Cours IHM

L. Buzer & B. Perret – 2017 - v0.95

1. Introduction

Une Interface Homme Machine (**IHM**) ou **GUI** (Graphical User Interface) définit les moyens mis en place pour que l'utilisateur (humain) puisse contrôler et communiquer avec la machine. Les termes équivalents **composants** et **contrôles** définissent les éléments de l'IHM : boutons, cases à cocher, boites d'édition, boites de défilement…

Pour désigner la fenêtre de votre application, on utilise le terme anglais « **Form** » littéralement « Formulaire » plutôt que le terme Window ! Le framework de .net pour la partie IHM porte ainsi le nom de « Windows Forms ». Il succède à l'ancien système MFC (Microsoft Fundation Classes) écrit en C++ et plus complexe à manipuler.

2. MVC : Modele-Vue-Contrôleur

Le modèle MVC s'impose naturellement dans les environnements Web (HTML/JS/MySQL). MVC, design pattern très utilisé dans l'industrie, prône la séparation entre :

- le MODELE de données (MySQL) stockant l'information
- la VUE c'est-à-dire l'interface affichée à l'écran (~HTML)
- le CONTROLEUR (~JavaScript) représentant la logique et les interactions entre les données et la vue

Cependant, dans les applications postes (backend et ingénierie), contrairement aux technologies du web en perpétuel changement, les composants d'une application poste sont aujourd'hui de grands classiques et leur fonctionnement est stable depuis une vingtaine d'années.

Ainsi dans les librairies d'IHM : .net, SWING, JFX… Les parties Vue et Contrôleur ont été fusionnées à l'intérieur des composants. Ainsi un composant bouton contient son apparence, les modifications de son apparence ainsi que sa logique de traitement : que se passe-t-il lorsque l'on clique avec le bouton gauche ? et avec le bouton droit ? Grâce aux techniques de la POO, on peut alors spécifier un composant en lui ajoutant des comportements ou des propriétés supplémentaires. Par exemple, on lui ajoute un écouteur spécifique qui réagit à l'appui sur une touche particulière pour créer un raccourci clavier.

L'inconvénient des librairies d'IHM est que vos applications vont avoir un look and feel très similaire. On reconnaît ainsi facilement le style WIN32 / QT / SWING …

3. Notion de processus et de thread

Un programme reçoit une portion de mémoire propre. Cette partie de la mémoire lui est réservée et seul lui peut lire ou écrire dans cette portion de mémoire. Ainsi, un programme

aura interdiction d'accéder à toute donnée en mémoire appartenant à un autre programme. Un programme ainsi conçu ne peut donc pas partager ses données avec d'autres. On parle de **processus**.

Un processus peut contenir un ou plusieurs **threads** s'exécutant en quasi-simultanéité ou simultanément. A la différence des processus, les threads partagent le même espace mémoire et les mêmes ressources.

Exemple d'un programme multithread : un lecteur vidéo. L'image et le son sont gérés par deux threads différents. Il faut lire ces informations, les décoder et les transmettre. On peut avoir une priorité haute pour le thread son pour éviter toute saccade et une priorité moindre pour le thread du décodage vidéo.

4. Les évènements

Dans un système d'exploitation moderne, c'est l'OS qui gère tous les évènements. Il se charge de déterminer quel programme ou quel service est concerné et l'OS lui envoie alors un message pour lui indiquer « qu'il se passe qqc » et que l'application doit agir en conséquence. Par exemple, l'appui sur le bouton éteindre de votre PC déclenche chez l'OS l'envoi du message « vous devez vous terminez » à l'ensemble des applications en cours. Le terme **message** est plus général (arrivée d'une trame réseau), le terme **évènement** est plus usité pour la partie interface utilisateur (clavier, souris, joystick).

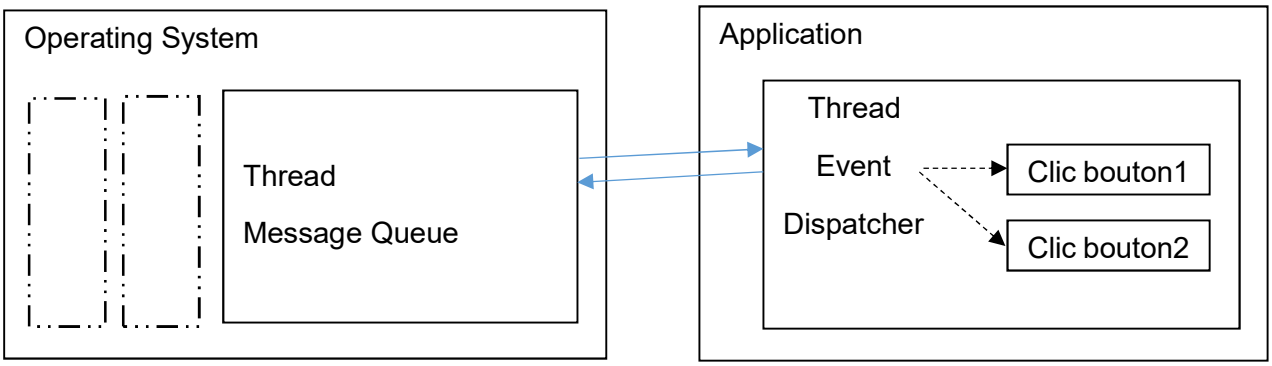
5. La pile des messages – Message queue

Attention, entre ChromeOS, Android, Windows32, Windows .net, OS X, chacun possède son propre verbiage et sa propre stratégie de gestion des évènements. Ainsi la pile des messages porte différents noms dans la littérature : Message queue, Event loop, Message loop… La pile des messages de l'OS est une file d'attente qui stocke les messages au fur et à mesure de leur arrivée. Elle peut aussi gérer des stratégies : message à traiter en priorité ou optimisation des messages de même type. Par exemple, imaginons que l'OS reçoive 100 fois en 1/20 de seconde le message : « réafficher la fenêtre X », l'OS n'enverra qu'un message « Affiche ta fenêtre » à l'application concernée par soucis d'optimisation des performances.

6. Le traitement d'un évènement – Event Dispatcher

On parle aussi de Message Pump. Contrairement à une application console où la fonction main joue le rôle de point d'entrée du programme, dans une application fenêtrée, il va s'agir de la boucle de l'Event Dispatcher. Elle joue le rôle de thread principal de votre programme :

- Le thread de l'Event Dispatcher (EDT) se met en attente d'un message provenant du système. Le thread est gelé et ne consomme pas de ressource CPU.
- Lorsque l'OS le décide, il traite l'évènement en première position dans la pile des messages.
- L'OS recherche l'application concernée et transmet alors un message à son EDT
- Le Thread EDT se réveille et reçoit l'évènement.
- Pour chaque contrôle concerné : appel de la fonction associée – phase de Dispatch



Les flèches en trait plein au centre désignent des flux de messages entre l'OS et l'EDT. Attention ce ne sont pas des appels de fonctions ! Les flèches en pointillés désignent un appel de fonctions de l'EDT à des parties du code du programme.

1. Exemple pour un clic souris – Dispatch dans l'arbre des composants

L'OS détecte qu'un clic souris s'est produit à l'intérieur d'une fenêtre, l'OS n'a pas connaissance des boutons et autres composants à l'intérieur de cette fenêtre, elle stocke donc les informations brutes dans la file d'attente : clic souris, positions x et y à l'écran. A un moment, l'EDT d'une application va récupérer ce message et le traiter :

- Arrivée de l'évènement sous sa forme brute : clic souris, posX, posY
- Recherche du ou des composants qui va intercepter cet évènement.
- Une fois le composant connu, appel de sa fonction Button_clic dans le thread de l'EDT

a) Par curiosité

A quoi ressemble le code contenu dans le thread de l'EDT ? En fait, historiquement, il est assez simple de conception car basé sur une simple boucle :

```
Do
{
        OS.WaitForMessage();      // gel du thread courant
        Msg = OS.GetMessage();
        DispatchMessage(Msg);
}  While ( Msg != OS.Message.ExitCode );
CloseWindow();
```

7. Cas pratiques associés à l'Event Dispatcher

a) Blocage

Prenons l'exemple de cette fonction gérant le clic souris sur le bouton1 :

```
Button1click()  {  While (true) { … }  }
```

Quel est le problème ?

- ⇨ Le thread principal de l'EDT boucle à l'infini
- ⇨ Plus d'affichage
- ⇨ Plus de gestion des évènements souris/clavier

L'application est « bloquée » sans possibilité de la sauver.

Attention, il n'y a qu'un seul thread « Event dispatcher » qui « prend » un à un chaque évènement dans la liste et appelle ensuite pour chacun la fonction adéquate. Si une fonction appelée par l'EDT bloque ou boucle à l'infini, l'exécution ne remontera jamais à la boucle principale de l'Event Dispatcher et votre application est inopérationnelle.

Ne faites pas l'erreur de croire que chaque clic sur un composant déclenche un traitement dans un thread indépendant. Il s'agit bien du thread principal qui appelle une fonction pour gérer l'évènement. En conclusion, rappelez-vous que le blocage du traitement d'un évènement entraine le blocage complet de l'interface !

b) Commande Invalidate

Une fenêtre n'a pas le pouvoir de s'afficher à l'écran !!! En effet, cette responsabilité appartient à l'OS. En réfléchissant, ceci est logique car par exemple, lorsque la fenêtre est recouverte par une autre application, elle ne le sait pas. Seul l'OS connait la position de l'ensemble des fenêtres et détermine qui peut être affiché. Ainsi la logique est la suivante :

- Application : je demande mon affichage à l'OS par l'envoi du message « Invalidate »
- OS : je décide ou non d'afficher la fenêtre. Si oui → envoi du message « Paint » à l'EDT
- L'EDT : à réception du Paint je dispatche ce message à tous les composants.

2. Remarque

La commande Invalidate est automatiquement lancée lors du changement d'état des contrôles, sinon cela alourdirait l'écriture du code. Ainsi pour les deux lignes suivantes, il y aura automatiquement deux commandes Invalidate lancées.

```
Label1.text = « toto » ;
Button1.Color = Color.Red ;
```

Ce mécanisme en deux temps : remontée à l'OS d'un message PAINT par la fenêtre puis réception de ce message plus tard depuis la Message Queue implique que l'affichage à l'écran est **asynchrone**. Cela veut dire qu'il n'est pas effectué juste après la commande Invalidate comme ce serait le cas dans un appel de fonction. Par exemple, entre la commande label.text = « toto » et l'affichage de toto à l'écran, il peut y avoir 1 seconde ou plus si l'OS est occupé.

Les évènements PAINT sont optimisés. Ils sont traités lorsque le système « a du temps » ou lorsqu'il considère que c'est adéquat. Une application de lecture vidéo avec une forte priorité peut geler ou ralentir ainsi l'interface des autres applications.

c) Les compteurs

Option 1 :

```
Bouton1_clic()  { for (int I = 0 ; I < 1000 ; i++) { … label1.Text = “”+i;  label1.Invalidate();  } }
```

- ⇨ Pas de mise à jour de l'affichage
- ⇨ Une fois le traitement terminé, le label affiche « Valeur : 999 »

Option 2 :

Utilisation de la méthode Update()

Update force l'affichage du composant. Pour cela, cette fonction contourne le mécanisme habituel de l'envoie d'un message PAINT et ainsi supprime le traitement par le gestionnaire d'évènements.

```
Bouton1_clic()  { for (int I = 0 ; I < 1000 ; i++) { label1.Text = …;  label1.Update();  } }
```

Option 3 :

Inconvénient du scénario2 : Il force 999 affichages à l'écran → pas optimisé → ralentit le traitement. Si tous les affichages ne sont pas nécessaires voici une option pour préserver l'interface en stressant moins le CPU :

```
if (  i % 100 == 0  )   label1.Update();
```

Vous ne rafraichissez ainsi le contrôle que toutes les 100 itérations, ce qui diminue par 100 la quantité de réaffichage ! Vous obtenez quelque chose de suffisamment dynamique. Le dernier affichage 999 sera traité à la fin de la fonction par la commande Invalidate implicite.