Rapport de projet d'AP2

“Game of Life”

Institut Universitaire Technologique de l'Université de Bordeaux 1

Année 2011

Auteurs :

•Khalif Lahbibi
•Dimitri Sabadie

Game of Life : table des matières

I.Introduction

Game of Life est un jeu initialement inventé par **John Conway**. Il s'agit d'une simulation cellulaire qui ne nécessite l'intervention d'aucun joueur. Il ne s'agit donc pas d'un jeu à proprement parlé, mais plus d'un automate cellulaire.
Le principe est simple : les cellules sont réparties sur un quadrillage (représenté par une simple matrice), et on s'intéresse toujours à leur état (mortes / vivantes) à un instant t. Elles changent toutes d'état au même moment. On peut ainsi, à partir de règles extrêmement simples, déterminer l'état d'une cellule donnée à l'instant t+1.

Le but de ce projet est d'implémenter Game of Life en lui ajoutant quelques fonctionnalités supplémentaires (comme l'apparition de virus, la gestion de sauvegarde/chargement, etc …), et surtout, de gérer une **GUI** (**G**raphic **U**ser **I**nterface, ou encore Interface Graphique Utilisateur) basée sur le principe des **Widgets**. Il est évident que la complexité du projet réside surtout dans l'implémentation des Widgets, la partie algorithmique étant inexistante, et les règles de Game of Life extrêmement simples.

Aussi, ce projet suit une architecture dérivée d'une architecture logicielle réputée pour son efficacité : l'architecture **MVC** (**M**odèle – **V**ue – **C**ontrôleur). Ces trois blocs permettent d'implémenter indépendamment les données (dans notre cas, les règles de Game of Life), les vues (c'est à dire l'interprétation graphique du modèle), et le contrôleur (une sorte de passerelle entre utilisateur et le modèle et la vue). Afin de simplifier son implémentation, le projet impose de suivre une stratégie MV (**M**odèle – **V**ue). Le contrôleur est ainsi inclus dans la partie Vue du projet (c'est à dire, la GUI).

Ce présent rapport explique toutes les démarches qui ont été prises afin de concevoir le projet. Il n'est en aucun cas une documentation, bien qu'il contienne du code. Tout code écrit dans ce présent rapport suivra une police bien précise afin de le distinguer : `ceci est du code`.

II. Le modèle

A. Les cellules

Le modèle regroupe plusieurs classes. Tout d'abord, la classe la plus représentative du jeu est tout simplement Cell (cellule). Une cellule a un comportement très limité : elle peut naître (`bear()`), mourir (`die()`) et retourner son état (`isAlive()`). Ce sont les uniques primitives d'une cellule de Game of Life. Il est important de noter que la méthode constante `isAlive()` a été la première implémentation de l'état d'une cellule. Cependant, nous verrons plus loin dans le présent rapport que c'est sujet à changement (notamment lorsque les virus incubent des cellules …).

Bien qu'une cellule soit extrêmement simple, les primitives qui lui sont associées sont

amplement suffisantes pour pouvoir supporter la simulaton. Afin de faciliter la gestion de la matrice de celllules, une nouvelle classe a été implémentée : la class `GameModel`. Elle représente tout le modèle. Elle contient donc des agrégats par composition, tel qu'une matrice de Cell. Il est possible, grâce à ses méthodes publiques telles que `setCol()` ou `setRow()` de redimensionner à la volée les dimensions de la matrice de simulation. Elle possède aussi une méthode public `next()` qui permet de passer à l'étape suivante de la simulation.

Pour passer d'une matrice de cellules donnée à instant t à une matrice de cellules à l'instant t+1, une méthode privée est nécessaire : `check8NeighAlive()`. Cette méthode très pratique retourne le nombre de cellule voisines vivantes d'une celulle donnée. Avant de continuer, il faut expliquer les règles de Game of Life :

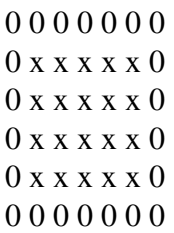
- si une cellule est vivante
  - elle reste vivante si elle possède 2 ou 3 voisines vivantes
  - elle meurt sinon
- si une cellule est morte
  - elle nait si elle possède exactement 3 voisines vivantes
  - elle reste morte sinon

Ces règles extrêmement simples doivent être appliquées à chaque cellule. Evidemment, on se rend compte qu'il n'est pas possible d'effectuer la mise à jour d'une matrice en place (c'est à dire sans consommer de mémoire supplémentaire) : une deuxième matrice est nécessaire afin d'effectuer la transition. Il suffit alors, pour chaque cellule, d'appeler le méthode privée `check8NeighAlive()`, et d'appliquer les règles afin de déterminer l'état de la cellule.

Intéressons-nous maintenant à cette méthode `check8NeighAlive()`. Il existe plusieurs façons de déterminer le nombre de cellules voisines d'une cellule donnée. La complexité de cet algorithme réside surtout dans le fait de limiter les comparaisons, notamment sur les indices, qui peuvent rapidemment devenir coûteuses. La première idée est, pour une cellule donnée d'indice (i,j), de faire une disjonction de cas : on commence par regarder si la cellule se situe à la première ligne. Si c'est le cas, on sait que la cellule a *au moins* 1 voisine (celle qui se situe juste en dessous d'elle). Sinon, on regarde si la cellule est sur la dernière ligne. Si tel est le cas, on sait qu'elle a une voisine juste au dessus d'elle. Sinon, elle se situe forcément au milieu, elle a donc une voisine au dessus et une voisine au dessous.
Pour chaque cas, on effectue le même procédé sur les colonnes. On obtient ainsi un code de if imbriqués, qui n'est pas facilement maintenable. Il reste cependant d'une bonne exécution puisqu'il écarte plusieurs possibilités d'un coup (si une cellule se situe, par exemple, dans le coin supérieur gauche, on a déjà 5 cellules d'écartées).
Le problème de cet algorithme réside dans le fait que les branchements dynamiques des if ne sont pas réellement justifiés. En effet, sur une matrice de 800x600 cellules, on a 2*800 + 2*(598) = 2796 cellules sur les bords, pour un total de 800 * 600 = 480 000 cellules. On se rend donc facilement compte que pour 477 204 cellules, des branchements dynamiques seront effectués … pour rien, étant donné que toutes ces cellules ont 8 voisines, vivantes ou pas. Un autre algorithme est donc nécessaire.

L'algorithme que nous avons trouvé demande de modifier un petit peu la matrice de cellules. Ainsi, nous avons besoin d'une matrice de (n+1)*(n+1) cellules. Nous consommons donc plus de mémoire avec cet algorithme, mais c'est négligeable. L'idée est d'inscrire notre matrice effective dans un cadre de cellules mortes (qui elles participent à la simulation mais ne sont pas affichées,

évidemment). On peut représenter cela à l'aide de ce schéma :



Ici, les x représentent les cellules effectives. On se rend alors compte que *toutes* les cellules effectives ont désormais 8 voisines, et qu'il n'est plus nécessaire de vérifier la validité des indices.
Cet algorithme nous offre donc une solution beaucoup plus élégante, car il est extrêmement simple de vérifier si les 8 cellules d'une cellule sont vivantes ou non (deux boucles suffisent). Voici la méthode publique `check8NeighAlive()` :

```
int GameModel::check8NeighAlive(int i, int j) const {
    int nb = 0;

    for (int p = 0; p <= 2; p += 2) {
        for (int k = 0; k < 3; ++k)
            nb += _cells[i-1+p][j-1+k].isAlive(); // hack
        nb += _cells[i][j-1+p].isAlive(); // hack
    }

    return nb;
}
```

On remarque par ailleurs la présence d'un hack qui nous permet d'écarter tout branchement dynamique : le cast implicite du booléen vers l'entier (`(int)true == 1` et `(int)false == 0`).
Cet algorithme est bien plus efficace que le précédent car il ne nécessite plus *aucun* branchement dynamique (ce qui n'est pas négligeable) étant donné que `isAlive()` retourne un simple booléen (et n'effectue pas de branchement dynamique non plus).
Cependant, on se rend là encore compte qu'il est possible de faire encore mieux. En effet, pour un certains nombre de cellules, on sait à l'avance combien de cellules sont visitables. Et notre algorithme n'en tient pas compte (pour les cellules sur les bords, on va aller lire l'état de cellules qui n'existent en réalité pas).
Nous nous sommes contentés de cet algorithme (car très efficace même avec un nombre élévé de cellules).

Finalement, il reste une chose à savoir sur les cellules. Il est possible de les paramétrer selon 4 paramètres :
- `minAlive`
- `maxAlive`
- `minDead`
- `maxDead`

Derrière ces 4 termes un peu étranges se cachent en réalité nos fameuses règles de Game of Life ! Les variables `{min,max}Alive` permettent de définir la condition de survie d'une cellule. On pourrait

traduire cela par : « Une cellule vivante reste en vie tant qu'elle a entre `minAlive` et `maxAlive` voisines en vie, sinon elle meurt ». Ainsi, si `minAlive = 2` et `maxAlive = 4`, une cellule vivante possédant ayant 2, 3 ou 4 voisines vivantes restera en vie. Si elle en possède strictement moins de 2 ou strictement plus de 4, elle meurt.
Les variables {`min`,`max`}`Dead` définissent la condition de naissance de cellules mortes. On peut traduire cela par : « Une cellule morte naît si elle a entre `minDead` et `maxDead` cellules vivantes voisines, sinon elle reste morte ». Par exemple, si `minDead = 3` et `maxDead = 3`, une cellule morte ayant exactement 3 voisines vivantes pourra naître, sinon elle restera morte.

Le paramétrage de ces deux règles peut parfois devenir complexe et générer des motifs extrêment complexes (pseudo-labyrintes par exemple). Aussi, veuillez noter que vous n'avez aucune contrainte d'intégrité sur les paramètres. Ainsi il vous est possible de définir des paramètres contradictoires, comme par exemple cette configuration :
•`minAlive = 3`
•`maxAlive = 2`
Ici, cette configuration génère la mort de toute cellule, où qu'elle soit. On peut par le même procédé empêcher la mort, empêcher la naissance, etc etc …

A.Les virus

Nouveauté par rapport au jeu de la vie de Conway, les virus ont été implémentés à notre jeu de façon à ajouter de nouveaux paramètres intéressants à la simulation. Ils ont un comportement complètement chaotique. En effet, ils bougent dans n'importe quelle direction et en changent à intervalles aléatoires. Aussi, ils ne s'inscrivent pas dans une “matrice” comme le sont les cellules : leur mouvement est libre et fluide, ce qui implique que contrairement aux cellules, les virus ont la capacité de se mouvoir dans l'espace de la simulation.

La première caractéristique notable des virus est leur caractère chaotique et agressif. Dans notre simulation, nous avons choisi d'implémenter les virus de cette façon afin de donner un peu de punch au jeu. L'utilisateur peut si il le souhaite faire apparaître un virus aléatoirement dans la simulation, enlever le dernier ajouté, ou enlever tous les virus de la simulation. Nous allons détailler dans l'ordre ces trois possibilités.

Tout d'abord, l'ajout de virus. Les virus sont définis dans une classe `Virus` qui fournit plusieurs méthodes publiques, ainsi qu'un constructeur paramétré prenant un `Frame` et un `float` en paramètres. Le `Frame` permet de spécifier les limites de déplacement du virus. Ce sera en l'occurence la zone de simulation. Le `float` spécifie la vitesse du virus. Ce paramètre permet d'adapter la vitesse des virus en fonction du nombre de lignes ou de colonnes par exemple.
Lorsqu'un `Virus` est instancié, son constructeur paramétré appelle une méthode privée nommée `changeDirection()` qui permet de définir une première direction aléatoire du virus. Cette fonction se charge aussi de définir une *fréquence* aléatoire, laquelle correspond en fait à la fréquence de changement de direction. Ainsi, à chaque changement de direction, une nouvelle fréquence est affectée. Grâce à ce procédé, on aura ainsi des virus qui pourront changé une fois de direction, attendre 100 ms pour changer de direction, changer de direction, puis attendre 58 ms, changer de direction, attendre 1,5 secondes, changer de direction, etc etc ... Cela amplifie donc grandement le caractère chaotique et imprévisible des virus.

Les virus sont placés dans une liste (`std::list`). Pourquoi le choix d'un tel containeur ? C'est bien simple : nous n'avons pas besoin de mémoire contiguë pour les virus, nous n'avons pas besoin de *random-access* sur les virus, nous avons uniquement besoin d'itérer sur l'ensemble. Une liste est donc parfaitement adaptée pour ce travail ! On pourrait penser qu'un `std::vector` serait aussi un bon choix, cependant ce containeur est bien plus lent et gourmand pour le travail requis ici. En effet, des réallocations peuvent avoir lieu, et la mémoire est gaspillée. Nous n'avons pas ce problème avec les listes, qui ont définitivement un bien meilleur rapport complexité temp/mémoire.
Ainsi, pour supprimer le dernier virus ajouté par l'utilisateur, un simple `pop_back()` de la liste suffit. De même, la suppression de l'intégralité des virus se fait de façon instantanée via la méthode publique `clear()` de `std::list`.

Abordons maintenant la partie déplacement. Lorsque les virus doivent bouger, il suffit d'appeler leur méthode `move(Frame const &limits)` pour les déplacer. Ici encore, `limits` permet de limiter la zone de déplacement. Une horloge est implémentée de façon à appeler la méthode `move()` de l'intégralité des virus au bout d'un certain moment (dépendant de la vitesse de génération). On remarque ici qu'une simple itération suffit pour mettre à jour les virus :

```
if (_virusClock.GetElapsedTime() >= _speed/50) {
    VirusList::iterator end = _viruses.end();
    for (VirusList::iterator it = _viruses.begin(); it != end; ++it)
        it->move(_frameBG);
    _virusClock.Reset();
}
```

Le dessin des virus se fait de façon similaire, mis à part le fait qu'il faut bouger un sprite et le dessiner pour chaque virus. Qui plus est, les cellules sont 1.5 fois plus gros que les virus.

Les virus possèdent aussi la capacité de se dupliquer après une incubation. Cette caractéristique est paramétrable (entre 1 et 8 enfants). Les premiers tests de cette fonctionnalité ont montré un énorme problème. En effet, les enfants étaient dans la première version de l'algorithme créés à la même position que le virus père. Cela devenait très vite problématique dans le cas d'une incubation de cellules “sécurisées” comme le montre ce schéma dans la configuration par défaut du jeu :

C C
C C

Ici, un seul virus ne peut pas casser cette configuration, car une fois qu'il a tué une cellule, elle renaît immédiatement (du fait de la présence des 3 autres cellules). Ce problème a été résolu en ajoutant un facteur de dispersion aux enfants. Ce facteur permet de faire apparaître les enfants dans un certain rayon du virus qui les a créés. Ainsi, il est maintenant possible pour un seul virus de casser la configuration vue plus haut.

Une fois cette fonctionnalité implémentée, des tests de “grande envergure” ont été menés (un seul virus laché dans un nuage de grand nombre de cellules). Les résultats ont été imparables : le nombre de virus générés croit exponentiellement et devient donc très rapidement trop grand pour permettre un jeu agréable à utiliser. Cela est tout simplement du au fait du non-recyclage de virus : des virus sont générés, mais aucun ne meurt ! Voila donc la dernière “fonctionnalité” des virus : leur mort.

Dans notre jeu, un seul facteur peut entrainer la mort d'un virus : la famine. L'idée est ici basique : tant qu'un virus n'incube pas, il s'affaiblit. Si il est trop faible, il meurt. Dès qu'il commence à incuber une cellule, il retrouve toute sa force. Evidemment, si l'incubation est un succès, le virus se remet à s'affaiblir. Ainsi, si vous faites appaître un virus dans une zone de simulation entièrement démunie de cellule, le sort de ce virus est directement scellé : il mourra très rapidemment. En revanche, un même virus peut rester en vie pendant très longtemps si la population en cellule est élevée. Il est facile de jouer sur tous les paramètres du jeu pour observer le comportement des virus vis à vis des cellules.

III.La vue

A. Les Widgets : définition

Dans cette partie, nous allons aborder les notions mises en oeuvre permettant d'afficher Game of Life, et, ainsi, de proposer une interface agréable avec l'utilisateur.

Dans notre programme, l'utilisateur intéragit via des Widgets. Les Widgets sont des objets graphiques permettant à l'utilisateur d'intéragir avec la simulation. Il est important de bien faire la distinction entre un Widget et ce qu'il représente. L'architecture MV nous oblige à faire une telle distinction, de façon à rendre indépendantes les deux portions de code. Ainsi, on pourra modifier l'aspect graphique d'un bouton sans pour autant modifier ce que représente le bouton. C'est là tout l'avantage du modèle MV.

B. Les Frames

Avant de rentrer dans les détails d'implémentation des Widgets dans notre programme, il faut d'abord parler d'une classe utilitaire qui est le pillier des Widgets. Sans cette classe, l'implémentation de Widgets dynamiques serait très certainement chaotique voire sous-optimale. Cette classe se nomme `Frame`. Comme son nom l'indique, elle représente … un simple cadre (ou une « boite englobante »). Elle nous permet donc, par essence, d'effectuer deux opérations principales : spécifier la position du cadre à l'écran (via les méthodes publiques `setPX()` et `setPY()`) et redimensionner le cadre (`setWidth()` et `setHeight()`). Evidemment, des accesseurs en écriture seule `get*()` sont disponibles pour récupérer les informations sur le cadre.
Tout ça pour quoi ? A l'aide de cette classe, on peut très facilement commencer à positionner et redimensionner des Widgets à l'écran. `Frame` compose tout simplement notre classe Widget de base (voir la partie Error: Reference source not found). Mais ce n'est pas tout. `Frame` possède une méthode public permettant d'optimiser grandement la gestion des Widgets : `focused() const`. Cette méthode prend en paramètre un `sf::Event` ou un `sf::Input` et retourne `true` si la souris se trouve dans le `Frame`, `false` sinon. L'intérêt principal de cette méthode vous apparaîtra un peu plus bas dans ce document.

C.Les Widgets : hierarchie d'héritage et designs patterns

Maintenant que vous êtes au courant pour `Frame`, nous allons pouvoir entrer dans le vif du sujet : les Widgets. Les Widgets sont par exemple des boutons cochables, du texte, des sliders, des onglets, mais aussi des pages ou des panneaux de configuration. On se rend vite compte que l'on peut

classer nos Widgets en deux catégories :
•les Widgets contenant d'autres widgets
•les Widgets ne contenant aucun autre widget
Cette appréciation est importante. En effet, des Widgets tels que les sliders ou les pages doivent pouvoir contenir d'autres Widgets. Cependant, ils sont eux-même des Widgets. Avant d'aller plus loins, nous parlerons désormais de Widget terminal ou Widget composant pour tout Widget ne contenant aucun autre Widget, et de Widget composite pour tout Widget contenant au moins un autre Widget (qu'il soit terminal ou composite). En réalité, je ne vous ai pas tout dit. Nous sommes ici en train de mettre en oeuvre un **design pattern** (**DP**) nommé **Composite**. Ce DP permet de manipuler un ensemble d'objets appartenant à une hierarchie en manipulant la classe la plus abstraite de la hierarchie. Il est donc parfaitement adapté pour représenter une arborescence, et c'est ici notre cas. Nous appelerons donc la classe la plus abstraite `WidgetBase`. Afin d'implémenter efficacement le DP, il nous faut une autre classe abstraite - celle représentant les Widgets composite : `WidgetComposite`. Evidemment, `WidgetComposite` hérite publiquement de `WidgetBase`.

Il est important de comprendre que nos Widgets auront tous au moins un agrégat (hérité protégé via `WidgetBase`) : le `Frame` représentant les dimensions et la position du Widget. On voit par exemple que nous en aurons besoin pour positionner un bouton à l'écran, un slider, un numeric bar, mais aussi un Widget composite, comme par exemple une page.

Afin d'harmoniser les Widgets et la zone de simulation, il faut englober cette dernière dans un `Frame`. Ainsi, si ce cadre a le focus (`focused()`), il sera inutile de traiter les Widgets puisque la souris sera occupée ailleurs. Cette approche porte un nom simple : le partionnement de l'espace. Ici, le partionnement est très simplifié, mais il reste très efficace.
Afin de ne pas avoir de « trou » entre deux Widgets, nous utilisons le principe du tiling. Le tiling, dans le domaine des Window Managers, est le fait qu'une fenêtre occupe tout l'espace qui lui est possible d'occuper. A chaque ajout de fenêtre, l'espace est coupé d'une façon précise, et les fenêtres sont redimensionnées afin de s'adapter l'une à l'autre. Nous utilisons ici une approche similaire pour les Widgets composite. Lorsqu'un Widget A est ajouté dans un autre Widget X, X est chargé de réorganiser tous les Widgets terminaux qu'il a déjà, et d'ajouter A en conséquence. Par exemple, dans le cas d'une list d'onglets, les Widgets sont ajoutés de façon simple :
1.la méthode public `X::add(WidgetBase *pA)` est appelée
2.`X` ajoute pA a ses enfants
3.`X` réorganise les Widgets. Le principe est simple : `X` possède le nombre de widgets qui sont désormais dans la liste d'onglets (i.e ses enfants). Il divise alors la longueur de son propre cadre par le nombre de widgets : `getFrame().getWidth() / nbWidgets`. Ainsi, il va pouvoir redimensionner tous les widgets par cette nouvelle valeur, et les repositionner par rapport à son cadre
On voit ainsi que cette approche est extrêmement puissante. En effet, lorsqu'un onglet est ajouté à une liste d'onglet, il n'est plus nécessaire de gérer la position ni la dimension de l'onglet : la liste d'onglets fait *tout* le travail !

Ce principe s'applique à tous les Widgets composite, bien qu'il varie d'un Widget à l'autre. Par exemple dans le cas des pages, aucune réorganisation préalable n'est effectuée à chaque ajout d'un Widget : le Widget est tout simplement ajouté sous les autres Widgets dans la page, et il centré selon sa longueur. Chaque Widget composite a sa propre règle de positionnement / redimensionnement à chaque ajout / mise à jour du Widget parent.

Nous avons donc vu que nous pouvons former de façons très simple des Widgets complexes. L'affichage de ces objets est simple : chaque Widget *peut s'il le souhaite* redéfinir la méthode virtuelle `WidgetBase::draw(sf::RenderWindow*) const`. Cette technique permet de faire du polymorphisme, et ainsi d'avoir un comportement différent selon que l'on dessine une page ou un bouton par exemple.

Parlons maintenant de la gestion des évènements. Il faut distinguer deux types de traitement évènementiel dans notre programme : les évènements qui doivent descendre la hierarchie d'héritage, et les évènements qui doivent remonter ou sortir de la hierarchie afin d'être capturé par d'autres objets (donc pas nécessairement des Widgets). Nous allons dans un premier cas expliquer le fonctionnement du premier type de traitement, puis nous verrons le second.

a.Traitements descendant la hierarchie

Le principe est simple : `WidgetBase` possède une méthode virtuelle `relayEvents(const sf::Event &event, const sf::Input &input)` qui retourne `true` si le Widget a traité l'évènement, `false` sinon. Par défaut, cette méthode vérifie simplement si le curseur de la souris se situe dans le `Frame` du Widget. Par définition, tout Widget qui vérifie cette condition doit retourner `true` pour dire que l'évènement a été traité. En effet, les Widgets ne s'interposent pas, ce qui signifie que si le curseur de la souris se situe dans un Widget, il ne peut pas se situer dans un autre Widget au même instant.
Il y a deux façons de traiter un évènement : soit le Widget est un terminal, et il proposera alors sa propre version de `relayEvents`() basée sur son propre type (par exemple un bouton va vérifier l'état du bouton gauche de la souris et changer l'aspect et la valeur du bouton en conséquence), bien que la plupart du temps il se contentera de `WidgetBase::relayEvents()`, soit le Widget est un composite, et là il proposera une version de `relayEvents()` un peu plus complexe, mais dans tous les cas, il appelera récursivement la méthode public `relayEvents()` de ses enfants. L'avantage de cette technique est de localiser précisément un évènement, et d'écarter tous les Widgets qui ne sont pas concernés. Et évidemment, on voit ici l'intérêt de la méthode public de `Frame` dont je parlais plus haut, `focused()`. Cela nous permet de tirer profit du partionnement spacial effectué par les `Frame`. Un autre avantage réside dans le fait que la zone jouable (i.e la zone où la simulation se déroule) possède son propre `Frame`. Ainsi, si la souris se situe dans ce `Frame`, il est inutile d'aller effectuer des tests et traitements sur la partie Widget !

b.Traitements remontant ou sortant de la hierarchie

Maintenant, imaginons que nous voulions lancer une certaine *action* lorsqu'un bouton est enfoncé. Ou alors, que nous voulions, à partir d'une Widget composite, surveillé le comportement des enfants afin de modifier le composite en conséquence. La solution de traitement descendant la hierarchie devient ici obsolète. En effet, cette solution est *statique* : le traitement est unique pour chaque Widget. Qui plus est, il ne peut se propager que versle bas. Par exemple, lorsque l'on clique dans une barre numérique, le traitement descendant est tout simplement de faire descendre l'évènement jusqu'aux enfants. Dans le cas d'une barre numérique, on aimerait pouvoir décrémenter la valeur lorsque l'on clique sur le bouton *moins* de gauche et l'incrémenter lorsque l'on clique sur le bouton *plus* de droite. On ne peut pas parvenir à ce but avec les traitements descendants. On a donc besoin d'une solution plus souple nous permettant de *paramétrer* les actions à réaliser par les Widgets.

En réalité, on aimerait pouvoir *surveiller* des objets. Cela nous permettrait d'éviter des branchements dynamiques. Par exemple, on aimerait pouvoir lancer un certain type d'action lorsqu'un toggle-button change d'état, ou bien lorsque la valeur d'un slider change.
La solution la plus élégante à ce problème réside là encore dans … un design pattern ! Et ce design pattern, c'est le DP **Observer** (ou **Listener**). Ce DP extrêmement simple mais extrêmement puissant permet de paramétrer le comportement à adapter lorsqu'un certain type d'évènement survient. Une classe susceptible de lever une notification se voit en mesure d'accepter par une simple agrégation un objet de type Listener. Lorsque la classe doit emettre une notification, elle vérifie si un Listener lui est attaché. Si c'est le cas, elle appelle la notification sur ce Listener, sinon elle ne fait rien.

Voici pour l'exemple comment est implémenté le changement de valeur d'un slider :

```
bool Slider::relayEvents(sf::Event const &event, sf::Input const
&input) {
    if (WidgetBase::relayEvents(event, input)) { // evenement capture
: traitement
        if (input.IsMouseButtonDown(sf::Mouse::Left)) {
            // on doit deplacer le curseur en consequence et modifier
la valeur du slider
            float px = input.GetMouseX();
            float d = _max - _min;
            float w = _frame.getWidth() / d;
            int value = (px - _frame.getPX() + w/2) / w + _min; //
valeur pointee par le curseur
            setValue(value);

            // notification
            if (_pListener)
                _pListener->notifyChangedValue(*this);
        }
        return true;
    }

    return false;
}
```

On remarque ici la présence de deux lignes pour gérer la notification. `_pListener` est un pointeur sur un `Slider::Listener`. Un `Listener` peut être spécifié dans le constructeur ou à l'aide d'une méthode publique annexe `setListener()` (cependant cette méthode n'est pas présente dans le cas d'un Slider car on n'en a pas réellement besoin). La classe de base des `Listener` pour les `Slider` est déclarée comme suit :

```
class Listener {
    public :
        Listener(void);
        ~Listener(void);

        // signaux ici
```

```
        virtual void notifyChangedValue(Slider &sender) = 0;
};
```

La méthode public virtuelle `notifyChangedValue()` doit être obligatoirement redéfinie lorsque l'on crée un `Listener`, et aucune implémentation par défaut n'est fournie. Si l'on veut par exemple faire varier la vitesse de génération des cellules, on pourrait écrire ce `Listener` :

```
// .hpp
class SpeedListener : public Slider::Listener {
private :
    GameView *_pGV;

public :
    SpeedListener(GameView *pGV);
    ~SpeedListener(void);

    void notifyChangedValue(Slider &sender);
};


// .cpp
SpeedListener::SpeedListener(GameView *pGV) :
    Slider::Listener(),
    _pGV(pGV) {
    assert(pGV);
}

SpeedListener::~SpeedListener() {
}

void SpeedListener::notifyChangedValue(Slider &sender) {
    float s = (sender.getMax() - sender.getValue()) / 1000.f;

    if (s <= 0.f)
          s = 0.001f; // limite
    _pGV->setSpeed(s);
}
```

On remarque parfaitement bien ici qu'il est extrêmement simple de paramétrer le fonctionnement d'un Widget à l'aide du design pattern **Listener**. Si nous passons un `SpeedListener` au constructeur de notre `Slider`, et si le `SpeedListener` a un pointeur vers un `GameModel` valide dans son constructeur, alors lorsque le curseur du `Slider` bougera, la méthode publique `notifyChangedValue()` sera appelée (cela est possible grâce au polymorphisme), et ainsi, la vitesse du jeu variera !

Les `Listener` sont donc de très bons amis pour gérer le paramétrage précis des Widgets. On y voit un seul inconvénient : l'explosion du nombre de classe pour effectuer de petits traitements. Cependant, comme vu plus haut, la simplicité de ces classes rend ce travail trivial.

Il reste une chose qui peut-être intéressant de détailler : la déclaration des `Listener` abstraits. Par soucis de lisibilité, il est plus agréable de dériver son propre listener de `Slider::Listener` plutôt que de `SliderListener`. Comment ce mécanisme est-il introduit ? Il s'agit tout simplement de sous-classes publiques, le Widget jouant un peu le rôle d'un `namespace` :

```
class Slider : public WidgetBase {
public :
    class Listener {
    public :
        Listener(void);
        ~Listener(void);

        virtual void notifyChangedValue(Slider &sender) = 0;
    };

    // … suite de la classe Slider
};
```