M2VC, un moteur MVC pour des applications GUI Java

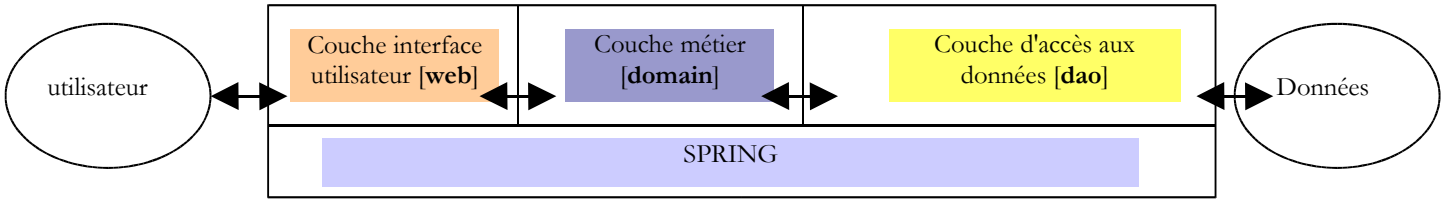
serge.tahe@istia.univ-angers.fr, juin 2005

Ce document présente le portage sur la plate-forme Java du moteur [**M2VC-win**] conçu pour la plate-forme .NET. Il reprend le texte de l'article [ftp://ftp-developpez.com/tahe/fichiers-archive/m2vc-win.pdf] en l'adaptant au monde Java et s'en écarte le moins possible.

1 Introduction

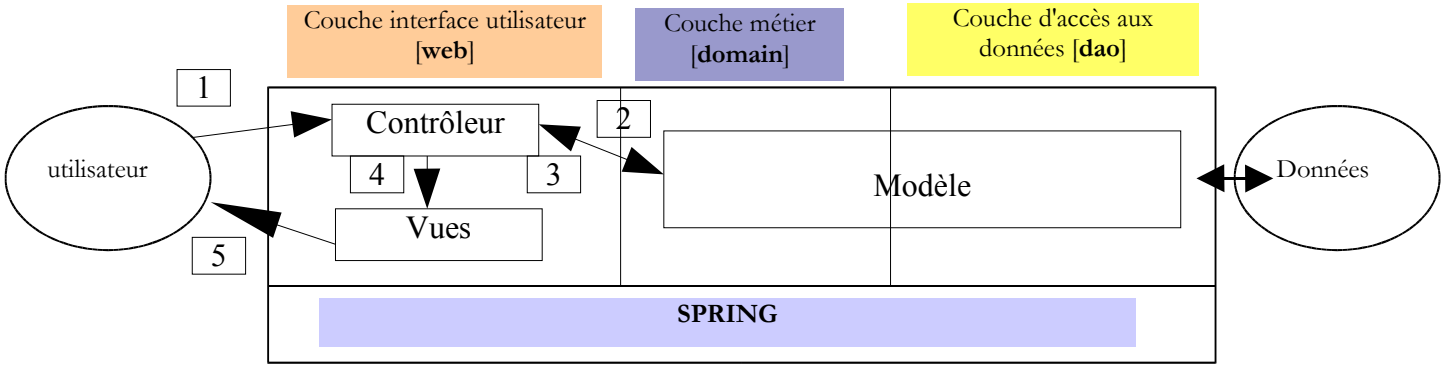
Nous nous proposons ici d'étudier un moteur MVC pour des application windows graphiques développées en Java. Si le modèle MVC (Modèle - Vue - Contrôleur) est désormais bien accepté dans le cadre des applications web, il ne semble pas qu'il ait percé dans le développement d'applications graphiques windows ou alors on n'en parle pas. L'idée de ce moteur MVC est venu à l'occasion du portage d'une interface web existante vers une interface à base de formulaires windows. L'interface web ayant une architecture MVC, j'ai souhaité reproduire celle-ci dans l'interface windows. En l'absence d'outils connus, j'ai été amené à développer le moteur M2VC-win pour la plate-forme .NET tout d'abord. Nous nous proposons ici de porter M2VC-win sur la plate-forme Java.

Considérons l'application web à trois couches suivante :



- les trois couches sont indépendantes grâce à l'utilisation d'interfaces
- l'intégration des différentes couches est réalisée avec **Spring**

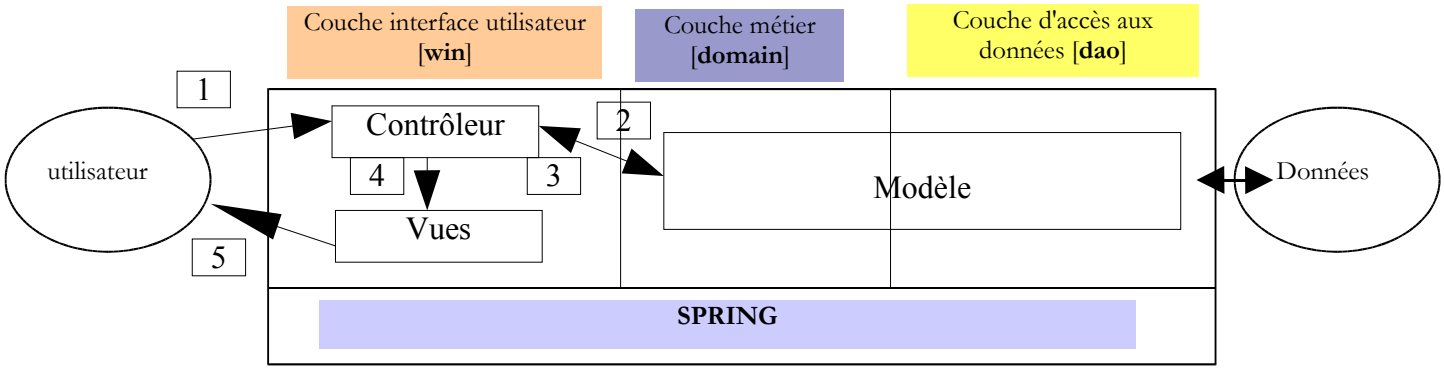
L'architecture MVC (Modèle - Vue - Contrôleur) s'intègre dans le schéma en couches ci-dessus, de la façon suivante :



Le traitement d'une demande d'un client se déroule ainsi :

1. le client fait une demande au contrôleur. Celui-ci voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le C de MVC.
2. le contrôleur traite cette demande. Pour ce faire, il peut avoir besoin de l'aide de la couche métier, ce qu'on appelle le modèle M dans la structure MVC.
3. le contrôleur reçoit une réponse de la couche métier. La demande du client a été traitée. Celle-ci peut appeler plusieurs réponses possibles. Un exemple classique est
   - une page d'erreurs si la demande n'a pu être traitée correctement
   - une page de confirmation sinon
4. le contrôleur choisit la réponse (= vue) à envoyer au client. Celle-ci est le plus souvent une page contenant des éléments dynamiques. Le contrôleur fournit ceux-ci à la vue.
5. la vue est envoyée au client. C'est le V de MVC.

L'architecture précédente transposée dans le monde des applications windows donne la chose suivante :



La couche [**web**] qui présentait des pages web à l'utilisateur est remplacée ici par une couche [**win**] qui lui, présente des formulaires windows. Le fonctionnement de l'ensemble, décrit plus haut, peut être repris ici à l'identique.

Différents outils du monde libre sont disponibles pour faciliter le développement de la couche [web] selon le concept MVC. Les plus connus sont dans le monde Java. On trouve notamment **Struts** et **Spring**. Cet article se propose de construire un outil inspiré de **Struts** pour faciliter l'implémentation du concept MVC dans la couche [win] de l'application à trois couches ci-dessus. Par la suite, nous appellerons cet outil, Moteur MVC et le désignerons par **M2VC**. Il sera développé en Java.

Avertissement

M2VC vise seulement à apporter une contribution à la recherche de méthodologies MVC dans le monde des applications graphiques Java. En aucun cas, il ne vise à être la solution définitive. Il n'est ainsi même pas sûr que le modèle utilisé pour son développement soit correct. Les quelques applications que j'ai pu développer avec, et dont on trouvera des exemples dans cet article, sont trop peu nombreuses pour le certifier. Cette contribution peut néanmoins donner des idées à d'autres développeurs.

En-dehors de tout but pratique, M2VC est un bon exemple des fantastiques possibilités apportées par [Spring Ioc].

Pré-requis et outils

Les outils utilisés dans cet articles sont les suivants :

- **JBuilder X Foundation** pour le développement des applications Java
- **Spring IoC** pour l'instanciation des objets nécessaires au moteur MVC

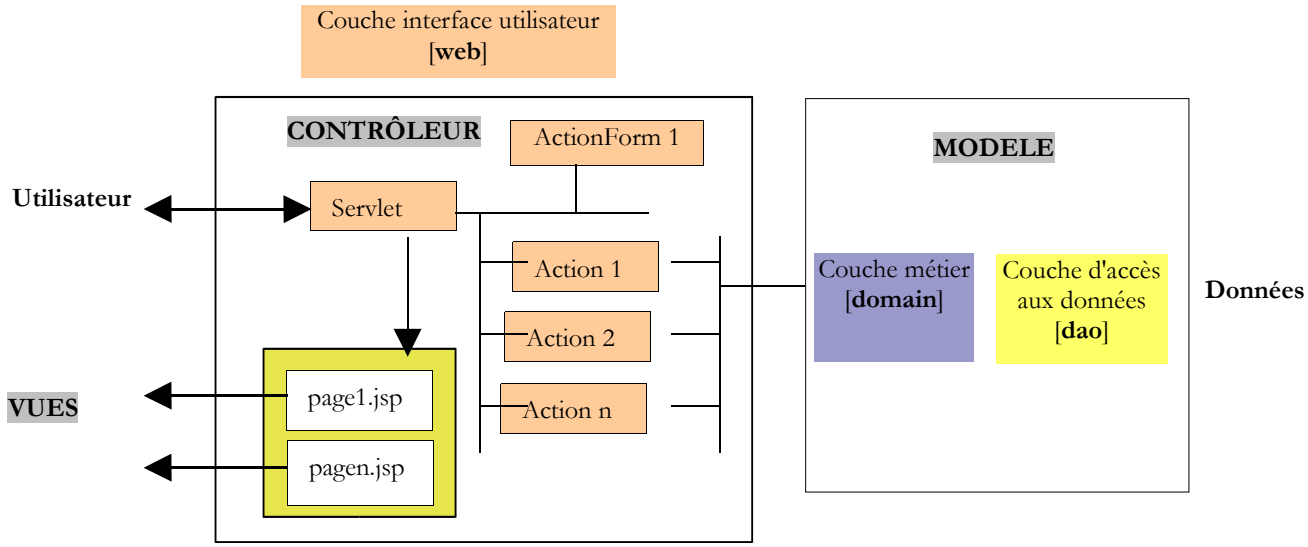
Dans une échelle [débutant-intermédiaire-avancé], ce document est dans la partie [avancé]. Sa compréhension nécessite divers pré-requis qu'on pourra trouver dans certains des documents que j'ai écrits :

- **langage Java** : [http://tahe.developpez.com/java/cours] : classes, interfaces, héritage, polymorphisme, threads, synchronisation
- **Spring IoC** : [http://tahe.developpez.com/java/springioc]
- **Struts** : [http://tahe.developpez.com/java/struts/]. Ce pré-requis n'est pas indispensable. La connaissance de Struts facilite simplement la compréhension de l'article.

Il est bien évident que le lecteur peut utiliser ses documents favoris.

2 La philosophie de Struts

Rappelons ou découvrons l'architecture MVC générique utilisée par STRUTS dans le cadre des applications web :

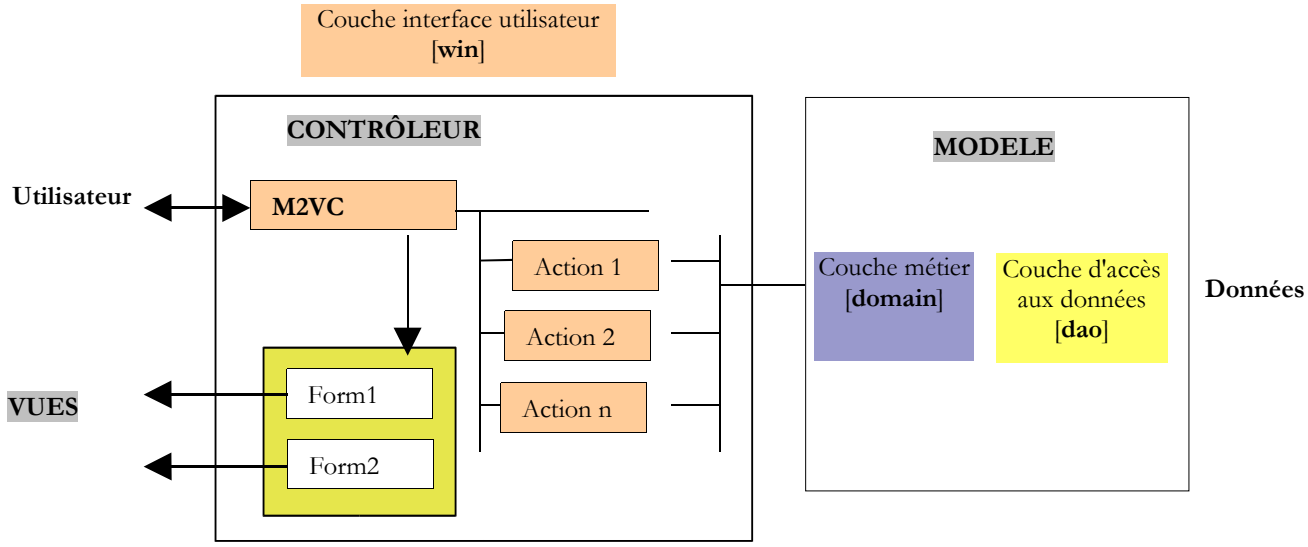


M=modèle	les classes métier, les classes d'accès aux données et la source de données
V=vues	les pages JSP
C=contrôleur	la servlet de traitement des requêtes clientes, les objets [Action] et [ActionForm]

- le contrôleur est le coeur de l'application. Toutes les demandes du client transitent par lui. C'est une servlet générique [ActionServlet] fournie par STRUTS. On peut dans certains cas être amené à la dériver. Pour les cas simples, ce n'est pas nécessaire. Cette servlet générique prend les informations dont elle a besoin dans un fichier le plus souvent appelé **struts-config.xml**.
- la demande du client vient par l'intermédiaire d'une requête HTTP. L'URL cible est l'action demandée par le client.
- si la requête du client contient des paramètres de formulaire, ceux-ci sont mis par le contrôleur dans un objet **[ActionForm]**.
- dans le fichier de configuration **struts-config.xml**, à chaque URL (=action) devant être traitée par programme (ne correspondant donc pas à une vue JSP qu'on pourrait demander directement) on associe certaines informations :
  - le **nom de la classe** de type **Action** chargée d'exécuter l'action
  - si l'URL demandée est paramétrée (cas de l'envoi d'un formulaire au contrôleur), le nom de l'objet [**ActionForm**] chargé de mémoriser les informations du formulaire
- muni de ces informations fournies par son fichier de configuration, à la réception d'une demande d'URL par un client, le contrôleur est capable de déterminer s'il y a un objet [**ActionForm**] à créer et lequel. Une fois instancié, cet objet [**ActionForm**] peut vérifier que les données qu'on vient de lui injecter et qui proviennent du formulaire, sont valides ou non. Une méthode de [**ActionForm**] appelée **validate** est appelée automatiquement par le contrôleur. L'objet [**ActionForm**] étant construit par le développeur, celui-ci a mis dans la méthode **validate** le code vérifiant la validité des données du formulaire. Si les données se révèlent invalides, le contrôleur n'ira pas plus loin. Il passera la main à une vue dont il trouvera le nom dans son fichier de configuration. L'échange est alors terminé.
- si les données de l'objet [**ActionForm**] sont correctes, ou s'il n'y a pas de vérification ou s'il n'y a pas d'objet [**ActionForm**], le contrôleur passe la main à l'objet de type [**Action**] associé à l'URL. Il le fait en demandant l'exécution de la méthode **execute** de cet objet à laquelle il transmet la référence de l'objet **[ActionForm]** qu'il a éventuellement construit. Les objets [**Action**] sont construits par le développeur. C'est là qu'il place le code chargé d'exécuter l'action demandée. Ce code peut nécessiter l'utilisation de la couche métier ou modèle dans la terminologie MVC. Les objets [**Action**] sont les seuls objets en contact avec cette couche. A la fin du traitement, l'objet [**Action**] rend au contrôleur une chaîne de caractères représentant le résultat de l'action.
- dans son fichier de configuration, le contrôleur trouvera l'URL de la vue associée à cette chaîne. Il envoie alors cette vue au client. L'échange avec le client est terminé.

3 La philosophie de M2VC

Nous nous inspirons ici librement de la philosophie Struts pour construire notre moteur MVC. L'architecture d'une application **M2VC** sera la suivante :

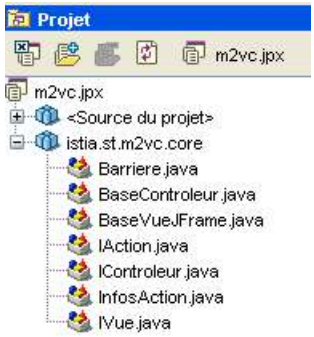


M=modèle	les classes métier, les classes d'accès aux données et la base de données
V=vues	les formulaires Windows
C=contrôleur	le moteur [MVC2] de traitement des requêtes clientes, les objets [Action]

Notre moteur **M2VC** joue le rôle de la servlet générique [**ActionServlet**] de Struts. **M2VC** n'utilisera pas d'objets [**ActionForm**] comme objets tampon entre le client et les classes [**Action**]. Il fera exécuter les actions demandées par l'utilisateur par des objets [**Action**] qui iront chercher les paramètres de la demande du client dans un objet qui sera défini par le développeur. Le moteur ne fait aucune hypothèse sur ce point.

4 Les éléments de M2VC

M2VC a été construit avec JBuilder. Le plus simple est peut-être de partir de la structure du projet JBuilder utilisé pour construire [M2VC] puis de détailler un à un ses différents éléments :



Le projet est formé des éléments suivants :

- trois interfaces : **IAction**, **IVue**, **IControleur**
- deux classes dérivables : **BaseControleur**, **BaseVueJFrame**
- une classe : **InfosAction**

Passons en revue le rôle de ces différents éléments sans encore entrer dans les détails. Les explications ci-dessous doivent être lues à la lumière de l'architecture présentée dans le paragraphe précédent :

IControleur	dit ce que doit savoir faire un contrôleur générique
BaseControleur	implémente l'interface [**IControleur**]. Assure toute la synchronisation [traitement actions] - affichage vues. Le développeur peut dériver cette classe si le contrôleur générique ne lui suffit pas.
IAction	dit ce que doit savoir faire un objet [**Action**]. Aucune implémentation n'est proposée.
IVue	dit ce que doit savoir faire une vue
BaseVueJFrame	implémente l'interface [**Ivue**] précédente. Est dérivée de l'objet [JFrame]. Les formulaires d'une application windows utilisant [M2VC] seront dérivés de cette classe de base. Il est possible de construire d'autres implémentations de l'interface [IVue].

`InfosAction`	classe qui définit les informations liées à une action. C'est grâce à cette classe, que le contrôleur générique va savoir ce qu'il doit faire.

Nous décrivons maintenant ces éléments un à un. Ils sont tous placés dans l'espace de noms [istia.st.m2vc.core].

4.1 L'interface [IControleur]

Son code est le suivant :

```
package istia.st.m2vc.core;

/**
 * @author serge.tahe@univ-angers.fr, juin 2005
 *
 */
public interface IControleur {
  // lance le contrôleur
  public void run();
}
```

On ne demandera qu'une chose à un contrôleur, c'est de s'exécuter. C'est la méthode [**run**] qui nous permettra de lancer le contrôleur de l'application windows.

4.2 L'interface [IAction]

Son code est le suivant :

```
package istia.st.m2vc.core;

/**
 * @author serge.tahe@univ-angers.fr, juin 2005
 *
 */
public interface IAction {
  // exécute l'action
  public String execute();
}
```

On ne demandera qu'une chose à un objet [Action], c'est d'exécuter l'action pour laquelle il a été construit. Rappelons ici que c'est le développeur qui fournira les classes [Action] implémentant l'interface [IAction]. Pour exécuter une action, on appellera donc la méthode [**execute**] de l'objet [Action] associée à cette action.

4.3 L'interface [IVue]

Son code est le suivant :

```
package istia.st.m2vc.core;

/**
 * @author serge.tahe@univ-angers.fr, juin 2005
 *
 */
public interface IVue {
  // affiche la vue
  public void affiche();
  // cache la vue
  public void cache();
  // récupère le nom de l'action demandée par la vue
  public String getAction();
  // fixe le nom de la vue
  public void setNom(String nom);
  // récupère le nom de la vue
  public String getNom();
}
```

Que peut-on demander à une vue ?

- qu'elle s'affiche (méthode **affiche**)
- qu'elle se cache (méthode **cache**)
- qu'elle donne son nom (propriété **nom**). Cette propriété n'a jamais paru indispensable mais elle a été néanmoins conservée. Je l'ai utilisée dans des phases de débogage

- qu'elle donne le nom de l'action demandée par l'utilisateur (propriété **action**). Rappelons qu'une vue sera une fenêtre windows. Celle-ci va réagir toute seule à certains événements. Ce sera la grande différence avec les vues web. Elle peut ainsi réagir à un événement *drag'n drop*. Elle s'interdira de faire appel elle-même aux couches métier dont elle n'est pas sensée avoir connaissance.. Dans ce cas, elle passera plutôt la main au contrôleur en lui donnant le nom de l'action demandée par l'utilisateur.

4.4 La classe [Barriere]

Lorsque le contrôleur fait afficher une vue, il se met ensuite en attente d'un événement. Une vue répond aux événements qui concernent son apparence. Lorsque l'utilisateur demandera une action qui nécessite un accès à la couche métier, la vue s'interdira de l'exécuter elle-même. Elle la fera exécuter par le contrôleur. Cette demande constitue ce que nous avons appelé ci-dessus un événement pour le contrôleur. Pour permettre au contrôleur d'être averti de celui-ci, on utilise la classe [Barriere] suivante :

```
1. package istia.st.m2vc.core;
2.
3. /**
4.  * @author serge.tahe@univ-angers.fr, juin 2005
5.  *
6.  */
7.
8. public class Barriere {
9.
10.  // état ouvert - fermé de la barrière
11.  private boolean ouverte;
12.
13.  /**
14.   * @return Returns the ouverte.
15.   */
16.  public boolean isOuverte() {
17.    return ouverte;
18.  }
19.
20.  /**
21.   * @param ouverte The ouverte to set.
22.   */
23.  public void setOuverte(boolean ouverte) {
24.    this.ouverte = ouverte;
25.  }
26.
27.  // fermeture barrière
28.  public synchronized void reset() {
29.    ouverte = false;
30.  }
31.
32.  // ouverture barrière
33.  public synchronized void set() {
34.    if (! ouverte){
35.      ouverte = true;
36.      this.notify();
37.    }
38.  }
39.
40.  // attente barrière
41.  public synchronized void waitOne() {
42.    if (!ouverte) {
43.      try {
44.        this.wait();
45.      }
46.      catch (InterruptedException ex) {
47.        throw new RuntimeException(ex.toString());
48.      }
49.    }
50.  }
51.}
```

- ligne 11, la classe gère une propriété [ouverte] pour indiquer si la barrière est ouverte ou non
- lignes 41-50, la méthode [waitOne] permet d'attendre (ligne 44) que ce booléen passe à vrai si on l'a trouvé à faux
- lignes 33-38, la méthode [set] ouvre la barrière et en avertit un éventuel thread en attente de cette ouverture (ligne 36).
- lignes 28-30, la méthode [reset] ferme la barriere.

Note : cette classe n'a pas fait l'objet d'une étude de synchronisation poussée. Il faudrait voir si elle passe bien tous les cas de synchronisation possible entre le contrôleur et les vues.

4.5 La classe [BaseVueJFrame]

[**BaseVue**] est une classe de base implémentant l'interface **Ivue**. Outre qu'elle implémente les caractéristiques de l'interface [IVue], elle sait également se synchroniser avec le contrôleur générique. En effet, lorsque celui-ci demande l'affichage d'une vue, il va :

- faire appel à la méthode [affiche] de la vue
- attendre que celle-ci envoie le nom d'une action à exécuter. Il y a donc une synchronisation [controleur - vue] à organiser.

Le code de la classe [BaseVueJFrame] est le suivant :

```
1. package istia.st.m2vc.core;
2.
3. import javax.swing.JFrame;
4.
5. /**
6.  * @author serge.tahe@univ-angers.fr, juin 2005
7.  *
8.  */
9. public class BaseVueJFrame
10.    extends JFrame implements IVue, Runnable {
11.
12.  // champs privés
13.  private String nom; // nom de la vue
14.  private String action; // nom de l'action que le contrôleur doit exécuter
15.  private Barriere synchro; // objet de synchronisation controleur - vues
16.  private Thread threadVue; // objet de synchronisation de la vue
17.
18.
19.  /**
20.   * @return Returns the action.
21.   */
22.  public String getAction() {
23.    return action;
24.  }
25.
26.  /**
27.   * @param action
28.   *          The action to set.
29.   */
30.  public void setAction(String action) {
31.    this.action = action;
32.  }
33.
34.  /**
35.   * @return Returns the nom.
36.   */
37.  public String getNom() {
38.    return nom;
39.  }
40.
41.  /**
42.   * @param nom
43.   *          The nom to set.
44.   */
45.  public void setNom(String nom) {
46.    this.nom = nom;
47.  }
48.
49.
50.  /**
51.   * @return Returns the synchro.
52.   */
53.  public Barriere getSynchro() {
54.    return synchro;
55.  }
56.  /**
57.   * @param synchro The synchro to set.
58.   */
59.  public void setSynchro(Barriere synchro) {
60.    this.synchro = synchro;
61.  }
62.
63.  // affiche la vue
64.  public void affiche() {
65.    // on s'affiche dans le thread d'affichage
66.    if (threadVue == null) {
67.      // on crée le thread d'affichage
68.      threadVue = new Thread(this);
69.      threadVue.start();
70.    } else {
71.      // on s'affiche
72.      this.setVisible(true);
73.    }
74.  }
75.
76.  // cache la vue
77.  public void cache() {
78.    // on se cache
79.    this.setVisible(false);
```

```
80.  }
81.
82.  // action asynchrone
83.  public void exécute(String action) {
84.    // on note l'action à exécuter
85.    this.setAction(action);
86.    // on passe la main au contrôleur
87.    synchro.set();
88.  }
89.
90.  // méthode du thread d'affichage
91.  public void run() {
92.    // affichage de la vue
93.    this.setVisible(true);
94.  }
95.}
```

Les points à comprendre sont les suivants :

- la classe [BaseVueJFRame] dérive de [JFrame] (ligne 10).
- la classe implémente l'interface [IVue] - ligne 10
- la classe implémente l'interface [Runnable] - ligne 10. En effet, elle contient la méthode [run] d'un thread qu'elle crée pour s'afficher.
- elle implémente la propriété [nom] de l'interface [Ivue] - lignes 34-47
- elle implémente la propriété [action] de l'interface [Ivue] - lignes 19-32
- elle implémente la méthode [affiche] de l'interface [Ivue] - lignes 64-74
- elle implémente la propriété [cache] de l'interface [Ivue] - lignes 77-80
- elle a un objet de synchronisation [synchro] de type [Barriere] - lignes 50-61. La propriété [synchro] va permettre à la vue de signaler au contrôleur qu'elle demande l'exécution d'une action.
- la fenêtre graphique va être affichée dans un thread différent de celui dans lequel s'exécute le contrôleur. Ce thread est déclaré en ligne 16

- **la méthode [affiche]**

```
1.   public void affiche() {
2.     // on s'affiche dans le thread d'affichage
3.     if (threadVue == null) {
4.       // on crée le thread d'affichage
5.       threadVue = new Thread(this);
6.       threadVue.start();
7.     } else {
8.       // on s'affiche
9.       this.setVisible(true);
10.    }
11.  }
12.
13.  // cache la vue
14.  public void cache() {
15.    // on se cache
16.    this.setVisible(false);
17.  }
18.
19.  // action asynchrone
20.  public void exécute(String action) {
21.    // on note l'action à exécuter
22.    this.setAction(action);
23.    // on passe la main au contrôleur
24.    synchro.set();
25.  }
26.
27.  // méthode du thread d'affichage
28.  public void run() {
29.    // affichage de la vue
30.    this.setVisible(true);
31.  }
32.}
```

La méthode [affiche] est destinée à être spécialisée par des classes dérivées. En effet, [BaseVueJFrame] ne sait pas quoi afficher. Seules les classes dérivées le sauront. Elles procèderont à l'affichage du formulaire de la façon suivante :

- initialisation des composants du formulaire
- appel de la méthode [affiche] de la classe de base [BaseVueJFrame] ci-dessus. C'est elle qui doit terminer l'affichage. Que fait-elle ?

- ✗ elle regarde si le formulaire a déjà été affiché. Si c'est le cas, le thread d'affichage [threadVue] a été initialisé - ligne 3
- ✗ si le thread d'affichage n'existe pas, alors on le crée (ligne 4) et on le lance (ligne 6). Le thread a été lancé pour exécuter la méthode privée [run] de la classe. La méthode [run] (lignes 28-31) s'exécute donc. Que fait-elle ? Elle affiche le formulaire (ligne 30). La méthode [affiche] est terminée.

- ✗ si le thread d'affichage existe déjà, alors on se contente d'afficher la vue (ligne 9). La méthode [affiche] est terminée.
- ✗ dans quelle situation sommes-nous lorsque la méthode [affiche] a été entièrement exécutée ?
  - ✗ l'utilisateur voit le formulaire. Il peut agir dessus. Le formulaire réagit aux événements.
  - ✗ celui-ci est affiché dans un thread [threadVue] séparé du thread du contrôleur
  - ✗ le contrôleur est en attente d'un événement : que la vue lui dise qu'une action lui est demandée. Elle signalera cet événement grâce à la variable de synchronisation [synchro] qui sera partagée entre le contrôleur et toutes les vues de l'application. Pour attendre, le contrôleur a émis la séquence d'instruction suivante :

```
      // on affiche la vue en se synchronisant avec elle
      synchro.reset();
      vue.affiche();
      synchro.waitOne();
```

  - ✗ la vue débloquera le contrôleur en faisant :

```
       synchro.Set()
```

- ✗ quand la situation se débloque-t-elle ? L'utilisateur interagit avec le formulaire. Celui-ci réagit avec les gestionnaires d'événements qu'on lui a mis. Pour certains événements, le développeur de la vue décide de passer la main au contrôleur. Pour cela il appelle la méthode [execute] de la classe de base [BaseVueJFrame] en lui passant le nom de l'action à exécuter.

- **la méthode [execute]**

Son code est le suivant :

```
1.   // action asynchrone
2.   public void exécute(String action) {
3.     // on note l'action à exécuter
4.     this.setAction(action);
5.     // on passe la main au contrôleur
6.     synchro.set();
7.   }
```

- ✗ l'action à eéxcuter est notée dans la propriété [action] - ligne 4. C'est ici que le contrôleur la récupèrera.
- ✗ on débloque le contrôleur - ligne 6. Celui-ci va pouvoir exécuter l'action qu'on lui demande
- ✗ on notera et **c'est très important de le comprendre** qu'en ligne 6, la méthode [execute] est terminée. Son exécution sera demandée à la suite d'un événement du formulaire affiché. La fin de [exécute] entraîne la fin de la gestion de l'événement. De nouveaux événements du formulaire peuvent alors survenir et être traités. **Cela signifie qu'alors que le contrôleur est en train d'accomplir une action pour le compte de la vue, celle-ci continue à répondre aux événements qui la concernent**. Nous n'avons pas voulu prendre de décision sur ce point. C'est au développeur de le faire. **Celui-ci doit savoir qu'à chaque fois qu'il fait appel au contrôleur, il déclenche une action asynchrone**. A lui de savoir comment son formulaire doit se comporter. Dans les exemples étudiés plus loin, nous proposons plusieurs réponses simples à cette situation.

- **la méthode [cache]**

Le contrôleur gère plusieurs vues. Après avoir exécuté l'action demandée par une vue, il peut vouloir afficher une autre vue. Il demande alors à la vue précédente de se cacher. Le code est simple :

```
1.   // cache la vue
2.   public void cache() {
3.     // on se cache
4.     this.setVisible(false);
5.   }
```

Les vues de l'application windows, dérivées de [BaseVueJFrame], seront instanciées par configuration avec [Spring IoC].

4.6 La classe [InfosAction]

La classe [InfosAction] spécifie une action à exécuter par le contrôleur. Son code est le suivant :

```
package istia.st.m2vc.core;

import java.util.Hashtable;
import java.util.Map;

/**
 * @author serge.tahe@univ-angers.fr, juin 2005
 *
 */
public class InfosAction {
```

```
  // champs privés
  private IAction action; // l'action à exécuter
  private IVue vue; // la vue à afficher
  private Map etats; // le dictionnaire des états de l'action à exécuter

  /**
   * @return Returns the action.
   */
  public IAction getAction() {
    return action;
  }

  /**
   * @param action
   *          The action to set.
   */
  public void setAction(IAction action) {
    this.action = action;
  }

  /**
   * @return Returns the états.
   */
  public Map getEtats() {
    return etats;
  }

  /**
   * @param états
   *          The états to set.
   */
  public void setEtats(Map etats) {
    this.etats = etats;
  }

  /**
   * @return Returns the vue.
   */
  public IVue getVue() {
    return vue;
  }

  /**
   * @param vue
   *          The vue to set.
   */
  public void setVue(IVue vue) {
    this.vue = vue;
  }
}
```

C'est un code simple. La classe [InfosAction] a simplement trois propriétés publiques destinées au contrôleur :

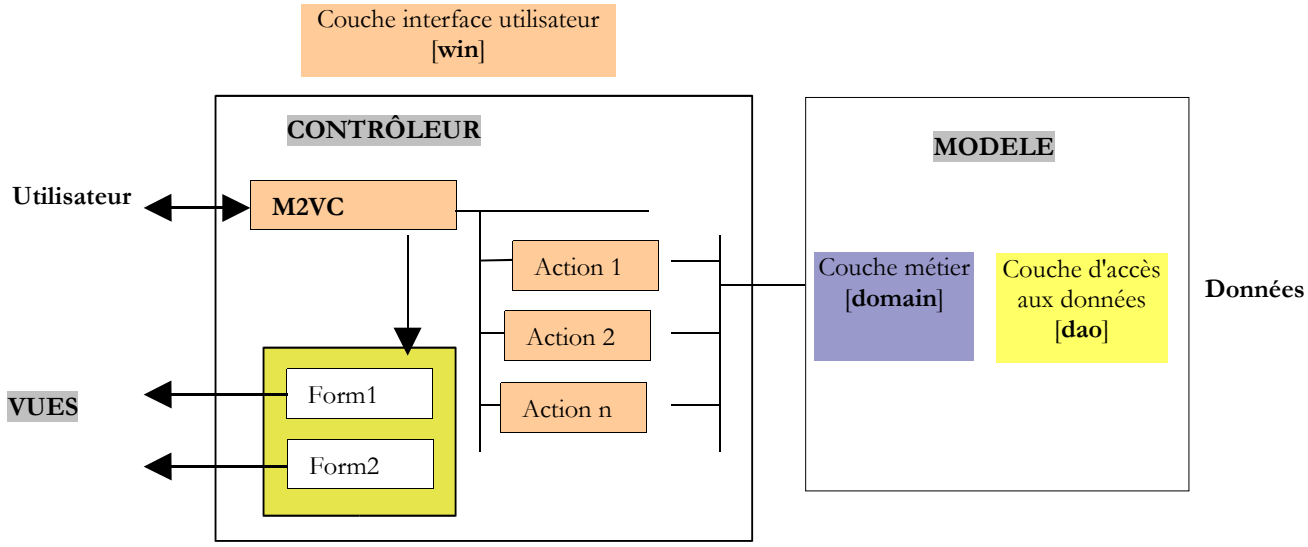
`action` une instance de classe implémentant [Iaction] à utiliser pour exécuter l'action. Peut être égal à [null]. Dans ce cas, l'attribut [vue] ci-dessous doit être initialisé.

`vue` une instance de classe implémentant [Ivue]. Désigne une vue à afficher si l'attribut [action] ci-dessus n'a pas été initialisé.

`etats` un dictionnaire d'états. N'est utile que si l'attribut [action] a été initialisé. Dans ce cas, une méthode [Iaction.execute] a été lancée. Celle-ci rend une chaîne de caractères représentant le résultat de l'action. Avec ce résultat, le contrôleur va symboliser le nouvel état de l'application en présentant à l'utilisateur la vue associée à cet état. Le dictionnaire [états] associe donc une vue [Ivue] à un résultat d'action de type [String]

Les instances de [InfosAction] seront créées par configuration avec [Spring IoC].

Pour comprendre le rôle de [InfosAction], revenons sur l'architecture de [M2VC] :



1. le contrôleur [M2VC] reçoit une demande de l'utilisateur. Celle-ci viendra sous la forme d'une chaîne de caractères indiquant l'action à entreprendre.
2. [M2VC] récupère l'instance [InfosAction] liée à cette action. Pour cela, il aura un dictionnaire associant le nom d'une action à une instance [InfosAction] rassemblant les informations nécessaires à cette action.
3. si l'attribut [vue] de [InfosAction] est non vide, alors la vue associée est affichée par [vue.affiche]. On passe ensuite à l'étape 7.
4. si l'attribut [action] de [InfosAction] est non vide, alors l'action est exécutée par [action.execute].
5. la méthode [action.execute] fait ce qu'elle a à faire puis rend au contrôleur une chaîne de caractères représentant le résultat auquel elle est parvenue.
6. le contrôleur utilise le dictionnaire [états] de [InfosAction] pour trouver la vue V à afficher. Il l'affiche par [V.affiche]
7. ici une vue a été affichée. Le contrôleur s'est synchronisé avec et attend que la vue déclenche une nouvelle action. Celle-ci va être déclenchée par une action particulière de l'utilisateur sur la vue, qui à cette occasion va repasser la main au contrôleur en lui donnant le nom de l'action à exécuter.
8. le contrôleur reprend à l'étape 1

4.7 La classe [BaseControleur]

Il ne nous reste plus qu'à présenter le code du contrôleur de base pour comprendre comment tout ce puzzle fonctionne :

```
1. package istia.st.m2vc.core;
2.
3. import java.util.Map;
4.
5. /**
6.  * @author serge.tahe@univ-angers.fr, juin 2005
7.  *
8.  */
9. public class BaseControleur
10.    implements IControleur {
11.
12.  // champs privés
13.  private Map actions; // les actions à contrôler
14.  private String firstActionName; // le nom de la 1ère action
15.  private String lastActionName; // le nom de la dernière action
16.  private Barriere synchro; // l'outil de synchronisation contrôleur - vues
17.
18.  /**
19.   * @return Returns the actions.
20.   */
21.  public Map getActions() {
22.    return actions;
23.  }
24.
25.  /**
26.   * @param actions
27.   *          The actions to set.
28.   */
29.  public void setActions(Map actions) {
30.    this.actions = actions;
31.  }
32.
33.  /**
34.   * @return Returns the firstActionName.
```

```
35.   */
36.  public String getFirstActionName() {
37.    return firstActionName;
38.  }
39.
40.  /**
41.   * @param firstActionName
42.   *          The firstActionName to set.
43.   */
44.  public void setFirstActionName(String firstActionName) {
45.    this.firstActionName = firstActionName;
46.  }
47.
48.  /**
49.   * @return Returns the lastActionName.
50.   */
51.  public String getLastActionName() {
52.    return lastActionName;
53.  }
54.
55.  /**
56.   * @param lastActionName
57.   *          The lastActionName to set.
58.   */
59.  public void setLastActionName(String lastActionName) {
60.    this.lastActionName = lastActionName;
61.  }
62.
63.  /**
64.   * @return Returns the synchro.
65.   */
66.  public Barriere getSynchro() {
67.    return synchro;
68.  }
69.
70.  /**
71.   * @param synchro
72.   *          The synchro to set.
73.   */
74.  public void setSynchro(Barriere synchro) {
75.    this.synchro = synchro;
76.  }
77.
78.  // le moteur d'exécution des actions
79.  public void run() {
80.
81.    // variables locales
82.    InfosAction configAction = null;
83.    String actionName = null;
84.    String état = null;
85.    IVue vue = null;
86.    IVue vuePrécédente = null;
87.
88.    // boucle d'exécution des actions
89.    actionName = firstActionName;
90.    while (!actionName.equals(lastActionName)) {
91.      // on récupère la config de l'action
92.      configAction = (InfosAction) actions.get(actionName);
93.      if (configAction == null) {
94.        // erreur de config - on s'arrête
95.        throw new RuntimeException("L'action [" + actionName
96.                                   + "] n'a pas été configurée correctement");
97.      }
98.      // exécution de l'action s'il y en a une
99.      if (configAction.getAction() != null) {
100.       // exécution de l'action
101.       état = configAction.getAction().execute();
102.       // on récupère la vue associée à l'état
103.       vue = (IVue) configAction.getEtats().get(état);
104.       // si vue == null, erreur de config
105.       if (vue == null) {
106.         throw new RuntimeException("L'état [" + état + "] de l'action ["
107.                                    + actionName +
108.                                    "] n'a pas été configuré correctement");
109.       }
110.     }
111.     else {
112.       // pas d'action - directement la vue
113.       état = "";
114.       vue = configAction.getVue();
115.     }
116.     // on cache la vue précédente si elle est différente de celle qui va être
117.     // affichée
118.     if (vue != vuePrécédente && vuePrécédente != null) {
119.       vuePrécédente.cache();
120.     }
121.     // on initialise la vue à afficher
```

```
122.      initVue(actionName, état, vue.getNom());
123.      // on affiche la vue en se synchronisant avec elle
124.      synchro.reset();
125.      vue.affiche();
126.      synchro.waitOne();
127.      // action suivante
128.      actionName = vue.getAction();
129.      vuePrécédente = vue;
130.    }
131.    // on cache la dernière vue affichée
132.    if (vue != null) {
133.      vue.cache();
134.    }
135.    // c'est fini
136.  }
137.
138.  public void initVue(String action, String état, String vue) {
139.    // action déléguée aux classes dérivées
140.  }
141.
142.}
```

Le contrôleur sera intancié par configuration à l'aide de [Spring Ioc]. Commençons par présenter les propriétés publiques du contrôleur.

`actions` lignes 18-31, ligne 13

un dictionnaire faisant le lien entre le nom d'une action et l'instance [InfosAction] à utiliser pour cette action. Initialisé par fichier de configuration.

`firstActionName` lignes 33-46, ligne 14

le nom de la première action à exécuter. En effet, les noms des actions sont normalement donnés par les vues. Or au départ, il n'y a pas de vue. Initialisé par fichier de configuration.

`lastActionName` lignes 48-61, ligne 15

le nom de la dernière action à exécuter. Lorsqu'une vue envoie cette action au contrôleur, celui-ci s'arrête. Initialisé par fichier de configuration.

`synchro` lignes 63-76, ligne 16

l'instance [Barriere] qui permet au contrôleur de se synchroniser avec les vues. Initialisé par fichier de configuration.

Le contrôleur implémente l'interface [IControleur] (ligne 10). Il a donc une méthode [run] (ligne 79). C'est cette méthode qui lance le contrôleur. Celui-ci, après quelques initialisations, passe son temps dans une boucle où il va exécuter les actions demandés par les vues. Il va commencer par la première, qui aura été positionnée par configuration (ligne 89). Il terminera par celle ayant été déclarée comme la dernière (ligne 90).

Inspectons la boucle d'exécution des actions :

```
1.     // boucle d'exécution des actions
2.     actionName = firstActionName;
3.     while (!actionName.equals(lastActionName)) {
4.       // on récupère la config de l'action
5.       configAction = (InfosAction) actions.get(actionName);
6.       if (configAction == null) {
7.         // erreur de config - on s'arrête
8.         throw new RuntimeException("L'action [" + actionName
9.                                    + "] n'a pas été configurée correctement");
10.      }
11.      // exécution de l'action s'il y en a une
12.      if (configAction.getAction() != null) {
13.        // exécution de l'action
14.        état = configAction.getAction().execute();
15.        // on récupère la vue associée à l'état
16.        vue = (IVue) configAction.getEtats().get(état);
17.        // si vue == null, erreur de config
18.        if (vue == null) {
19.          throw new RuntimeException("L'état [" + état + "] de l'action ["
20.                                     + actionName +
21.                                     "] n'a pas été configuré correctement");
22.        }
23.      }
24.      else {
25.        // pas d'action - directement la vue
26.        état = "";
27.        vue = configAction.getVue();
28.      }
29.      // on cache la vue précédente si elle est différente de celle qui va être
30.      // affichée
31.      if (vue != vuePrécédente && vuePrécédente != null) {
```

```
32.         vuePrécédente.cache();
33.       }
34.       // on initialise la vue à afficher
35.       initVue(actionName, état, vue.getNom());
36.       // on affiche la vue en se synchronisant avec elle
37.       synchro.reset();
38.       vue.affiche();
39.       synchro.waitOne();
40.       // action suivante
41.       actionName = vue.getAction();
42.       vuePrécédente = vue;
43.     }
44.     // on cache la dernière vue affichée
45.     if (vue != null) {
46.       vue.cache();
47.     }
48.     // c'est fini
49.
```

- l'attribut public [actions] du contrôleur contient les associations nom d'action <-> instance [InfosAction]. Le contrôleur commence donc par chercher dans son dictionnaire [actions] les informations concernant l'action à exécuter (ligne 5). S'il ne trouve rien, il lance une exception en expliquant l'erreur (ligne 8).
- une action peut soit demander l'exécution d'un objet [IAction] soit l'affichage d'une vue [Ivue].
- commençons par le cas où une instance [IAction] doit être exécutée. Dans ce cas, l'attribut [action] de l'instance [InfosAction] de l'action en cours n'est pas vide (ligne 12).
- l'instance [IAction] est alors exécutée (ligne 14). Cette exécution rend un résultat sous forme de chaîne de caractères placée ici dans la variable [état].
- on se rappelle que dans [InfosAction] on a un dictionnaire associant un état à une vue. Le contrôleur utilise ce dictionnaire pour récupérer l'instance [IVue] à afficher (ligne 16). Si cette instance n'est pas trouvée, une exception est lancée avec un message expliquant l'erreur (ligne 19).
- dans le cas où l'action ne demande que l'affichage d'une vue, celle-ci est récupérée ligne 27
- arrivé en ligne 31, on est prêt à afficher une vue [Ivue]. Si la vue à afficher n'est pas la même que la vue précédemment affichée, on cache cette dernière (lignes 31-33).
- on s'apprête à afficher une vue [IVue]. On veut laisser une chance au développeur de préparer celle-ci. On fait alors appel à une méthode [initVue] (ligne 35) en lui transmettant les informations dont on dispose :
  - le nom de l'action en cours
  - l'état rendu par cette action
  - le nom de la vue à afficher

  La méthode [initVue] a une implémentation locale qui ne fait rien (lignes 138-140 du code de la classe complète).Le développeur pourra la redéfinir s'il le souhaite.
- maintenant on peut afficher la vue. C'est fait dans les lignes 36-39. Le contrôleur ferme la barriere [synchro] (ligne 37). Il affiche la vue (ligne 38). Il attend que la vue affichée ouvre la barrière (ligne 39) pour continuer avec une nouvelle action.
- lorsque le contrôleur est débloqué (ligne 39), il récupère la nouvelle action à exécuter auprès de la vue qu'il a affichée (ligne 35).
- il note que cette vue devient désormais la vue précédente (ligne 36) avant de reboucler pour exécuter la nouvelle action (ligne 41).
- tout cela va durer un certain temps jusqu'à ce que l'action à exécuter soit la dernière action [lastActionName] (ligne 2). On se retouve alors ligne 45 où on cache la dernière vue qu'on a affichée.
- le contrôleur a fini son travail. Il rend la main au programme qui l'a lancé (ligne 136 du code complet).

4.8 Un lanceur pour M2VC

Pour instancier et lancer le contrôleur M2VC, nous utiliserons la classe suivante :

```
1. package istia.st.m2vc.appli;
2.
3. import java.io.BufferedReader;
4. import java.io.InputStreamReader;
5. import istia.st.m2vc.core.*;
6. import org.springframework.beans.factory.xml.XmlBeanFactory;
7. import org.springframework.core.io.ClassPathResource;
8. import istia.st.m2vc.core.IControleur;
9.
10./**
11. * @author serge.tahe@istia.univ-angers.fr, juin 2005
12. *
13. */
14.public class Main {
15.
16.  public static void main(String[] args) throws Exception {
17.    // variables locales
18.    IControleur monControleur = null;
19.    // message de patience
```

```
20.     System.out.println("Application en cours d'initialisation. Patientez...");
21.     try {
22.       // on instancie le contrôleur de l'application
23.       monControleur = (IControleur) (new XmlBeanFactory(new ClassPathResource(
24.           "m2vc.xml"))).getBean("controleur");
25.     }
26.     catch (Exception ex) {
27.       // on s'arrête
28.       abort("Erreur lors de l'initialisation du contrôleur M2VC-win", ex, 1);
29.     }
30.     // exécution application
31.     System.out.println("Application lancée...");
32.     try {
33.       monControleur.run();
34.     }
35.     catch (Exception ex) {
36.       // on affiche l'erreur et on s'arrête
37.       abort("Erreur d'exécution", ex, 2);
38.     }
39.     // fin normale
40.     System.out.println("Application terminée...");
41.     System.exit(0);
42.   }
43.
44.   // fin anormale
45.   private static void abort(String message, Exception ex, int exitCode) throws
46.       Exception {
47.     // on affiche l'erreur
48.     System.out.println(message);
49.     System.out.println("---------------------------------------");
50.     System.out.println(ex.toString());
51.     System.out.println("---------------------------------------");
52.     // on laisse l'utilisateur voir le message
53.     System.out.println("Tapez [enter] pour terminer l'application...");
54.     BufferedReader IN = new BufferedReader(new InputStreamReader(System.in));
55.     IN.readLine();
56.     // on s'arrête
57.     System.exit(exitCode);
58.   }
59.}
```

- le contrôleur est créé lignes 21 à 29. On demande à [Spring] une instance d'un objet appelé [controleur] (lignes 23-24). Spring va alors exploiter un fichier [m2vc.xml] pour instancier le contrôleur. Il le cherchera dans les dossiers du [classpath] de l'application. Nous alons voir bientôt la structure de ce fichier. Il est assez complexe à construire. Les erreurs de configuration vont générer des exceptions chez Spring. Le lanceur les affiche alors et s'arrête (lignes 26-29).
- si le contrôleur est instancié, il est lancé (ligne 33). A partir de maintenant, c'est le contrôleur qui a la main. Il ne la rendra qu'à deux occasions :
  - des erreurs de programmation ou de configuration qui engendrent des exceptions. Celles-ci sont affichées sur la console (lignes 35-38) et le programme est arrêté.
  - l'exécution de la dernière action [lastActionName]. Le lanceur s'arrête alors (lignes 40-41).

4.9 Le fichier de configuration [m2vc.xml]

[Spring IoC] nous permet d'avoir un contrôleur très simple. Cette simplicité est obtenue grâce au fichier de configuration [m2vc.xml]. Celui-ci n'est pas, au début, très simple à construire. Nous allons donner par la suite divers exemples. Parce que l'application ira chercher ce fichier dans les dossiers du [classpath] de l'application, nous le mettrons systématiquement dans le dossier [**classes**] du projet, là où JBuilder range les classes de celui-ci.

Examinons un premier exemple :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.     <!-- la synchronisation -->
5.     <bean id="synchro" class="istia.st.m2vc.core.Barriere"/>
6.     <!-- la session -->
7.     <bean id="session" class="istia.st.m2vc.appli.Session" />
8.     <!-- les vues -->
9.     <bean id="vueSaisir" class="istia.st.m2vc.appli.VueSaisir">
10.      <property name="nom">
11.        <value>saisir</value>
12.      </property>
13.      <property name="session">
14.        <ref bean="session" />
15.      </property>
16.      <property name="synchro">
17.        <ref bean="synchro" />
18.      </property>
19.    </bean>
```

```
20.    <bean id="vueConsulter" class="istia.st.m2vc.appli.VueConsulter">
21.      <property name="nom">
22.        <value>consulter</value>
23.      </property>
24.      <property name="session">
25.        <ref bean="session" />
26.      </property>
27.      <property name="synchro">
28.        <ref bean="synchro" />
29.      </property>
30.    </bean>
31.    <bean id="vueErreurs" class="istia.st.m2vc.appli.VueErreurs">
32.      <property name="nom">
33.        <value>erreurs</value>
34.      </property>
35.      <property name="session">
36.        <ref bean="session" />
37.      </property>
38.      <property name="synchro">
39.        <ref bean="synchro" />
40.      </property>
41.    </bean>
42.    <!-- les actions -->
43.    <bean id="actionConsulter" class="istia.st.m2vc.appli.ActionConsulter">
44.      <property name="session">
45.        <ref bean="session" />
46.      </property>
47.    </bean>
48.    <!-- la configuration des actions -->
49.    <bean id="infosActionConsulter" class="istia.st.m2vc.core.InfosAction">
50.      <property name="action">
51.        <ref bean="actionConsulter" />
52.      </property>
53.      <property name="etats">
54.        <map>
55.          <entry key="erreurs">
56.            <ref bean="vueErreurs" />
57.          </entry>
58.          <entry key="consulter">
59.            <ref bean="vueConsulter" />
60.          </entry>
61.        </map>
62.      </property>
63.    </bean>
64.    <bean id="infosActionSaisir" class="istia.st.m2vc.core.InfosAction">
65.      <property name="vue">
66.        <ref bean="vueSaisir" />
67.      </property>
68.    </bean>
69.    <!-- le contrôleur -->
70.    <bean id="controleur" class="istia.st.m2vc.core.BaseControleur">
71.      <property name="synchro">
72.        <ref bean="synchro" />
73.      </property>
74.      <property name="firstActionName">
75.        <value>saisir</value>
76.      </property>
77.      <property name="lastActionName">
78.        <value>quitter</value>
79.      </property>
80.      <property name="actions">
81.        <map>
82.          <entry key="saisir">
83.            <ref bean="infosActionSaisir" />
84.          </entry>
85.          <entry key="consulter">
86.            <ref bean="infosActionConsulter" />
87.          </entry>
88.        </map>
89.      </property>
90.    </bean>
91.  </beans>
```

- les lignes 1 à 2 sont standard. On les gardera toujours en l'état.
- la description du contrôleur commence ligne 3 où on commence à décrire tous les objets qui le composent.
- ligne 5 - on déclare l'objet [synchro] de type [Barriere] qui sert à synchroniser le contrôleur et les vues. Cet objet sera toujours présent. Il doit être injecté dans chaque vue [Ivue] et dans le controleur [IControleur].
- ligne 7 - on déclare un objet [Session]. Nous n'avons encore jamais rencontré cet objet. Ce n'est pas un objet propre au contrôleur mais à l'application particulière contrôlée ici. Nous retrouverons cet objet dans la plupart de nos exemples. En effet, le contrôleur fait interagir des vues [Ivue] et des actions [IAction] sans qu'on sache comment ces objets échangent de l'information. Il faut bien pourtant qu'elles en échangent puisqu'en général une vue affiche des informations produites par une action. C'est l'objet [Session] qui nous servira à cela. Il sera injecté dans chaque vue [Ivue] et chaque action [IAction]. Tous ces objets se partageront un même et unique objet [Session] qui leur servira à communiquer.

- lignes 8-41 - on déclare les vues [IVue] de l'application.
- lignes 9-19 - on déclare une vue identifiée par [id="vueSaisir"]. On sait qu'une vue a deux propriétés publiques : [nom] et [synchro]. Celles-ci sont initialisées lignes 10-12 et 16-18. Ces initialisations seront toujours à faire. La propriété [session] définie lignes 13-15 est propre à l'application particulière contrôlée ici et non pas une propriété initiale de [IVue]. Nous avons déjà expliqué son rôle.
- lignes 42-47, on déclare les objets [Action] de l'application. Ici, il n'y en a qu'une appelée [actionConsulter]. Rappelons que les actions sont définies par le développeur et non par le moteur M2VC.
- lignes 48-68 - on déclare les objets [InfosAction] des actions que sera amené à exécuter le contrôleur.
- lignes 49-63 - on déclare un objet [InfosAction] identifié par [id=infosActionConsulter]. On sait qu'un tel objet a trois propriétés publiques : **action** de type [IAction], **vue** de type [IVue], e**tats** de type [Map]. Les propriétés **action** et **vue** sont exclusives l'une de l'autre. L'objet [InfosAction] ici ne déclare l'action (lignes 50-52). Les états associés à l'action sont définis lignes 53-62. Ces lignes disent que si l'action rend
  - la chaîne "erreurs", alors la vue [VueErreurs] doit être affichée (lignes 55-57)
  - la chaîne "consulter", laors la vue [VueConsulter] doit être affichée (lignes 58-60)
- lignes 64-68 - on déclare un objet [InfosAction] identifié par [id=infosActionSaisir]. On n'y définit que la propriété **vue**. Cela signifie que sur cette action, la contrôleur doit se contenter d'afficher la vue précisée ligne 66, la vue [VueSaisir].
- lignes 70-90, on définit le contrôleur. Nous savons que le contrôleur a les propriétés publiques suivantes : le nom de la première action à exécuter [**firstActionName**] (lignes 74-76), le nom de la derrnière action à exécuter [**lastActionName**] (lignes 77-79), l'outil de synchronisation avec les vues [**synchro**] (lignes 71-73), le dictionnaire [**actions**] (lignes 80-89). Celui-ci est un dictionnaire associant un nom d'action (String) à un objet de type [InfosAction].

Nous venons de décrire les grandes lignes du fichier de configuration [m2vc]. La structure de celui-ci obéit aux règles de [Spring IoC]. Maîtriser celles-ci demande un peu de temps. Nous allons les revoir sur trois exemples.

5 Application 1

Notre première application n'affiche qu'une unique fenêtre :

C'est une fenêtre presque normale. Elle a cependant deux particularités :

- elle ne se ferme pas
- le bouton [Quitter] fait appel au contrôleur. C'est lui qui va fermer la fenêtre

5.1 La structure

La projet JBuilder de cette application a la structure suivante :

- les bibliothèques du projet [appli] sont les suivantes :

Le fichier [m2vc-core.jar] est l'archive issue du projet [M2VC] étudié précédemment. Les fichiers [spring-core.jar, commons-logging.jar] sont nécessaires pour utiliser Spring. Nous placerons systématiquement ces trois fichiers dans un dossier [lib] de nos exemples que le projet référencera.

5.2 Le lanceur [Main.java]

La classe [**Main.java**] du projet ci-dessus est la classe que nous avons décrite paragraphe 4.8. Elle servira à lancer l'application dans tous nos exemples.

5.3 Les vues

Comme nous l'avons dit, il n'y a qu'une vue :

Celle-ci est implémentée par la classe [MaVue.java] suivante :

```
1. package istia.st.m2vc.appli;
2.
3. import java.awt.*;
4. import java.awt.event.*;
5. import javax.swing.*;
6. import istia.st.m2vc.core.*;
7.
8.
9. public class MaVue extends BaseVueJFrame {
10.  JPanel contentPane;
11.  JSpinner jSpinner1 = new JSpinner();
12.  JButton jButtonQuitter = new JButton();
13.
14.  //Construire le cadre
15.  public MaVue() {
16.    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
17.    try {
18.      jbInit();
19.    }
20.    catch(Exception e) {
21.      e.printStackTrace();
22.    }
23.  }
24.
25.  //Initialiser le composant
26.  private void jbInit() throws Exception  {
27.    contentPane = (JPanel) this.getContentPane();
28.    contentPane.setLayout(null);
29.    this.setSize(new Dimension(220, 92));
30.    this.setTitle("MaVue");
31.    jSpinner1.setBounds(new Rectangle(21, 25, 58, 28));
32.    jButtonQuitter.setBounds(new Rectangle(92, 26, 86, 27));
33.    jButtonQuitter.setText("Quitter");
34.    jButtonQuitter.addActionListener(new MaVue_jButtonQuitter_actionAdapter(this));
35.    contentPane.add(jSpinner1, null);
36.    contentPane.add(jButtonQuitter, null);
37.  }
38.
39.  //Redéfini, ainsi nous pouvons sortir quand la fenêtre est fermée
40.  protected void processWindowEvent(WindowEvent e) {
41.  }
42.
43.  void jButtonQuitter_actionPerformed(ActionEvent e) {
44.    // action asynchrone - on gèle le formulaire
45.    this.setEnabled(false);
46.    // on passe l'action à la classe parent
47.    super.exécute("quitter");
48.  }
49.
50.  // affiche
51.  public void affiche() {
52.    // on dégèle le formulaire
53.    this.setEnabled(true);
54.    // on affiche la vue parent
55.    super.affiche();
56.  }
57.}
58.
59.class MaVue_jButtonQuitter_actionAdapter implements java.awt.event.ActionListener {
60.  MaVue adaptee;
61.
62.  MaVue_jButtonQuitter_actionAdapter(MaVue adaptee) {
```

```
63.    this.adaptee = adaptee;
64.  }
65.  public void actionPerformed(ActionEvent e) {
66.    adaptee.jButtonQuitter_actionPerformed(e);
67.  }
68.}
```

Nous ne commentons que le code propre à l'utilisation du moteur [M2VC]. Le reste est du code classique d'instance [JFrame].

- ligne 9 : la classe [MaVue] dérive de la classe [BaseVueJFrame] définie dans le contrôleur. C'est obligatoire.
- lignes 51-57 : la classe [MaVue] redéfinit la méthode [affiche]. Elle commence par "dégeler" la fenêtre (ligne 53) qui avait été gelée ligne 45. Puis elle affiche sa classe parent (ligne 55). C'est obligatoire. Toute vue redéfinissant la méthode [affiche] pour ses propres besoins doit se terminer par l'appel à la méthode [affiche] de sa classe parent.
- un clic sur le bouton [Quitter] provoque l'exécution de la méthode [jButtonQuitter_actionPerformed], ligne 43. On s'apprête à exécuter l'action asynchrone "quitter". On décide de geler le formulaire (ligne 53) pour éviter que l'utilisateur ne provoque des événements pendant l'exécution de l'action asynchrone. Le formulaire sera réactivé lors d'un futur réaffichage (qui peut ne pas avoir lieu - c'est le cas ici) dans la méthode [affiche], ligne 53. Une fois le formulaire inhibé, on demande l'exécution asynchrone de l'action "quitter" (ligne 47).
- lignes 40-41, l'événement de fermeture de la fenêtre n'est pas géré. L'utilisateur ne pourra donc la fermer. Il en sera ainsi dans toutes les vues de tous nos exemples.

5.4 Le fichier de configuration [m2vc.xml]

Le fichier de configuration [m2vc.xml] de l'application sera le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <!-- la synchro -->
5.  <bean id="synchro" class="istia.st.m2vc.core.Barriere"/>
6.  <!-- les vues -->
7.  <bean id="maVue" class="istia.st.m2vc.appli.MaVue">
8.    <property name="nom">
9.      <value>mavue</value>
10.   </property>
11.   <property name="synchro">
12.     <ref bean="synchro" />
13.   </property>
14. </bean>
15. <!-- les actions -->
16. <!-- la configuration des actions -->
17. <bean id="infosActionMaVue" class="istia.st.m2vc.core.InfosAction">
18.   <property name="vue">
19.     <ref bean="maVue" />
20.   </property>
21. </bean>
22. <!-- le contrôleur -->
23. <bean id="controleur" class="istia.st.m2vc.core.BaseControleur">
24.   <property name="synchro">
25.     <ref bean="synchro" />
26.   </property>
27.   <property name="firstActionName">
28.     <value>afficherMaVue</value>
29.   </property>
30.   <property name="lastActionName">
31.     <value>quitter</value>
32.   </property>
33.   <property name="actions">
34.     <map>
35.       <entry key="afficherMaVue">
36.         <ref bean="infosActionMaVue" />
37.       </entry>
38.     </map>
39.   </property>
40. </bean>
41.</beans>
```

- ligne 5 : on définit l'objet [synchro] qui synchronise le contrôleur et les vues.
- lignes 7-13 : l'unique vue [maVue] est définie avec ses propriétés [nom] et [synchro]
- ligne 15 : pas d'objets [IAction]
- lignes 17-21 : les objets [InfosAction] sont définis. Il n'y en a qu'un. L'objet [infosActionMaVue] demande un affichage de la vue [maVue]
- lignes 23-40 : le contrôleur est défini.
- la propriété [synchro] est définie lignes 24-26. C'est bien sûr le même objet [synchro] que pour la vue [maVue]
- la propriété [firstActionName] est définie lignes 27-29. Il s'agit de l'action nommée " afficherMaVue "
- la propriété [lastActionName] est définie lignes 30-32. Il s'agit de l'action nommée "quitter"

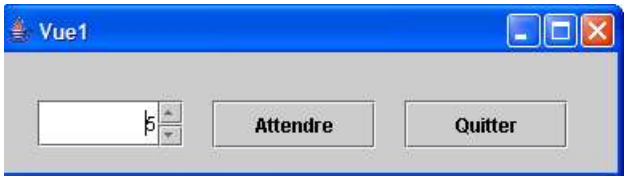
- la liste des actions autres que [lastActionName] que le contrôleur doit gérer est définie lignes 33-39. C'est ainsi que l'action nommée " afficherMaVue " est associée à l'objet [infosActionMaVue]. Si on passe à la définition de cet objet (lignes 17-21), on voit que l'action nommée " afficherMaVue " va au final provoquer l'affichage de la vue [maVue], donc l'affichage de notre formulaire.
- lorsque l'utilisateur va cliquer sur le bouton [Quitter], nous avons vu que le formulaire va alors demander au contrôleur l'exécution de l'action appelée " quitter ". Comme celle-ci est définie comme la dernière action du contrôleur [lastActionName], celui-ci va terminer l'application.

5.5 Les tests

Le lecteur est invité à tester la solution disponible en téléchargement sur le site de cet article.

6 Application 2

Notre deuxième application va, là encore, n'afficher qu'une fenêtre :



L'application est analogue à la précédente. Nous avons ajouté un bouton [Attendre] qui va déclencher une action qui sera exécutée par le contrôleur. Celui-ci fera alors exécuter une action [IAction] particulière. On va donc introduire la notion d'objets [IAction] qui ne l'avait pas encore été. L'objet [IAction] associé au bouton [Attendre] ne fera rien si ce n'est d'attendre 5 secondes. Avant de passer la main au contrôleur, lors du clic sur le bouton [Attendre], la vue [Vue1] va inhiber les boutons [Attendre] et [Quitter] mais pas l'incrémenteur. L'utilisateur va donc pouvoir pendant les 5 secondes d'attente jouer avec l'incrémenteur. On veut mettre en lumière ici, l'aspect asynchrone des actions exécutées par le contrôleur.

6.1 La structure

Le projet JBulder de l'application a la structure suivante :

6.2 Les vues

L'unique vue est implémentée par la classe [Vue1.java] :

```
1. package istia.st.m2vc.appli;
2.
3. import java.awt.*;
4. import java.awt.event.*;
5. import javax.swing.*;
6. import istia.st.m2vc.core.BaseVueJFrame;
7.
8. public class Vue1 extends BaseVueJFrame {
9.   JPanel contentPane;
10.  JSpinner jSpinner1 = new JSpinner();
11.  JButton jButtonAttendre = new JButton();
12.  JButton jButtonQuitter = new JButton();
13.
14.  //Construire le cadre
15.  public Vue1() {
16.    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
```

```
17.    try {
18.      jbInit();
19.    }
20.    catch(Exception e) {
21.      e.printStackTrace();
22.    }
23.  }
24.
25.  //Initialiser le composant
26.  private void jbInit() throws Exception  {
27.    contentPane = (JPanel) this.getContentPane();
28.    contentPane.setLayout(null);
29.    this.setSize(new Dimension(391, 109));
30.    this.setTitle("Vue1");
31.    jSpinner1.setBounds(new Rectangle(21, 30, 91, 29));
32.    jButtonAttendre.setBounds(new Rectangle(130, 30, 102, 30));
33.    jButtonAttendre.setText("Attendre");
34.    jButtonAttendre.addActionListener(new Vue1_jButtonAttendre_actionAdapter(this));
35.    jButtonQuitter.setText("Quitter");
36.    jButtonQuitter.addActionListener(new Vue1_jButtonQuitter_actionAdapter(this));
37.    jButtonQuitter.setBounds(new Rectangle(250, 30, 102, 30));
38.    contentPane.add(jSpinner1, null);
39.    contentPane.add(jButtonAttendre, null);
40.    contentPane.add(jButtonQuitter, null);
41.  }
42.
43.  //Redéfini, ainsi nous pouvons sortir quand la fenêtre est fermée
44.  protected void processWindowEvent(WindowEvent e) {
45.  }
46.
47.  // évt sur bouton [Quitter]
48.  void jButtonQuitter_actionPerformed(ActionEvent e) {
49.    // on passe la main au contrôleur
50.    super.exécute("quitter");
51.  }
52.
53.  // affichage fenêtre
54.  public void affiche(){
55.    // on met le curseur flèche
56.    this.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
57.    // état normal
58.    this.setExtendedState(JFrame.NORMAL);
59.    // on autorise le formulaire
60.    jButtonAttendre.setEnabled(true);
61.    jButtonQuitter.setEnabled(true);
62.    // on affiche la vue parent
63.    super.affiche();
64.  }
65.
66.  void jButtonAttendre_actionPerformed(ActionEvent e) {
67.    // action asynchrone - on gèle certains éléments du formulaire
68.    jButtonAttendre.setEnabled(false);
69.    jButtonQuitter.setEnabled(false);
70.    // sablier
71.    this.setCursor(new Cursor(Cursor.WAIT_CURSOR));
72.    // on passe la main au contrôleur
73.    super.exécute("attendre");
74.  }
75.}
76.
77.class Vue1_jButtonQuitter_actionAdapter implements java.awt.event.ActionListener {
78.  Vue1 adaptee;
79.
80.  Vue1_jButtonQuitter_actionAdapter(Vue1 adaptee) {
81.    this.adaptee = adaptee;
82.  }
83.  public void actionPerformed(ActionEvent e) {
84.    adaptee.jButtonQuitter_actionPerformed(e);
85.  }
86.}
87.
88.class Vue1_jButtonAttendre_actionAdapter implements java.awt.event.ActionListener {
89.  Vue1 adaptee;
90.
91.  Vue1_jButtonAttendre_actionAdapter(Vue1 adaptee) {
92.    this.adaptee = adaptee;
93.  }
94.  public void actionPerformed(ActionEvent e) {
95.    adaptee.jButtonAttendre_actionPerformed(e);
96.  }
97.}
```

- lignes 54-64, on définit la méthode [affiche] de la vue. Après quelques opérations qui lui sont propres, elle affiche sa classe de base (ligne 63). On rappelle que c'est obligatoire.

- lignes 48-51, on définit la gestion du clic sur le bouton [Quitter]. On demande au contrôleur d'exécuter l'action " quitter ". Cette action sera définie comme la dernière [lastActionName] du contrôleur et l'application se terminera donc.
- lignes 66-75, on définit la gestion du clic sur le bouton [Attendre]. On demande au contrôleur d'exécuter l'action "attendre". Avant de passer la main au contrôleur, les boutons [Attendre] et [Quitter] sont gelés et le sablier affiché (lignes 68-71). Les boutons retrouveront un état actif, lorsque la vue sera réaffichée (lignes 60-61). L'incrémenteur n'ayant pas été gelé, on devrait pouvoir l'utiliser pendant que le contrôleur exécute l'action " attendre ".

6.3 Les actions

L'unique action est implémentée par [ActionAttendre.java] :

```
1. package istia.st.m2vc.appli;
2.
3. import istia.st.m2vc.core.IAction;
4.
5. public class ActionAttendre implements IAction{
6.
7.   // méthode execute
8.   public String execute(){
9.     // implémente l'action [attente] - on attend 5 secondes
10.    try{
11.      Thread.sleep(5000);
12.    }catch(Exception ex){
13.      throw new RuntimeException(ex.toString());
14.    }
15.    // on retourne le résultat
16.    return "succès";
17.  }
18.}
```

- ligne 5 : la classe [ActionAttente] implémente l'interface [IAction]. C'est obligatoire.
- elle doit donc définir la méthode [execute] qui sera appelée par le contrôleur. C'est ce qui est fait lignes 8-17.
- dans [execute], on ne fait rien sinon attendre 5 secondes (ligne 11). La méthode [execute] doit rendre une chaîne de caractères comme résultat. Souvent nous utiliserons les résultats " succès " et " échec " pour symboliser la réussite ou non de l'action. Parfois le résultat d'une action peut être plus fin que succès/échec. On aura alors plusieurs résultats possibles. On choisira pour chacun d'eux un nom significatif. Ici rien ne peut échouer. On rend la chaîne " succès ".

6.4 Le fichier de configuration

Le fichier de configuration [m2vc.xml] de l'application est le suivant :

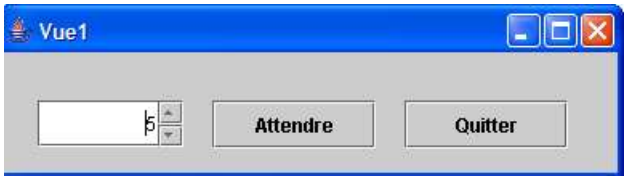
```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.  <!-- la synchronisation -->
5.  <bean id="synchro" class="istia.st.m2vc.core.Barriere"/>
6.  <!-- les vues -->
7.  <bean id="vue1" class="istia.st.m2vc.appli.Vue1">
8.    <property name="nom">
9.      <value>vue1</value>
10.   </property>
11.   <property name="synchro">
12.     <ref bean="synchro" />
13.   </property>
14. </bean>
15. <!-- les actions -->
16.         <bean id="actionAttendre" class="istia.st.m2vc.appli.ActionAttendre"/>
17. <!-- la configuration des actions -->
18. <bean id="infosActionCommencer" class="istia.st.m2vc.core.InfosAction">
19.   <property name="vue">
20.     <ref bean="vue1" />
21.   </property>
22. </bean>
23. <bean id="infosActionAttendre" class="istia.st.m2vc.core.InfosAction">
24.   <property name="action">
25.     <ref bean="actionAttendre" />
26.   </property>
27.   <property name="etats">
28.                   <map>
29.                     <entry key="succès">
30.                         <ref bean="vue1"/>
31.                     </entry>
32.                   </map>
33.   </property>
34. </bean>
35. <!-- le contrôleur -->
36. <bean id="controleur" class="istia.st.m2vc.core.BaseControleur">
```

```
37.   <property name="synchro">
38.     <ref bean="synchro" />
39.   </property>
40.   <property name="firstActionName">
41.     <value>commencer</value>
42.   </property>
43.   <property name="lastActionName">
44.     <value>quitter</value>
45.   </property>
46.   <property name="actions">
47.     <map>
48.       <entry key="commencer">
49.         <ref bean="infosActionCommencer" />
50.       </entry>
51.       <entry key="attendre">
52.         <ref bean="infosActionAttendre" />
53.       </entry>
54.     </map>
55.   </property>
56. </bean>
57.</beans>
```

- l'objet [synchro] nécessaire à la synchronisation du contrôleur et des vues est défini ligne 5
- l'unique vue [vue1] est définie lignes 7-14
- l'unique action [actionAttendre] est définie ligne 16
- les informations sur les actions sont données lignes 18-34
- l'objet [infosActionCommencer] sera associé à la première action du contrôleur. Les lignes 18-22 disent que sur cette action, la vue [vue1] doit être affichée.
- l'objet [infosActionAttendre] sera associé à l'action " attendre ", celle qui sera déclenchée par le clic sur le bouton [Attendre]. Les lignes 23-34 disent les choses suivantes :
  - lignes 24-26 : que le contrôleur doit commencer par exécuter l'action [actionAttendre] définie ligne 16.
  - lignes 27-33 : que sur le résultat " succès " de l'action [actionAttendre], la vue [vue1] doit être affichée. On voit donc qu'après l'attente de 5 secondes, la vue [vue1] sera réaffichée.
- le contrôleur est défini lignes 36-56. Les attributs [synchro, firstActionName, lastActionName] sont définis lignes 37-45, la liste des actions lignes 46-56.

6.5 Les tests

Le lecteur est invité à tester la solution disponible en téléchargement sur le site de cet article.

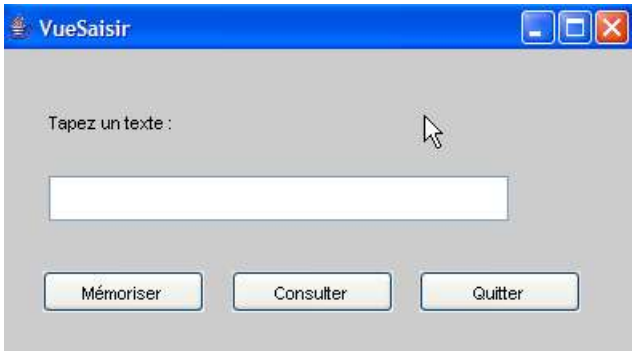


- l'incrémenteur s'utilise normalement
- le clic sur le bouton [Attendre] change la vue :

- pendant les 5 secondes de présence du sablier, on constate que l'incrémenteur est actif. Il suffit de cliquer sur sa barre de défilement avec le sablier pour s'en rendre compte.
- un clic sur le bouton [Quitter] de la vue redevenue active arrête l'application.

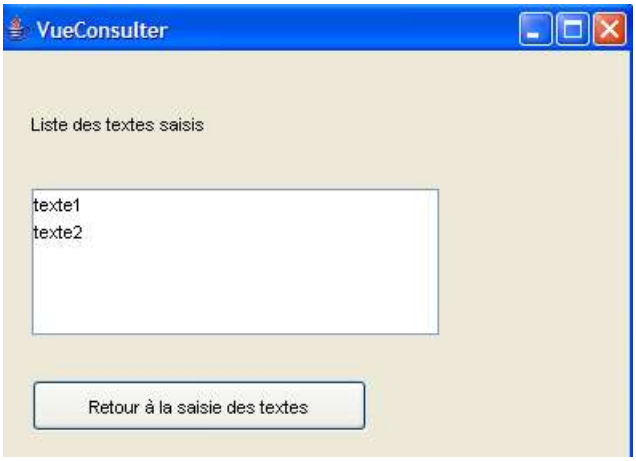
7 Application 3

Nous abordons ici une application à trois vues. L'application commence par afficher la vue [Saisir] suivante :



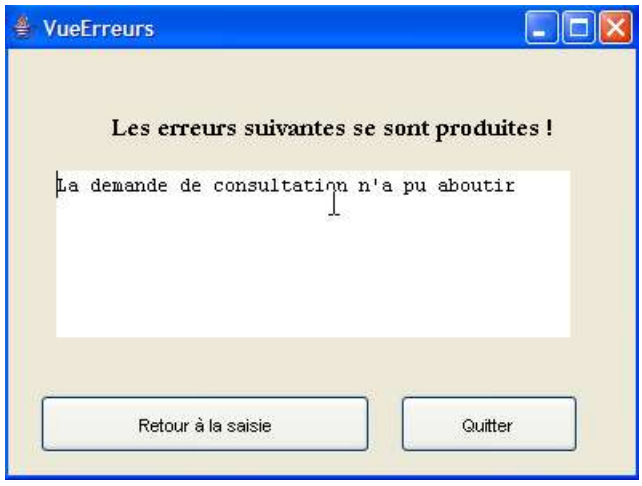
L'utilisateur entre des textes dans le champ de saisie. Le bouton [Mémoriser] les mémorise au fur et à mesure. Le bouton [Consulter] sert à voir la liste des textes saisis. L'action associée a été construite pour échouer une fois sur deux.

Lorsque l'action [Consulter] réussit, on a la vue [Consulter] suivante :

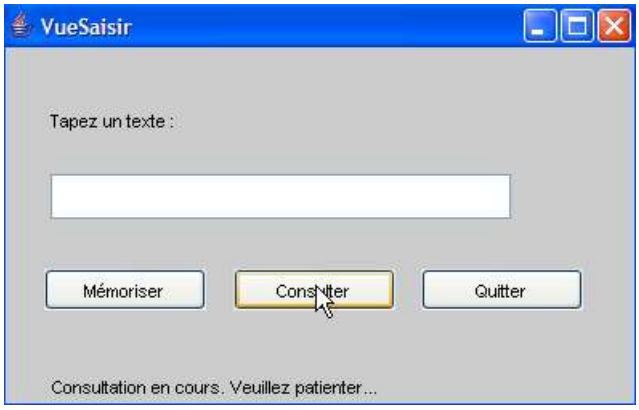


Un bouton permet de revenir à la saisie des textes.

Lorsque l'action [Consulter] échoue, on a la vue [Erreurs] suivante :

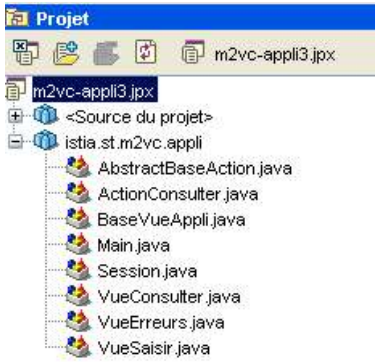


On peut alors soit revenir à la saisie des textes, soit quitter l'application. En cas d'erreurs, l'action [Consulter] a été construite pour attendre trois secondes pour rendre son résultat. Pendant cette attente, la vue [Saisir] est complètement gelée. Elle affiche un message pour faire patienter l'utilisateur :



7.1 La structure de l'application

Le projet JBuilder a la structure suivante :



7.2 La session

Dans les exemples étudiés précédemment, les actions [IAction] et les vues [IVue] n'échangeaient pas d'informations. Ce n'est pas le cas courant. Dans cette application, la vue [VueSaisir] remplit un objet [ArrayList] avec les textes qu'elle a saisis. Lorsque la vue [VueConsulter] va vouloir afficher les textes saisis, elle devra avoir accès à cet objet [ArrayList]. Le contrôleur [M2VC] ne donne aucune aide pour ces échanges d'informations entre vues et actions. C'est au développeur d'organiser ceux-ci. Nous proposons ici une méthode simple qui devrait suffire pour beaucoup d'applications. Il s'agit de créer un unique objet qui contiendra toutes les informations à partager entre les vues et les actions. Il n'y a pas de problème de synchronisation. Les objets du contrôleur ne sont jamais amenés à accéder à cet objet partagé de façon simultanée. Nous appellerons cet objet [Session] par similitude avec l'objet [Session] des applications web dans lequel on stocke tout ce qu'on veut garder au fil des échanges client-serveur. Tous les objets d'une application web ont accès à cet objet [Session] comme ce sera le cas ici.

Notre classe [Session] sera la suivante :

```
package istia.st.m2vc.appli;

import java.util.ArrayList;

public class Session {
  // liste de textes
  private ArrayList textes=new ArrayList();;
  // liste d'erreurs
  private ArrayList erreurs=new ArrayList();

  // getters-setters
  public ArrayList getErreurs() {
    return erreurs;
  }
  public ArrayList getTextes() {
    return textes;
  }
  public void setErreurs(ArrayList erreurs) {
    this.erreurs = erreurs;
```

```
  }
  public void setTextes(ArrayList textes) {
    this.textes = textes;
  }

}
```

Elle contiendra la liste des textes mémorisés par la vue [VueSaisir] et les erreurs à afficher par la vue [VueErreurs].

7.3 Les vues

Il ya trois vues. Elles sont implémentées par les fichiers [BaseVueAppli.java, VueConsulter.java, VueErreurs.java, VueSaisir.java].

7.3.1 La vue [BaseVueAppli]

La vue [BaseVueAppli] est une classe de base pour les vues [VueConsulter, VueErreurs, VueSaisir]. On y a mis le comportement commun des trois vues :

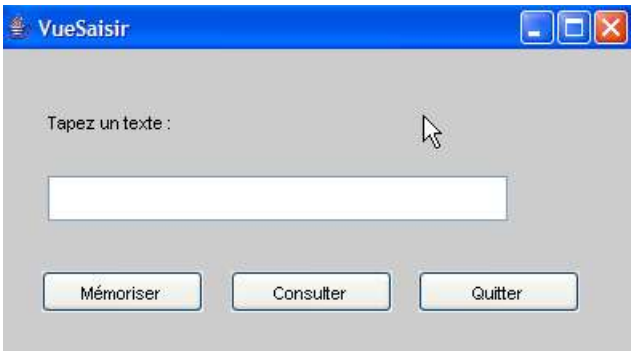
```
1. package istia.st.m2vc.appli;
2.
3. import istia.st.m2vc.core.BaseVueJFrame;
4. import java.awt.*;
5. import java.awt.event.*;
6. import javax.swing.*;
7.
8. public class BaseVueAppli
9.     extends BaseVueJFrame {
10.  // session de l'application commune aux vues et actions
11.  private Session session;
12.
13.  // getters-setters
14.  public Session getSession() {
15.    return session;
16.  }
17.
18.  public void setSession(Session session) {
19.    this.session = session;
20.  }
21.
22.  public BaseVueAppli() {
23.    try {
24.      // on fixe le look and feel des composants
25.      UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
26.    }
27.    catch (Exception ex) {
28.      throw new RuntimeException(ex.toString());
29.    }
30.  }
31.
32.  // affiche
33.  public void affiche() {
34.    // dimension normal
35.    this.setExtendedState(JFrame.NORMAL);
36.    // formulaire dégelé
37.    this.setEnabled(true);
38.    // affichage vue parent
39.    super.affiche();
40.  }
41.
42.  // exécute
43.  protected void exécuteAction(String action) {
44.    // action asynchrone - on gèle le formulaire
45.    this.setEnabled(false);
46.    // on passe la main à la classe parent
47.    super.exécute(action);
48.  }
49.}
```

- l'objet [Session] défini précédemment devra être injecté dans tous les actions [IAction] et les vues [IVue]. La classe [BaseVueAppli] définit donc une propriété publique appelée [session] pour intégrer cet objet (lignes 11-20).
- la méthode [affiche] (lignes 33-40) définit un mode d'affichage standard pour toutes les vues de l'application. Elle se termine par l'affichage de la classe de base (ligne 39). C'est obligatoire.
- la méthode [exécuteAction] a pour but de donner un comportement standard à l'exécution des actions asynchrones des différentes vues :
  - la vue est gelée - ligne 45
  - l'exécution de l'action est déléguée à la classe de base [BaseVueAppli] (ligne 47).
  - la vue retrouvera un état actif lorsqu'elle sera réaffichée - ligne 37

- le constructeur, lignes 22-30, défini comme "LookAndFeel" des vues, celui du système.

7.3.2 La vue [VueSaisir]

Rappelons l'aspect visuel ce cette vue :



Le code de la classe [VueSaisir] est le suivant :

```
1. package istia.st.m2vc.appli;
2.
3. import java.awt.*;
4. import java.awt.event.*;
5. import javax.swing.*;
6. import istia.st.m2vc.core.BaseVueJFrame;
7.
8. public class VueSaisir
9.     extends BaseVueAppli {
10.  JPanel contentPane;
11.  JLabel jLabel1 = new JLabel();
12.  JTextField jTextFieldSaisie = new JTextField();
13.  JButton jButtonMemoriser = new JButton();
14.  JButton jButtonConsulter = new JButton();
15.  JButton Quitter = new JButton();
16.  JLabel jLabelEtat = new JLabel();
17.
18.  //Construire le cadre
19.  public VueSaisir() {
20.    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
21.    try {
22.      jbInit();
23.    }
24.    catch (Exception e) {
25.      e.printStackTrace();
26.    }
27.  }
28.
29.  //Initialiser le composant
30.  private void jbInit() throws Exception {
31.    contentPane = (JPanel)this.getContentPane();
32.    jLabel1.setText("Tapez un texte :");
33.    jLabel1.setBounds(new Rectangle(28, 32, 144, 27));
34.    contentPane.setLayout(null);
35.    this.setSize(new Dimension(400, 256));
36.    this.setTitle("VueSaisir");
37.    jTextFieldSaisie.setSelectionStart(11);
38.    jTextFieldSaisie.setText("");
39.    jTextFieldSaisie.setBounds(new Rectangle(28, 79, 287, 28));
40.    jButtonMemoriser.setBounds(new Rectangle(24, 138, 101, 26));
41.    jButtonMemoriser.setToolTipText("");
42.    jButtonMemoriser.setText("Mémoriser");
43.    jButtonMemoriser.addActionListener(new
44.                                       VueSaisir_jButtonMemoriser_actionAdapter(this));
45.    jButtonConsulter.setText("Consulter");
46.    jButtonConsulter.addActionListener(new VueSaisir_jButtonConsulter_actionAdapter(this));
47.    jButtonConsulter.setBounds(new Rectangle(142, 138, 101, 26));
48.    jButtonConsulter.setToolTipText("");
49.    jButtonConsulter.setActionCommand("jButton1");
50.    Quitter.setText("Quitter");
51.    Quitter.addActionListener(new VueSaisir_Quitter_actionAdapter(this));
52.    Quitter.setBounds(new Rectangle(259, 138, 101, 26));
53.    Quitter.setToolTipText("");
54.    jLabelEtat.setText("Consultation en cours. Veuillez patienter...");
55.    jLabelEtat.setBounds(new Rectangle(29, 196, 343, 30));
56.    contentPane.add(jLabel1, null);
57.    contentPane.add(jTextFieldSaisie, null);
58.    contentPane.add(jButtonMemoriser, null);
59.    contentPane.add(jButtonConsulter, null);
```

```
60.    contentPane.add(Quitter, null);
61.    contentPane.add(jLabelEtat, null);
62.  }
63.
64.  //Redéfini, ainsi nous pouvons sortir quand la fenêtre est fermée
65.  protected void processWindowEvent(WindowEvent e) {
66.  }
67.
68.  void jButtonMemoriser_actionPerformed(ActionEvent e) {
69.    // on récupère le texte saisi
70.    String texte = jTextFieldSaisie.getText().trim();
71.    // on refuse les textes vides
72.    if ("".equals(texte)) {
73.      // message d'erreur et retour à la saisie
74.      JOptionPane.showMessageDialog(this, "Vous n'avez pas saisi de texte",
75.                                    "Erreur", JOptionPane.INFORMATION_MESSAGE);
76.      return;
77.    }
78.    // on mémorise le texte dans la session
79.    getSession().getTextes().add(texte);
80.    // on nettoie
81.    jTextFieldSaisie.setText("");
82.  }
83.
84.  void jButtonConsulter_actionPerformed(ActionEvent e) {
85.  // on affiche le label d'état
86.  jLabelEtat.setVisible(true);
87.  // on passe la main à la classe parent
88.  super.exécuteAction("consulter");
89.  }
90.
91.  void Quitter_actionPerformed(ActionEvent e) {
92.  // on passe la main à la classe parent
93.  super.exécute("quitter");
94.  }
95.
96.  // affiche
97.  public void affiche(){
98.    // on cache le label d'état
99.    jLabelEtat.setVisible(false);
100.    // affiche classe parent
101.    super.affiche();
102.  }
103.}
104.
105.class VueSaisir_jButtonMemoriser_actionAdapter
106.    implements java.awt.event.ActionListener {
107.  VueSaisir adaptee;
108.
109.  VueSaisir_jButtonMemoriser_actionAdapter(VueSaisir adaptee) {
110.    this.adaptee = adaptee;
111.  }
112.
113.  public void actionPerformed(ActionEvent e) {
114.    adaptee.jButtonMemoriser_actionPerformed(e);
115.  }
116.}
117.
118.class VueSaisir_jButtonConsulter_actionAdapter implements java.awt.event.ActionListener {
119.  VueSaisir adaptee;
120.
121.  VueSaisir_jButtonConsulter_actionAdapter(VueSaisir adaptee) {
122.    this.adaptee = adaptee;
123.  }
124.  public void actionPerformed(ActionEvent e) {
125.    adaptee.jButtonConsulter_actionPerformed(e);
126.  }
127.}
128.
129.class VueSaisir_Quitter_actionAdapter implements java.awt.event.ActionListener {
130.  VueSaisir adaptee;
131.
132.  VueSaisir_Quitter_actionAdapter(VueSaisir adaptee) {
133.    this.adaptee = adaptee;
134.  }
135.  public void actionPerformed(ActionEvent e) {
136.    adaptee.Quitter_actionPerformed(e);
137.  }
138.}
```

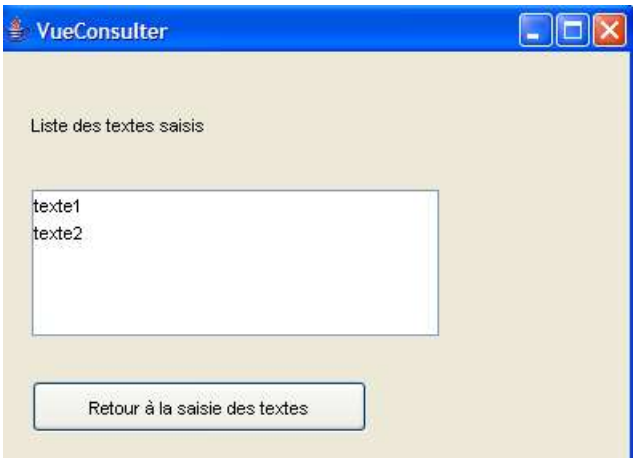
- lignes 8-9 : la classe [VueSaisir] dérive de la classe [BaseVueAppli]. Rappelons que cette dernière dérive elle-même de la classe [BaseVueJFrame] de [M2VC].
- lignes 96-103 : la méthode [affiche] est définie. Elle ne fait que ce qui est propre à la vue puisqu'elle passe ensuite la main à sa classe de base [BaseVueAppli] (ligne 101). Rappelons que celle-ci va geler le formulaire puis passer à son tour la main à sa classe

de base [BaseVueJFrame]. L'exécution de la méthode [VueSaisir.affiche] se termine donc par celle de la méthode [BaseVueJFrame.affiche]. C'est obligatoire.

- lignes 68-82 : on traite le clic sur le bouton [Mémoriser]. Le texte saisi par l'utilisateur est mémorisé dans l'attribut [textes] de l'objet [Session] de la vue (ligne 79). Cet objet est une propriété de la classe de base [BaseVueAppli].
- lignes 84-89 : on traite le clic sur le bouton [Consulter]. On rend visible le texte "Consultation en cours. Patientez..." (ligne 86) et on passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "consulter" (ligne 88). Le texte "Consultation en cours. Patientez..." sera caché lorsque la vue redeviendra active (ligne 99).
- lignes 91-94 : on traite le clic sur le bouton [Quitter]. On passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "quitter" (ligne 93).

7.3.3 La vue [VueConsulter]

Rappelons l'aspect visuel ce cette vue :



Le code de la classe [VueConsulter] est le suivant :

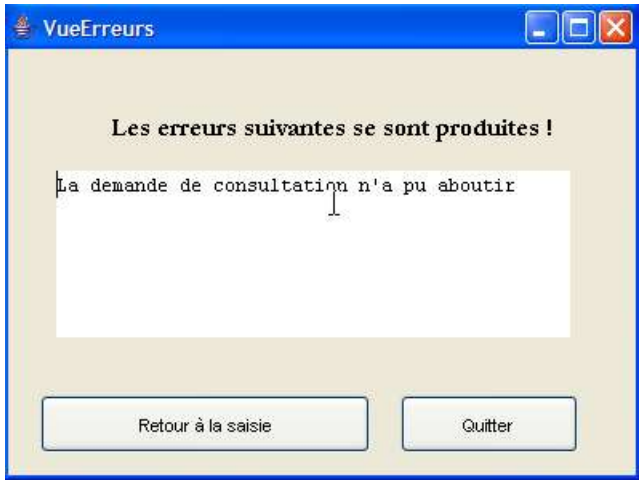
```
1. package istia.st.m2vc.appli;
2.
3. import java.awt.*;
4. import java.awt.event.*;
5. import javax.swing.*;
6. import java.util.ArrayList;
7.
8. public class VueConsulter
9.     extends BaseVueAppli {
10.  JPanel contentPane;
11.  JLabel jLabel1 = new JLabel();
12.  JScrollPane jScrollPane1 = new JScrollPane();
13.  DefaultListModel listTextes = new DefaultListModel();
14.  JList jList1 = new JList(listTextes);
15.  JButton jButtonRetour = new JButton();
16.
17.  //Construire le cadre
18.  public VueConsulter() {
19.    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
20.    try {
21.      jbInit();
22.    }
23.    catch (Exception e) {
24.      e.printStackTrace();
25.    }
26.  }
27.
28.  //Initialiser le composant
29.  private void jbInit() throws Exception {
30.    contentPane = (JPanel)this.getContentPane();
31.    jLabel1.setText("Liste des textes saisis");
32.    jLabel1.setBounds(new Rectangle(19, 30, 210, 30));
33.    contentPane.setLayout(null);
34.    this.setSize(new Dimension(400, 300));
35.    this.setTitle("VueConsulter");
36.    jScrollPane1.setBounds(new Rectangle(19, 86, 254, 91));
37.    jButtonRetour.setBounds(new Rectangle(19, 205, 209, 32));
38.    jButtonRetour.setText("Retour à la saisie des textes");
39.    jButtonRetour.addActionListener(new
40.                                    VueConsulter_jButtonRetour_actionAdapter(this));
41.    contentPane.add(jLabel1, null);
42.    contentPane.add(jScrollPane1, null);
43.    contentPane.add(jButtonRetour, null);
```

```
44.     jScrollPane1.getViewport().add(jList1, null);
45.   }
46.
47.   //Redéfini, ainsi nous pouvons sortir quand la fenêtre est fermée
48.   protected void processWindowEvent(WindowEvent e) {
49.   }
50.
51.   // bouton [retour]
52.   void jButtonRetour_actionPerformed(ActionEvent e) {
53.     // retour aux saisies
54.     super.exécuteAction("saisir");
55.   }
56.
57.   // affiche
58.   public void affiche(){
59.     // affiche les textes de la session
60.     ArrayList textes=getSession().getTextes();
61.     // nettoyage préalable
62.     listTextes.clear();
63.     // remplissage
64.     for(int i=0;i<textes.size();i++){
65.       listTextes.addElement(textes.get(i));
66.     }
67.     // affiche la classe parent
68.     super.affiche();
69.   }
70.}
71.
72.class VueConsulter_jButtonRetour_actionAdapter
73.     implements java.awt.event.ActionListener {
74.   VueConsulter adaptee;
75.
76.   VueConsulter_jButtonRetour_actionAdapter(VueConsulter adaptee) {
77.     this.adaptee = adaptee;
78.   }
79.
80.   public void actionPerformed(ActionEvent e) {
81.     adaptee.jButtonRetour_actionPerformed(e);
82.   }
83.}
```

- lignes 8-9 : la classe [VueConsulter] dérive de la classe [BaseVueAppli] qui dérive elle-même de la classe [BaseVueJFame] de [M2VC].
- lignes 58-69 : la méthode [affiche] est définie. Elle fait ce qui est propre à la vue puis passe la main à sa classe de base [BaseVueAppli] (ligne 68). La méthode [affiche] locale affiche les textes mémorisés par la vue [VueSaisir]. Elle les trouve dans l'objet [Session] de sa classe de base, objet qu'elle partage avec toutes les autres vues.
- lignes 52-55 : on traite le clic sur le bouton [Retour...]. On passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "saisir" (ligne 54).

7.3.4 La vue [VueErreurs]

Rappelons l'aspect visuel ce cette vue :



Le code de la classe [VueErreurs] est le suivant :

```
1. package istia.st.m2vc.appli;
2.
3. import java.awt.*;
4. import java.awt.event.*;
```

```
5. import javax.swing.*;
6. import java.util.ArrayList;
7.
8. public class VueErreurs
9.     extends BaseVueAppli {
10.  JPanel contentPane;
11.  JLabel jLabel1 = new JLabel();
12.  JTextArea jTextAreaTextes = new JTextArea();
13.  JButton jButtonRetour = new JButton();
14.  JButton jButtonQuitter = new JButton();
15.
16.  //Construire le cadre
17.  public VueErreurs() {
18.    enableEvents(AWTEvent.WINDOW_EVENT_MASK);
19.    try {
20.      jbInit();
21.    }
22.    catch (Exception e) {
23.      e.printStackTrace();
24.    }
25.  }
26.
27.  //Initialiser le composant
28.  private void jbInit() throws Exception {
29.    contentPane = (JPanel)this.getContentPane();
30.    jLabel1.setFont(new java.awt.Font("Garamond", 1, 16));
31.    jLabel1.setHorizontalAlignment(SwingConstants.CENTER);
32.    jLabel1.setText("Les erreurs suivantes se sont produites !");
33.    jLabel1.setBounds(new Rectangle(15, 27, 374, 44));
34.    contentPane.setLayout(null);
35.    this.setSize(new Dimension(400, 300));
36.    this.setTitle("VueErreurs");
37.    jTextAreaTextes.setText("");
38.    jTextAreaTextes.setBounds(new Rectangle(30, 76, 321, 104));
39.    jButtonRetour.setBounds(new Rectangle(20, 216, 206, 36));
40.    jButtonRetour.setText("Retour à la saisie");
41.    jButtonRetour.addActionListener(new VueErreurs_jButtonRetour_actionAdapter(this));
42.    jButtonQuitter.setBounds(new Rectangle(245, 216, 111, 36));
43.    jButtonQuitter.setText("Quitter");
44.    jButtonQuitter.addActionListener(new
45.                                     VueErreurs_jButtonQuitter_actionAdapter(this));
46.    contentPane.add(jLabel1, null);
47.    contentPane.add(jTextAreaTextes, null);
48.    contentPane.add(jButtonRetour, null);
49.    contentPane.add(jButtonQuitter, null);
50.  }
51.
52.  //Redéfini, ainsi nous pouvons sortir quand la fenêtre est fermée
53.  protected void processWindowEvent(WindowEvent e) {
54.  }
55.
56.  // bouton [retour]
57.  void jButtonRetour_actionPerformed(ActionEvent e) {
58.    // passage de l'action à la classe parent
59.    super.exécuteAction("saisir");
60.  }
61.
62.  // bouton [quitter]
63.  void jButtonQuitter_actionPerformed(ActionEvent e) {
64.    // passage de l'action à la classe parent
65.    super.exécuteAction("quitter");
66.  }
67.
68.  // affiche
69.  public void affiche() {
70.    // affiche les erreurs de la session
71.    ArrayList erreurs = getSession().getErreurs();
72.    // prépare le texte à afficher
73.    String texte = "";
74.    for (int i = 0; i < erreurs.size(); i++) {
75.      texte += erreurs.get(i).toString() + "\n";
76.    }
77.    // affichage du texte
78.    jTextAreaTextes.setText(texte);
79.    // affichage classe parent
80.    super.affiche();
81.  }
82.}
83.
84.class VueErreurs_jButtonRetour_actionAdapter
85.    implements java.awt.event.ActionListener {
86.  VueErreurs adaptee;
87.
88.  VueErreurs_jButtonRetour_actionAdapter(VueErreurs adaptee) {
89.    this.adaptee = adaptee;
90.  }
91.
```

```
92.  public void actionPerformed(ActionEvent e) {
93.    adaptee.jButtonRetour_actionPerformed(e);
94.  }
95.}
96.
97.class VueErreurs_jButtonQuitter_actionAdapter
98.    implements java.awt.event.ActionListener {
99.  VueErreurs adaptee;
100.
101.  VueErreurs_jButtonQuitter_actionAdapter(VueErreurs adaptee) {
102.    this.adaptee = adaptee;
103.  }
104.
105.  public void actionPerformed(ActionEvent e) {
106.    adaptee.jButtonQuitter_actionPerformed(e);
107.  }
108.}
```

- lignes 8-9 : la classe [VueErreurs] dérive de la classe [BaseVueAppli] qui dérive elle-même de la classe [BaseVueJFrame] de [M2VC].
- lignes 69-82 : la méthode [affiche] est définie. Elle fait ce qui est propre à la vue puis passe la main à sa classe de base [BaseVueAppli] (ligne 80). La méthode [affiche] locale affiche les erreurs construites par l'action [ActionConsulter] qui va être présentée prochainement. Elle trouve ces erreurs dans l'objet [Session] de sa classe de base, objet partagé entre toutes les vues et actions.
- lignes 57-60 : on traite le clic sur le bouton [Retour...]. On passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "saisir" (ligne 59).
- lignes 63-66 : on traite le clic sur le bouton [Quitter]. On passe la main à la classe de base [BaseVueAppli] pour faire exécuter l'action asynchrone "quitter" (ligne 65).

7.4 Les actions

Nous n'avons ici qu'une action. La classe [ActionConsulter] est chargée d'exécuter l'action asynchrone "consulter". Nous avons éprouvé le besoin de nous placer dans un cadre plus général où il y aurait plusieurs actions qui dériveraient toutes d'une classe abstraite [AbstractAction]. Celle-ci factoriserait tout ce que les différentes actions ont en commun.

7.4.1 L'action [AbstractAction]

Le code de [AbstractAction] est le suivant :

```
1. package istia.st.m2vc.appli;
2.
3. import istia.st.m2vc.core.IAction;
4.
5. public abstract class AbstractBaseAction
6.     implements IAction {
7.   // la session commune aux actions et vues
8.   private Session session;
9.
10.  // getters-setters
11.  public Session getSession() {
12.    return session;
13.  }
14.
15.  public void setSession(Session session) {
16.    this.session = session;
17.  }
18.
19.  // méthode execute laissée à la charge des classes dérivées
20.  public abstract String execute();
21.}
```

- la classe implémente l'interface [IAction] (ligne 6). Elle doit donc implémenter la méthode [execute] de cette interface. C'est fait ligne 20. On ne sait pas quoi exécuter. Seules les classes dérivées le sauront. La méthode [execute] est donc marquée abstraite (abstract) ce qui entraîne que la classe est elle-même abstraite (attribut abstract, ligne 5). On rappelle qu'une classe abstaite est une classe qu'on doit obligatoirement dériver pour en avoir des instances.
- nous avons dit que la communication [actions-vues] se ferait via un unique objet [Session] partagé par tous les objets [actions-vues]. L'objet [Session] sera injecté dans [AbstractBaseAction] grâce à l'attribut public [session] (lignes 8-17).

7.4.2 L'action [ActionConsulter]

Le code de [ActionConsulter] est le suivant :

```
1. package istia.st.m2vc.appli;
```

```
2.
3. import java.util.Random;
4.
5. public class ActionConsulter
6.     extends AbstractBaseAction {
7.
8.   // bascule
9.   private boolean bascule=false;
10.
11.  // exécution de l'action [consulter]
12.  public String execute() {
13.    // on change la bascule
14.    bascule=! bascule;
15.    if (bascule) {
16.      // on patiente 3 secondes
17.      try {
18.        Thread.sleep(3000);
19.      }
20.      catch (Exception ex) {
21.        throw new RuntimeException(ex.toString());
22.      }
23.      // on signale une erreur
24.      getSession().getErreurs().clear();
25.      getSession().getErreurs().add("La demande de consultation n'a pu aboutir");
26.      // on rend le résultat [erreurs]
27.      return "erreurs";
28.    }
29.    else {
30.      // on rend l'état [consulter]
31.      return "consulter";
32.    }
33.  }
34.}
```

- lignes 5-6 : la classe [ActionConsulter] dérive de la classe [AbstractAction]
- lignes 12-33 : la classe [ActionConsulter] implémente la méthode [execute] que n'avait pas implémentée sa classe de base [AbstractAction]. Qu'y fait on ?
- ligne 9, on défini une bascule qui aura alternativement les valeurs vrai - faux
- si la bascule est vraie (ligne 15), on s'arrête pendant 3 secondes (ligne 18). Puis lignes 24-25, on insère un message d'erreur arbitraire dans l'attribut [erreurs] de l'objet [Session] de la classe de base [AbstractAction].
- ligne 27 - l'action est terminée. On rend la chaîne "erreurs" pour signaler qu'on a rencontré des problèmes.
- si la bascule est fausse, on ne fait rien et on rend directement la chaîne "consulter" pour signaler que tout s'est bien passé (ligne 31).

7.5 Le fichier de configuration

Le fichier de configuration [m2vc.xml] de l'application est le suivant :

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>
2. <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3. <beans>
4.   <!-- la synchronisation -->
5.   <bean id="synchro" class="istia.st.m2vc.core.Barriere"/>
6.     <!-- la session -->
7.     <bean id="session" class="istia.st.m2vc.appli.Session" />
8.     <!-- les vues -->
9.     <bean id="vueSaisir" class="istia.st.m2vc.appli.VueSaisir">
10.      <property name="nom">
11.        <value>saisir</value>
12.      </property>
13.      <property name="session">
14.        <ref bean="session" />
15.      </property>
16.      <property name="synchro">
17.        <ref bean="synchro" />
18.      </property>
19.    </bean>
20.    <bean id="vueConsulter" class="istia.st.m2vc.appli.VueConsulter">
21.      <property name="nom">
22.        <value>consulter</value>
23.      </property>
24.      <property name="session">
25.        <ref bean="session" />
26.      </property>
27.      <property name="synchro">
28.        <ref bean="synchro" />
29.      </property>
30.    </bean>
31.    <bean id="vueErreurs" class="istia.st.m2vc.appli.VueErreurs">
32.      <property name="nom">
```

```
33.       <value>erreurs</value>
34.     </property>
35.     <property name="session">
36.       <ref bean="session" />
37.     </property>
38.     <property name="synchro">
39.       <ref bean="synchro" />
40.     </property>
41.   </bean>
42.   <!-- les actions -->
43.   <bean id="actionConsulter" class="istia.st.m2vc.appli.ActionConsulter">
44.     <property name="session">
45.       <ref bean="session" />
46.     </property>
47.   </bean>
48.   <!-- la configuration des actions -->
49.   <bean id="infosActionConsulter" class="istia.st.m2vc.core.InfosAction">
50.     <property name="action">
51.       <ref bean="actionConsulter" />
52.     </property>
53.     <property name="etats">
54.       <map>
55.         <entry key="erreurs">
56.           <ref bean="vueErreurs" />
57.         </entry>
58.         <entry key="consulter">
59.           <ref bean="vueConsulter" />
60.         </entry>
61.       </map>
62.     </property>
63.   </bean>
64.   <bean id="infosActionSaisir" class="istia.st.m2vc.core.InfosAction">
65.     <property name="vue">
66.       <ref bean="vueSaisir" />
67.     </property>
68.   </bean>
69.   <!-- le contrôleur -->
70.   <bean id="controleur" class="istia.st.m2vc.core.BaseControleur">
71.     <property name="synchro">
72.       <ref bean="synchro" />
73.     </property>
74.     <property name="firstActionName">
75.       <value>saisir</value>
76.     </property>
77.     <property name="lastActionName">
78.       <value>quitter</value>
79.     </property>
80.     <property name="actions">
81.       <map>
82.         <entry key="saisir">
83.           <ref bean="infosActionSaisir" />
84.         </entry>
85.         <entry key="consulter">
86.           <ref bean="infosActionConsulter" />
87.         </entry>
88.       </map>
89.     </property>
90.   </bean>
91. </beans>
```

- l'objet [synchro] nécessaire à la synchronisation du contrôleur et des vues est défini ligne 5
- l'objet [Session] partagé par les actions et les vues est défini ligne 7. Il sera injecté dans chaque vue et chaque action.
- les vues [VueSaisir, VueConsulter, VueErreurs] sont définies lignes 9-41. On y définit leurs attributs [nom, synchro, session].
- l'unique action [ActionConsulter] est définie lignes 43-47. On y injecte l'objet [Session] défini ligne 7.
- les informations sur les différentes actions du contrôleur sont données lignes 49-68.
- l'objet [infosActionSaisir] sera associé à la première action du contrôleur. Les lignes 64-68 disent que sur cette action, la vue [VueSaisir] doit être affichée.
- l'objet [infosActionConsulter] sera associé à l'action "consulter", celle qui sera déclenchée par le clic sur le bouton [Consulter]. Les lignes 49-63 disent les choses suivantes :
  - lignes 50-52 : que le contrôleur doit commencer par exécuter l'action [actionConsulter] définie lignes 43-47.
  - lignes 54-61 : que sur le résultat "erreurs" de l'action [actionConsulter], la vue [VueErreurs] doit être affichée et que sur le résultat "consulter", la vue [VueConsulter] doit être affichée.
- le contrôleur est défini lignes 70-90. Les attributs [synchro, firstActionName, lastActionName] sont définis lignes 71-79, la liste des actions contrôlées, lignes 80-89.

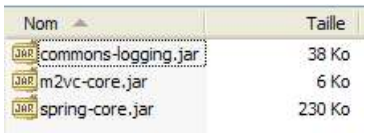
7.6 Les tests

Le lecteur est invité à tester la solution disponible en téléchargement sur le site de cet article.

8 Conclusion

Nous avons décrit dans ce document un moteur MVC pour des applications graphiques Java que nous avons appelé [M2VC]. Son architecture est inspirée de celle de Struts, le moteur MVC très utilisé dans le domaine des applications web Java. Nous avons construit trois applications élémentaires utilisant le moteur [M2VC]. Seule l'utilisation intensive du moteur dans différentes applications pourrait attester de sa stabilité et de son intérêt. Pour l'instant c'est de la bêta...

[M2VC] est un bon exemple de l'intérêt de [Spring IoC]. Regardons la taille des archives des différents exemples :



On voit que le moteur [m2vc-core] est peu volumineux (6 K). Le code principal de [Spring] (spring- core) occupe lui 230K. C'est l'effet [Spring]. On s'est déchargé sur [Spring] de tout un travail lourd et fastidieux d'instanciations et de gestions de singletons. Cela a conduit à une réduction drastique du code de [M2VC] qui se concrétise par le faible poids de son archive.

Table des matières