Centre Informatique pour les Lettres et les Sciences Humaines

Apprendre C++ avec QtCreator : Utilisation de la classe `CZones`

Pour télécharger la dernière version des fichiers source et de ce manuel : http://www.jlpys.fr/SB_600/CZones/

Document du 03/11/2011

La classe CZones étant en développement, ces fichiers sont susceptibles d'être mis à jour assez fréquemment. Veillez à bien utiliser la dernière version disponible.

1 - Objectif

La classe CZones est destinée à fournir une fenêtre de base à des programmes dont l'interface simule le fonctionnement d'un appareil "Boutons + écran à cristaux liquides[1] affichant des icônes".

> Les dialogues dessinés avec QtDesigner ne permettent que difficilement de maquetter le type d'appareil qui nous intéressent, d'une part parce que l'aspect des widgets (boutons, en particulier) est trop standardisé et d'autre part parce que les actions utilisateur prises en comptent ne conviennent pas exactement (notamment du fait de la non gestion des "pressions prolongées").

La raison d'être de la classe CZones est donc de prendre en charge la gestion de l'affichage des icônes, la détection des actions de l'utilisateur et la tenue d'une chronique retraçant la succession de ces actions.

2 - Principe général

La classe CZones utilise une série d'images de tailles identiques, qu'il vous faudra fournir sous la forme de fichier .png.

L'une de ces images représente l'appareil lui même et sert de "fond d'écran" à la fenêtre principale du programme.

Les autres images ne comportent en général qu'un élément figurant sur un fond transparent. Cet élément peut, par exemple, définir une icône affichable par l'appareil, une zone correspondant à un bouton, ou une zone dans laquelle l'appareil peut afficher une valeur numérique (affichage à 7 segments, typique des écrans ACL).

La classe CZones réalise une superposition de ces images, exactement comme on pouvait, au siècle dernier, superposer physiquement des transparents en acétate pour réaliser un "montage" à l'aide d'un rétro-projecteur.

La classe CZones va donc offrir quatre types de fonctionnalités :

- des fonctions permettant de construire l'interface (essentiellement par ajout d'images) ;

- des fonctions permettant de contrôler l'aspect de cette interface (afficher/masquer les images disponibles) ;

- un système de gestion des évènements qui permet notamment que certaines des fonctions que vous allez définir dans votre classe de dialogue soient automatiquement appelées lorsque surviennent certains évènements (un clic, par exemple).

- des fonctions permettant d'exploiter la chronique automatique (insertion de commentaires, sauvegarde, etc.)

La limite principale de la classe CZones est qu'elle ne peut évidemment pas assurer une modélisation de l'état de l'appareil maquetté (puisque cet état est intimement lié à la nature de l'appareil en question). L'essentiel du travail de programmation qui vous incombe sera donc lié à cette modélisation.

3 - Techniques recommandées

De par sa nature, la classe CZones encourage l'utilisation de nombreuses images, ce qui pose deux problèmes :
- ces images doivent rester disponibles pour que le programme puisse s'exécuter correctement ;
- l'écriture du programme exige la mise en place d'un système efficace de désignation des éléments de l'interface utilisateur définis par ces images.

Utilisation de ressources internes

Les images sont, au départ, contenues dans des fichiers issus d'un appareil photo ou d'un scanner, ou créés par simple copier-coller à partir de la version électronique du mode d'emploi de l'appareil maquetté.

Il est possible de faire en sorte que, lors de son lancement, le programme ouvre ces différents fichiers pour aller y récupérer les images qu'il devra afficher. Cette approche comporte cependant un inconvénient important : la copie du programme d'un ordinateur à l'autre exige de respecter des règles strictes (dont un utilisateur final n'est pas forcément conscient), faute de quoi le programme risque, le moment venu, de ne pas retrouver les fichiers dont il a besoin.

L'expérience suggère que, pour ce type de projets, il est généralement préférable d'importer les images dans un fichier ressource. Lorsque QtCreator génère le fichier exécutable correspondant à votre programme, il y intègre alors directement les images. Le risque que le code exécutable ne retrouve pas ces images se trouve donc radicalement éliminé.

Utilisation de constantes symboliques

Les fonctions de la classe CZones désignent les éléments de l'interface utilisateur (boutons, icônes...) en s'appuyant sur un simple système de numérotation. Comme il n'est guère réaliste d'espérer se souvenir de quel numéro est associé à chacun de ces éléments, l'écriture du programme est grandement facilitée par le recours à des constantes dont le nom évoque directement la nature de l'élément d'interface auquel leur valeur correspond.

1 Ce qu'on appelle habituellement un écran ACL ou, en franglais, un LCD.

En clair, plutôt que de devoir écrire

```
affiche(317); //rend l'icône de la cloche visible
```

on préférera définir une constante

```
const int ICO_CLOCHE = 317;
```

qui permettra ensuite de limiter considérablement les risques d'erreur en écrivant

```
affiche(ICO_CLOCHE); //sans commentaires...
```

La création d'une maquette conduit souvent à utiliser plusieurs dizaines de constantes de ce type, dont les valeurs sont en fait sans importance (il suffit qu'elles soient toutes différentes). Plutôt que de définir ces constantes une par une comme le suggère l'exemple ci-dessus, on peut préférer recourir à une énumération :

```
enum {BOU_ON_OFF, BOU_GAUCHE, BOU_DROIT, ICO_CLOCHE, ICO_SNOOZE};
```

Toutes les étiquettes insérées dans une même `enum` correspondent automatiquement à des valeurs différentes.

L'énumération peut être aussi longue que nécessaire, et il est souvent pratique de la placer dans un fichier identifiants.h, qui fera l'objet d'une directive d'inclusion dans chacun des fichiers .cpp contenant des instructions qui désignent les éléments d'interface par leur code.

> Le recours à un fichier .h dédié aux identifiants est quasiment obligatoire dans les projets impliquant plusieurs fenêtres basées sur la classe `CZones`.

4 - Création d'un projet basé sur des `CZones`

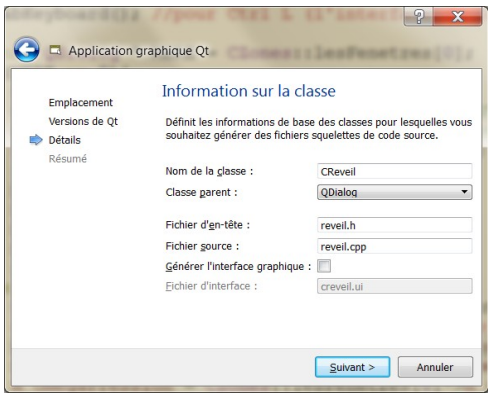
Pour utiliser cette classe, il faut créer un projet à interface graphique basé sur un dialogue pour lequel l'utilisation de QtDesigner sera remplacée par celle d'une instance de la classe `CZones`.

Renoncer à QtDesigner

Il suffit pour cela de décocher la case "Générer l'interface graphique" dans le troisième écran de la séquence de création du projet (cf. ci-contre).

> Donnez à la classe le nom de l'appareil que vous allez maquetter. Dans la suite de ce document, on suppose généralement qu'il s'agit d'un réveil matin.

Si votre programme comporte des dialogues secondaires, ceux-ci **pourront** être dessinés à l'aide de QtDesigner.



Adopter CZones

Cette adoption exige **quatre** interventions de votre part. Il faut :

1) Placer une copie des fichiers zones.h et zones.cpp dans le dossier du projet.

> Cette façon de rendre disponible la classe CZones vous donne accès à sa définition, et un examen du code correspondant peut éventuellement vous aider à comprendre comment elle fonctionne. Cet examen reste cependant purement optionnel et, quoi qu'il en soit,
>
> L'utilisation normale de la classe CZones se fait **sans modifier** les fichiers zones.h et zones.cpp
>
> Si votre programme utilise une classe CZones modifiée par vos soins, vous devrez en assumer les conséquences, notamment lors des mises à jour de la version commune des fichiers.

2) Ajouter ces deux fichiers au projet (Commande "Ajouter des fichiers existants..." du menu local obtenu avec un clic droit sur le nom du projet dans la zone de navigation entre fichiers).

3) Modifier la fonction main() de façon à ce que l'instanciation de la classe de dialogue mentionne le nom du fichier qui contient l'image qui va servir de base à l'interface utilisateur du programme, et le nom du programme lui-même :

avant

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    CReveil w;
    w.show();
    return a.exec();
}
```

après

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    CReveil w("reveil.png", "Trophy 01");
    return a.exec();
}
```

4) Modifier votre classe de dialogue pour qu'elle devienne une instance de CZones plutôt qu'un simple QDialog :

fichier	Avant	Après

.h	`#include <QtGui/QDialog>`
	
	`class CReveil : public QDialog`
	`{`
	`  Q_OBJECT`
	`public:`
	`  CReveil(QWidget *parent = 0);`
	`  ~CReveil();`
	`};`
		`#include "zones.h"`
		
		`class CReveil : public CZones`
		`{`
		`  Q_OBJECT`
		`public:`
		`  CReveil(QString image, QString titre);`
		`  ~CReveil();`
		`};`

.cpp	`#include "reveil.h"`
	
	`CReveil::CReveil(QWidget *parent)`
	`  : QDialog(parent)`
	`{`
	`}`
	`CReveil::~CReveil()`
	`{`
	`}`
		`#include "reveil .h"`
		
		`CReveil::CReveil(QString i, QString t)`
		`  : CZones(i,t)`
		`{`
		`}`
		`CReveil::~CReveil()`
		`{`
		`}`

5 - Construction de l'interface

Les opérations de construction de l'interface sont des opérations qui prennent naturellement place dans le constructeur de votre classe de dialogue : elles doivent être effectuées une seule fois, et pratiquement rien n'est possible tant qu'elles n'ont pas été effectuées.

Les fonctions de la classe `CZones` qui permettent la mise en place des éléments de l'interface utilisateur du programme exigent en général qu'on leur passe une chaîne de caractères désignant une image.

Si l'image en question est une ressource interne, cette chaîne sera quelque chose comme "`:/images/fondEcran`", par exemple. Dans le cas contraire (utilisation directe de fichiers), cette chaîne devra être un chemin d'accès au fichier (quelque chose comme "`c:/projet/image.png`", par exemple).

> Si vous n'utilisez pas une ressource interne, il vous appartient de déterminer le chemin (absolu ou relatif) adéquat et de faire en sorte qu'il continue à désigner correctement le fichier concerné lorsque le programme est déplacé ou lorsque ses conditions d'exécution changent...

L'ajout d'un élément à l'interface au moyen des fonctions présentées ci-dessous ne rend pas l'élément en question visible et disponible immédiatement. Cette visibilité/disponibilité doit être gérée explicitement, en appelant les fonctions de contrôle de l'aspect de l'appareil.

La fonction `ajouteBouton()`

Cette fonction attend normalement qu'on lui passe trois paramètres : une valeur entière et deux chaînes de caractères.

La valeur entière est un code arbitraire qui servira à identifier le bouton par la suite (la fonction `clicSur()`, par exemple, sera appelée et recevra cette valeur lorsque l'utilisateur cliquera sur ce bouton).

La première chaîne de caractères est le nom qui sera utilisé dans le journal pour désigner le bouton dans les messages le concernant.

La seconde chaîne désigne une image définissant la zone qui doit être considérée comme occupée par le bouton.

Lorsque le pointeur de la souris passe dans la zone ainsi définie, l'image est affichée en superposition sur l'image de base.

> Si vous ne souhaitez aucun effet visuel particulier, utiliser simplement une image extraite de votre image de base : la superposition sera invisible. Une autre approche est de modifier légèrement l'aspect du bouton (léger changement de teinte, par exemple) ce qui renforce le feed-back donné à l'utilisateur sur le fait qu'il rentre dans une zone cliquable.

Il est également possible de passer à `ajouteBouton()` une troisième chaîne, qui désigne l'image qui sera superposée lorsque le pointeur de la souris passe dans la zone définie par la première image. Cette seconde image permet de donner un feed-back qui concerne une zone différente de la zone cliquable, ce qui est notamment utile pour simuler la pression simultanée sur deux boutons.

Exemples d'utilisation :

```
1 ajouteBouton(BOU_OK, "OK", "bou_OK.png");
2 ajouteBouton(BOU_CAD, "CTRL ALT DEL", ":/images/bou_CAD", ":/images/feedBack_CAD");
```

> Dans ce dernier exemple, l'image `bou_CAD` désigne une zone cliquable unique. Lorsque le pointeur de la souris entre dans cette zone, c'est l'image `feedBack_CAD` qui est affichée, ce qui permet de faire comprendre à l'utilisateur qu'un clic à cet endroit simulerait la pression sur trois touches à la fois.

La fonction `ajouteIcone()`

Cette fonction reçoit deux valeurs : un entier et une chaîne de caractères.

L'entier est un code qui servira ensuite à désigner l'icône pour en contrôler la visibilité (affichée, masquée ou clignotante).

La chaîne de caractère désigne l'image de l'icône à sa position d'affichage.

> Rappel : toutes les images doivent avoir la même taille (celle de l'image de l'appareil qui sert de "fond d'écran". L'image d'une icône (ou d'un bouton) est typiquement composée d'un fond transparent et de quelques pixels noirs perdus au milieux...

Exemple d'utilisation :

```
ajouteIcone(ICO_CLOCHE, ":/images/cloche");
```

> Les icônes ajoutées à l'interface sont initialement invisibles. Utilisez les fonctions `affiche()` et `masque()` pour contrôler l'aspect de l'appareil.

La fonction ajouteImageCalculee()

Lorsque le nombre d'états d'un élément de l'interface est trop important pour qu'il soit facile de fournir une icône (et donc un fichier png) pour chacun d'entre eux, il peut être préférable de recourir à une image calculée.

> Si, par exemple, l'interface comporte une simulation d'horloge analogique, il n'est guère envisageable de fournir une image pour chacune des positions possibles des aiguilles...

On passe à la fonction ajouteImageCalculee() exactement les mêmes arguments qu'à la fonction ajouteIcone() : un code et la désignation d'une image.

Exemple d'utilisation :

```
ajouteImageCalculee(CAL_HORLOGE, ":/images/zoneHorloge");
```

L'image confiée à ajouteImageCalculee() ne sera pas affichée : elle ne sert qu'à définir une zone rectangulaire, dans laquelle sera affichée l'image produite par la fonction **calculeImage()**, que votre programme **devra** définir.

La fonction ajouteLCD()

Cette fonction reçoit deux valeurs : un entier et une chaîne de caractères.

L'entier est un code arbitraire qui servira ensuite à désigner le LCD pour en contrôler la visibilité (affiché, masqué ou clignotant) et changer la valeur qu'il affiche.

La chaîne de caractère désigne une image définissant la taille et la position du LCD.

> Il s'agir, une fois encore, d'une image ayant la même taille que celle qui sert de "fond d'écran". Elle est typiquement composée d'un fond transparent sur lequel figure un simple rectangle qui indique où le LCD devra être affiché.

Exemple d'utilisation :

```
ajouteLCD(LCD_MINUTES, ":/images/minutes");
```

Les LCD ainsi créés sont simplement des QLCDNumber, tout à fait identiques à ceux mis en place à l'aide de QtDesigner dans un projet graphique classique. Les possibilités offertes et les limitations imposées sont donc les mêmes.

Remarques :

- Les LCD de la classe CZones adoptent par défaut un style de segment "flat" et un affichage sur deux chiffres. L'utilisation de la fonction ecran() (cf. chapitre suivant) permet de modifier ces caractéristiques.

- Les LCD de la classe CZones affichent par défaut, à gauche du premier chiffre significatif, autant de zéros qu'il en faut pour occuper toutes les positions d'affichage. Ce comportement peut être contrôlé à l'aide de la fonction modeLCD() :

```
modeLCD(PAS_DE_ZEROS_A_GAUCHE);     //serait annulé par modeLCD(ZEROS_A_GAUCHE)
```

> Tous les LCD d'un projet doivent adopter le même comportement du point de vue des zéros à gauche du premier chiffre significatif. La fonction modeLCD() ne reçoit donc aucun code lui indiquant le LCD concerné par le changement de mode.

La fonction ajouteEcran()

Cette fonction reçoit deux valeurs : un entier et une chaîne de caractères.

L'entier est un code arbitraire qui servira ensuite à désigner l'écran pour en contrôler la visibilité (affiché, masqué ou clignotant) et modifier le texte qu'il affiche.

La chaîne de caractère désigne une image définissant la taille et la position de l'écran.

> Il s'agir, une fois encore, d'une image ayant la même taille que celle qui sert de "fond d'écran". Elle est typiquement composée d'un fond transparent sur lequel figure un simple rectangle qui indique où l'écran devra être affiché.

Exemple d'utilisation :

```
ajouteEcran(ECRAN, "minutes.png");
```

Les écrans ainsi créés sont simplement des QTextEdit, tout à fait identiques à ceux mis en place à l'aide de QtDesigner dans un projet graphique classique. Les possibilités offertes et les limitations imposées sont donc les mêmes.

Remarque : les écrans de la classe CZones sont, par défaut, en mode read-only. L'utilisation de la fonction ecran() (cf. chapitre suivant) permet de modifier cette caractéristique.

La fonction ajouteClavier()

Cette fonction permet d'utiliser une seule image pour définir plusieurs boutons à la fois.

Par convention, l'image se compose d'un fond au moins partiellement transparent (pixels dont la valeur alpha est inférieure à 255) sur lequel figurent des zones totalement opaques (pixels contigus dont la valeur alpha est 255).

> La fonction ajouteClavier() analyse l'image dans le sens de la lecture (ie. en partant du coin supérieur gauche et en explorant ligne par ligne). Dès qu'elle rencontre un pixel opaque, elle repère tous les pixels opaques adjacents et considère qu'ils définissent la zone occupée par un bouton. Ces pixels sont ignorés lors de la recherche des boutons suivants).

La fonction ajouteClavier() exige qu'on lui passe trois paramètres : un code, une chaîne de caractères désignant l'image à utiliser, et une QStringList contenant les noms à associer aux différents boutons.

Exemple d'utilisation :

```
enum {CLAVIER_UN, BOU_Zz, BOU_ALARME, BOU_PLUS, BOU_MOINS, BOU_CAL, BOU_QUESTION};
```

```
1  QStringList lesNoms;
2  lesNoms << "Zz" << "alarme" << "plus" << "moins" << "calendrier" << "question";
3  ajouteClavier(CLAVIER_UN, ":/images/clavier", lesNoms);
```

Les boutons ainsi définis se verront attribuer des codes croissants et des noms puisés dans la liste fournie, dans l'ordre où ils seront détectés.

Il convient d'être vigilant lors de la mise en place d'un clavier, de façon à ce que l'ordre de détection des boutons ne vienne pas perturber l'association entre les boutons et les identifiants (code et nom) prévus à leur intention. Un simple contrôle des messages affichés dans la fenêtre chronique met immédiatement en évidence toute interversion accidentelle, qui peut être corrigée soit en modifiant l'image, soit en changeant la définition des constantes et l'ordre des noms dans la liste.

La fonction ajouteSon()

Cette fonction reçoit deux valeurs : un entier et une chaîne de caractères.

L'entier est un code arbitraire qui servira ensuite à désigner le son pour en contrôler la reproduction (mise en route, arrêt).

La chaîne de caractères désigne un fichier .wav qui contient un enregistrement du son concerné.

Exemple d'utilisation :

```
ajouteEcran(SONNERIE, "sonnerie.wav"); //chemin relatif...
```

Le système de gestion des ressources de QtCreator ne permet actuellement pas d'utiliser des ressources de type son. Il faut donc obligatoirement recourir à un fichier qui restera distinct de l'exécutable du programme, mais devra être disponible pour que celui-ci puisse produire le son...

6 - Contrôle de l'aspect de l'appareil

Une fois l'interface mise en place, la classe `CZones` fournit un certain nombre de fonctions qui permettent au programme de modifier l'apparence de l'image de l'appareil qui est présentée à l'écran.

> Ces fonctions étant membre de `CZones`, elles sont accessibles aux fonctions membre de votre classe de dialogue (qui *est* une `CZones`). Si le projet comporte d'autres classes (une classe `CEtat` permettant de représenter l'état de l'appareil, peut-être ?), les fonctions de ces autres classes n'auront évidemment pas la possibilité d'utiliser les fonctions décrites ici. Ceci est une excellente chose : contrôler l'aspect visuel de l'appareil est le rôle de la classe de dialogue, et d'elle seule.

La fonction `affiche()`

Cette fonction reçoit une valeur entière : le code correspondant à l'élément d'interface qu'elle doit rendre visible.

Exemple d'utilisation :

```
affiche(ICO_CLOCHE);
```

Si l'élément concerné est un LCD, la fonction `affiche()` accepte un second argument indiquant la valeur que le LCD en question doit afficher.

Exemple d'utilisation :

```
affiche(LCD_MINUTES, 59);
```

Si l'élément concerné est un écran, la fonction `affiche()` accepte comme second argument une chaine de caractères dont le contenu sera ajouté à la fin du texte affiché par l'écran.

Exemple d'utilisation :

```
affiche(ECRAN, "coucou !");
```

La fonction `masque()`

Cette fonction peut être utilisée pour rendre invisibles tous les éléments ajoutés à l'interface.

Exemple d'utilisation :

```
masque();
```

Elle peut également être utilisée pour masquer un seul de ces éléments : il faut alors lui désigner cet élément en lui en passant le code.

Exemple d'utilisation :

```
masque(ICO_CLOCHE);
```

La fonction `clignote()`

Cette fonction peut être utilisée pour faire clignoter tous les éléments ajoutés à l'interface.

Exemple d'utilisation :

```
clignote();
```

Elle peut également être utilisée pour faire clignoter un seul de ces éléments : il faut alors lui désigner cet élément en lui en passant le code.

Exemple d'utilisation :

```
clignote(ICO_CLOCHE);
```

> Par défaut, les objets clignotants changent de visibilité deux fois par seconde. Il est possible de modifier ce rythme en appelant la fonction `periodeClignotement()`, à laquelle il faut passer une durée exprimée en millisecondes :
>
> ```
> periodeClignotement(250); //double la cadence
> ```

La fonction `joueSon()`

Cette fonction peut être utilisée pour obtenir la restitution audio du son désigné par le code qu'on lui passe.

Exemple d'utilisation :

```
joueSon(SONNERIE);
```

On peut également lui spécifier un nombre de répétitions successives. Si ce nombre est -1, le son sera rejoué indéfiniment, jusqu'à exécution de `stopSon()`.

Exemple d'utilisation :

```
joueSon(SONNERIE, 10);
```

> Les sons sont restitués de façon asynchrone : le programme poursuit son exécution sans attendre que la restitution soit terminée. Plusieurs sons peuvent être restitués simultanément (mais c'est rarement agréable à entendre...).

La fonction `stopSon()`

Cette fonction arrête immédiatement la restitution audio du son dont on lui passe le code (elle est sans effet si le son en question n'est pas en cours de restitution audio).

Exemple d'utilisation :

```
stopSon(SONNERIE);
```

Si on ne lui passe aucun argument, cette fonction arrête tous les sons en cours de restitution audio, ce qui permet notamment d'interrompre un son joué en boucle sans avoir à déterminer de quel son il s'agit.

Exemple d'utilisation :

```
stopSon();
```

La fonction `lcd()`

Cette fonction renvoie l'adresse du `QLCDNumber` correspondant au LCD dont on lui passe le code. Cette information permet d'utiliser toutes les fonctionnalités de la classe `QLCDNumber`.

Exemple d'utilisation :

```
lcd(LCD_TIME_H)->setSegmentStyle(QLCDNumber::Outline);
```

La fonction `ecran()`

Cette fonction renvoie l'adresse du `QTextEdit` correspondant à l'écran dont on lui passe le code. Cette information permet d'utiliser toutes les fonctionnalités de la classe `QTextEdit`.

Exemple d'utilisation :

```
ecran(ECRAN)->setReadOnly(false);
```

La fonction `activeClavier()`

Cette fonction reçoit le code du clavier qui doit être activé. Elle est surtout utile aux programmes qui utilisent plusieurs claviers (claviers virtuels sélectionnables à volonté), mais, **en cas de clavier unique, il est quand même nécessaire d'activer explicitement celui-ci**.

Exemple d'utilisation :

```
activeClavier(CLAVIER_UN);
```

> Si le programme dispose de plusieurs claviers, un seul d'entre eux peut être actif à un instant donné : la simple activation de l'un désactive implicitement tous les autres.
>
> L'activation d'un clavier provoque son affichage. Si le "fond d'écran" ne comporte aucune image de touche, l'interface affichera en permanence le clavier actif, et lui seul.

La fonction calcule`Image()`

A la différence des précédentes, cette fonction n'est pas une fonction qu'il s'agit d'appeler, mais **une fonction qu'il s'agit de définir** : c'est elle qui est responsable de l'aspect visuel des `imagesCalculees`.

Elle sera automatiquement appelée lorsqu'il sera nécessaire de redessiner la zone d'écran attribuée à une image calculée, et elle doit disposer de deux paramètres : le premier recevra le code associé à la zone concernée, le second est une référence qui sera associée à l'image finalement dessinée à l'écran dans la zone en question.

Exemple de définition :

```
1 void Dialog::calculeImage(const int CODE, QPixmap & pix)
```

```
2 {
3   if(CODE == CAL_COULEUR){
4     pix.fill(m_couleur);        }
5 }
```

La fonction `fill()` utilisée à la ligne *4* se borne à remplir uniformément la zone rectangulaire avec la couleur contenue dans la variable membre `m_couleur`. Il suffit donc de changer la valeur de cette variable pour obtenir un affichage différent, sans avoir pour cela à fournir un fichier png pour chacune des nuances utilisées.

Contrairement à ce que cet exemple pourrait laisser croire, l'écriture d'une fonction `calculeImage()` produisant des images non triviales n'est pas forcément une tâche très simple pour un programmeur débutant.

L'usage d'images calculées n'est conseillé que si ces images sont réellement faciles à générer, ou si le maquette dont il s'agit **l'exige** absolument.

Un des cas où le calcul d'une image intéressante peut rester assez simple est celui où il est possible de se contenter de modifier une image contenue dans un fichier png. L'exemple suivant affiche les aiguilles d'une horloge dans la position correspondant à l'heure courante :

```
1  void Clock::calculeImage(const int CODE, QPixmap & pix)
2  {
3    QPainter p(&pix);
4    QPoint axeRotation(pix.width()/2, pix.height()/2);
5    p.translate(axeRotation);
6    p.setRenderHint(QPainter::Antialiasing);
7    QRect rect(-axeRotation, pix.size());
8    QTime instant = QTime::currentTime();
9    p.save(); //sauvegarde la position initiale du repère
10   p.rotate(30 * instant.hour() + 0.5 * instant.minute());
11   p.drawPixmap(rect, QPixmap(":/png/courte"));
12   p.restore(); //annule la rotation précédente
13   p.rotate(instant.minute()*6);
14   p.drawPixmap(rect, QPixmap(":/png/longue"));
15 }
```

La rotation des aiguilles est obtenue en affichant leur image *après avoir fait pivoter le repère* utilisé par le `QPainter` *(10, 13)*. Si l'aiguille doit afficher autre chose que l'heure, il faut bien entendu effectuer les calculs nécessaires pour déterminer l'ampleur de la rotation (qui doit être exprimée en degrés). Dans le cas d'un quadrant qui n'occupe pas un disque complet (vu-mètre, par exemple), il faudra évidemment modifier aussi le calcul de la position de l'axe *(4)*.

Les images contenues dans les fichiers mentionnés aux lignes *11* et *15* sont dessinées *dans la zone attribuée à l'image calculée*. A la différence des images utilisées lors de la mise en place de l'interface, elles n'ont donc pas forcément la même taille (ni même les mêmes proportions hauteur/largeur) que l'image servant de fond. Elles doivent cependant habituellement présenter les mêmes proportions hauteur/largeur que la zone attribuée à l'image calculée (faute de quoi leur dessin dans cette zone se traduira par un étirement ou un écrasement).

7 - Gestion des évènements

Si la mise en place de l'interface et le contrôle de l'aspect de la représentation visuelle de l'appareil reposent essentiellement[2] sur l'appel de fonctions définies dans la classe `CZones`, la gestion des évènements s'appuie sur une logique inverse : il s'agit d'ajouter à votre classe de dialogue des fonctions qui seront appelées (sans que vous ayez à faire quoi que ce soit pour cela) lorsque les évènements qui vous intéressent surviendront.

Votre tâche consiste donc à définir le corps de ces fonctions, de façon à réaliser les conséquences que ces actions impliquent pour l'appareil que vous simulez.

La fonction `clicSur()`

Cette fonction doit être membre de votre classe de dialogue et doit être déclarée comme ceci :

```
  void clicSur(const int CODE);
```

Elle sera appelée lorsque l'utilisateur cliquera sur l'un des boutons mis en place par appel à `ajouteBouton()` ou à `ajouteClavier()`. Lors de chaque appel, elle reçoit le code associé au bouton concerné.

Typiquement, le corps de cette fonction comporte un `switch` qui lui permet de prendre les mesures appropriées à la nature du bouton qui a été cliqué :

2 L'exception notable étant évidemment le contrôle de l'aspect des `imagesCalculees`.

```
void Dialog::clicSur(const int BOUTON)
{
  switch(BOUTON) {
     case BOU_ON_OFF:
        //....
     break;
     case BOU_SIEGE_EJECTABLE:
        //...
     break;
     default:
        remarque("clic sur un bouton non pris en charge !");
  }
}
```

La fonction pressionSur()

Cette fonction doit être membre de votre classe de dialogue et doit être déclarée comme ceci :

```
  void pressionSur(const int CODE);
```

Elle sera appelée lorsque l'utilisateur maintiendra enfoncé l'un des boutons mis en place par appel à ajouteBouton() ou à ajouteClavier() pendant une durée dépassant 2 secondes. Lors de chaque appel, elle reçoit le code associé au bouton concerné.

Typiquement, le corps de cette fonction comporte un switch analogue à celui utilisé par clicSur().

> La durée seuil au delà de laquelle une pression est considérée comme un événement "pression prolongée" peut être modifiée en passant la durée souhaitée (exprimée en millisecondes) à la fonction dureePressionLongue().

8 - Exploitation de la chronique

La chronique est un texte, une sorte de "journal" de bord dans lequel le programme enregistre la séquence des actions de l'utilisateur et, éventuellement, les variations de l'état de l'appareil.

Dès l'instant où des boutons ont été ajoutés à l'interface par appel à `ajouteBouton()` ou à `ajouteClavier()`, toute action (clic ou pression longue) sur l'un de ces boutons crée automatiquement une entrée dans la chronique.

La fonction `remarque()`

Si les clics et pressions longues sur les boutons sont automatiquement notés dans la chronique, il est en général intéressant d'ajouter à celle-ci des notes indiquant dans quel état l'appareil se trouve. La fonction `remarque()` permet d'insérer ce genre d'information :

```
remarque("10 secondes d'inactivité : passage en mode veille");
```

La fonction `nomBouton()`

Cette fonction renvoie simplement la première chaîne de caractères qui a été passée à `ajouteBouton()` lors de la mise en place du bouton associé au code sur lequel elle est interrogée

Exemple d'utilisation :

```
QString texte = nomBouton(BOU_OK);
```

La fonction `afficheChronique()`

La fenêtre présentant le texte de la chronique est masquée par défaut. Elle peut être rendu visible en appelant la fonction `afficheChronique()` ou à l'aide de la combinaison de touches Ctrl L.

> Pour masquer la fenêtre de la chronique, utilisez sa case de fermeture ou appuyez à nouveau sur Ctrl L.

La fonction `sauveChronique()`

Cette fonction permet d'enregistrer dans un fichier texte le contenu de la chronique.

Si on ne lui passe rien, elle utilise un dialogue secondaire pour demander à l'utilisateur d'indiquer où et sous quel nom de fichier les informations doivent être enregistrées.

Si on lui passe une chaîne de caractères, elle essaie de l'utiliser comme chemin/nom de fichier.

9 - Projets multi-fenêtrés

Certains projets gagnent à utiliser plusieurs fenêtres, soit parce que l'appareil maquetté dispose d'éléments d'interface situés sur plusieurs de ses côtés (face avant et face arrière, par exemple), soit parce que plusieurs appareils sont concernés (l'accordeur, par exemple, ne prend tous son sens que si une simulation d'instrument de musique permet de le tester).

Dans tous les cas, la création du projet doit suivre la procédure décrite ci-dessus, car

La fenêtre principale de l'application doit être basée sur la classe `CZones`.

Les fenêtres secondaires peuvent, pour leur part, soit être également des `CZones`, soit être de simples `QDialog` (dessinés ou non à l'aide de QtDesigner).

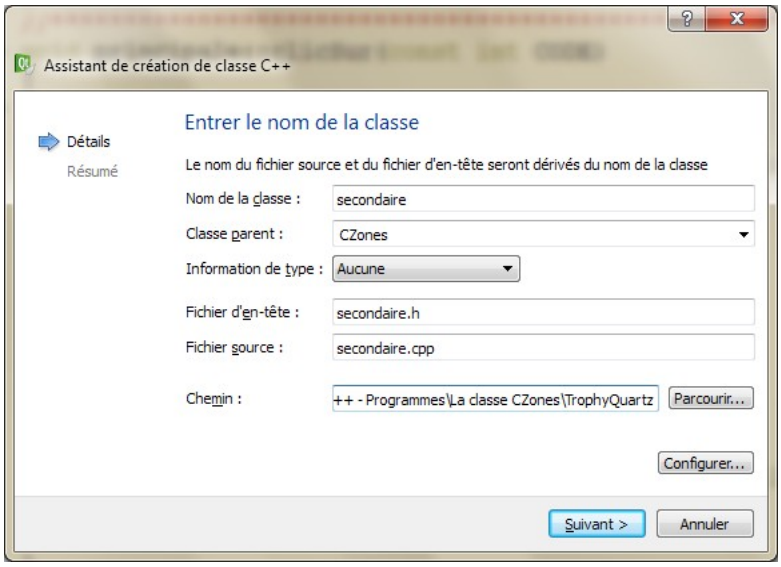
Fenêtres secondaires de type `QDialog`

La seule difficulté concerne les communications entre fenêtres. Selon les circonstances, on peut envisager d'établir ces communications soit à l'aide du mécanisme signaux/slots (un signal émis par une fonction d'un des dialogues pouvant être connecté à un slot d'une autre fenêtre), soit en faisant du dialogue secondaire une variable membre du dialogue principal (ce qui permet aux fonctions de ce dernier d'accéder aux membres publics du dialogue secondaire).

D'une façon générale, lorsque les fenêtres secondaires sont de simples `QDialog`, le fait que la fenêtre principale soit basée sur la classe `CZones` n'affecte en rien la problématique des communications entre fenêtres.

Fenêtres secondaires de type `CZones`

Dans le cas d'une fenêtre secondaire, l'utilisation de la classe `CZones` est en fait plus simple que dans le cas d'une fenêtre principale car, lors création d'une classe, QtCreator nous laisse spécifier librement la classe de base (alors que, lors de la création du projet, la classe de base ne peut être que `QMainWindow` ou `QDialog`) :



Il reste néanmoins nécessaire de rajouter manuellement au constructeur les deux paramètres de type `QString` qui spécifieront, lors de l'instanciation, l'image servant de fond d'écran et le nom de la fenêtre.

Cette instanciation peut être réalisée dans la fonction `main()` :

```
1  int main(int argc, char *argv[])
2  {
3      QApplication a(argc, argv);
4      principale w(":/png/fondMere", "TrophyQuartz");
5      secondaire x(":/png/fondFille", "Dos");
6      x.show();
7      return a.exec();
8  }
```

Le rôle des fenêtres est défini par l'ordre d'instanciation : la première variable créée sera la fenêtre principale, les suivantes seront des fenêtres secondaires.

Par comparaison avec la fenêtre principale, les fenêtres secondaires présentent plusieurs particularités notables :

- Elles ne sont pas visibles par défaut (ie. elles ne deviennent visibles qu'au prix d'un appel à `show()`, cf. *6*).

- Elles ne disposent pas de leur propre journal, mais utilisent celui de la fenêtre principale (ie. si l'une de leurs fonctions membre appelle `remarque()`, le texte est inséré dans le seul journal qui existe, celui de la fenêtre principale).

- Leur case de fermeture est désactivée par défaut, et elles ne se referment que lorsque la fenêtre principale est fermée (fin du programme). Il reste évidemment possible de les masquer à l'aide de la fonction `hide()`.

- Elles peuvent signaler à la fenêtre de base les évènements qui la concernent, sans qu'il soit nécessaire de mettre en place un système signaux/slots :

```
1  void secondaire::clicSur(const int CODE)
2  {
3     switch(CODE){
4     case BOU_HR :
5           remarque("géré dans secondaire");
6           break;
7     case BOU_12H :
8           masque(CUR_24H);
9           affiche(CUR_12H);
10          CZones::clicSur(CODE);
11          break;
12    default: CZones::clicSur(CODE);
13    }
14 }
```

Ce fragment de code illustre trois gestions différentes du clic sur un bouton de la fenêtre secondaire :

Le bouton `BOU_HR` ne concerne que la fenêtre secondaire, qui le gère donc exactement comme si elle était une fenêtre principale.

Le bouton `BOU_12H` exige un traitement de la part de la fenêtre secondaire (ici, le déplacement d'un curseur dont les différentes positions possibles sont représentées par des icônes). Le déplacement de ce curseur doit cependant aussi être pris en compte par la fenêtre principale, qui va, dans cet exemple, passer d'un affichage "européen" (ie. sur 24 heures) à un affichage "américain" (sur 12 heures, où 13 h devient 1 h).

Bien qu'elle n'ait aucun moyen d'accéder explicitement à la fenêtre principale, la fonction `secondaire::clicSur()` peut transmettre à celle-ci une demande de prise en compte du clic sur le bouton : il lui suffit *(10)* d'appeler `CZones::clicSur()` en lui passant le code associé au bouton concerné.

> Il s'agit normalement du code reçu par `secondaire::clicSur()`, mais il peut parfois s'avérer intéressant que le clic sur un bouton de la fenêtre secondaire soit équivalent au clic sur un autre bouton situé, lui, dans la fenêtre principale : il suffit alors de passer le code de cet autre bouton à `CZones::clicSur()`.

Cette façon de procéder suppose évidement que la la fenêtre principale dispose d'une fonction `clicSur()` prenant en charge non seulement les codes de ses propres boutons, mais aussi ceux des boutons des fenêtres secondaires qui la concernent.

Les autres boutons ne nécessitent aucun traitement de la part de la fenêtre secondaire, qui se contente de répercuter sur la fenêtre principale les clics dont ils font l'objet.

> Du point de vue de la gestion de l'interface, tout se passe donc comme si ces boutons étaient en fait dans la fenêtre principale.

Si AUCUN des boutons de la fenêtre secondaire ne doit avoir d'effet sur celle-ci, il est inutile de créer une fonction `secondaire::clicSur()` car, **en son absence**, tous les clics seront automatiquement signalés à la fenêtre de base.

> Remarquez bien que, dès l'instant où la fonction `secondaire::clicSur()` existe, cette transmission automatique disparaît et que seuls les appels explicites à `CZones::clicSur()` permettent alors ce type de communication.

La fonction `pressionSur()` d'une fenêtre secondaire peut être gérée exactement comme la fonction `clicSur()`.

10 - Les commandes "hors simulation"

Il peut s'avérer utile d'ajouter au programme des commandes qui ne sont pas celles de l'appareil maquetté, mais qui concernent l'utilisation de la maquette elle-même. Un exemple simple serait une commande qui enregistrerait le contenu du journal et effacerait celui-ci, ce qui permettrait de faire passer un nouveau sujet sans avoir à quitter et relancer le programme.

Pour que ces commandes n'interfèrent pas avec celles de l'appareil, la classe `CZones` propose de les regrouper dans un menu local, qui apparaît lorsqu'on clique avec le bouton droit.

La fonction `ajouteCommande()`

Cette fonction dispose de deux paramètres qui permettent de lui communiquer le code d'identification attribué à la commande et le texte qui apparaitra dans le menu.

Exemple d'utilisation :

```
ajouteCommande(COM_CHRONIQUE, "Chronique visible");
```

Les commandes sont ajoutées à la fin du menu, ce qui signifie qu'elles figureront dans celui-ci dans l'ordre où elles y ont été ajoutées.

La fonction `ajouteSeparateur()`

Cette fonction ajoute une ligne horizontale dans le menu, ce qui permet de séparer visuellement des groupes de commandes.

Exemple d'utilisation :

```
ajouteSeparateur();
```

La fonction `commande()`

Cette fonction doit être membre de votre classe de dialogue et doit être déclarée comme ceci :

```
  void commande(const int CODE);
```

Elle sera appelée lorsque lorsqu'un des items du menu sera activé. Lors de chaque appel, elle reçoit le code associé à la commande concernée.

Typiquement, le corps de cette fonction comporte un `switch` qui lui permet d'effectuer les traitements correspondant aux différentes commandes :