HAL
open science

Architecture des applications graphiques en Java

Philippe Andary

To cite this version:

Philippe Andary. Architecture des applications graphiques en Java. Doctorat. France. 2020. hal-02484898

HAL Id: hal-02484898

https://normandie-univ.hal.science/hal-02484898v1

Submitted on 2 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Architecture des applications graphiques en Java

Philippe Andary

février 2020

1 Introduction

Quand on construit du logiciel objet qui présente ses données à travers une interface graphique, on a l'habitude d'organiser les classes qui le constituent en plusieurs parties, chacune dédiée à une fonctionnalité bien précise. Ceci dans le but d'optimiser les qualités de lisibilité, de simplicité, d'extensibilité et de maintenance du logiciel.

Dès que le moteur applicatif est un peu conséquent, mais surtout lent à fonctionner par rapport au rendu graphique de l'application, on utilise la notion de fil d'exécution (*thread*) afin d'augmenter la fluidité du fonctionnement de l'application graphique. Or, en Java, la bibliothèque Swing de composants graphiques n'est pas sûre relativement aux accès concurrents à ses éléments. Plus précisément, il ne faut accéder aux composants Swing qu'à travers un seul thread, le thread d'affichage (*Event Dispatch Thread*, noté EDT dans la suite).

Ce document présente une architecture possible, en Java, pour de telles applications graphiques : le moteur applicatif utilise les threads autant que de besoin et de manière disciplinée, afin de ne pas figer l'affichage de l'interface graphique, tout en assurant la manipulation des composants graphiques et de leurs modèles sur EDT. Comme on le verra, cette architecture ne nécessite pas la délicate mise en place de synchronisation.

La section 2 rappelle comment on découpe une application graphique respectant l'architecture MVC. La section 3 aborde l'architecture à modèle séparable des composants Swing en remarquant qu'elle n'est qu'une version ad'hoc de la précédente. Enfin, la section 4, après avoir rappelé que la bibliothèque Swing n'est pas sûre dans le cadre de la programmation concurrente et avoir posé les bases du problème qui nous intéresse, expose une manière sûre de gérer l'activité du modèle de l'application sur différents threads, en fonction de la complexité structurelle de celui-ci.

2 Architecture MVC

2.1 MVC à l'origine

Au sein d'une application développée selon une architecture MVC, le code est organisé en trois parties distinctes qui collaborent étroitement ([4], [1]). Elles concernent respective-

ment la gestion des données (modèle), le rendu graphique (vue) et la communication entre l'application et l'utilisateur (contrôleur).

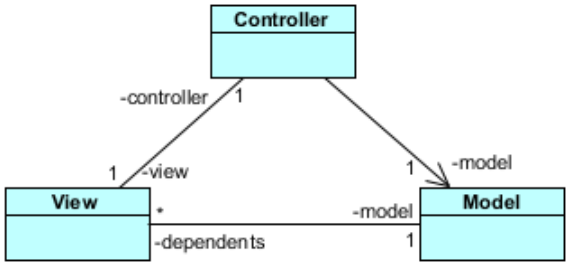

FIGURE 1 – Relations entre les classes de l'architecture MVC d'origine.

— Le *modèle* gère la logique applicative du logiciel, c'est-à-dire qu'il encapsule les données nécessaires à l'application, le moyen d'y accéder et la manière de les modifier. Lorsque ses données changent de valeur, le modèle notifie les vues préalablement enregistrées afin qu'elles puissent se mettre à jour.
— La *vue* gère le rendu graphique de l'application, c'est-à-dire qu'elle spécifie la manière dont il faut afficher les données que contient le modèle, auxquelles elle accède à travers les services que propose ce dernier. Elle se déclare auprès du modèle qui pourra ainsi la notifier lorsqu'une mise à jour sera nécessaire. Enfin, elle transmet les actions de l'utilisateur au contrôleur.
— Le *contrôleur* définit le comportement de l'application. Il reçoit les actions de l'utilisateur qui lui sont transmises par la vue, récupère éventuellement des informations à partir de la vue et traduit ces actions ainsi paramétrées en modifications adaptées pour le modèle. Il peut arriver aussi qu'il commande directement la vue sans passer par le modèle dans certains cas simples.

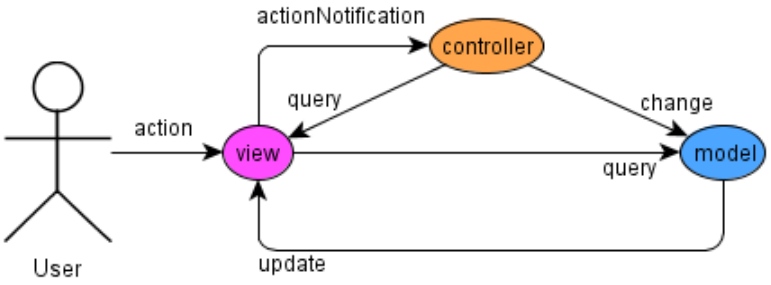

FIGURE 2 – Collaboration entre les éléments de l'architecture MVC d'origine.

L'association entre une vue et un contrôleur est unique et bidirectionnelle, ils se connaissent donc mutuellement.

À un modèle on peut associer plusieurs vues qui le représenteront, chacune totalement ou en partie seulement. Le mode de notification des vues par le modèle peut être de deux sortes :

— *push*, la notification contient toutes les informations nécessaires à la mise à jour de la vue ;
— *pull*, la vue devra consulter le modèle pour y puiser les informations nécessaires à son rafraîchissement.

Il faut remarquer que l'implantation, dans le modèle, du mécanisme de notification des vues est nécessaire dès lors que d'autres objets que les contrôleurs peuvent changer l'état de celui-ci puisque c'est alors le seul moyen efficace d'assurer la cohérence entre l'état du modèle et l'apparence de la vue. Dans le cas extrême où le contrôleur serait le seul à modifier le modèle, la relation entre ce dernier et la vue qui en dépend pourrait disparaître sous réserve que le contrôleur rafraîchisse la vue à chaque fois qu'il modifie le modèle. En dehors du mécanisme de notification, le modèle n'a pas connaissance du comportement de ses vues ni de celui des contrôleurs qui leurs correspondent.

Cette description correspond à l'architecture d'origine, telle qu'elle était réalisée en Smalltalk-80. Dans ce langage, MVC était implanté directement dans des classes bas-niveau (`View`, `Controller`, `Model` et `Object`) qu'il suffisait de réutiliser par héritage.

2.2 MVC en Java

Dans un langage comme Java, les développeurs sont fortement incités à réutiliser cette architecture ([2]) ; le mécanisme de notification (de type *pull*) est alors encapsulé dans une classe `Observable` dont devra dériver le modèle. La dépendance entre le modèle et ses vues est réduite à sa plus simple expression par utilisation de l'interface `Observer`. Ces deux types sont présents dans le paquetage `java.util`. Concernant la relation vue-contrôleur, c'est la notion d'écouteur qui est utilisée. Un écouteur est un objet très léger (souvent instance d'une classe interne anonyme dont le code est très court) dont le comportement est réduit à la gestion d'un type précis d'actions de l'utilisateur comme la manipulation de la souris, ou celle du clavier, mais pas les deux ensemble. La notion de contrôleur se trouve donc éclatée en deux parties : un observateur (pour la relation contrôleur-vue utilisant le modèle) et un écouteur (pour la relation contrôleur-modèle utilisant la vue). Les relations entre les différentes classes et interfaces sont résumées dans la figure 3.

Par ailleurs, les notifications d'actions sur les écouteurs se font de manière événementielle, c'est-à-dire pilotées par un thread dédié (EDT) qui puise indéfiniment dans une file d'objets (les événements), elle-même alimentée par le noyau système, le système de fenêtrage ou l'application. Le diagramme de collaboration entre les différents objets est présenté à la figure 4 où, pour distinguer le mode de communication par événement, la flêche qui relie *view* à *view listener* est notée en pointillés fins. On obtient ainsi la version adaptée à Java de l'architecture MVC.

Cette fois, on notera bien que la vue et le modèle sont complètement indépendants l'un de l'autre ; il suffit que la vue puisse gérer une séquence d'écouteurs (mécanisme implanté dans les composants graphiques de l'API) et le modèle une séquence d'observateurs

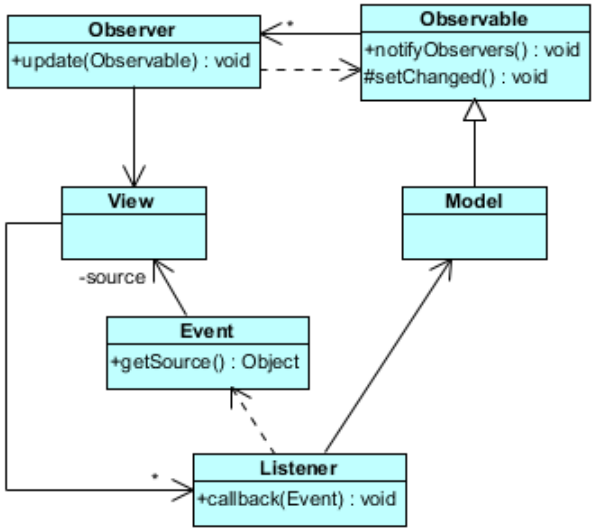

FIGURE 3 – Relations entre les classes de l'architecture MVC en Java.

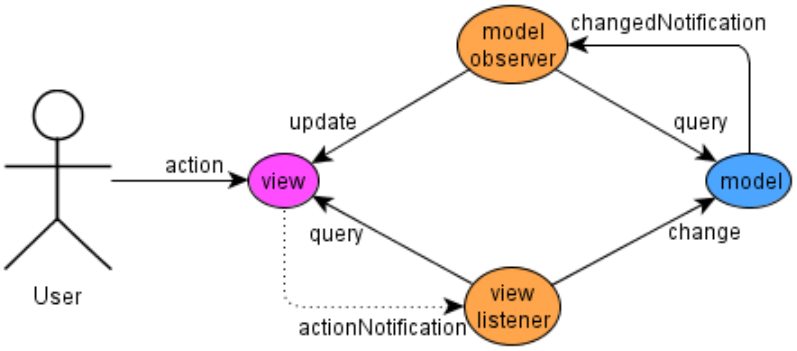

FIGURE 4 – Collaboration entre les éléments de l'architecture MVC en Java.

(mécanisme implanté dans la classe `Observable` de l'API). Enfin, puisque la vue agit sur l'écouteur associé à travers le mécanisme de gestion des événements, l'ensemble des actions de ce schéma se déroule par défaut sur EDT.

3 Swing et MVC

Dans ses différents composants graphiques, Swing regroupe par le biais de classes internes les parties vue et contrôleur en une entité unique : le gestionnaire graphique (*UI-delegate*). Cette architecture (dite « à modèle séparable » et dérivée de MVC) permet de minimiser l'interface entre vue et contrôleur, tout en mettant la puissance de MVC au service du développement du composant (voir figure 5).

Lorsqu'on utilise la bibliothèque de composants Swing en Java, on ne fait plus de diffé-

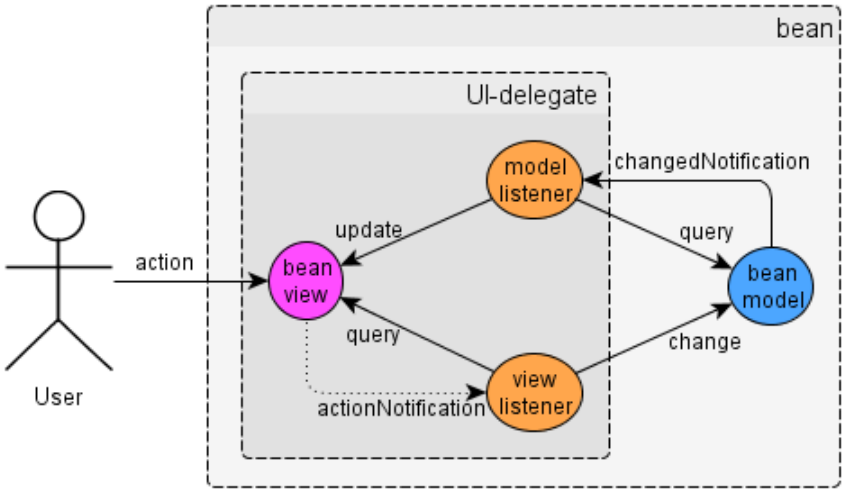

FIGURE 5 – Architecture à modèle séparable des composants Swing.

rence entre écouteur et observateur : tout n'est qu'écouteur. Un mécanisme de notification de type *pull* peut être réalisé au sein du modèle via des implantations idoines de l'interface `ChangeListener` et l'utilisation de la classe `EventListenerList`. Cette implantation est légère car elle permet de n'instancier qu'un seul événement par modèle, à réutiliser lors de chaque notification. Elle est à privilégier dans le cas où les notifications par le modèle sont fréquentes.

De plus, un mécanisme de notification de type *push* peut être facilement réalisé par la mise en place de propriétés (au sens de la norme JavaBean) observables à l'aide d'instances de `PropertyChangeListener`, le mécanisme de notification étant lui-même encapsulé dans la classe `PropertyChangeSupport`.

Il est ici important de remarquer que les interactions entre les différents protagonistes sont toutes initiées par un événement de notification d'action de la vue vers ses écouteurs. Or cet événement, puisque nous sommes dans le contexte d'une action de l'utilisateur, est géré par EDT. Par conséquent l'accès à la vue et à son modèle se fait uniquement par le biais de ce dernier thread.

Si, dans ces conditions, un thread autre que EDT était à l'origine d'une modification du modèle, cela se terminerait par une mise à jour de la vue qui serait effectuée sur cet autre thread. Comme on va le voir dans la section suivante, cela n'est pas permis. C'est d'ailleurs à cause de cette interdiction, et pour être certain que tous les composants graphiques sont bien réalisés sur EDT (et pas sur le *main thread*), que toutes nos applications se lancent à l'aide de la « formule magique » :

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        ... code de démarrage de l'application
    }
```

```
}
```

4 Swing et la programmation concurrente

Swing n'est pas *thread-safe*[1]. Cela signifie que l'invocation de méthodes sur les composants graphiques ou sur leurs modèles, à partir de plusieurs threads, risque de provoquer des problèmes d'interférence entre les threads et des erreurs liées à l'incohérence des mises à jour de la mémoire. Pratiquement, cela a pour conséquence l'obligation d'accéder aux composants graphiques (ainsi qu'à leurs modèles) à l'aide d'un seul thread. Et en Java, ce thread, c'est EDT.

Le problème ? Une application graphique dont la logique métier est un peu lourde va avoir tendance à geler l'interface graphique. Par exemple, télécharger un gros fichier sur le Web prend plusieurs minutes ; or si le téléchargement est initié par un clic souris sur un bouton, c'est EDT qui va être spontanément sollicité et pendant ce temps-là, l'interface graphique ne répondra plus puisque EDT est accaparé par une longue tâche...

Par conséquent, il est normal et souhaitable qu'une tâche lourde ne soit pas exécutée sur EDT mais sur un thread utilisateur. Par contre, si au cours de l'exécution de cette tâche il est nécessaire de manipuler des composants Swing ou leurs modèles, il faudra revenir le faire sur EDT.

4.1 Architecture supportant la concurrence (cas simple)

Dans les cas où le modèle doit traiter de lourdes tâches, on aboutit donc au diagramme de la figure 6, où la flêche en pointillés plats étiquetée par *change* indique un envoi de message asynchrone via un thread utilisateur. Lorsque l'envoi d'un message asynchrone doit se faire via EDT (message *update*), on procède, en Java, au moyen d'une instance d'`ActiveEvent` (à ne pas confondre avec `ActionEvent`) placée sur la file des événements (de type `EventQueue`) que gère EDT (un appel à `SwingUtilities.invokeLater`, quoi), le code correspondant à ce message devant être encapsulé dans une instance de `Runnable`. Dans les cas les plus simples on utilisera le schéma de code suivant :

```
// dans view listener
final ViewInfo vi = guiView.query(...);
Runnable target = new Runnable() {
    public void run() {
        applicationModel.change(vi);
    }
};
new Thread(target).start();
---------------><8
// dans model listener
```

1. On pourrait se demander pourquoi ? Un élément de réponse dans [3].

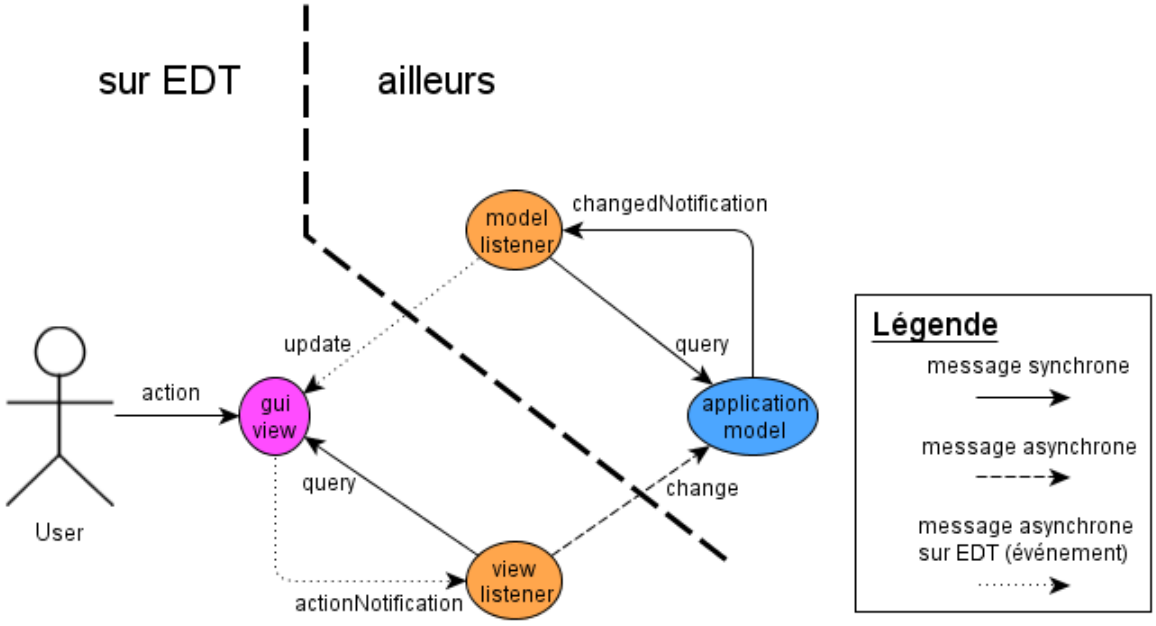

FIGURE 6 – Collaboration entre les éléments de l'architecture MVC supportant la concurrence (modèle simple).

```
final ModelInfo mi = applicationModel.query(...);
Runnable target = new Runnable() {
    public void run() {
        guiView.update(mi);
    }
};
SwingUtilities.invokeLater(target);
```

Alternativement, Swing propose d'utiliser une classe abstraite générique : `SwingWorker`. Une instance de cette classe permet d'exécuter ailleurs que sur EDT la tâche codée dans la méthode `SwingWorker.doInBackground` (qui joue le rôle de la méthode `run` du code cible d'un thread, mais qui permet tout de même de calculer une valeur qu'elle retournera). Cette méthode peut appeler `SwingWorker.publish` pour communiquer des données qui seront traitées sur EDT par appel à `SwingWorker.process`. Il suffit alors de définir cette dernière de sorte qu'elle implante le comportement souhaité pour la mise à jour des composants graphiques. La tâche se termine par appel de `SwingWorker.done` (qui ne fait rien par défaut mais qui peut-être redéfinie) et le fruit du calcul de `doInBackground` est accessible par `SwingWorker.get`, qui est une méthode bloquante (donc attention si cette dernière méthode est appelée sur EDT).

4.2 Architecture supportant la concurrence (cas complexe)

Que l'on utilise `SwingWorker` ou que l'on gère soi-même ses propres threads, il peut arriver que le modèle de l'application soit suffisamment complexe et consommateur de temps CPU pour que l'on soit amené à découper l'application comme sur la figure 7.

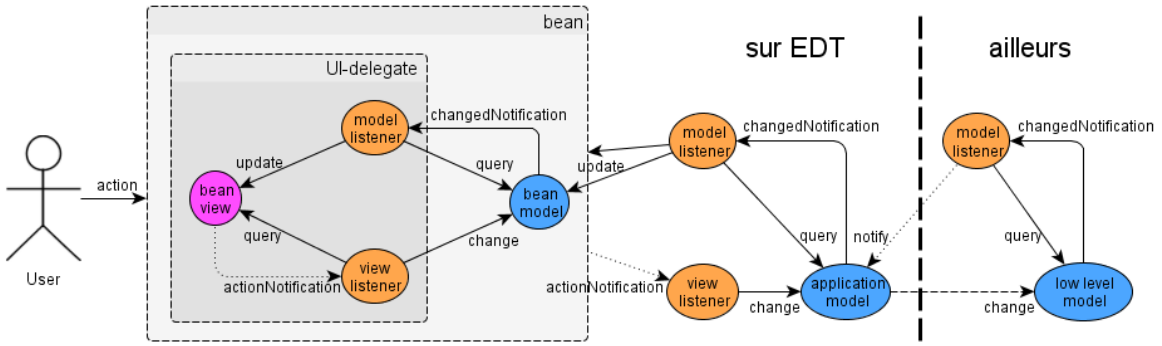

FIGURE 7 – Collaboration entre les éléments de l'architecture MVC supportant la concurrence (modèle complexe).

Il y a maintenant trois modèles en jeu :

— le modèle de l'application (au centre sur le schéma), qui définit les commandes métier de l'application, les services en quelque sorte (par exemple, dans un éditeur de texte, « ouvrir un fichier texte pour l'édition ») ;

— le modèle bas-niveau (à droite), qui définit les commandes d'interaction avec le système (par exemple, « charger le contenu du fichier en mémoire », ou plus généralement la couche d'accès aux données) ;

— le modèle de l'interface graphique de l'application (à gauche), qui définit l'état sous-jacent aux composants de l'interface graphique (par exemple, le `Document` utilisé par la zone de texte qui affiche le texte en cours d'édition).

Le modèle de l'application délègue l'exécution des calculs longs, ou pouvant bloquer le fil courant d'exécution, au modèle bas-niveau à l'aide de messages asynchrones, c'est-à-dire exécutés sur des threads utilisateurs, afin de ne pas accaparer EDT.

Ensuite, le modèle bas-niveau travaille sur un ou plusieurs threads utilisateurs. Son fonctionnement est observé (au sens du patron de conception OBSERVER) par un écouteur préalablement défini à l'aide d'une classe locale anonyme dans le modèle de l'application. Cet écouteur a, de plus, la charge de rétablir le flot de contrôle sur EDT, ce qui est absolument nécessaire puisque seul EDT a le droit de mettre à jour les composants graphiques.

Enfin, le modèle de l'interface graphique est parfois dispersé au sein des composants graphiques qui constituent la partie « vue » de l'application, comme suggéré sur la figure 7. Mais on peut préférer centraliser dans un seul objet toutes les informations nécessaires à la vue, en regroupant une partie des modèles de certains composants graphiques (complexes) ainsi que des informations du genre « tel bouton doit être activable ou non. » Cette façon

de faire peut grandement simplifier le traitement des mises à jour de la vue.

Une dernière remarque, de taille, avant de terminer cette section. Le lecteur attentif aura deviné que la récupération des informations du modèle bas-niveau par celui de l'application peut être critique puisque ces deux modèles sont manipulés sur deux threads distincts. Or si le mécanisme de notification entre les deux modèles est de type *push* il n'y a alors pas lieu de prendre des précautions particulières car toutes les données transitent entre les modèles sous forme d'événements, donc par l'intermédiaire de paramètres plutôt que de variables partagées. En revanche, si ce mécanisme était de type *pull*, le modèle de l'application devrait envoyer des requêtes au modèle bas-niveau (ce cas n'est pas indiqué sur le diagramme de la figure 7) ; il faudrait alors s'assurer que la communication entre les modèles bénéficie d'un schéma de synchronisation adapté pour un échange sûr de données cohérentes. La technique exposée, par l'écoute du changement de valeurs de propriétés du modèle bas-niveau plutôt que par l'observation concurrente de ses changements d'état, dispense donc le développeur de synchronisation et simplifie grandement le développement de l'application.

5 Conclusion

Après avoir rappelé ce qu'était l'architecture MVC et la manière dont elle pouvait être implantée en Java, nous avons présenté l'architecture dite « à modèle séparable » des composants de la bibliothèque Swing. Nous avons expliqué que cette dernière forme était une adaptation de la première, permettant de minimiser l'interface entre les parties vue et contrôleur de ces composants.

L'utilisation de la bibliothèque Swing impose une contrainte forte : tout composant graphique et son modèle doivent être manipulés sur EDT (car *Swing is not thread-safe*). Ceci impose que l'activité globale de l'application se déroule sur EDT, et il semblerait que l'on tombe alors dans un cercle vicieux : l'activité du modèle se déroule sur EDT car elle trouve son origine dans l'action de l'utilisateur, or si cette activité est trop longue il faut qu'elle se déroule sur un autre thread que EDT et, si le modèle notifie ses vues, certains composants Swing ne seront pas mis à jour sur EDT !

Nous avons alors proposé une méthode qui permet au développeur, tout en respectant la contrainte engendrée par l'utilisation de la bibliothèque Swing (modification des composants graphiques sur EDT), d'utiliser autant de threads que nécessaire à la bonne exécution de son application. Il suffit pour cela de découper le modèle en plusieurs parties (entre une et trois selon la complexité du modèle de l'application) et nous avons montré comment exécuter, d'une part les tâches longues (sur des threads utilisateur), et d'autres part les manipulations des composants graphiques et de leurs modèles (sur EDT systématiquement). L'usage de cette technique a l'avantage de permettre l'échange de données sans synchronisation entre les différents modèles.

Références

[1] Steeve Burbeck. Applications Programming in Smalltalk-80(TM) : How to use Model-View-Controller (MVC). https://www.researchgate.net/publication/238719652_Applications_programming_in_smalltalk-80_how_to_use_model-view-controller_mvc, 1992.

[2] Robert Eckstein. Java SE Application Design With MVC. http://www.oracle.com/technetwork/articles/javase/mvc-136693.html, 2007. (Avec les images : http://browse.feedreader.com/c/The_skiing_cube/251397735).

[3] Graham Hamilton. Multithreaded toolkits : A failed dream ? https://community.oracle.com/blogs/kgh/2004/10/19/multithreaded-toolkits-failed-dream, 2004.

[4] Trygve Reenskaug. The original MVC reports. http://heim.ifi.uio.no/~trygver/2007/MVC_Originals.pdf, 1979.