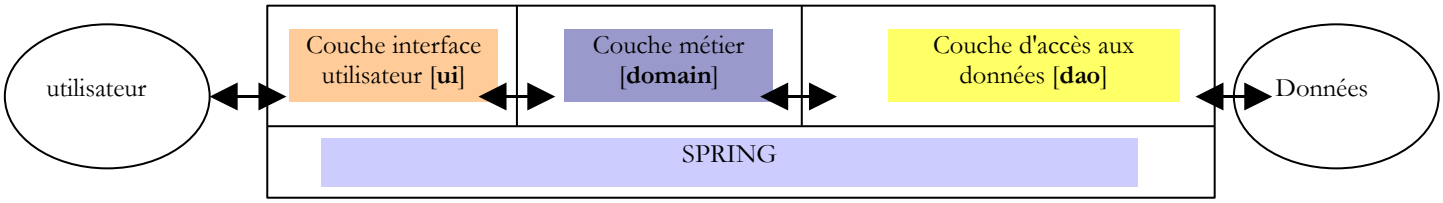
ERRATA - MAGASIN VIRTUEL

1 28 juillet 2005

Entre mars et juillet 2005, un certain nombre d'articles sont parus sur [http://tahe.developpez.com]. Ils avaient pour but de présenter le framework Spring aussi bien dans le monde Java que dans le monde .NET. Une application simplifiée d'achats de produits sur le web a servi de fil conducteur à tous les articles. Celle-ci a l'architecture suivante :



Une erreur de conception s'est glissée dès les premiers articles parus en mars 2005, dans l'implémentation de la couche [domain]. Je n'ai détecté l'erreur que le 27 juillet. Mieux vaut tard que jamais...

Cet article explique quelle est l'erreur et quelles sont ses conséquences pour les applications utilisant la couche [domain]. Pour comprendre cet "Errata", il faut avoir lu les premiers articles de la série afin de comprendre le rôle de la couche [domain]. En fait, nous conseillons au lecteur de lire d'abord les articles sans tenir compte de cet "errata" et de ne revenir qu'ensuite sur ce dernier. En effet, l'erreur n'apparaît pas lors d'une lecture rapide des articles et ne nuit en rien à leur compréhension. Il est probable que seuls des développeurs expérimentés la détecteront dès la première lecture.

Nous présentons tout d'abord l'erreur dans sa version Java. Elle existe à l'identique dans la version .NET. Nous proposons ensuite des exemples de codes corrigeant l'erreur. Ceux-ci sont pris dans les versions .NET.

1.1 L'erreur dans sa version Java

L'interface [IArticlesDomain] et sa classe d'implémentation [AchatsArticles] s'avèrent avoir une conception incorrecte pour ce qui est de l'achat du panier. La classe [AchatsArticles] est la suivante :

```
package istia.st.articles.domain;

// Imports
import istia.st.articles.dao.IArticlesDao;
import istia.st.articles.exception.UncheckedAccessArticlesException;
import java.util.ArrayList;
import java.util.List;

public class AchatsArticles implements IArticlesDomain {

  // Champs
  private IArticlesDao articlesDao;
  private ArrayList erreurs;

  // Constructeurs
  public AchatsArticles(IArticlesDao articlesDao) { }

  // Méthodes
  public ArrayList getErreurs() {}
  public List getAllArticles() {}
  public Article getArticleById(int id) {}
  public void acheter(Panier panier) { }
}
```

L'achat d'un panier se fait en deux temps :

1. la méthode [acheter] est tout d'abord utilisée pour décrémenter les stocks des articles achetés,
2. la méthode [getErreurs] est ensuite utilisée pour obtenir la liste des éventuelles erreurs. Celles-ci listent les articles dont les stocks sont insuffisants pour satisfaire la demande du client.

Si cette méthode est acceptable (quoique maladroite) lorsqu'il n'y a qu'un client, elle ne l'est plus lorsqu'il y en a plusieurs comme c'est le cas dans une application web où les clients sont multiples. Il faut tout d'abord comprendre qu'il n'y a qu'une seule instance de la classe [AchatsArticles] pour satisfaire les demandes des clients. On a affaire à un singleton. Prenons la situation suivante dans le cadre d'une application web :

1. le client 1 veut acheter le panier [panier1]. Il est servi par un thread T1 côté serveur. Celui-ci commence par faire appel à la méthode [acheter] du singleton [AchatsArticles]. Celle-ci se termine. Les erreurs d'achats ont été stockées dans le champ privé [erreurs] du singleton. Le thread T1 est alors interrompu pour une raison quelconque.
2. le client 2 veut acheter le panier [panier2]. Il est servi par un thread T2 côté serveur. Celui-ci commence par faire appel à la méthode [acheter] du singleton [AchatsArticles]. Celle-ci se termine. Les erreurs d'achats ont été stockées dans le champ privé [erreurs] du singleton écrasant donc les erreurs du client 1. Le thread T2 n'est pas interrompu. Il fait alors appel à la méthode [getErreurs] du singleton et obtient la liste d'erreurs des achats de [panier2]. Le thread T2 envoie sa réponse au client 2 et se termine.
3. Le thread T1 récupère alors le processeur et fait ce qu'il n'a pas pu eu le temps de faire. Il fait appel à la méthode [getErreurs] du singleton [AchatsArticles] et obtient la liste des erreurs d'achats de [panier2] et non celle de [panier1] qui n'existe plus.

On a un problème...

Lorsqu'on a affaire à un singleton servant plusieurs clients, il est important de vérifier que ses données d'état ne concernent pas un client particulier. Dans la classe [AchatsArticles], les données d'état sont les suivantes :

```
  // Champs
  private IArticlesDao articlesDao;
  private ArrayList erreurs;
```

`articlesDao` représente l'instance d'accès à la couche [dao] - est elle-même un singleton. N'est pas lié à un client particulier.

`erreurs` la liste des erreurs qui se sont produites sur l'achat d'un panier. Cette donnée est liée au panier d'un client particulier. Elle ne peut faire partie de l'état du singleton qui est partagé par tous les clients.

La solution à notre problème est ici relativement simple. Il faut changer la signature de la méthode [acheter]. Elle doit devenir :

```
  public ArrayList acheter(Panier panier);
```

Le résultat de type [ArrayList] est la liste des erreurs survenues lors de l'achat du panier. Celle-ci n'a donc plus à faire partie de l'état du singleton implémentant l'interface [IArticlesDomain]. Cette dernière devient la suivante :

```
1. package istia.st.articles.domain;
2.
3. import istia.st.articles.dao.Article;
4.
5. import java.util.ArrayList;
6. import java.util.List;
7.
8. /**
9.  * @author serge.tahe@istia.univ-angers.fr
10. *
11. */
12.public interface IArticlesDomain {
13.  // liste de tous les articles
14.  public List getAllArticles();
15.  // obtenir un article particulier
16.  public Article getArticleById(int id);
17.  // acheter un panier
18.  public ArrayList acheter(Panier panier);
19.}
```

La méthode [getErreurs] a disparu. La méthode [acheter] a changé de signature, ligne 18.

La classe d'implémentation [AchatsArticles] devient la suivante :

```
1. package istia.st.articles.domain;
2.
3. // Imports
4. import istia.st.articles.dao.IArticlesDao;
5. import istia.st.articles.exception.UncheckedAccessArticlesException;
6. import java.util.ArrayList;
7. import java.util.List;
8.
9. public class AchatsArticles implements IArticlesDomain {
10.
11.  // Champs
12.  private IArticlesDao articlesDao;
13.  private ArrayList erreurs;
14.
15.  // Constructeurs
16.  public AchatsArticles(IArticlesDao articlesDao) { }
17.
18.  // Méthodes
19.  public List getAllArticles() {...}
```

```
20.  public Article getArticleById(int id) {...}
21.  public ArrayList acheter(Panier panier) {...}
22.}
```

La méthode [getErreurs] a disparu. La méthode [acheter] a changé de signature et devient la suivante :

```
1.   // valider un panier d'achats
2.   public ArrayList acheter(Panier panier) {
3.     // la liste des erreurs
4.     ArrayList erreurs = new ArrayList();
5.     // on parcourt les achats
6.     ArrayList achats = panier.getAchats();
7.     Article article = null;
8.     Achat achat = null;
9.     for (int i = achats.size() - 1; i >= 0; i--) {
10.      // on récupère l'achat
11.      achat = (Achat) achats.get(i);
12.      // on tente de modifier le stock de l'article dans la base
13.      int nbarticles = articlesDao.changerStockArticle(achat.getArticle()
14.          .getId(), -achat.getQte());
15.      // a-t-on réussi ?
16.      if (nbarticles != 0) {
17.        achats.remove(i);
18.      } else {
19.        erreurs.add("Achat article [" + achat.getArticle() + ","
20.            + achat.getQte() + "] impossible - Vérifiez son stock");
21.      }
22.    }
23.    // on rend la liste des erreurs
24.    return erreurs;
25.  }
```

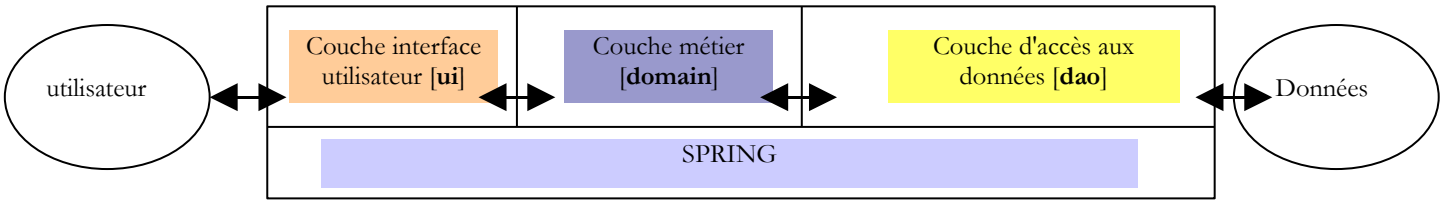
- la liste des erreurs est devenue une variable locale à la méthode - ligne 4
- elle est rendue comme résultat - ligne 24

Ce code est-il suffisant pour garantir l'étanchéité entre deux clients ? Dans cette méthode, seules des variables locales à la méthode sont utilisées sauf la variable [articlesDao] qui est une variable d'instance du singleton partagée par tous les clients. La méthode [articlesDao.changerStockArticle] utilisée ligne 13 est synchronisée (synchronized). Deux threads ne peuvent l'exécuter simultanément. Il semble donc que la méthode [acheter] assure bien l'étanchéité des clients.

Nous utiliserons désormais cette nouvelle interface [IArticlesDomain] et sa nouvelle classe d'implémentation [AchatsArticles]. On peut conclure de cette erreur que les tests de la couche [domain] avaient été insuffisants. On avait négligé de tester l'étanchéité des clients entre-eux.

1.1 Conséquences de l'erreur

Revenons sur l'architecture de l'application d'achats d'articles :



Nous avons modifié l'interface [IArticlesDomain] de la couche [domain]. Très exactement nous avons :

- supprimé la méthode [getErreurs]
- modifié la signature de la méthode [acheter]

Cela entraîne des modifications sur les éléments de la couche [ui] qui utilisent ces deux méthodes. Dans ces éléments, la modification à faire est la suivante :

Dans l'ancienne version, l'achat d'un panier se faisait de la façon suivante (version Java) :

```
1. Panier panier;
2. ArrayList erreurs;
3. ...
4. [domain].acheter(panier);
5. ...
6. erreurs=[domain].getErreurs();
```

où [domain] représente une instance d'une classe implémentant [IArticlesDomain].

Dans la nouvelle version, l'achat d'un panier se fera de la façon suivante (version Java) :

```
7. Panier panier;
8. ArrayList erreurs;
9. ...
10.erreurs=[domain].acheter(panier);
```

Nous présentons des exemples de corrections apportées aux versions .NET des articles.

1.2 Corrections .NET - Exemples

1.2.1 L'interface IArticlesDomain

```
1. Imports istia.st.articles.dao
2.
3. Namespace istia.st.articles.domain
4.  Public Interface IArticlesDomain
5.    ' méthodes
6.    Function acheter(ByVal panier As Panier) As ArrayList
7.    Function getAllArticles() As IList
8.    Function getArticleById(ByVal idArticle As Integer) As Article
9.  End Interface
10.End Namespace
```

- à noter le changement de signature de la méthode [acheter] ligne 6

1.2.2 La classe [AchatsArticles]

```
1. Imports istia.st.articles.dao
2.
3. Namespace istia.st.articles.domain
4.  Public Class AchatsArticles
5.    Implements IArticlesDomain
6.
7.    'champs privés
8.    Private _articlesDao As IArticlesDao
9.
10.   ' constructeur
11.   Public Sub New(ByVal articlesDao As IArticlesDao)
12.     _articlesDao = articlesDao
13.   End Sub
14.
15.   ' méthodes
16.   Public Function getAllArticles() As IList Implements IArticlesDomain.getAllArticles
17.     ' liste de tous les articles
18.     Return _articlesDao.getAllArticles
19.   End Function
20.
21.   Public Function getArticleById(ByVal idArticle As Integer) As Article Implements
    IArticlesDomain.getArticleById
22.     ' un article particulier
23.     Return _articlesDao.getArticleById(idArticle)
24.   End Function
25.
26.   Public Function acheter(ByVal panier As Panier) As ArrayList Implements IArticlesDomain.acheter
27.     ' achat d'un panier - les stocks des articles achetés doivent être décrémentés
28.     Dim erreurs As New ArrayList
29.     Dim achat As achat
30.     Dim achats As ArrayList = panier.achats
31.     For i As Integer = achats.Count - 1 To 0 Step -1
32.       ' décrémenter stock article i
33.       achat = CType(achats(i), achat)
34.       Try
35.         If _articlesDao.changerStockArticle(achat.article.id, -achat.qte) = 0 Then
36.           ' on n'a pas pu faire l'opération
37.           erreurs.Add("L'achat " + achat.ToString + " n'a pu se faire - Vérifiez les stocks")
38.         Else
39.           ' l'opération s'est faite - on enlève l'achat du panier
40.           panier.enlever(achat.article.id)
41.         End If
42.       Catch ex As Exception
43.         erreurs.Add("Erreur d'accès aux données : " + ex.Message)
44.       End Try
45.     Next
46.     ' on rend les erreurs
47.     Return erreurs
48.   End Function
```

```
49. End Class
50.
51.End Namespace
```

- à noter le changement de signature de la méthode [acheter] ligne 26. La liste des erreurs est rendue ligne 47.

1.2.3 Article [http://tahe.developpez.com/dotnet/win3tier]

Il s'agit de l'article intitulé "Construction d'une application windows MVC à trois couches avec Spring, M2VC-win et VB.NET".

L'action [ActionValiderPanier] décrite au paragraphe 6.7.7 page 36 devient la suivante :

```
1. Namespace istia.st.cmv.magasin
2.  Public Class ActionValiderPanier
3.    Inherits AbstractBaseAction
4.
5.    ' validation du panier
6.    Public Overrides Function execute() As String
7.      ' au départ, pas d'erreurs
8.      Session.erreurs = New ArrayList
9.      Try
10.       ' on tente de valider le panier
11.       session.erreurs = session.articlesDomain.acheter(session.panier)
12.     Catch ex As Exception
13.       ' on note l'erreur
14.       session.erreurs.Add(String.Format("Erreur lors de la validation du panier [{0}]", ex.Message))
15.     End Try
16.     ' état application
17.     If session.erreurs.Count <> 0 Then
18.       ' problème
19.       Return "échec"
20.     Else
21.       ' c'est bon
22.       Return "succès"
23.     End If
24.   End Function
25. End Class
26.
27.End Namespace
```

- la modification a lieu ligne 11 lors de l'achat du panier

1.2.4 Article [http://tahe.developpez.com/dotnet/web3tier-part3]

Il s'agit de l'article intitulé "Construction en VB.NET d'une application web MVC multi-couches formée d'un client riche et d'un service web".

La méthode [acheterPanier] de la classe [WebserviceArticles] décrite page 25 devient la suivante :

```
1.    ' achat du panier
2.    <WebMethod()> _
3.    Public Function acheterPanier(ByVal panier As Panier) As WSPanier Implements
   IArticlesWebService.acheterpanier
4.      ' on achète le panier
5.      Dim erreurs As ArrayList = articlesDomain.acheter(panier)
6.      ' on prépare le résultat
7.      Dim wspanier As New wspanier
8.      For i As Integer = 0 To panier.achats.Count - 1
9.        wspanier.ajouter(CType(panier.achats(i), Achat))
10.     Next
11.     wspanier.erreurs = erreurs
12.     ' on rend le résultat
13.     Return wspanier
14.   End Function
```

- la modification a lieu ligne 5 lors de l'achat du panier

La méthode [acheter] de la classe [WebServiceArticlesProxy] décrite paragraphe 6.3 page 43 devient la suivante :

```
1.    ' achat d'un panier
2.    Public Function acheter(ByVal panier1 As Panier) As ArrayList Implements IArticlesDomain.acheter
3.      ' on envoie panier1 au serveur - on reÃ§oit en retour panier2
4.      Dim wsPanier2 As WSPanier = webServiceProxy.acheterPanier(panier1)
5.      ' on met les achats restants dans le panier original panier2 -> panier1
6.      For i As Integer = panier1.achats.Count - 1 To 0 Step -1
7.        panier1.enlever(CType(panier1.achats(i), Achat).article.id)
8.      Next
```

```
9.      For i As Integer = 0 To wsPanier2.achats.Count - 1
10.       panier1.ajouter(CType(wsPanier2.achats(i), Achat))
11.     Next
12.     ' on rend les erreurs
13.     Return wsPanier2.erreurs
14.   End Function
```

- ligne 2 : la signature de la méthode [acheter] a changé
- ligne 13 : la méthode [acheter] rend désormais la liste des erreurs d'achats

1.3 Conclusion

Nous n'avons passé en revue que trois des différents clients .NET ou Java de la couche [domain] utilisés dans les articles de [http://tahe.developpez.com]. Pour les autres clients, les modifications à apporter sont analogues et consistent toutes à transformer une séquence du genre :

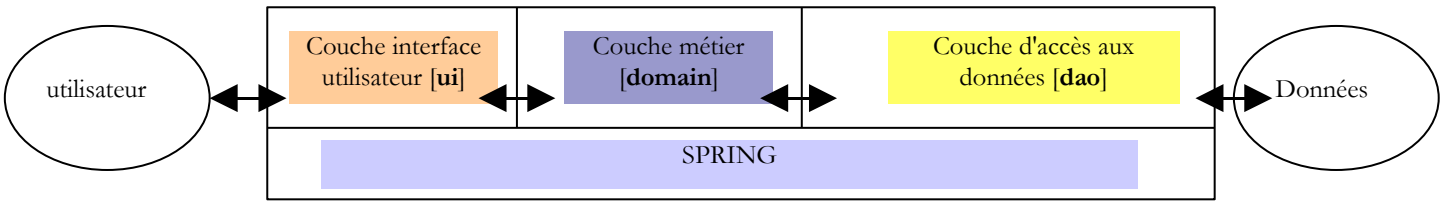
```
11.Panier panier;
12.ArrayList erreurs;
13....
14.[domain].acheter(panier);
15....
16.erreurs=[domain].getErreurs();
```

en une séquence :

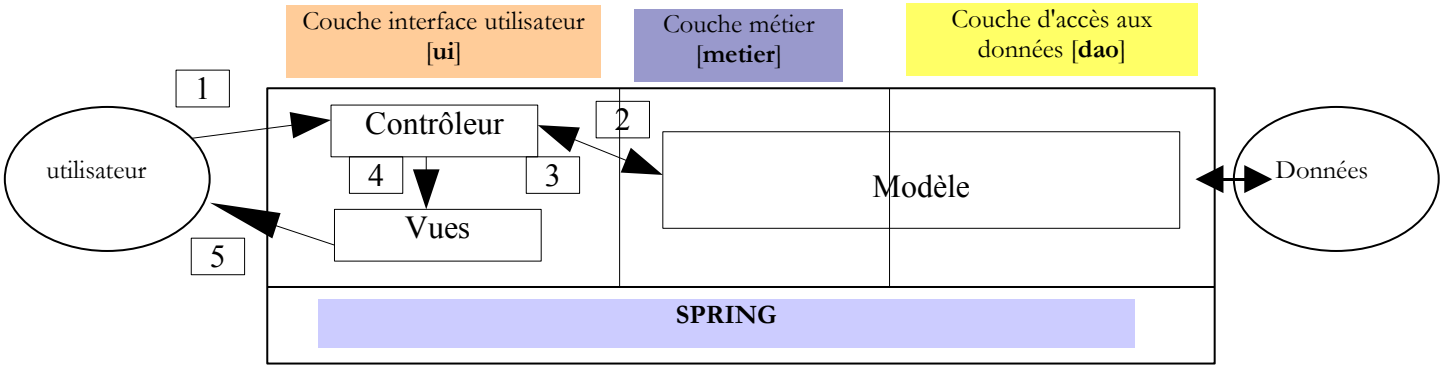
```
17.Panier panier;
18.ArrayList erreurs;
19....
20.erreurs=[domain].acheter(panier);
```

2 29 décembre 2005 - Schéma du modèle MVC

Entre mars et juillet 2005, un certain nombre d'articles sont parus sur [http://tahe.developpez.com]. Ils avaient pour but de présenter le framework Spring aussi bien dans le monde Java que dans le monde .NET. Une application simplifiée d'achats de produits sur le web a servi de fil conducteur à tous les articles. Celle-ci a l'architecture suivante :



Sur cette architecture 3tier était superposée une architecture MVC dont on donnait le schéma suivant :



Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

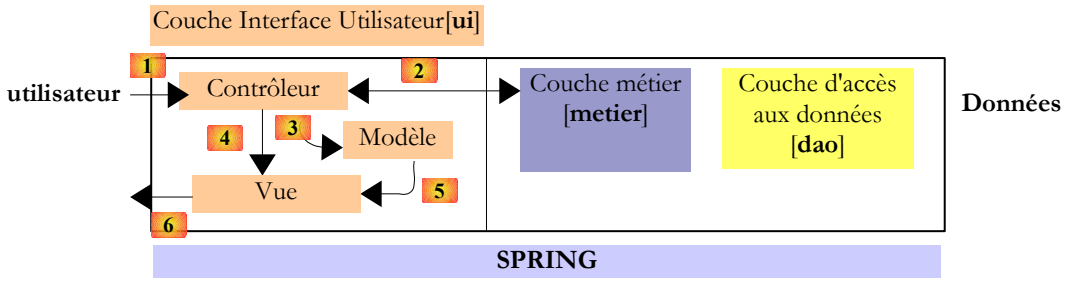
1. le client fait une demande au contrôleur. Ce contrôleur est une classe qui voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le C de MVC.
2. le contrôleur traite cette demande. Pour ce faire, il peut avoir besoin de l'aide de la couche métier, ce qu'on appelle 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.

La position du modèle M dans ce schéma est incorrecte. Mon premier contact avec le modèle MVC s'est fait au travers d'un livre exposant la méthodologie Struts. Au travers de cette lecture, j'avais cru comprendre que le modèle M du MVC était constitué par le modèle des données de l'application, aussi bien le modèle des données du SGBD utilisé par la couche DAO que le modèle objet utilisé par les couches [métier] et [dao]. Cette interprétation est reflétée dans tous les articles écrits en 2004 et 2005 où le modèle est d'abord placé du côté du SGBD (article Struts) puis s'étend à gauche pour recouvrir la totalité des couches [metier] et [dao] (articles Spring / java, Spring / dotnet).

En abordant la techologie web MVC de Spring, j'ai réalisé que Spring ne conçoit pas du tout le modèle de cette façon. Le modèle M est simplement l'ensemble des données que fournit le contrôleur C à une vue V pour s'afficher. Le modèle M d'une vue est alors simplement l'ensemble de ses éléments dynamiques auxquels on doit fournir une valeur avant de demander à la vue de s'afficher. On pourrait tout simplement appeler cela les paramètres de la vue.

Le nouveau schéma MVC superposé à une architecture 3tier devient alors le suivant :



Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

1. le client fait une demande au contrôleur qui 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. 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
3. le contrôleur choisit la réponse (= vue) à envoyer au client. Choisir la réponse à envoyer au client nécessite plusieurs étapes :
   - choisir l'objet qui va générer la réponse. Ce choix dépend en général du résultat de l'exécution de l'action demandée par l'utilisateur.
   - lui fournir les données dont il a besoin pour générer cette réponse. En effet, celle-ci contient le plus souvent des informations calculées par le contrôleur. C'est ce qu'on appelle le modèle de la vue, le **M** de MVC.

   L'étape 3 ci-dessus consiste donc en le choix d'une vue et en la construction du modèle nécessaire à celle-ci.
4. le contrôleur demande à la vue choisie de s'afficher. Il s'agit le plus souvent de faire exécuter une méthode particulière de l'objet chargé de générer la réponse au client. Ce générateur de vue est le **V** de MVC. Nous appelons vue ici, aussi bien l'objet qui génère la réponse au client que cette réponse elle-même. La littérature MVC n'est pas explicite sur ce point.
5. le générateur de vue utilise le modèle préparé par le contrôleur pour initialiser les parties dynamiques de la réponse qu'il doit envoyer au client.
6. la réponse est envoyée au client. La forme exacte de celle-ci dépend du générateur de vue. Ce peut être un flux HTML, PDF, Excel, ...