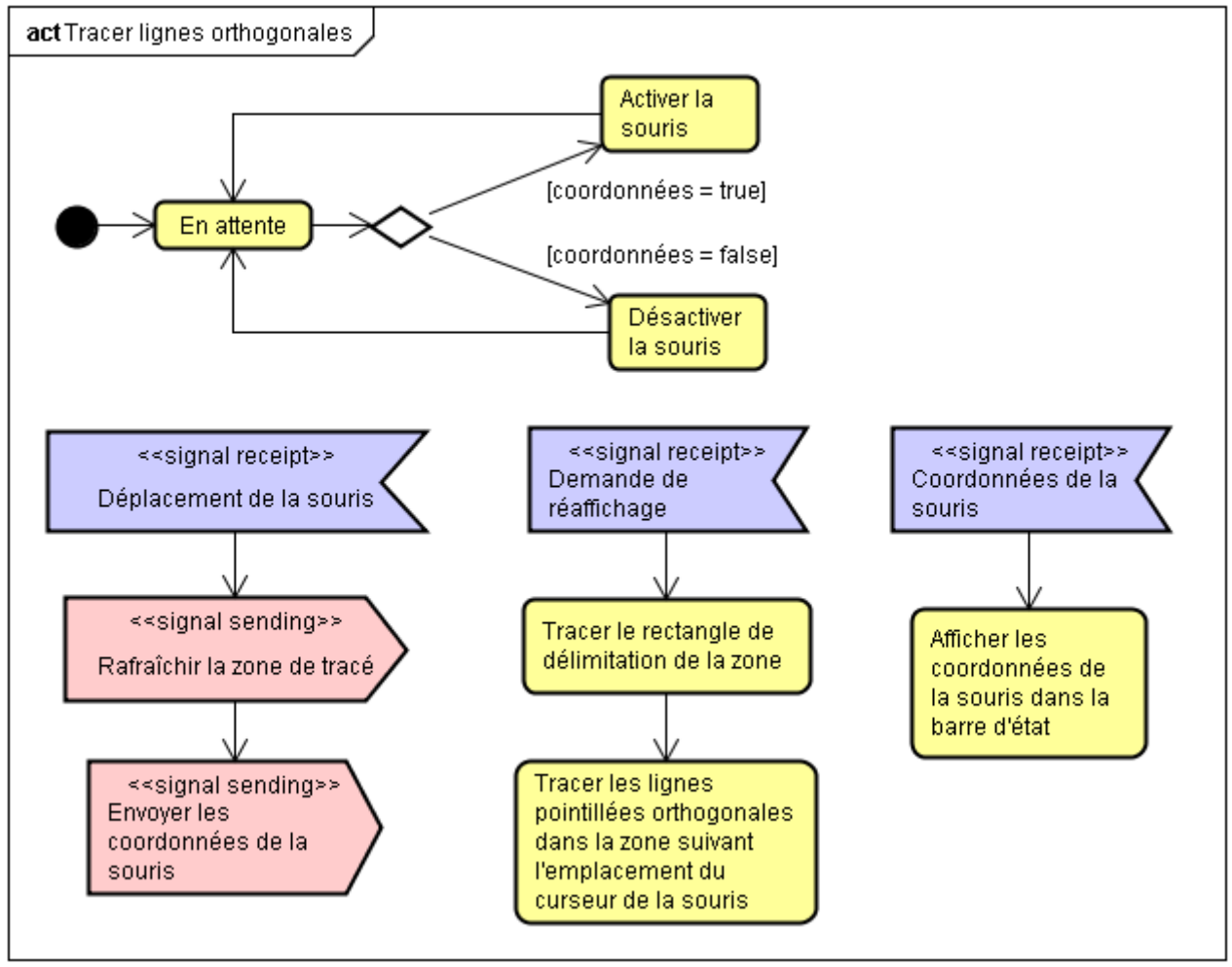
- *Les coordonnées de la souris sont récupérées après chaque déplacement et sont précisées dans la barre d'état.*
- *La partie principale trace deux lignes orthogonales en pointillées suivant le curseur de la souris.*
- *Le pointeur de la souris est d'ailleurs en forme de croix.*
- *Toutefois, le tracé ne s'effectue que si l'utilisateur le désire au moyen d'un bouton prévu à cet effet.*
- *Des raccourcis clavier sont rajoutés pour augmenter la convivialité et donc l'ergonomie de l'application.*
- *Pour finir, je place un **Timer** qui donne le temps écoulé en secondes depuis le démarrage du programme. La visualisation de ce Timer se fait au travers du composant **QLCDNumber**.*



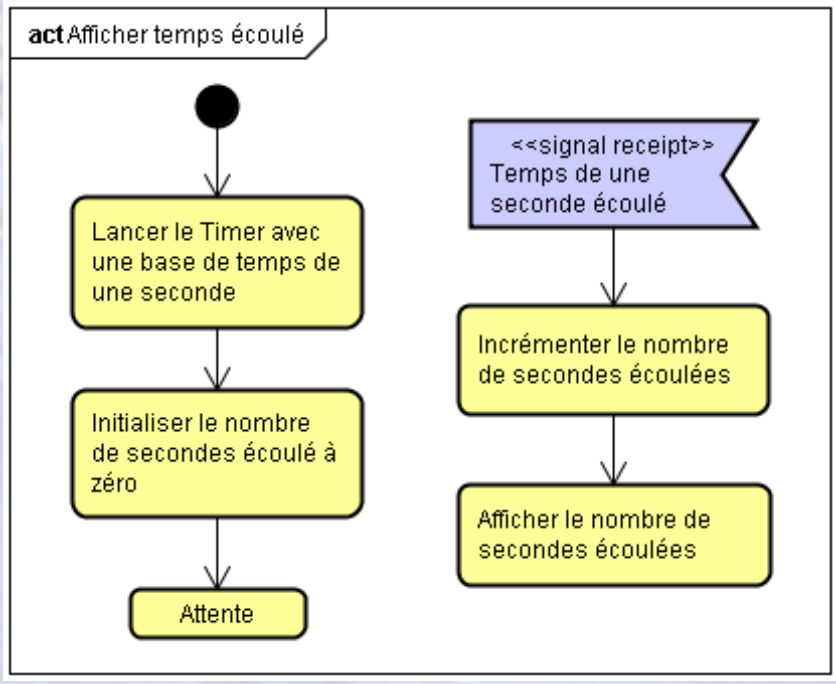
MODÉLISATION – DIAGRAMME DE CAS D'UTILISATION



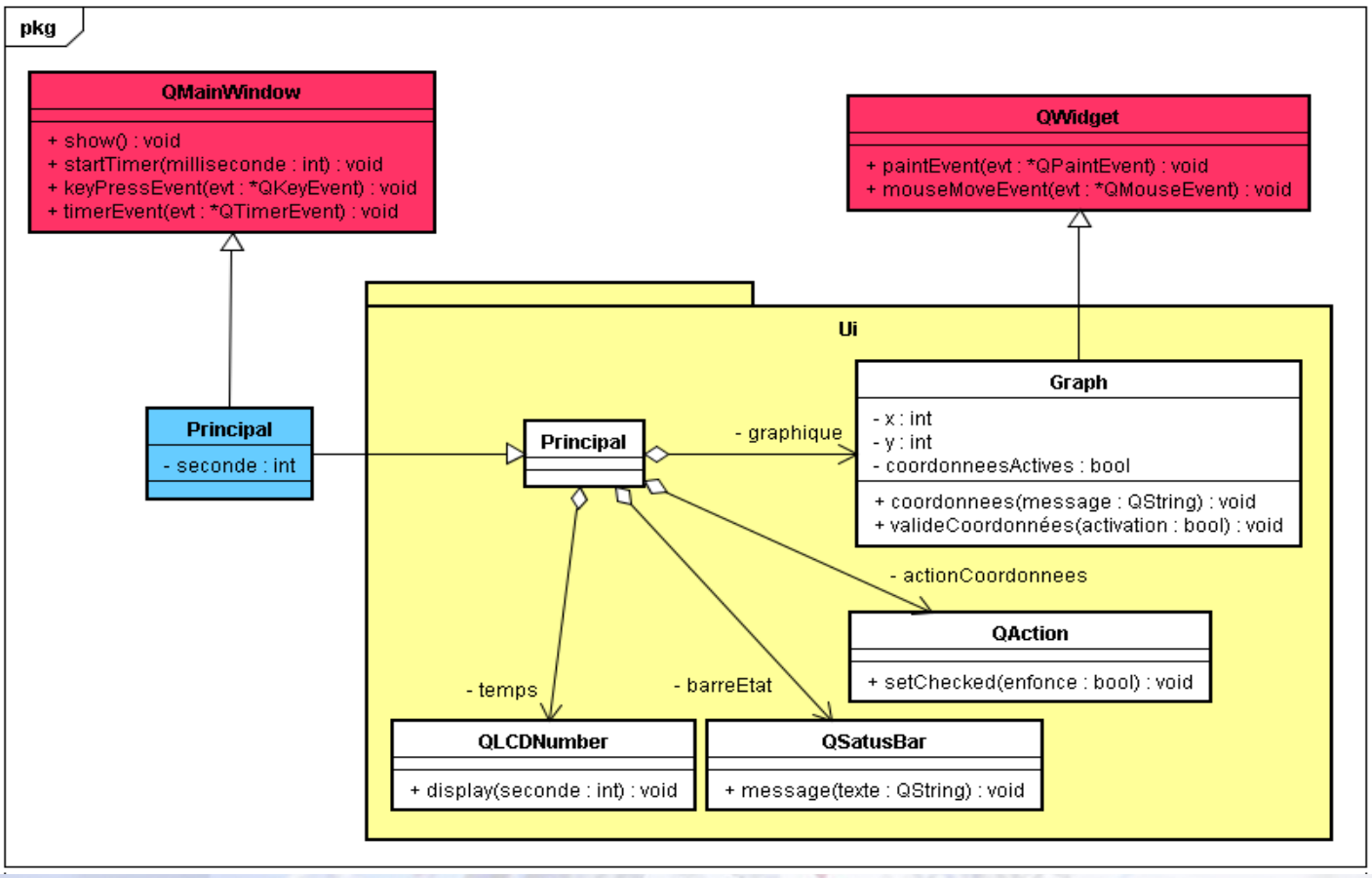
x MODÉLISATION – DIAGRAMME D'ACTIVITÉ DE TRACER LIGNES ORTHOGONALES



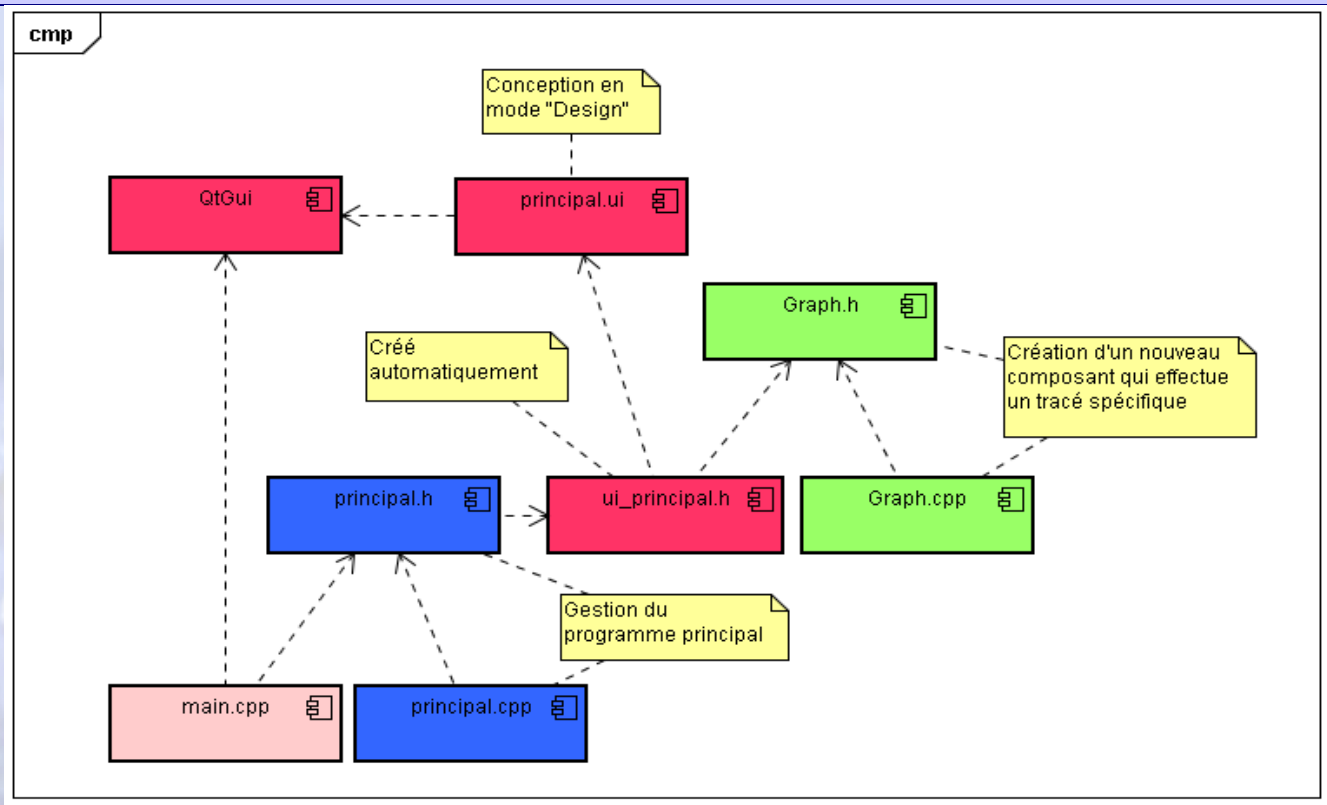
x MODÉLISATION – D'ACTIVITÉ AFFICHER TEMPS ÉCOULÉ



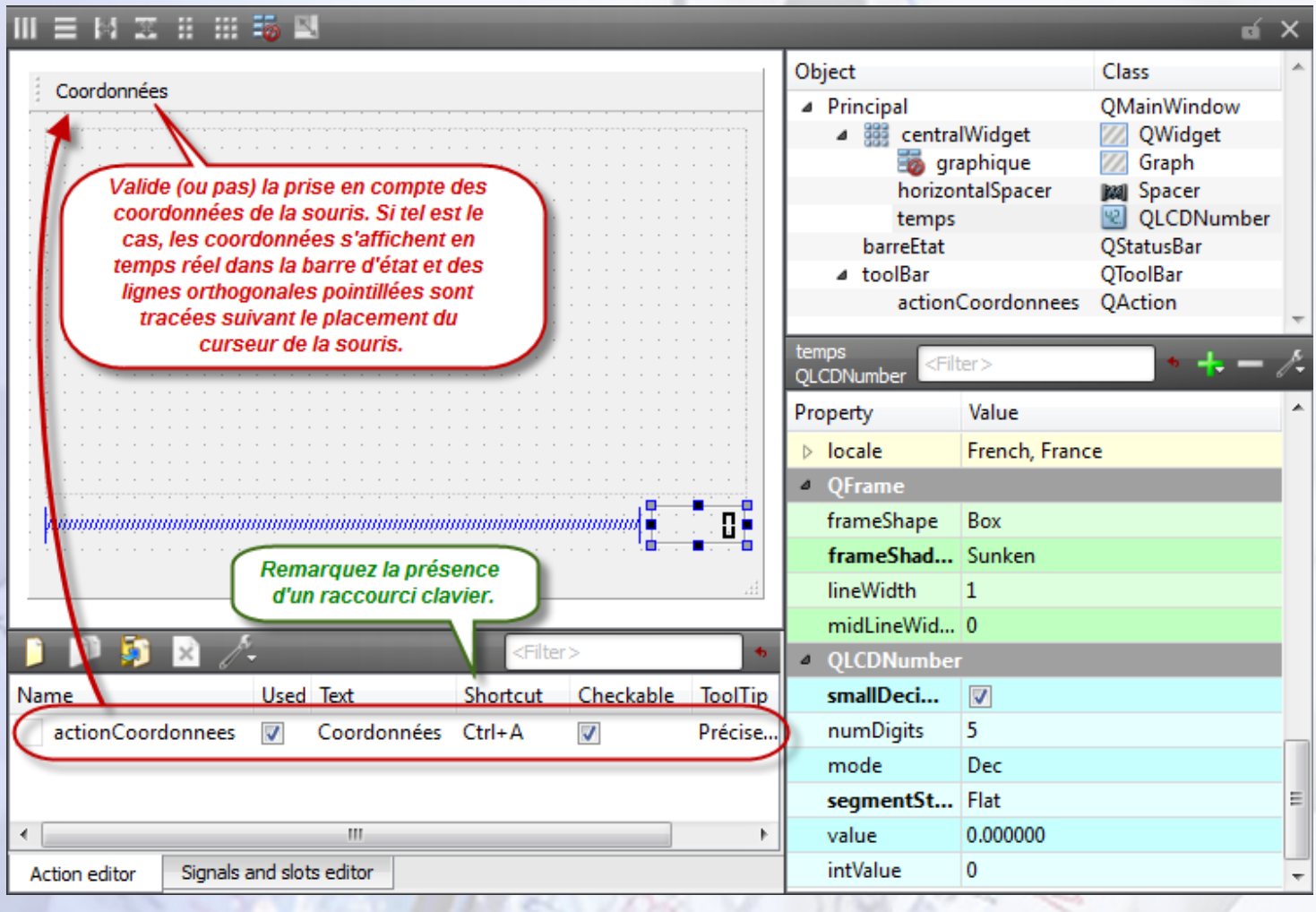
MODÉLISATION – DIAGRAMME DE CLASSES



MODÉLISATION – DIAGRAMME DE COMPOSANTS



PRINCIPAL.UI



PRINCIPAL.H

principal.h — <No Symbols>

```
1   #ifndef PRINCIPAL_H
2   #define PRINCIPAL_H
3
4   #include <QtGui/QMainWindow>
5   #include "ui_principal.h"
6
7   class Principal : public QMainWindow, public Ui::Principal
8   {
9     Q_OBJECT
10  public:
11    Principal(QWidget *parent = 0);
12  protected:
13    void keyPressEvent(QKeyEvent *evt);
14    void timerEvent(QTimerEvent *evt);
15  private:
16    int secondes;
17  };
18
19  #endif // PRINCIPAL_H
```

Redéfinition des méthodes de gestion événementielle.

Nombre de secondes écoulées depuis le début du programme.

- *Il est nécessaire de redéfinir le comportement propre aux méthodes de gestion événementielle qui prennent en compte, d'une part l'écoulement du temps et d'autre part le fait d'utiliser les touches du clavier pour activer ou pas le système de coordonnées (raccourcis clavier supplémentaires).*
- *Nous avons également besoin d'un attribut **secondes** qui sert à spécifier le nombre de secondes déjà écoulées depuis le lancement du programme.*

PRINCIPAL.CPP

Nous devons nous préoccuper de plusieurs comportement spécifiques en relation avec la déclaration proposée plus haut :

- ***Phase d'initialisation – Constructeur :** Il faut démarrer le **timer** au travers de la méthode **startTimer()**. Cette dernière prend en argument le nombre de millisecondes souhaité entre chaque impulsion d'horloge, laquelle va générer un événement de type **timeEvent()**. Nous profitons de cette phase pour mettre à zéro le nombre de secondes écoulées.*
- ***Evénement de type appui sur une touche du clavier – keyPressEvent() :** Dans cette méthode, nous allons contrôler si la touche **« Echappe »** est sollicité. Dans ce cas, l'action de gestion des coordonnées est désactivée. Nous contrôlons également l'appui sur la touche **« A »** qui réalise l'opération inverse, c'est-à-dire qui active la gestion des coordonnées.*
- ***Evénement de type timer – timerEvent() :** Dans cette méthode, nous incrémentons le nombre de secondes écoulées et nous l'affichons au travers du composant **QLCDNumber**.*

principal.cpp* | Principal::keyPressEvent(QKeyEvent *)

```
1   #include "principal.h"
2   #include <QStatusBar>
3   #include <QKeyEvent>
4   #include <QLCDNumber>
5
6   Principal::Principal(QWidget *parent) : QMainWindow(parent)
7   {
8     setupUi(this);
9     startTimer(1000);
10    secondes = 0;
11  }
12
13  void Principal::keyPressEvent(QKeyEvent *evt)
14  {
15    switch (evt->key())
16    {
17      case Qt::Key_Escape : actionCoordonnees->setChecked(false); break;
18      case Qt::Key_A      : actionCoordonnees->setChecked(true); break;
19    }
20  }
21
22  void Principal::timerEvent(QTimerEvent *evt)
23  {
24    temps->display(++secondes);
25  }
```

Lancement du timer avec un top d'horloge toutes les secondes.

Un événement est généré à chaque fois que l'uilisateur tape sur une touche du clavier.

Activation ou pas du système de coordonnées.

A chaque seconde un événement est envoyé.

*Tous les composants héritent de la classe ancêtre **QObject**. C'est la classe de base par excellence qui possède ainsi un certain nombre de méthodes qui vont être utiles pour tous les objets de notre interface. C'est cette classe qui gère la fonctionnalité du timer, ce qui sous-entend que chaque composant peut avoir son propre timer.*

*La classe **QObject** possède ainsi la méthode **startTimer()** qui crée et active son propre timer en spécifiant la cadence en millisecondes. Cette méthode renvoie une valeur entière qui correspond à l'identification du timer utilisé. Il est alors possible de supprimer ce timer à l'aide de la méthode **killTimer()**, en spécifiant en argument de la méthode, le numéro d'identification issu de la méthode **startTimer()**.*

GRAPH.H

Nous allons maintenant fabriquer un nouveau composant graphique que nous allons placer dans la zone centrale de la fenêtre principale de l'application.

*Un nouveau composant est tout simplement une classe. Si vous désirez qui soit visible, c'est-à-dire graphique, cette classe doit hériter de la classe **Qwidget,** ou bien l'une des classes qui héritent elles-même de la classe **QWidget**.*

*A ce sujet, la classe **QWidget** est la classe de base de tous les composants graphiques, comme **QPushButton, QLCDNumber, QTextEdit, QLabel,** etc.*

Graph.h | Graph | Line: 24, Col: 1

```
1   #ifndef GRAPH_H
2   #define GRAPH_H
3
4   #include <QWidget>
5
6   class Graph : public QWidget
7   {
8     Q_OBJECT
9   signals:
10    void coordonnees(const QString &message);
11  public:
12    Graph(QWidget *parent = 0);
13  protected:
14    void paintEvent(QPaintEvent *evt);
15    void mouseMoveEvent(QMouseEvent *evt);
16  private slots:
17    void valideCoordonnees(bool activation);
18  private:
19    int x, y;
20    bool coordonneesActives;
21  };
22
23  #endif //
24
```

Pour prendre en compte les signaux et les slots, vous devez intégrer cette MACRO.

Nouveau composant graphique.

Envoi d'un message qui va apparaître dans la barre d'état.

Le constructeur de la classe doit impérativement prendre un paramètre de type QWidget pour permettre l'imbrication des composants.

Redéfinition des événements issus de la classe QWidget pout réaliser un tracé spécifique et pour récupérer les coordonnées de la souris.

Mémorisations des différents événements.

Méthode de traitement qui est exécutée lorsque l'utilisateur clique sur le bouton "Coordonnées".

GRAPH.CPP

A chaque fois que vous désirez réaliser votre propre tracé, vous devez redéfinir la méthode ***paintEvent()***. Il est possible alors de provoquer un ré-affichage au moyen de la méthode ***update()***.

Un ré-affichage est automatiquement lancé pour chaque composant, lorsque la fenêtre se replace en premier plan sur le bureau, notamment lorsque notre application se trouvait au préalable dans la barre des tâches.

Graph.cpp — Graph::mouseMoveEvent(QMouseEvent *) — Line: 28, Col: 16

```
1  #include "Graph.h"
2  #include <QtGui>
3
4  Graph::Graph(QWidget *parent) : QWidget(parent)
5  {
6    x = y = 0;
7    coordonneesActives = false;
8  }
9
10 void Graph::paintEvent(QPaintEvent *evt)
11 {
12   QPainter dessin(this);
13   int largeur = rect().width()-1;
14   int hauteur = rect().height()-1;
15   dessin.drawRect(0, 0, largeur, hauteur);
16   if (coordonneesActives) {
17     QPen crayon(Qt::red);
18     crayon.setStyle(Qt::DashLine);
19     dessin.setPen(crayon);
20     dessin.drawLine(0, y, largeur, y);
21     dessin.drawLine(x, 0, x, hauteur);
22   }
23 }
24
25 void Graph::mouseMoveEvent(QMouseEvent *evt)
26 {
27   x = evt->x();
28   y = evt->y();
29   update();
30   coordonnees(QString("(x=%1, y=%2)").arg(x).arg(y));
31 }
32
33 void Graph::valideCoordonnees(bool activation)
34 {
35   setMouseTracking(coordonneesActives = activation);
36   coordonnees("");
37   update();
38 }
```

Précision du composant parent (imbrication) par héritage.

Tracé du composant personnalisé.

Récupération de la surface de travail (**contexte graphique**).

Récupération des dimensions actuelles du composant dans la fenêtre.

Tracé du contour du composant.

Tracé des lignes orthogonales pointillée suivant les coordonnées de la souris.

Traitement spécifique à chaque mouvement de la souris.

Enregistrer les coordonnées de la souris.

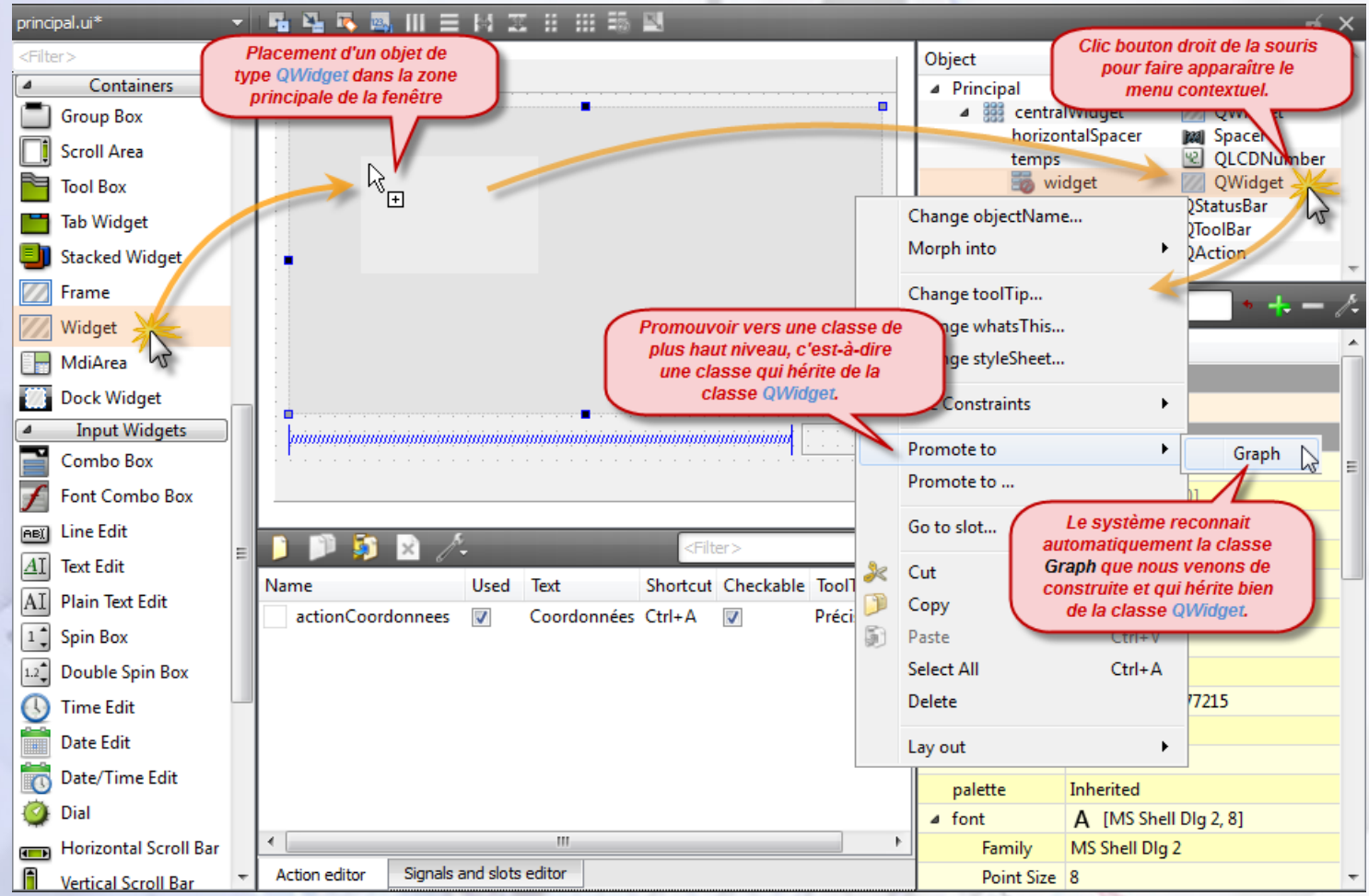
Provoquer le réaffichage complet du composant.

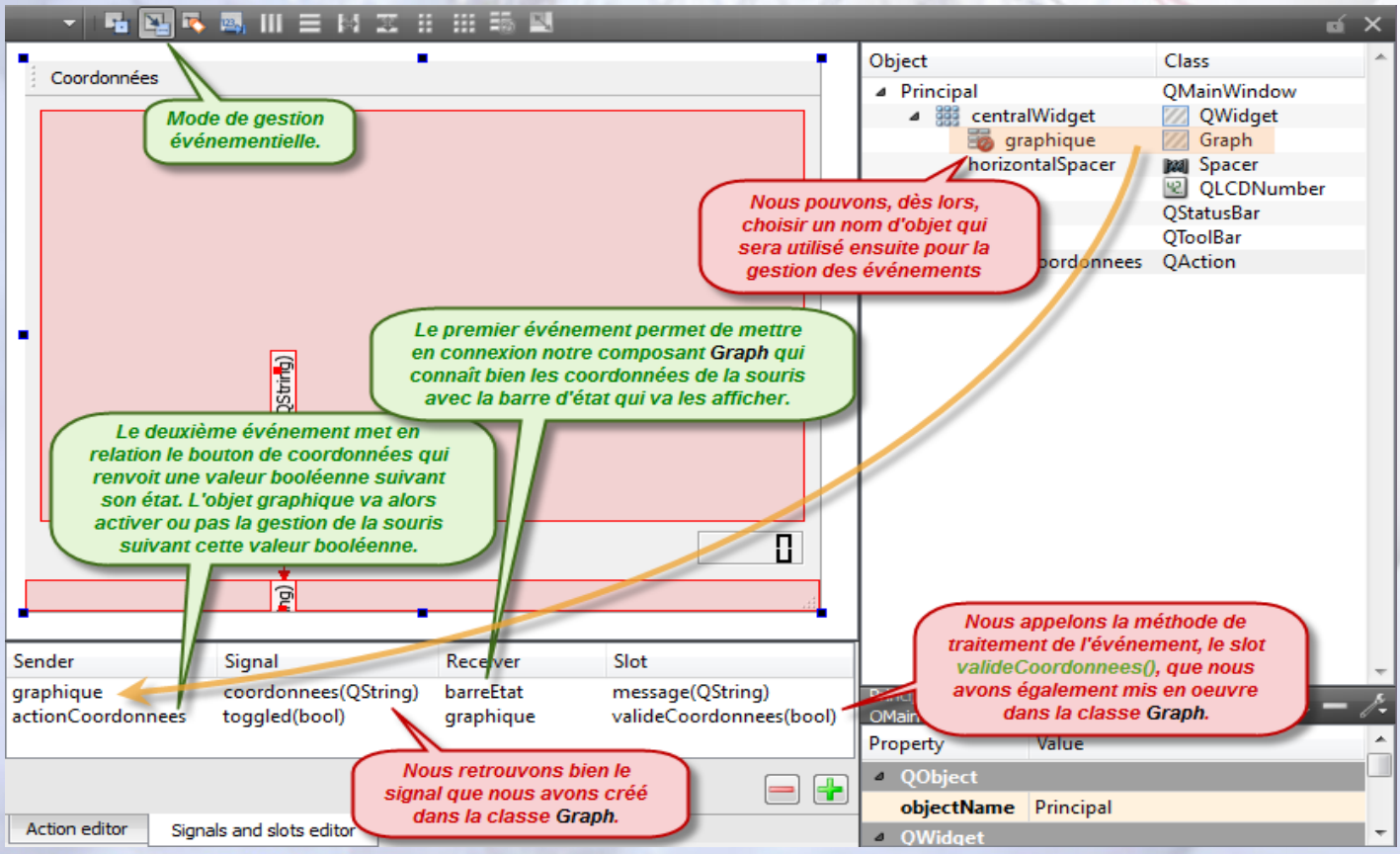
Création et envoi d'un message dont le destinataire sera la barre d'état.

Prendre en compte ou pas le déplacement de la souris.

*La méthode **mouseMoveEvent()** gère les événements « déplacement de souris ». Par défaut, ces événements sont déclenchés lorsque l'utilisateur enfonce un des boutons de la souris. Il est possible de changer ce comportement en appelant explicitement la méthode **setMouseTracking()**, ce que nous faisons dans notre code.*

x PLACEMENT DU COMPOSANT PERSONNALISÉ EN MODE « DESIGN »

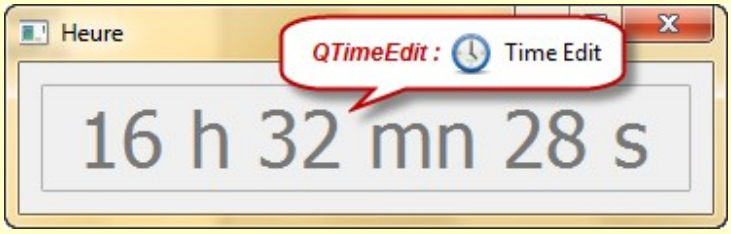




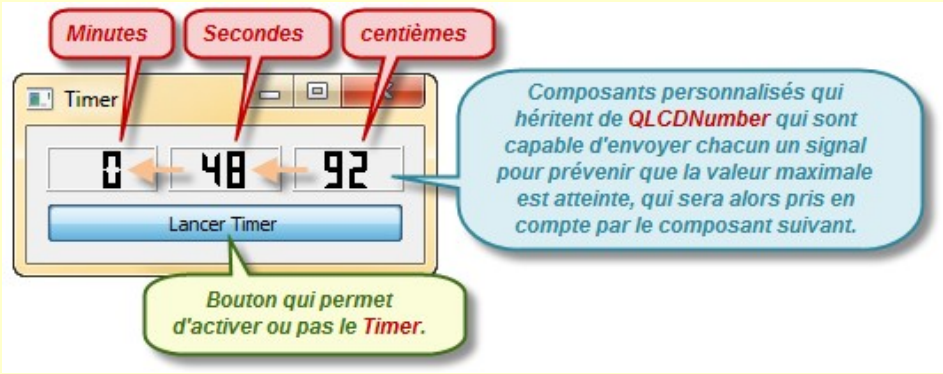
TRAVAUX PRATIQUES EN AUTONOMIE

Je vous propose maintenant de créer vos propres composants au travers de deux projets, d'une part pour revoir la notion de timer et d'autre part pour faire vos propres tracés graphiques.

*Toutefois, afin de bien maîtriser la notion de **Timer**, je vous propose de réaliser une horloge qui nous donne les heures, les minutes et les secondes, sans pour cela créer un objet personnalisé. Dans ce projet, vous allez utiliser les compétences du composant **QTimerEdit**. Tous les dixièmes de seconde, vous remettrez à jour l'affichage de l'heure actuelle. Pour connaître l'heure courante, faites appel à la méthode statique **QTime::currentTime()**.*



*Le deuxième projet consiste à visualiser un timer au centième de seconde près à l'aide de trois composants personnalisés qui héritent de la classe **QLCDNumber** et qui représentent respectivement, les minutes, les secondes et les centièmes de seconde. Pour cela, je vous donne la déclaration complète de la classe dans le fichier « Timer.h ». Il vous suffit alors de définir correctement l'ensemble des méthodes dans le fichier « Timer.cpp » correspondant et de régler précisément, en mode « Design », l'enchaînement des événements comme cela vous est présenté ci-dessous.*



timer.h — Timer

```
1  #ifndef TIMER_H
2  #define TIMER_H
3
4  #include <QLCDNumber>
5
6  class Timer : public QLCDNumber
7  {
8    Q_OBJECT
9    Q_PROPERTY(int baseDeTemps READ baseDeTemps WRITE setBaseDeTemps)
10
11 signals:
12   void tempsEcoule();
13 public:
14   Timer(QWidget *parent = 0);
15   int baseDeTemps();
16   void setBaseDeTemps(int base);
17 protected:
18   void timerEvent(QTimerEvent *);
19 private slots:
20   void lancer(bool etat);
21   void incrementer();
22 private:
23   int periode;
24   int idTimer;
25 };
26
27 #endif // TIMER_H
28
```

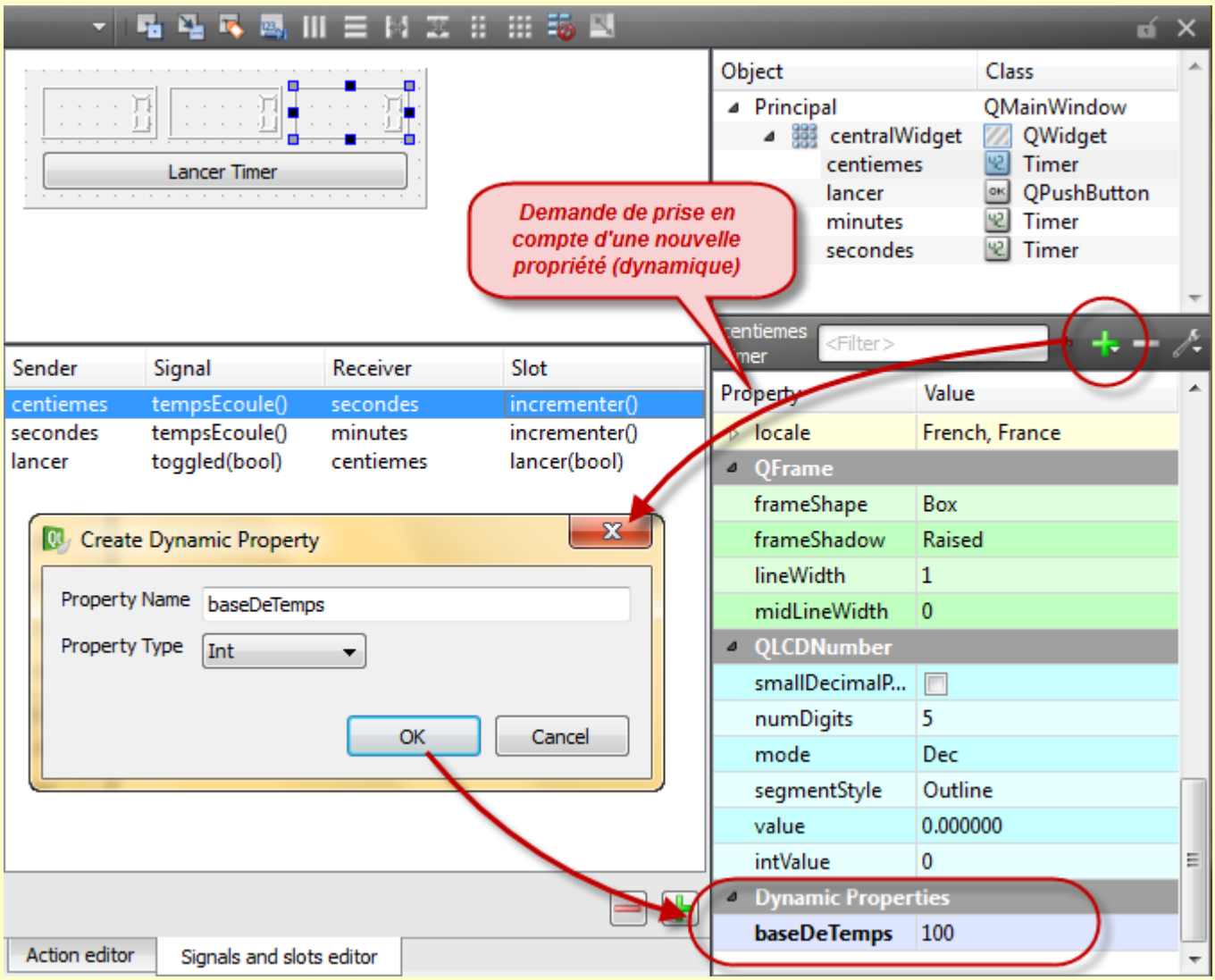
Possibilité de mettre en place une propriété qui pourra être exploitée dans le mode "Design".

Signal à envoyer lorsque l'afficheur atteint la valeur maximale.

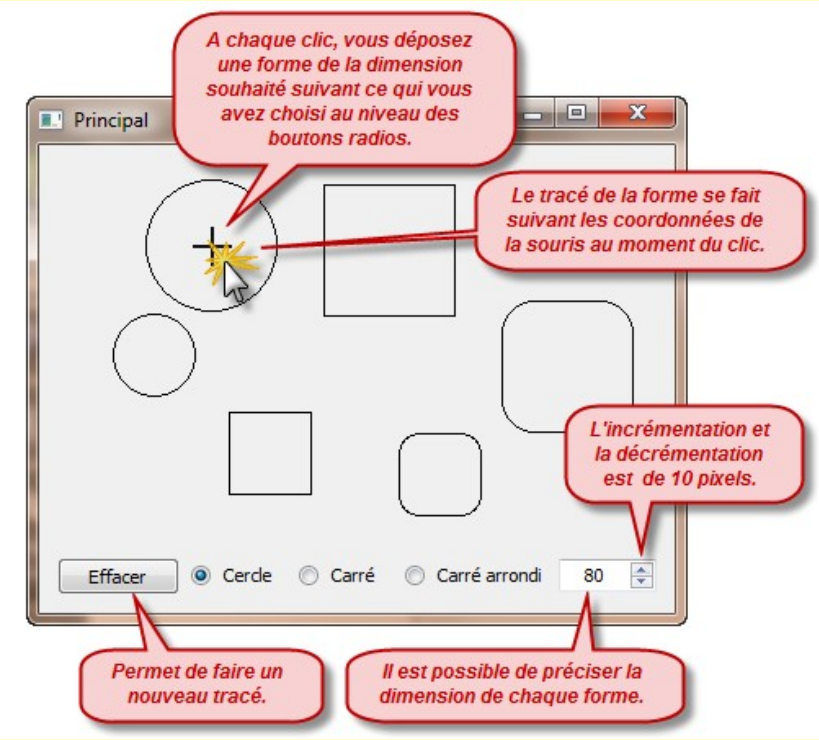
Relatifs à la propriété baseDeTemps. Par défaut, la base de temps doit être de 60.

Méthode employée pour activer ou pas le timer.

Augmenter d'une unité et vérifier si la valeur maximale est atteinte. Dans ce dernier cas , il faut envoyer le signal tempsEcoule().


Lancer Timer
Object
Class
Principal
QMainWindow
centralWidget
QWidget
centiemes
Timer
lancer
QPushButton
minutes
Timer
secondes
Timer
Demande de prise en compte d'une nouvelle propriété (dynamique)
Sender
Signal
Receiver
Slot
centiemes
tempsEcoule()
secondes
incrementer()
secondes
tempsEcoule()
minutes
incrementer()
lancer
toggled(bool)
centiemes
lancer(bool)
Create Dynamic Property
Property Name
baseDeTemps
Property Type
Int
OK
Cancel
Property
Value
locale
French, France
QFrame
frameShape
Box
frameShadow
Raised
lineWidth
1
midLineWidth
0
QLCDNumber
smallDecimalP...
numDigits
5
mode
Dec
segmentStyle
Outline
value
0.000000
intValue
0
Dynamic Properties
baseDeTemps
100
Action editor
Signals and slots editor

- *Dans le troisième projet, nous plaçons des formes prédéfinis sur la zone principale de la fenêtre. Nous pouvons placer des cercles, des carrés et des carrés à bords arrondis. Il est possible de choisir la dimension de chaque forme. Le placement de la forme souhaité s'effectue à l'aide de la souris à l'endroit voulu. Toutefois, vous devez choisir au préalable la forme souhaité avec ses dimensions avant de la placer dans la zone d'affichage. Il est également possible de tout effacer pour recommencer un nouveau tracé.*



-