Modélisation Objet -- MOBJ

Contents

TMEs 8, 9 & 10 -- Visualisateur de Netlist . . . 1
Compilation . . . 2
Fichiers Fournis. . . . 2
Documentation de Qt 4 . . . 2
Spécifications des Widgets. . . . 3
SaveCellDialog . . . 3
OpenCellDialog. . . . 3
CellViewer . . . 4
InstancesWidget & InstancesModel . . . 5
CellsLib & CellsModel . . . 6
Synchronisation entre CellViewer et CellsLib . . . 6
CellWidget . . . 7
  Systèmes de Coordonnées . . . 7
  Conversion des Points & des Boîtes . . . 8
  Gestion des Évènements . . . 8
  Récapitulatif de la Classe `CellWidget` . . . 9
  Charte Graphique . . . 10
Travail à Réaliser . . . 10
Annexes . . . 11

TMEs 8, 9 & 10 -- Visualisateur de Netlist

L'objectif des TME 8, 9 & 10 est d'implanter l'interface graphique de visualisation des NETLIST. La base du code de cette interface, s'appuyant sur QT vous a été présentée en cours, il s'agira de compléter les parties manquantes. Les classes/*widgets* à implanter sont, dans l'ordre:

- `CellViewer (.h, .cpp)`, la fenêtre principale de l'application (dérivant de `QMainWindow`).
- `Main.cpp`, contenant la `QApplication` (gestion de la boucle principale des évènements).
- `SaveCellDialog (.h, .cpp)`, boîte de dialogue pour sauver une `Cell`.
- `OpenCellDialog (.h, .cpp)`, boîte de dialogue pour charger une `Cell`.
- `InstancesWidget (.h, .cpp)` et le modèle correspondant `InstancesModel (.h,.cpp)`, table contenant les instances présentes dans le modèle actuellement chargé, ainsi que les *master Cells* associées.
- `CellsLib (.h, .cpp)` et le modèle correspondant `CellsModel (.h,.cpp)`, table contenant la liste des `Cell` actuellement présentes en mémoire.
- `CellWidget (.h, .cpp)`, la fenêtre de tracé de la `Cell`. Pour pouvoir commencer les TMEs, une version dégénérée de ce *widget* vous est fournie.

Compilation

En prélude à la compilation, récupérer le fichier CMakeLists.txt qui ajoute la gestion de QT à votre projet. Les modifications sont principalement liées au support de `moc` (Meta-Object Compiler) qui implique de créer des **.cpp** additionnels pour chaque **.h** contenant (au moins) un `QWidget`. Pour les curieux:

```
# detection de Qt4.
find_package(Qt4     REQUIRED)

# Chargement des macros additionnelles de Qt.
include(${QT_USE_FILE})

# [...]

# Liste des fichiers include contenant des classes dérivées de QWidget.
# L'ensemble des fichiers include du projet se trouve ainsi divisé en
# deux, suivant qu'ils contiennent ou non des QWidgets.
set( mocIncludes  SaveCellDialog.h
                  CellWidget.h
                  CellViewer.h
   )

# Appel au Meta-object Compiler de Qt, pour les mocIncludes.
# Construit la liste des .cpp additionnels "mocCpps".
qt4_wrap_cpp( mocCpps  ${modIncludes} )

# L'exécutable est construit avec les .cpp ordinaires ET les
# .cpp générés par MOC.
add_executable( tme810  ${cpps} ${mocCpps} )

# Ajout des bibliothèques Qt (il y en a plusieurs) pour l'édition de liens.
target_link_libraries( tme810  ${QT_LIBRARIES} ${LIBXML2_LIBRARIES} )

# On installe *tous* les includes (non-Qt & Qt).
install( FILES  ${includes} ${mocIncludes}  DESTINATION include )
```

Note

Rappel la méthode pour organiser votre code et compiler est décrite dans procédure de compilation.

Fichiers Fournis

- CellWidget.h et CellWidget.cpp : implantation tronquée du widget mais qui vous permet d'implanter immédiatement `CellViewer`.
- CMakeLists.txt : la configuration de `cmake`, il vous faudra ajouter les fichiers au fur et à mesure que vous les écrirez.

Documentation de Qt 4

Est accessible en local sur chaque machine à l'adresse suivante:

`file:///usr/share/doc/qt4/html/index.html`

Spécifications des Widgets

SaveCellDialog

La spécification et l'implantation de ce *widget* a été présentée en cours.

Note

Le `SaveCellDialog`, contrairement à ce que son nom laisse penser, n'effectue pas lui-même la sauvegarde de la cellule. Il se contente de demander à l'utilisateur quel nom il souhaite donner à la cellule à sauvegarder. Ce qui est récupéré par l'appel à la méthode `SaveCellDialog::run(QString& name)` est juste le nom sous lequel on va effectuer la sauvegarde. D'où le passage par référence de l'argument `name`.
La sauvegarde sera réellement faite dans la méthode `CellViewer::saveCell()`.

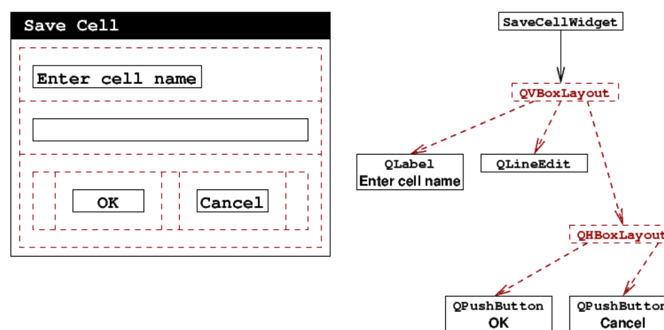

Figure 1: Figure 1 -- Structure de SaveCellDialog

OpenCellDialog

Le *widget* `OpenCellDialog` a une structure quasi-identique à celle du `SaveCellDialog`. La même remarque que précédemment s'applique. Ce *widget* n'effectue pas le chargement lui-même, mais retourne simplement le nom du modèle que l'utilisateur veut charger (passage par référence d'une `QString`).

La chargement sera réellement fait dans la méthode `CellViewer::openCell()`.

Note

Exercice de style: dans `SaveCellDialog`, le *widget* est créé dans le `CellViewer` puis lancé (affiché) selon les besoins avec la méthode `SaveCellDialog::run()`. Dans `OpenCellDialog`, transformer la méthode `OpenCellDialog::run()` en méthode statique qui créera *à la volée* un `OpenCellDialog` (et le détruira sitôt le résultat obtenu).

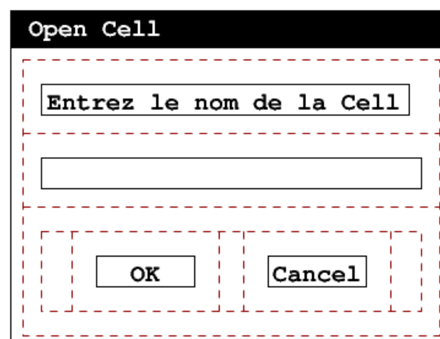

Figure 2: Figure 2 -- Structure de OpenCellDialog

CellViewer

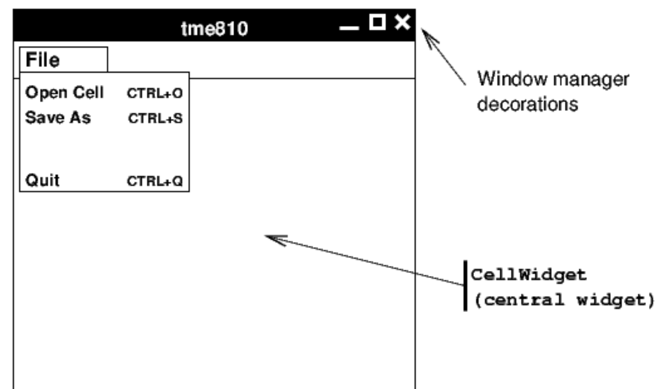

Figure 3: Figure 3 -- Structure du CellViewer

Le `CellViewer` a lui aussi été présenté en cours. On rappelle sa déclaration:

```
class CellViewer : public QMainWindow {
    Q_OBJECT;
  public:
                      CellViewer          ( QWidget* parent=NULL );
    virtual          ~CellViewer          ();
            Cell*     getCell             () const;
    inline  CellsLib* getCellsLib         ();   // TME9+.
  public slots:
            void      setCell             ( Cell* );
            void      saveCell            ();
            void      openCell            ();
            void      showCellsLib        ();   // TME9+.
            void      showInstancesWidget ();   // TME9+.
  private:
    CellWidget*       cellWidget_;
    CellsLib*         cellsLib_;          // TME9+.
    InstancesWidget*  instancesWidget_;   // TME9+.
    SaveCellDialog*   saveCellDialog_;
};
```

InstancesWidget & InstancesModel

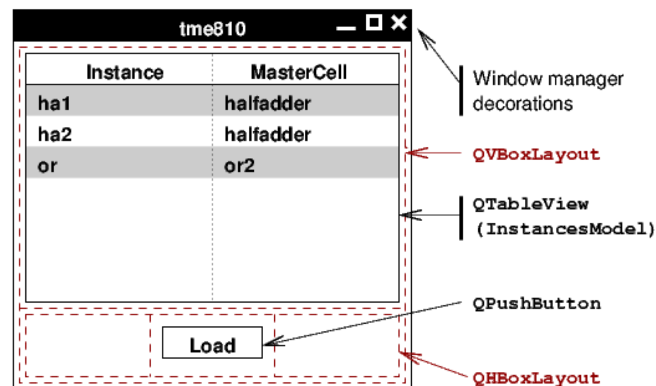

Figure 4: Figure 4 -- Structure de l'InstanceWidget

Déclaration de l'`InstancesWidget`:

```
class InstancesWidget : public QWidget {
    Q_OBJECT;
  public:
                  InstancesWidget ( QWidget* parent=NULL );
            void  setCellViewer   ( CellViewer* );
            int   getSelectedRow  () const;
    inline  void  setCell         ( Cell* );
  public slots:
            void  load            ();
  private:
            CellViewer*     cellViewer_;
            InstancesModel* baseModel_;
            QTableView*     view_;
            QPushButton*    load_;
};
```

Principales Caractéristiques:

- L'appui sur le bouton *close* du bandeau de la fenêtre ne provoquera pas sa destruction (`delete`). Elle sera simplement *réduite*, c'est à dire non-affichée.
- La `QTableView` sera configurée de façon à sélectionner une ligne entière, et une seule à la fois.
- Le modèle associé à la `QTableView` (attribut `view_`) de type `InstancesModel` (attribut `baseModel_`) présentera deux colonnes:
  1. Le nom de l'instance.
  2. Le nom du modèle de l'instance (le nom de la *master cell*).

  Ne pas oublier de fournir le nom des colonnes (*headers*).
- En cas de *clic* sur le boutton **Load**, on chargera dans le visualisateur la *master cell* dans la ligne actuellement sélectionnée. Si aucune ligne n'est sélectionnée, ne rien faire. On s'aidera de la méthode `::getSelectedRow()` et du *slot* `::load()`.
- La méthode `::getSelectedRow()` renverra l'index de la ligne (*row*) selectionnée. Si aucune ligne n'est actuellement sélectionnée, elle renverra `-1`. Pour implanter cette méthode, on consultera avec profit la documentation QT de la classe `QItemSelectionModel`.

CellsLib & CellsModel

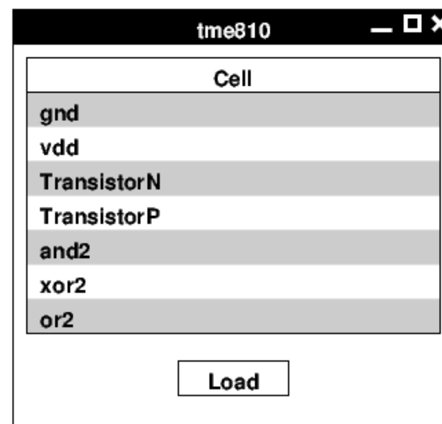

Figure 5: Figure 5 -- Le CellsLib

Déclaration du `CellsLib`:

```
class CellsLib : public QWidget {
    Q_OBJECT;
  public:
                         CellsLib       ( QWidget* parent=NULL );
           void          setCellViewer  ( CellViewer* );
           int           getSelectedRow () const;
    inline CellsModel*   getBaseModel   ();
  public slots:
           void          load           ();
  private:
    CellViewer*   cellViewer_;
    CellsModel*   baseModel_;
    QTableView*   view_;
    QPushButton*  load_;
};
```

Synchronisation entre CellViewer et CellsLib

La liste des `Cell` présentée dans le `CellsLib` contient *uniquement* les modèles qui ont été chargés en mémoire. Lorsque l'on charge, en appelant le `OpenCellDialog` un modèle, deux cas sont possibles:

1. Le modèle *est déjà* présent en mémoire (directement accessible avec `Cell::getAllCells()`, dans ce cas on le recharge dans le `CellViewer`. Aucune autre action n'est nécessaire.
2. Le modèle *n'est pas* présent en mémoire, il faut donc le charger depuis le disque (fichier XML) avec la méthode `Cell::load()`. L'appel à cette dernière méthode va donc modifier la liste des modèles en mémoire. **Il faut informer le CellsLib de ce changement**, plus précisément, son modèle associé (`CellsModel`).

Pour informer le `CellsModel` d'un changement dans la liste des modèles en mémoire, nous allons utiliser le mécanisme de *signal/slot*.

- L'émetteur (*sender*) est le `CellViewer`, nous allons définir un signal à son niveau: `void cellLoaded()`. C'est une méthode sans corps (son implantation est générée par `moc`).

- Le récepteur (*receiver*) sera le `CellsModel`, un slot va y être défini: `void updateDatas()`. C'est la méthode qui effectue l'actualisation des données, son écriture est à votre charge.
- Le *signal* provoquant la synchronisation sera émis quelque part dans le corps de la méthode `CellViewer::openCell()`:

```
void   CellViewer::openCell ()
{
   // du code...
   emit cellLoaded();
   // Encore du code...
}
```

CellWidget

Systèmes de Coordonnées

On rappelle ici la relation entre le système de coordonnées de la *netlist* (en noir sur la figure) et le celui de l'écran (en rouge, utilisé par QT).

- Pour simplifier les conversions entre système de coordonnées, on n'implantera pas de fonctionalité de zoom. L'unité des deux repères sera donc le *pixel*. Les *tailles* des rectangles seront donc identiques dans les deux repères, seules leurs origines vont différer. Attention cependant au renversement de l'axe vertical.
- Le `viewport_` représente la zone actuellement affichée par le visualisateur, *mais exprimé dans le système de coordonnées de la netlist*.

Note
Par définition, l'origine du `viewport_` exprimée dans le repère de l'écran est `(0,0)`. Sa *taille* étant identique.

- Les formules de conversion d'un repère à l'autre sont donc:
  1. *Netlist* (ou schéma) vers écran:
     ```
     Xscreen = Xschema      - X1_viewport
     Yscreen = Y2_viewport - Yschema
     ```
  2. Écran vers *netlist* (ou schéma):
     ```
     Xschema = Xscreen      + X1_viewport
     Yschema = Y2_viewport - Yscreen
     ```

Le `CellWidget` implémentera ces quatre formules de conversion dans les quatre méthodes suivantes:

```
inline int  xToScreenX ( int x ) const;
inline int  yToScreenY ( int y ) const;
inline int  screenXToX ( int x ) const;
inline int  screenYToY ( int y ) const;
```

Toutes les autres méthodes convertissant des points ou des boîtes (`Box`) devront y faire appel. On ne doit pas réécrire une formule de transformation à plusieurs endroits dans le code (augmente la fiabilité et facilite le déboguage).

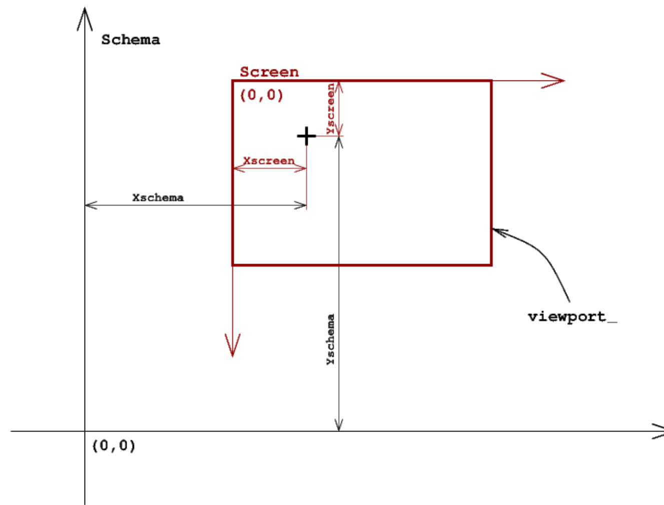

Figure 6: Figure 6 -- Coordonnées Graphiques (QT)

Conversion des Points & des Boîtes

Les coordonnées simples, dans la structure de donnée *netlist* comme dans QT sont exprimées avec des `int`. Les objets plus complexes ont en revanche des versions spécifiques différentes.

Netlist	**attributes**	**QT**	**attributes**
`Point`	`x, y`	`QPoint`	`x, y`
`Box`	`x1, y1, x2, y2` les coins inférieur droit et supérieur gauche	`QRect`	`x, y, w, h` Le coin supérieur gauche, la largeur (`w`) et hauteur (`h`)

D'où les déclarations des fonctions de conversion suivantes:

```
inline QRect   boxToScreenRect     ( const Box& ) const;
inline QPoint  pointToScreenPoint  ( const Point& ) const;
inline Box     screenRectToBox     ( const QRect& ) const;
inline Point   screenPointToPoint  ( const QPoint& ) const;
```

Gestion des Évènements

La gestion des évènements que l'on souhaite gérer au niveau du `CellWidget` se fait par la surcharge de méthodes spécifiques du `QWidget`.

Note

Le prototype, c'est à dire la signature *et* le nom de ces méthodes est fixé par le `QWidget`. Pour qu'elles fonctionnent il est donc *obligatoire* d'utiliser le même prototype (noms et arguments). Ce sont, bien ententu, des méthodes virtuelles.

Redimensionnement du `CellWidget`:

```
virtual void  resizeEvent ( QResizeEvent* );
```

Tracé ou retracé du contenu du `CellWidget`:

```
virtual void  paintEvent ( QPaintEvent* );
```

Traitement de l'appui sur une touche du clavier au dessus du `CellWidget`:

```
virtual void  keyPressEvent ( QKeyEvent* );
```

Pour mieux structurer le code, on écrira aussi des méthodes dédiées pour le déplacement du `viewport_` dans les quatre directions (haut, bas, gauche, droite). Le déplacement se fera par incrément de 20 pixels.

```
void  goLeft  ();
void  goRight ();
void  goUp    ();
void  goDown  ();
```

Récapitulatif de la Classe `CellWidget`

Notez que la déclaration de la classe n'est pas tout à fait complète.

```
class CellWidget : public QWidget {
    Q_OBJECT;
  public:
                     CellWidget         ( QWidget* parent=NULL );
    virtual         ~CellWidget         ();
            void     setCell            ( Cell* );
    inline  Cell*    getCell            () const;
    inline  int      xToScreenX         ( int x ) const;
    inline  int      yToScreenY         ( int y ) const;
    inline  QRect    boxToScreenRect    ( const Box& ) const;
    inline  QPoint   pointToScreenPoint ( const Point& ) const;
    inline  int      screenXToX         ( int x ) const;
    inline  int      screenYToY         ( int y ) const;
    inline  Box      screenRectToBox    ( const QRect& ) const;
    inline  Point    screenPointToPoint ( const QPoint& ) const;
    virtual QSize    minimumSizeHint    () const;
    virtual void     resizeEvent        ( QResizeEvent* );
  protected:
    virtual void     paintEvent         ( QPaintEvent* );
    virtual void     keyPressEvent      ( QKeyEvent* );
  public slots:
            void     goLeft             ();
            void     goRight            ();
            void     goUp               ();
            void     goDown             ();
  private:
    Cell* cell_;
    Box   viewport_;
};
```

Charte Graphique

On adoptera les conventions suivantes pour le tracé de la *netlist*.

Lorsqu'un node est connecté à plus de deux `Line`, on affiche un gros point bleu. Cela permet de distinguer entre un simple croisement de fils *sans connexion* d'une vraie connexion.

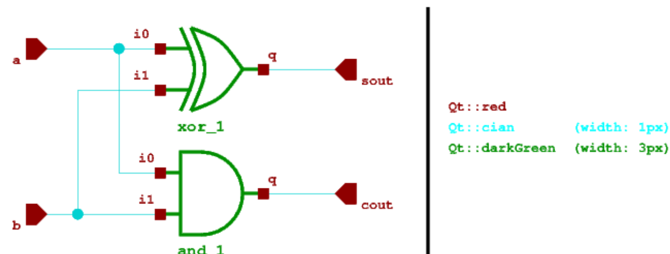

Figure 7: Figure 7 -- Exemple de Dessin de NETLIST

Travail à Réaliser

Calendrier indicatif:

TME	Widgets	Remarques
8	`CellViewer`	Fenêtre principale, avec la version dégénérée (fournie) du `CellWidget`
	`SaveCellDialog`	Dialogue de sauvegarde d'une `Cell`
	`OpenCellDialog`	Dialogue de chargement d'une `Cell`
9	`InstancesWidget`, `InstancesModel`	Fenêtre affichant la liste des instances et leurs modèles
	`CellsLib`, `CellsModel`	Fenêtre affichant la liste de toutes les cells chargées en mémoire
10	`CellWidget`	Widget affichant le dessin complet d'une `Cell`. On implantera progessivement l'affichage des différents éléments: 1. Dessin du symbole des instances. 2. Dessin des connecteurs des instances. 3. Dessin des connecteurs du modèle. 4. Dessin des fils. 5. Fonctions de déplacement (haut, bas, gauche droite).

Annexes

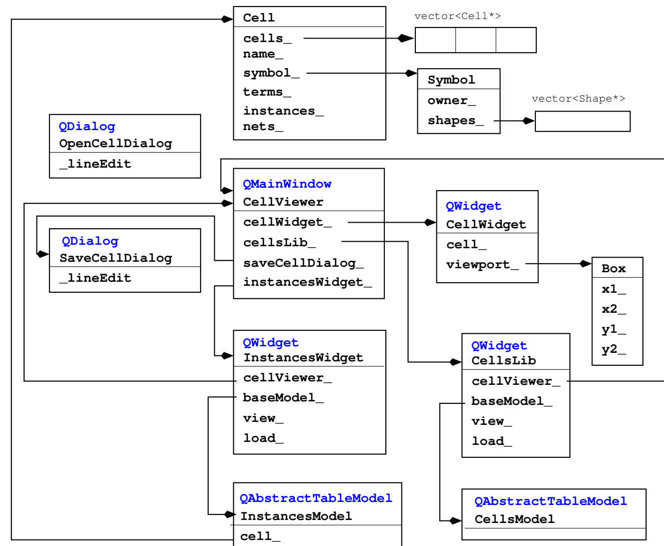

Figure 8: Figure 8 -- Relations entre classes graphiques

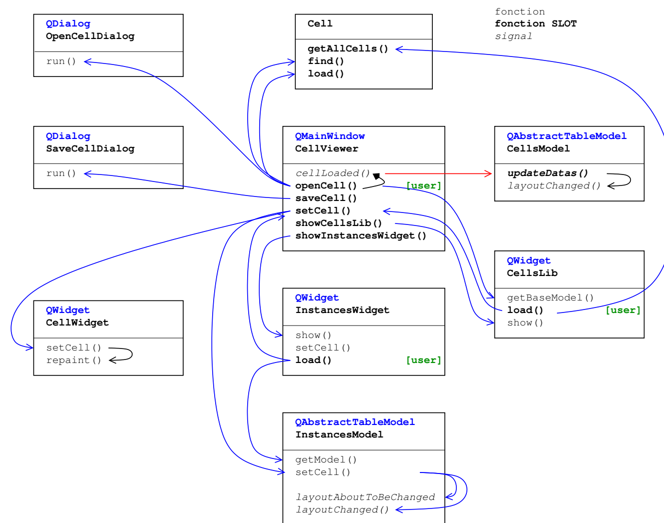

Figure 9: Figure 9 -- Relations signal/slot et fonctions

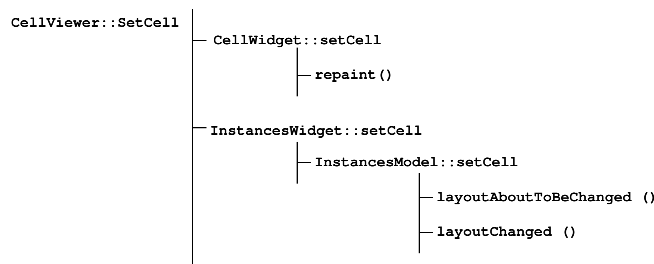

Figure 10: Figure 10 -- Distribution de la fonction setCell