Code Weblog

... une journée dans la vie d'un développeur



# Développement d'une appli Android basée sur un MVC

posté dans Android, Design Patterns, Java, Tutoriel écrit le 26 février 2011 par Nicolas PARTAGER

**Objectifs du tutoriel** : Développer une application Android basée sur le design pattern MVC (Model View Controller). L'application développée au cours du tutoriel est une ébauche de calculatrice. En fait la calculatrice n'est qu'un exemple qui vous permettra de réutiliser le pattern pour des applications plus intéressantes. Elle est donc volontairement simplifiée...

**Niveau** : Intermédiaire++

**Pré-requis** :

- Avoir un environnement Java sur sa machine.
- Avoir installé le SDK Android (suivez le tutoriel officiel: http://developer.android.com/sdk/installing.html).
- Avoir une version de l'IDE Eclipse qui gère Java (« Eclipse Java » par exemple): http://www.eclipse.org/downloads/
- Avoir ajouté le plugin ADT à Eclipse: http://developer.android.com/sdk/installing.html
- Des connaissances en Java sont indispensables.
- Des connaissances sur les design patterns sont conseillées (en particulier les patterns : Stratégie et Observateur qui sont à la base du pattern MVC)
- Des connaissances sur Android faciliteraient la compréhension du tuto...

**Ressources** : Vous pouvez télécharger le code source de l'intégralité du tutoriel sous la forme d'un projet Eclipse. Vous n'avez qu'à compiler et exécuter l'application!

Code source

**Sommaire** [Masquer]



# Introduction

Dans ce tutoriel, nous allons voir comment utiliser un MVC maison (que nous bâtirons de A à Z) pour structurer une application Android.

J'ai pu lire de nombreuses discutions sur le web concernant les MVC et Android. Pas mal de personnes pensent qu'une « Activity » Android est un amalgame d'une Vue et d'un Contrôleur et que le MVC n'est, de fait, pas utilisable. Je ne pense pas que le MVC soit impossible… L'Activity, qui pour moi, fait purement partie de la Vue, va simplement écouter les évènements et en informer le Contrôleur.

Dans un premier temps, nous allons parcourir un code « métier » qui constituera notre modèle. Ce sera une calculatrice qui effectuera un parsing d'un objet String contenant l'opération à calculer, qui testera la syntaxe et qui évaluera cette expression. Nous passerons assez vite sur ce code. Nous discuterons du design de ce code pour se chauffer avec les design patterns. Il met en œuvre un décorateur : nous allons décorer des nombres avec des opérations ! Quelle poésie… 😉

Dans un second temps, nous allons revoir la théorie du MVC dans le détail.

Enfin nous réaliserons le MVC appliqué à Android. Nous verrons que la classe « Activity » Android constitue partiellement la « Vue » du MVC que nous tacherons d'organiser proprement.

**NB**: La méthode que nous allons utiliser n'est pas la seule possibilité pour développer une application pour Android. Si vous ne devez pas interagir avec le hardware du smartphone (ce qui notre cas pour réaliser une calculatrice : nous n'avons ni besoin de gps, ni de l'appareil photo, etc.), il vaut mieux réaliser une application web. Une appli web se réalise avec des technologies différentes (html, css et Javascript). Le gros avantage est que votre appli pourra tourner sur tous les smartphones ! Le développement est en plus bien moins long qu'en utilisant Java. Donc vous avez tout à gagner à privilégier cette deuxième possibilité dès que vous le pouvez. Sachez qu'il existe des frameworks qui permettent de réaliser une appli native (ce que nous allons réaliser dans le tutoriel) à partir d'une appli web et d'accéder à pas mal de primitives bas niveau. Si vous êtes intéressés, regardez du côté de: http://www.appcelerator.com, http://www.phonegap.com ou encore http://rhomobile.com.

Dans le cadre de ce tutoriel, j'utilise donc sciemment la méthode la plus « lourde » uniquement à des fins pédagogiques.

# Partie métier: la calculatrice

## Spécifications

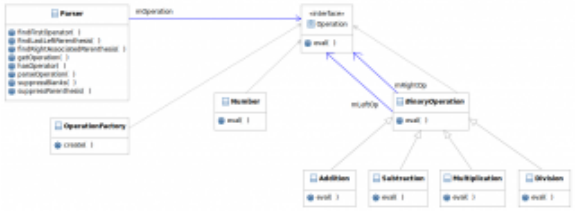
La spécification des fonctionnalités de notre calculatrice (par moi même! c'est plus simple que quand il s'agit d'un client :-P) est de pouvoir parser une chaîne de caractères et d'évaluer cette expression. En cas de problème de syntaxe, le parsing lèvera une exception. L'expression à saisir pourra comprendre des nombres, des opérateurs et des parenthèses. Nous n'implémenterons pas de priorité d'opérateurs, ce sera donc à l'utilisateur d'utiliser des parenthèses. L'opérateur « moins » unaire ne sera pas implémenté. Son ajout est très facile au niveau du modèle, mais plus complexe au niveau du parsing. À vous de le rajouter si vous le souhaitez...

Exemple de saisies :

- 3.542 : ok (rien à évaluer mais valide)
- ((3.542)) : ok (rien à évaluer et parenthèses inutiles, mais valide)
- 1.5+-5 : ko, les opérateurs unaires ne sont pas gérés
- 1.5*(-5) : ko, les opérateurs unaires ne sont pas gérés
- 1.5*(0-5) : ok, c'est l'astuce à utiliser pour obtenir : 1.5*(-5) (l'opérateur unaire « moins » n'étant pas géré
- 1.5-5 : ok
- 5.12+(1.032*2.12) : ok
- 5.12+1.032*2.12 : ok mais le résultat pourra être erroné puisque les priorités ne sont pas gérées
- 5.12+((1.032*2.12) : ko, exception à lever (erreur de parenthèses)
- 5.12z+((1.032*2.12) : ko, exception à lever (l'expression comporte une lettre)
- ( (((500)) *2 )/(1 -8.7) ) + 1.3 : ok (les espaces inutiles sont acceptés, les parenthèses inutiles aussi)

## Conception

Nous allons concevoir ce modèle de calculatrice à l'aide du design pattern Décorateur. Voici le schéma UML de notre modèle :



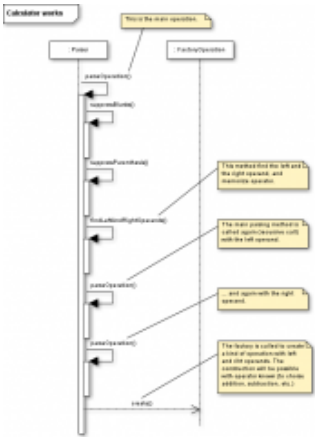
Quelques explications :

- Nous allons travailler en manipulant une abstraction d'opérations (à l'aide d'une interface « Operation »). Nous pourrons ainsi demander l'évaluation d'une opération, qu'elle soit un nombre, une addition, une multiplication, etc.
- « BinaryOperation » est un décorateur abstrait. Il décore une opération binaire (addition, multiplication, etc) avec 2 autres opérations (opérande de gauche et opérande de droite).
- « Addition », « Subtraction », etc. sont des décorateurs concrets.
- Parser est la classe principale qui utilisera le reste de l'ensemble du modèle.
- « OperationFactory » est une classe basée sur le design pattern « Factory » (ou «Usine» en français). Elle a pour vocation de nous allouer de nouvelles opérations (quelque soit la nature réelle de l'opération : nombre, addition, soustraction, etc).
- Vous allez me dire, c'est bien joli les design patterns partout, mais ça complique tout ! Vous aurez (partiellement) raison... Pour ma défense : on fait un tuto, donc un code à vocation pédagogique. Ensuite ce design est extrêmement souple et on applique la bonne pratique suivante : les classes doivent être conçues pour pouvoir facilement étendre leur fonctionnement sans pour autant les modifier. En effet, si on souhaite ajouter un nouvel opérateur (un modulo % par exemple), il suffira de créer une nouvelle classe « Modulo » qui hérite de « BinaryOperation ». On ira ensuite dans la classe « OperationFactory » renseigner comment construire une telle opération. Enfin on ajoutera un opérateur supplémentaire dans la liste contenue dans l'interface « Operation ». En gros, on a rien touché à la classe « Parser » ! Tout notre code déjà existant, bien éprouvé et validé n'a subi aucune modification! On peut donc rester confiant ! Pas mal non ? 😉

Le mécanisme offert par le décorateur est intéressant. Nous allons pouvoir wrapper des nombres dans des opérations. Exemples :

```
// Pour calculer : 2.31*0.4
Operation op = new Multiplication(new Number(2.31f), new Number(0.4f));
// Pour calculer : 2.31*(0.4+0.2)
Operation op = new Multiplication(new Addition(new Number(2.31f), new Number(0
```

Le fonctionnement du parser est quelque peu décrit par le schéma de séquence suivant:



### Code source du modèle de calculatrice

L'objectif premier du tuto n'étant pas de faire une calculatrice industrielle, nous ne rentrerons pas plus dans le détail... Promis on développera le MVC vraiment dans le détail!

Le code est inclus dans l'archive contenant tout le code source du tutoriel (package com.weblog.calculatrice) que vous pouvez télécharger ICI.

## Aspect théorique du MVC

### Définition

> Le Modèle-Vue-Contrôleur (en abrégé MVC, de l'anglais Model-View-Controller) est une architecture et une méthode de conception qui organise l'interface homme-machine (IHM) d'une application logicielle. Ce paradigme divise l'IHM en un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (logique de contrôle, gestion des événements, synchronisation), chacun ayant un rôle précis dans l'interface. Cette méthode a été mise au point en 1979 par Trygve Reenskaug, qui travaillait alors sur Smalltalk dans les laboratoires de recherche Xerox PARC.
>
> **source: Wikipedia**

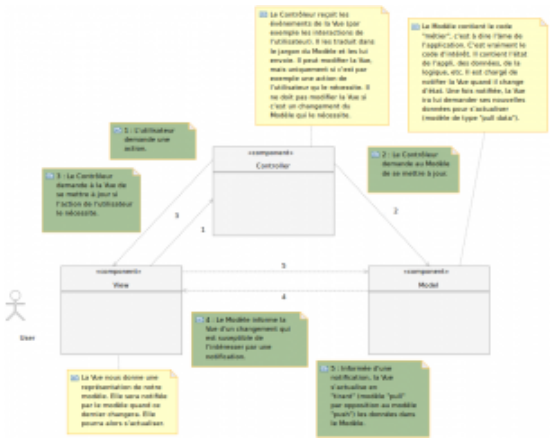
Concrètement, un MVC permet de séparer les responsabilités dans une application. Les avantages sont nombreux.

Tout d'abord, ce pattern étant extrêmement connu, il permet de formaliser une application. Il sera plus facile pour une tierce personne de comprendre la structure de votre appli si vous avez utilisé ces principes, ce pattern étant largement documenté.

Ensuite, il permet de créer des limites très claires et bien définies entre les différents morceaux de code. La réusabilité est alors beaucoup plus facile. Avec un MVC, vous pouvez décider demain de changer un algorithme au cœur de votre modèle sans pour autant modifier une seule ligne de code au sein de la Vue ou du Contrôleur. Vous pourrez de plus décider de changer la façon de représenter vos données (responsabilité de la Vue) sans pour autant affecter votre Modèle.

## Exemple de fonctionnement

Le schéma suivant présente un exemple de fonctionnement:




## Structure détaillée du MVC

La puissance du MVC repose sur deux design patterns : la Stratégie et l'Observateur (« Strategy » et « Observer » en anglais).

Présentons brièvement ces 2 patterns qui permettront de mieux comprendre le MVC.

> **Design pattern «Strategy»**

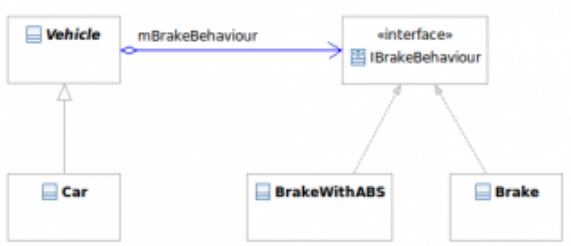
Ce pattern est un des plus simples Son principe est de manipuler une abstraction plutôt qu'une implémentation. Il permet de rendre des algorithmes interchangeables. Les algorithmes peuvent être remplacés indépendamment des clients qui les utilisent.

Les bonnes pratiques de conception orientée objet que l'on trouve dans ce design sont:

- l'encapsulation de ce qui peut changer dans le temps,
- l'utilisation de la composition préférée à l'héritage,
- l'utilisation d'interfaces plutôt que des classes concrètes.

Imaginons que nous souhaitions manipuler des matrices de nombres. Si une matrice contient des nombres en virgule fixe simple précision par exemple, et que demain nous souhaitions utiliser des entiers, le travail sera long et nous devrons revalider pas mal de code. En revanche, si la matrice contient des objets « Nombre » (abstraction d'un vrai nombre), alors peut importe le type de la donnée réellement manipulée ! C'est le principe appelé « polymorphisme ».

Voici le schéma UML d'un exemple du design pattern « Strategy » :




Dans cet exemple (source Wikipedia), nous voyons qu'une sorte de véhicule (une voiture par exemple) peut changer de comportement de freinage, ceci au run-time (c'est à dire pendant l'exécution du programme) ! De plus, les différents comportements de freinage pourront être utilisés avec n'importe quel véhicule.

**NB**: Dans l'exemple, « Vehicle » est une classe abstraite, « IBrakeBehaviour » une interface et des classes instanciables pour le reste.

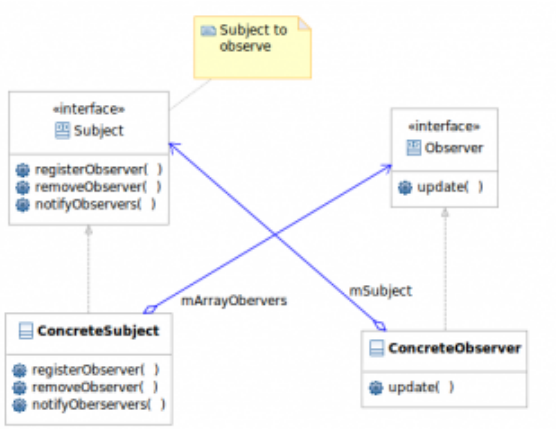
> Design pattern « Observer »

Ce design pattern est assez connu. Il permet à des classes d'observer une classe particulière, ceci tout en conservant un couplage faible entre observateurs et observé. Le principe est le suivant:

- Quand l'état de la classe observée (aussi nommée le « sujet ») change, cette classe va notifier les observateurs de son changement.
- Un fois notifiés, les observateurs s'actualisent en « tirant » les nouvelles données dans le sujet observé.
- On dit que le couplage est faible car:
  - un sujet observé ne connait qu'une liste d'observateurs (via une interface, donc il ne connaît rien de l'intérieur d'une classe observatrice),
  - un observateur ne connaît qu'un certain nombre de données très limité de la classe observée, et par ailleurs, il n'a pas le droit d'aller examiner ces données quand il le souhaite (il lui faut une notification de changement d'état pour cela).

On retrouve souvent ce pattern. Le JDK nous fourni même les 2 interfaces (« Observable » et « Observer ») dans java.util. À noter que ce mécanisme offert n'est pas très apprécié car « Observable » n'est pas vraiment une vraie interface. C'est en réalité une classe abstraite. En Java, n'ayant pas le mécanisme d'héritage multiple, votre sujet concret à observer ne pourra donc pas hériter d'une autre classe, ce qui est fort dommage...

Voici le schéma UML générique de ce design pattern :




> Le Modèle-Vue-Contrôleur

Les fondements du MVC repose donc sur les 2 design patterns que nous venons de passer au crible. Donc facile !

La partie Modèle du MVC est notre sujet observé. La Vue observe ce sujet. Le modèle notifiera la Vue dès lors qu'il aura un changement d'état (que nous jugerons significatif pour en informer la Vue). La Vue pourra alors s'actualiser en allant « tirer » les données ayant changées dans le Modèle.

Le design pattern Stratégie est plus difficile à percevoir au sein de toute l'architecture du MVC. C'est juste que nous allons travailler le plus souvent possible avec des interfaces plutôt qu'avec des classes. Nous pourrons ainsi changer facilement un de nos 3 morceaux du MVC.

Bon je pense que nous sommes OK au niveau théorique pour se jeter dans la pratique ;-) !

# Réalisation d'une application Android

Nous allons apprendre à réaliser une application Android mono thread. Notre appli ne sera donc basée que sur une seule fenêtre graphique.

Notre fenêtre graphique comprendra:

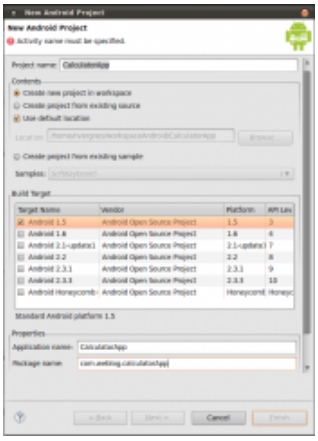
- une zone de saisie d'une opération,
- un bouton « Calculate » pour ordonner le calcul et l'affichage du résultat dans la zone de saisie énoncée ci-dessus,
- une zone colorée indiquant si la syntaxe de l'expression saisie est correcte (verte) ou incorrecte (rouge), ceci après chaque changement de l'expression saisie,
- un texte indiquant le nombre de décimales souhaité (3 par défaut) pour afficher le résultat,
- un bouton pour augmenter ce nombre de décimales,
- un bouton pour diminuer ce nombre de décimales.

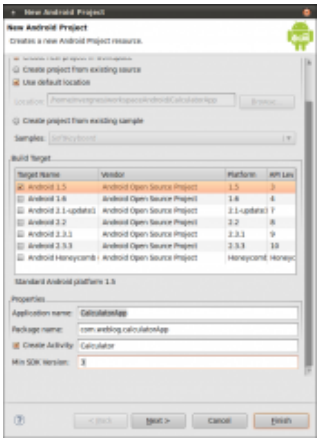
**RQ**: Sous Android il existe des widgets très jolis mais nous allons aller au plus simple au point de vue graphique. Notre appli ne sera pas des plus belle mais notre but n'est pas encore de la vendre ;-).

## Création d'un projet Android sous Eclipse

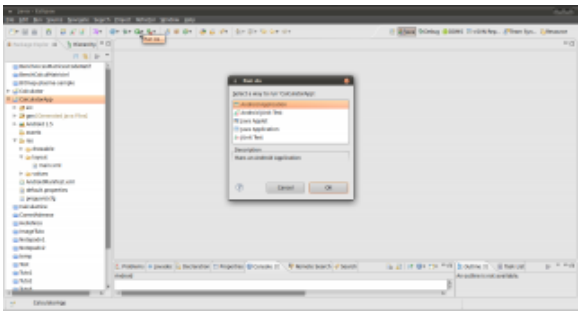
Commençons par créer un projet Android sous Eclipse : File / New / Android Project

Il faut choisir un certains nombres de noms : nom du projet Eclipse nom de votre package Java de base, nom de l'activité principale (titre notre unique fenêtre graphique). Ensuite il nous faut choisir notre cible. Nous prendrons Android 1.5 car nous n'aurons pas besoin d'API très spécifiques développées récemment... Le niveau correspondant (API Level) est le niveau 3.



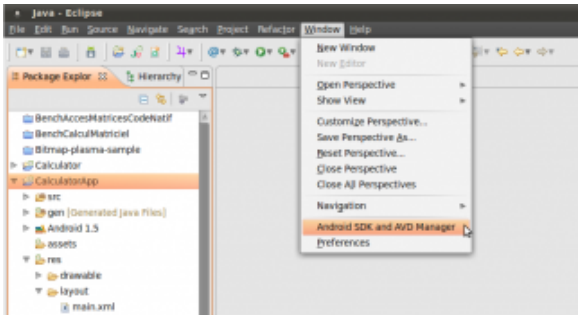


Nous pouvons dès à présent faire tourner notre application sous l'émulateur AVD (Android Virtual Device) :

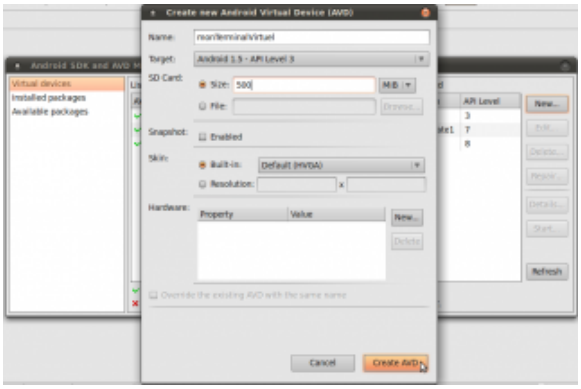


Si vous n'avez jamais crée de terminal Android virtuel, Eclipse va vous demander de le créer avant de pouvoir lancer AVD, installer l'appli et la lancer.

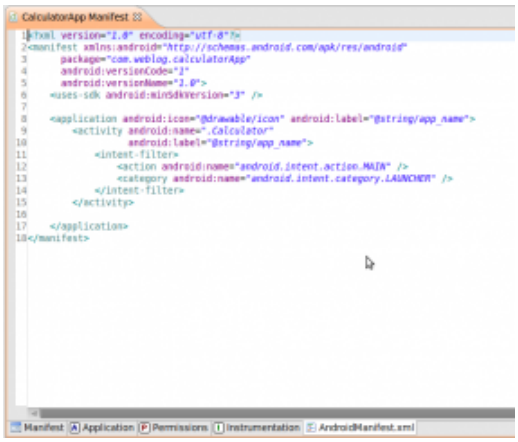
Pour gérer ses machines virtuelles, on peut aller dans le menu AVD-Manager:




Ensuite en cliquant sur New, on crée une nouvelle machine virtuelle:

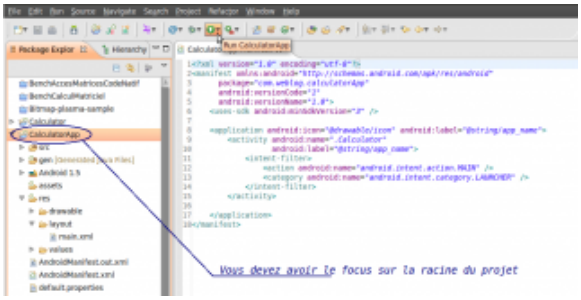



Pour les besoins du tuto, pas de stress, les paramètres importent peu… Choisissez juste « Android 1.5 – API Level 3 » histoire de rester cohérent avec notre définition du projet. Au passage, notez que votre projet contient un Manifest qui contient un ensemble de données importante sur votre projet (dont la cible minimale que nous avions choisie lors de la création du projet):

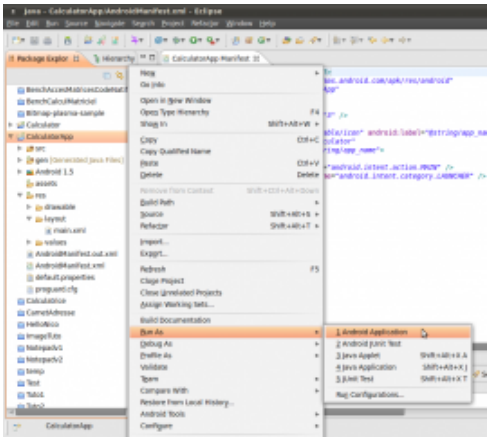



Vous devriez maintenant pouvoir lancer l'émulateur en faisant Run.

**Note**: si vous avez un problème de lancement de l'AVD, vérifiez bien que le focus soit bien sur la racine du projet (et pas sur un fichier quelconque du projet) ou moment ou vous cliquez sur Run:




Si ça ne fonctionne toujours pas, tenter le Run par une autre méthode : Clic droit sur le projet, puis « Run As », puis « Android Application »



YES ! Voilà notre appli 😀

Bon, passons à présent aux choses sérieuses!

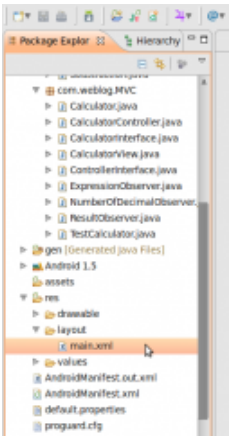
## Layout

La première étape à réaliser pour le développement d'une appli Android est de définir un layout (mise en page en français). Nous en aurons qu'un seul à réaliser, notre appli étant mono thread... Pour cela on peut procéder de 2 façons différentes.

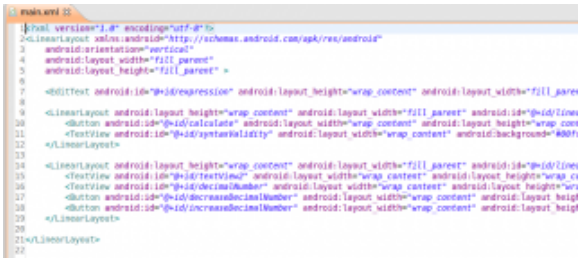
La première méthode consiste à créer le layout au run-time en utilisant le Java pour créer des boutons, des zone de saisie de texte, etc. Cette méthode est la plus performante mais la lisibilité est assez mauvaise. L'activité Android (c'est notre fenêtre graphique porté par notre unique thread) comprendra alors tout ce code de mise en forme graphique.

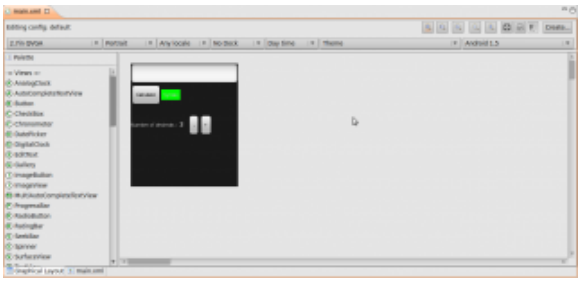
L'autre méthode (qui est préférable dans une majorité de cas), consiste à définir un layout en xml. Pour les personnes qui font du web, cette méthode ressemble pas mal à la création d'un layout html. Nous allons utiliser cette méthode. Je vous disais qu'elle était moins performante, c'est exact! C'est l'inflation (parsing du xml et récupération des données dans un objet Java) du layout xml qui est couteuse... Cela dit, une bonne lisibilité permet de minimiser les problèmes... Donc à vous de voir suivant la criticité de votre application !

Ce layout se situe dans le répertoire des ressources de votre projet : ./res/layout/main.xml




Sous Eclipse, et grâce au plugin ADT, vous pouvez éditer ce layout de manière graphique. C'est plutôt pratique même s'il faut bien reconnaître que c'est très limité. Le plus simple est de jongler entre le rendu graphique et le calque xml brut (non interprété). Vous pouvez également utiliser des outils comme DroidDraw (http://www.droiddraw.org/) par exemple...





Sous Android, il existe plusieurs types de Layout:

- LinearLayout
  - C'est un positionnement qui respecte le flux naturel (comme en html, le positionnement par défaut « static »)
- RelativeLayout
  - C'est un positionnement relatif dont les éléments enfants sont placés relativement aux autres éléments enfants et/ou par rapport au conteneur. Il y a une certaine

ressemblance à un
html positionné en relatif et dont les éléments enfants sont positionnés en absolu.

- TableLayout
  - Pour disposer des widget dans des lignes et colonnes (comme on faisait jadis en html).
- ScrollView
- etc

Pour notre besoin basique, nous allons utiliser des LinearLayout.

Au niveau des propriétés des bloc xml, nous pouvons choisir soit d'adapter les tailles de nos widgets par rapport au contenu (wrap_content), soit par rapport au conteneur parent (fill_parent). Par exemple, le champ de saisie de l'opération à calculer sera un bloc de type EditText (vous l'aurez compris, pour pouvoir éditer un texte !), et nous souhaitons remplir la fenêtre en largeur (donc android:layout_height= »fill_parent » si vous suivez toujours…) et adapter la hauteur en fonction du contenu (donc android:layout_height= »wrap_content »).

Ensuite nous allons utiliser deux sous-layout afin de pouvoir obtenir des éléments centrés verticalement dans leur conteneur.
Voici la structure xml contenant les widgets que nous allons utiliser:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >

    <EditText android:id="@+id/expression" android:layout_height="wrap_content"
android:layout_width="fill_parent" android:paddingTop="15px" android:gravity="right"/>

    <LinearLayout android:layout_height="wrap_content" android:layout_width="fill_parent"
android:id="@+id/linearLayoutExpression" android:layout_marginBottom="20px">
        <Button android:id="@+id/calculate" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="Calculate"/>
       <TextView android:id="@+id/syntaxValidity" android:layout_width="wrap_content"
android:background="#00ff00" android:text="Syntax" android:layout_height="wrap_content"
android:padding="5px" />
   </LinearLayout>

   <LinearLayout android:layout_height="wrap_content" android:layout_width="fill_parent"
android:id="@+id/linearLayoutDecimalControl" >
        <TextView android:id="@+id/textView2" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="Number of decimals : "/>
     <TextView android:id="@+id/decimalNumber" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="3" android:textStyle="bold" android:textSize="14px"/>
        <Button android:id="@+id/decreaseDecimalNumber" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="-" android:layout_marginLeft="10px" />
       <Button android:id="@+id/increaseDecimalNumber" android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="+" />
  </LinearLayout>

</LinearLayout>
```

Toutes les propriétés définies ci-dessus dans notre xml pourront bien sur être modifiées au run-time.

À ce stade, vous devriez pouvoir exécuter l'application et obtenir le rendu suivant:

Pas mal mais c'est encore une appli factice... Aucun comportement n'est rattaché aux widgets. Bon on va attaquer le code alors !

## Classes d'interfaces

Pour appliquer la bonne pratique de manipuler des abstractions plutôt que des classes concrètes (design pattern Strategy), commençons par définir un certain de classes d'interfaces:

- ControllerInterface
  - Cette interface va définir un certain nombre de méthodes qui vont être sollicitées par la Vue lors d'une interaction de l'utilisateur.
  - Ces méthodes vont interpréter cette action et demander au Modèle d'effectuer des opérations ou des changements en conséquence. Si le Modèle se modifie lors de sa sollicitation, il notifiera la Vue.
  - Lors d'une action de l'utilisateur, la Vue peut directement demander un changement dans la Vue si c'est sa responsabilité. Par exemple si notre calculatrice avait un bouton pour changer la couleur du fond de l'appli, la Vue remonterait une requête du type « l'utilisateur a actionné le bouton couleur ». Le Contrôleur interprèterait cette action en demandant directement à la Vue (donc sans passer par le Modèle qui ne doit absolument pas connaître ceci) de changer sa couleur de fond.
  - Liste des méthodes de l'interface:
    - checkSyntax( String ) : void
      - Cette méthode demande au Modèle se faire une vérification syntaxique. Le Contrôleur délègue simplement cette opération car ce n'est pas son rôle.
    - calculate : void
      - Idem que pour checkSyntax, ce n'est pas le rôle du Contrôleur, il délègue donc cette opération au Modèle.
    - increaseNumberOfDecimal() : void
      - Cette méthode est appelée quand l'utilisateur appui sur le bouton « + » de la Vue. Le Contrôleur demande donc au Modèle quel est son « nombre de

décimales » courant, incrémente celui-ci, et demande au Modèle de prendre en compte cette nouvelle valeur.
- Le Modèle subi donc un changement d'état, il notifiera donc la Vue et cette dernière actualisera son affichage du nombre de décimales en allant « tirer » la donnée dans le Modèle.
- **RQ**: Depuis pas mal de temps je vous parle de « tirer » ou de modèle « pull » (ce sont les mêmes choses, en français ou en anglais). Dans le design pattern « Oberver », il y a deux implémentations possibles. Dans la première (modèle « push »), le Modèle notifie la Vue et lui « pousse » les nouvelles données par la même occasion. Dans le deuxième (celle que nous allons implémenter), c'est l'observateur qui ira « tirer » la donnée après avoir été notifié. Le modèle « pull » (la deuxième méthode est admise comme la meilleure pratique).

- decreaseNumberOfDecimal() : void
  - Similaire à la méthode increaseNumberOfDecimal, en décrémentant la valeur du nombre de décimale (avec un min à zéro).

- Code:

```
package com.weblog.MVC;

/**
 * This interface allows us to work with a Strategy Design Pattern. So we can
 * easily change our controller.
 *
 * This is in the Controller part of the MVC Pattern.
 *
 * @author nvergnes
 *
 */
public interface ControllerInterface {
   /**
  * This method will ask to the model part of the MVC to check the expression's syntax.
  * @param newOperation to be set in Calculator, and to be checked
    */
 void checkSyntax( String newOperation );

    /**
  * This method will ask to the model part of the MVC to evaluate all the expression.
     */
 void calculate();

   /**
  * This method increase the number of decimals to show in the gui (View
  * part of the MVC)
  */
 void increaseNumberOfDecimals();

    /**
  * This method decrease the number of decimals to show in the gui (View
  * part of the MVC)
  */
 void decreaseNumberOfDecimals();

    /**
```

```
     * Getter
       * @return the view part of the MVC
     */
    public CalculatorView getView();

       /**
     * Adapt the result to have the right number of decimals
        * @param result is the String which contains result
        * @return result with the right number of decimals
     */
    public String setRightNumberOfDecimal( String result );
}
```

- ModelInterface
  - Cette interface comporte un certain nombre de méthodes qui sont sollicitées par le Modèle, ce sont les méthodes du code métier, c'est à dire le code d'intérêt (le reste étant du code pour implémenter des mécanismes objets appuyés sur des design patterns).
    - checkSyntax() : void : pour vérifier la syntaxe de l'opération saisie par l'utilisateur
    - calculate() : void : pour demander le calcul de l'opération
  - Cette interface comporte d'autres méthodes qui sont requises par le pattern Observer:
    - registerObserver(...) : void : Cette méthode sera surchargé pour chaque type d'observateur. En construisant la Vue (qui implémente 3 observateurs : ExpressionObserver, NumberOfDecimalObserver, ResultObserver), nous demanderons au Modèle de les enregistrer dans sa liste des observateurs afin de bien les notifier lors de ses changements.
    - removeObserver(...) : void : Cette méthode sera également surchargée pour les 3 types d'observateurs. Elle permettrait de dé-enregistrer des observateurs. Nous ne l'utiliserons pas dans notre appli, mais elle est susceptible d'être un jour utilisée. Par exemple, nous pourrions désinscrire une Vue pour en inscrire une nouvelle pour changer la façon de présenter les données.
  - Code:

```
package com.weblog.MVC;

/**
 * This interface allows us to work with a Strategy Design Pattern. So we can
 * easily change our calculator.
 *
 * This is in the Model part of the MVC Pattern.
 *
 * @author nvergnes
 *
 */
public interface ModelInterface {
 /**
  * This method will check the expression's syntax, then notify observers with the result of the test.
   */
 void checkSyntax();

 /**
```

```
     * This method will evaluate all the expression, then notify observers with the result.
     */
    void calculate();

      /**
     * This method, based on Observer Design Pattern, allows "expression" observers to be registered.
     * @param observer to register
      */
    void registerObserver( ExpressionObserver observer );

      /**
     * This method, based on Observer Design Pattern, allows "expression" observers to be unregistered.
       * @param observer to remove
        */
    void removeObserver( ExpressionObserver observer );

    /**
     * This method, based on Observer Design Pattern, allows "number of decimals" observers to be registered.
     * @param observer to register
      */
    void registerObserver( NumberOfDecimalObserver observer );

     /**
     * This method, based on Observer Design Pattern, allows "number of decimals" observers to be unregistered.
       * @param observer to remove
        */
    void removeObserver( NumberOfDecimalObserver observer );

       /**
     * This method, based on Observer Design Pattern, allows "result" observers to be registered.
     * @param observer to register
      */
    void registerObserver( ResultObserver observer );

      /**
     * This method, based on Observer Design Pattern, allows "result" observers to be unregistered.
       * @param observer to remove
        */
    void removeObserver( ResultObserver observer );

    /**
     * Getter
       * @return the mNumberOfDecimals
        */
    public int getNumberOfDecimals();

      /**
     * Setter
       * @param numberOfDecimals New value to be set
      */
    public void setNumberOfDecimals( int numberOfDecimals );

       /**
     * Setter
       * @param newOperation to be set
        */
    public void setOperation( String newOperation );

   }
```

- ExpressionObserver
  - Cette interface sera implémentée par la Vue. La Vue sera pourra alors être inscrite dans la liste des observateurs du Modèle qui souhaitent toujours être informés des changement de l'expression à calculer.
  - Code:

```
package com.weblog.MVC;

/**
 * @author nvergnes
 *
 * This interface is based on Observer Design Pattern
 *
 */
public interface ExpressionObserver {
   /**
  * This method will be called when user changes the current expression (whenever
     * user add or suppress a new character in the current expression).
  */
 void updateExpressionValidity( boolean syntaxValidity );
}
```

- NumberOfDecimalObserver
  - Cette interface sera implémentée par la Vue. La Vue sera pourra alors être inscrite dans la liste des observateurs du Modèle qui souhaitent toujours être informés des changement du nombre de décimale utilisé pour afficher le résultat.
  - Code:

```
package com.weblog.MVC;

/**
 * @author nvergnes
 *
 * This interface is based on Observer Design Pattern
 *
 */
public interface NumberOfDecimalObserver {
  /**
  * This method will be called when user asks a new number of decimal.
    */
 void updateNumberOfDecimal();
}
```

- ResultObserver
  - Cette interface sera implémentée par la Vue. La Vue sera pourra alors être inscrite dans la liste des observateurs du Modèle qui souhaitent toujours être informés des changement de du résultat contenu dans le modèle.
  - Code:

```
package com.weblog.MVC;

/**
 * @author nvergnes
 *
 * This interface is based on Observer Design Pattern
```

```
 *
 */
public interface ResultObserver {
   /**
  * This method will be called when user ask to perform the current expression
    * evaluation.
   * @param result of the expression evaluation
    */
 void updateExpression( float result );
}
```

## Structuration de l'application

### Schéma de classes

Grâce à notre analyse de toutes les interfaces utilisées, nous pouvons à présent aboutir au schéma de classes suivant:


On voit bien les 3 morceaux du MVC de gauche à droite : Vue / Contrôleur / Modèle.

À l'extrême gauche, nous avons une classe « Calculator » qui est l'activité (activité au sens Android du terme) principale. Cette classe étend la classe Activity, cette dernière n'étant pas dans notre appli (mais dans le SDK Android). Calculator est donc une classe un peu spéciale mais elle sera incluse dans notre partie « Vue » du MVC.

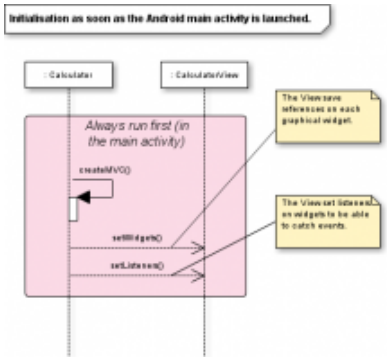
Vous pouvez constater d'une manière générale que les classes concrètes agrègent (traits bleu sur le schéma UML) le plus souvent possible des interfaces plutôt que d'autres classes concrètes. Souvenez vous : c'est le design Strategy à l'oeuvre ! Du jour au lendemain, nous pourrons changer notre Modèle sans modifier une ligne de code dans la Vue et dans le Contrôleur. La seule contrainte sera que notre nouveau modèle devra implémenter ModelInterface. Bon c'est normal, la conception orientée objet, c'est puissant mais pas magique 🙂

### Comportement de l'application

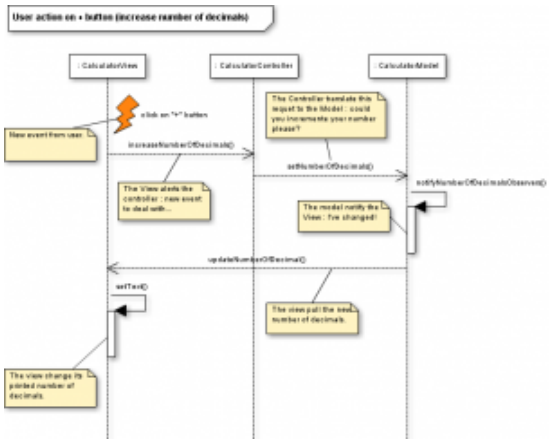
J'espère que le schéma de classes ne vous a pas donné mal à la tête. Normalement, avec les explications complémentaires vous avez du suivre à peu près.

Rentrons un peu plus en détail dans cette appli. Je vais vous présenter les schéma de séquence qui vont illustrer des scénarios d'utilisation de l'application. Pour ceux qui ne connaissent pas trop UML, les schémas de séquences permettent de décrire le comportement dynamique d'un système. Le strict minimum pour documenter un projet est le schéma de classes et de séquence.

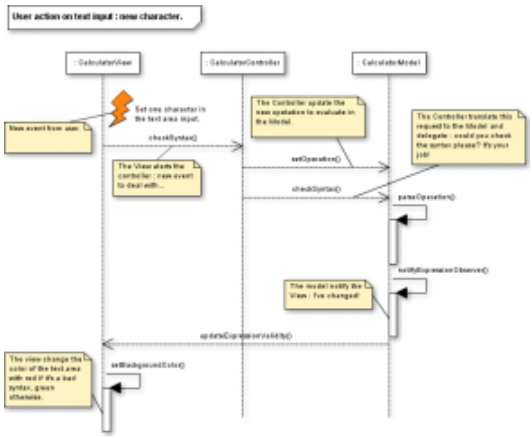
**Scénario 1: Toujours exécuté au lancement de l'application**



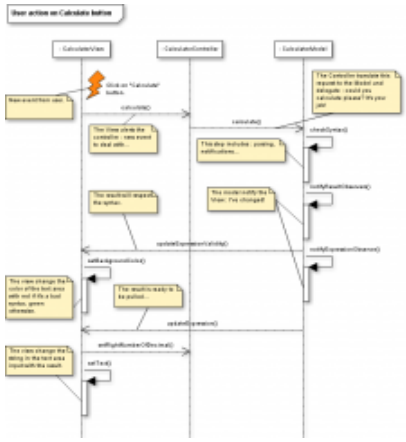

### Scénario 2: L'utilisateur clique sur le bouton «+» pour incrémenter le nombre de décimales à afficher sur le résultat.




### Scénario 3: L'utilisateur saisi un caractère dans la zone de saisie de l'expression à calculer.




### Scénario 4: L'utilisateur clique sur le bouton «Calculate».



Bon, nous pourrions rentrer encore plus en détail ou de manière plus exhaustive dans ces schémas de séquence mais nous allons en rester là…

J'espère qu'à ce stade, vous avez du pu le fonctionnement du MVC dans cette application. Il n'est sans doute pas parfait, mais les grands principes y sont.

Vous pouvez télécharger l'ensemble du projet Eclipse contenant tout le code source:

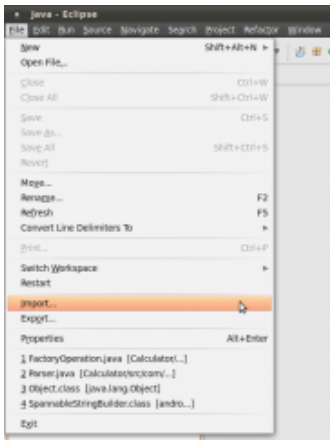
Code source

> **Comment exécuter le projet Eclipse téléchargé**

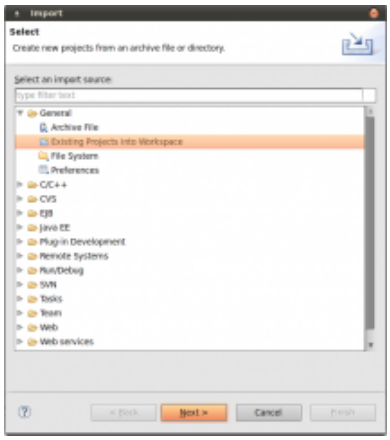
Pour les débutants avec Eclipse, vous pouvez directement faire tourner le projet que je vous propose de télécharger. Tout sera déjà paramétré et vous n'aurez plus qu'à configurer votre Eclipse avec le SDK Android et ADT.

Téléchargez le projet. Après extraction de l'archive zip, placez le dans le répertoire de travail Eclipse. Si vous ne l'avez pas modifié, ce sera un répertoire nommé « Workspace ».

Ensuite, vous ne devez pas créer un nouveau fichier mais vous devez importer ce projet. Pour cela : File / Import




Ensuite : Existing Projects into Workspace




Il ne vous reste plus qu'à indiquer le projet et c'est OK

# Conclusion

J'espère que ce tutoriel vous aura intéressé et vous aura permis de mieux comprendre le design pattern MVC ainsi que les bonnes pratiques de la conception orientée objet.

N'hésitez pas à me poser des questions ou à me signaler des erreurs si vous en voyez. Il peut y en avoir quelques unes...



Android, Application, Design pattern, développement, Eclipse, Java, MVC, UML

## À PROPOS DE L'AUTEUR

NICOLAS

Je suis ingénieur développement logiciel avec orientation scientifique (traitement du signal, estimation Bayésienne, optimisation par exemple). J'ai acquis une solide expérience dans un grand groupe naval, dans le domaine de l'autoguidage de véhicules autonomes. Aujourd'hui, je développe et je m'occupe de la qualité logicielle chez RDTronic dans le domaine de la cartographie, du traitement et de la sécurisation de la donnée. Je suis par ailleurs passionné d'escalade et de canyoning.

BILLET PRÉCÉDENT

← Tutoriel Aptana Studio 3

BILLET SUIVANT

Appli Android pour enquête de terrain →

# 24 Commentaires

→

## Développement d'une appli Android basée sur un MVC

**JBBI**
*13 avril 2011 à 16 h 20 min*

Bonjour,

Très bon article. N'est-il pas possible d'utiliser l'Activity comme vue ? je trouve votre solution bonne mais très lourde surtout pour de l'embarqué. De plus beaucoup d'éléments de vue demandent le context de l'Activity pour être crée à la volée tel les boite de dialogue par exemple.

Qu'en pensez-vous ?

RÉPONDRE ↓

**NICOLAS**
*13 avril 2011 à 17 h 25 min*

Oui, vous avez tout à fait raison. J'ai souhaité décorréler le MVC des mécanismes propres à Android juste à des fins pédagogique. Ça permet aux débutants sur Android de ne pas prendre peur avec notre instance de calculatrice héritée de Activity. Ça change un peu du traditionnel main().

En pratique, il est clair que mon choix n'est pas performant. D'une manière générale, les design patterns peuvent nuire à la performance. Leur but est vraiment d'abstraire l'architecture pour faciliter l'évolutivité et la réusabilité, ceci parfois au prix d'un code plus dense et moins performant :-(.

Dans mon exemple, on se demande bien pourquoi aussi lourd pour si peux, je vous l'accorde lol. Dans l'optique d'un vrai cycle de vie d'une appli, je touve que le MVC permet de bien clarifier les responsabilités. Après chacun doit évaluer ses besoins au niveau de la performance attendue et coder en conséquence. D'ailleurs, je n'ai pas du tout respecté les conseils de Google pour améliorer les performances: http://developer.android.com/guide/practices/design/performance.html.

Merci pour votre commentaire :-). Ça fait plaisir d'avoir des retours!

RÉPONDRE ↓

**YASSINE**
*14 août 2011 à 18 h 18 min*

bon article

RÉPONDRE ↓

**WTC0DER**
*8 novembre 2011 à 22 h 11 min*

Article très sympa et bien pédagogique, ça rappel des souvenirs (UML).
Bref, je cherchais des expérimentations d'application d'MVC pour Android et j'ai trouvé.

Merci à toi !

RÉPONDRE ↓

**NICOLAS**
*9 novembre 2011 à 5 h 24 min*

Merci pour le feedback!

RÉPONDRE ↓

**PHILIPPE**
*14 février 2012 à 18 h 10 min*

Merci pour l'article.
Dans le cas de multiples activités, comment peut-on organiser le code ?
Une activité = une Vue + un Contrôleur + un Modèle ?

Est-il préférable d'avoir un super-contrôleur ?

Quelles pistes prendre ?...

Merci pour vos indications.

RÉPONDRE ↓

**NICOLAS**
*15 février 2012 à 6 h 46 min*

Bonjour Philippe,

Une activité ne peut pas être assimilée à un MVC tout entier. On pourrait plutôt l'assimiler à la partie « Vue » du MVC mais encore ce propos est à modérer (je vais essayer d'expliquer pourquoi dans la suite).

Une activité est un mécanisme de gestion du cycle de vie d'une fenêtre graphique (création d'un thread lors de l'appel de onCreate() et destruction du même thread lors de l'appel de onDestroy()). Une activité est donc un mécanisme plutôt « bas niveau » (gestion de threads) tandis que la « Vue » d'un MVC conventionnel est plutôt un mécanisme « abstrait », « haut niveau ».

C'est pour cette raison que j'ai volontairement utilisé 2 classes distinctes : une pour l'activité (Calculator) et une pour la vue (CalculatorView). Ce choix n'est pas le meilleur en terme de performance mais il a l'avantage de mieux coller à la théorie du MVC. Autre avantage à ce choix d'utiliser 2 classes : on ne pollue pas la partie « Vue » conventionnelle du MVC (CalculatorView dans le tuto) avec les mécanismes d'interactions entre activités (lancement d'autres activités, destructions d'autres activités, etc). On découple donc « Android » de notre conception « métier » (pur Java).

En pratique, on pourrait également choisir amalgamer ces 2 classes (Calculator et CalculatorView) pour n'en constituer qu'une seule. C'est moins gourmand car on alloue une seule classe… C'était d'ailleurs la question de « Jbbi » (voir commentaires précédents).

Conclusion : la partie « Vue » du MVC peut être considérée comme (Calculator + CalculatorView) ou bien comme seulement CalculatorView. Tout est question de point de vue, c'est un point discutable… Dans le tuto, j'ai considéré que la vue était l'ensemble des 2 classes (cf. http://code-weblog.com/developpement-dune-appli-android-basee-sur-un-mvc/#titre4-4-1).

Maintenant pour revenir à votre problématique, il faut vraiment prendre du recul sur le sujet et s'interroger : est-ce que les différentes activités interagissent avec les mêmes données, est-ce qu'elles n'ont rien avoir, etc…

Un exemple concernant la calculatrice…
- Si demain, on souhaite modifier le nombre de décimales en remplaçant les boutons « - » et « + » par un widget de type « scrollbar », on va créer une nouvelle vue mais conserver le contrôleur et le modèle à l'identique. On aura juste modifier la façon d'interagir avec l'utilisateur.
- Si demain, on souhaite faire le calcul sur un serveur distant. On va alors ajouter un modèle et probablement utiliser un design pattern « Adaptateur » pour faire coller ce nouveau modèle avec la classe d'interface « ModelInterface ». La vue sera conservée à l'identique, et pour le contrôleur il faudra voir…
- Si demain on souhaite ajouter un formulaire d'inscription à un jeu concours à la fin de chaque calcul effectué par notre calculatrice, on devra concevoir un nouveau MVC car ces 2 fonctionnalités n'ont rien en commun.

Au sujet du MVC je peux vous conseiller de lire « Head First Design Patterns » qui est vraiment un excellent bouquin, très simple et très pratique. Le MVC est vraiment abordé simplement (ce qui change de la majorité des autres livres qui rendent le sujet imbuvable…) dans le chapitre 12.

En espérant vous avoir guidé et ne pas trop vous avoir embrouillé 😉
Bonne continuation!

RÉPONDRE ↓

**PHILIPPE**
*16 février 2012 à 14 h 36 min*

Effectivement, c'est bien ce que je pensais. Il faudra créer des classes suivant le pattern MVC pour des activités n'ayant pas de lien entre elles.
C'est l'imbrication de ces MVC qui n'est pas encore très clair. Il est difficile de trouver des exemples montrant plus d'une vue ou activité…
Dans votre exemple, vous utilisez une interface ControllerInterface. Cette interface ne pourra pas être générique et être utilisée pour un deuxième contrôleur chargé de gérer une autre activité. Faut-il recréer une deuxième interface pour ce contrôleur ?

Merci pour ces explications.

Philippe

RÉPONDRE ↓

**NICOLAS**

*16 février 2012 à 15 h 50 min*

Effectivement dans ce cas, il faudra recréer une autre classe d'interface « ControllerInterface2″ et un contrôleur « Controller2″ implémentant cette interface. S'il s'agit d'une activité indépendante, n'ayant rien en commun avec la première au point de vue fonctionnel, on recrée un deuxième MVC exactement comme dans le tutoriel.

Vous vous demandez alors pourquoi cette classe d'interface qui complique tout! C'est le design pattern Strategy (voir ici) qui nous l'impose. Ce design pattern ne sert pas pour l'instant mais n'oublions pas que le but de la conception objet au travers des design pattern est de respecter des règles de bonne conduite, ceci pour la maintenance, évolutivité et la réutilisabilité.

Ici, en utilisant le pattern Strategy, on permet de décolérer la Vue du contrôleur. En effet, la vue ne connait que l'interface du contrôleur. On peut ainsi modifier les méthodes du contrôleur qui ne sont pas définies dans l'interface sans devoir modifier une seule ligne de code de la vue. La vue n'en saura rien! 😉

Si demain on souhaite utiliser un autre contrôleur (toujours pour ce MVC dans cette même activité) que l'on avait déjà codé pour une application un peu similaire, on ajoutera un adaptateur pour que le nouveau contrôleur « colle » à notre classe d'interface. Ainsi, une fois de plus, l'interface reste identique et donc pas une ligne de code à modifier dans la vue.

Pour conclure, les design-patterns alourdissent la conception et bien souvent nuisent aux performances. En revanche, quand on souhaite faire évoluer l'application (ce qui est l'essence même du soft il me semble…), les design patterns nous permettent de ne pas devoir tout casser et saloper le code existant. L'ancien code étant déjà testé et validé, on a intérêt à ne plus jamais le toucher! C'est un des principes de la philosophie des design patterns! Je vous rappelle les basiques (extraits du livre que je vous ai conseillé) : encapsuler ce qui varie, favoriser la composition plutôt que l'héritage, **utiliser des interfaces dès que possible** (pour les raisons expliquées ci-dessus), utiliser des couplages faibles entre les entités qui interagissent, **les classes doivent être ouvertes aux extensions mais fermées aux modifications**, faire des dépendances vers des abstractions plutôt que des classes concrètes, respecter et bien isoler les responsabilités des classes, etc… (Les principes en gras devraient vous évoquer notre MVC ;))

J'espère que vous y voyez un peu plus clair maintenant sur l'esprit du MVC et des design-patterns : ils structurent l'applicatif et pérennisent les évolutions. Sinon on peut tout à fait s'en passer…

Bonne soirée

RÉPONDRE ↓

**PHILIPPE**

*16 février 2012 à 17 h 36 min*

OK, il me reste plus qu'a mettre ça en application !
Merci
Philippe

**LHOUSSAIN**
*5 mars 2012 à 20 h 17 min*

Merci pour votre article j'aimerais bien savoir comment je exploiter cette application dans Adroid 2.2 ça marche pas !!

RÉPONDRE ↓

**NICOLAS**
*5 mars 2012 à 21 h 07 min*

Si vous voulez de l'aide merci d'être d'avantage constructif! Que se passe-t-il? Message d'erreur?

Chez moi, l'application fonctionne parfaitement sur une cible Android 2.2.

RÉPONDRE ↓

**LHOUSSAIN**
*5 mars 2012 à 21 h 25 min*

Je m'excuse alors mais déjà je vous félicite sur votre travail, j'ai importé votre application et il me signale que le package de chaque classe est inconnue :
« Description Resource Path Location Type
Unable to resolve target 'android-3' Calculator Unknown Android Target Problem
»
Voilà le problème qui affiche merci pour votre aide précieux.

RÉPONDRE ↓

**NICOLAS**
*6 mars 2012 à 6 h 24 min*

Juste une remarque par rapport à votre message d'erreur : vous essayer de compiler pour une cible Android 3 et non pas Android 2.2.

Mais l'erreur semble être une erreur de Java « pur » et non pas d'une erreur liée à Android. Avez-vous modifié des noms dans le projet?

En Java, la définition d'un package se fait comme ceci :
package com.weblog.MVC;

La structure physique dans le projet doit alors impérativement être cohérente avec cette définition. Tous les fichiers sources qui appartiennent à ce package devront se trouver dans :

racineProjet / src / com / weblog / MVC /

Votre message d'erreur fait penser à une structure physique (noms et hiérarchie des répertoires de votre projet) incohérente avec le nom du package.

Le plus simple, vous téléchargez à nouveau le projet, vous le dézippez, vous le placez dans votre workspace et vous faites : File / Import / Existing Projets Into Workspace

Vous repartirez ainsi sur des bases saines et j'espère que la projet sera correctement compilé cette fois…

Astuce : Pour modifier le nom d'une variable, d'une classe ou d'un package par exemple, il faut toujours utiliser la fonction « Refactor / Rename » d'Eclipse (ou un autre IDE). L'IDE connaît tout le projet, il est donc capable de modifier tous les fichiers et tous les répertoires du projet pour conserver la cohérence. Par exemple, il serait très dangereux de modifier un nom de package Java sans utiliser cette fonctionnalité. En effet, il faudrait modifier le nom de certains répertoire, le nom du package dans tous les fichiers, etc… On est certain d'en oublier et de galérer quelques heures! Alors qu'avec Refactor / Rename, ça prend 15 secondes et l'IDE ne se trompe pas… 😉

Bon courage, en espérant vous avoir aidé.

RÉPONDRE ↓

**NICOLAS**
*6 mars 2012 à 6 h 29 min*

Je viens de penser à une autre piste…

Tout simplement avez vous bien crée un émulateur ayant la version 3 de Android? Si vous demandez à faire le Run sur un émulateur qui n'existe pas, vous obtenez un message d'erreur.

Pour vérifier :
Window / Android SDK and AVD Manager

. Vous devriez avoir un AVD (Android Virtuel Device, c'est à dire un émulateur) ayant la propriété « Target Name » valant « Android 3.0″ (API Level = 11).

**ZARK**
*5 juillet 2012 à 13 h 24 min*

bonjour,

Excellent tutoriel qui m'a bien aidé à mettre en place mon architecture MVC, merci!
Pour ma part j'ai opté pour un contrôleur unique et des models et vues qui changent selon les fonctionnalités.

J'aurais aussi aimé connaitre le logiciel utilisé pour les diagrammes UML de classes et de séquences.

RÉPONDRE ↓

**NICOLAS**
*5 juillet 2012 à 15 h 29 min*

Merci pour le feedback Zark!

Pour le modeler va voir ce lien: http://www.eclipse.org/modeling/mdt/downloads/?project=uml2 Attention il est parfois instable et toutes les fonctionnalités ne sont pas opérationnelles…

Sinon j'ai trouvé bien mieux, c'est Modelio : http://www.modeliosoft.com/en/download/free-downloads.html C'est basé sur un socle Eclipse, les schémas sont sympas et le standard UML2 n'est pas trop saccagé comme dans d'autres modeleurs… Par contre je crois qu'il n'y a pas d'engineering et de reverse dans la version community. Donc juste pour du schéma c'est au top.

RÉPONDRE ↓

**GASPOUTE**
*6 octobre 2012 à 11 h 41 min*

Bonjour,

Super tuto ! J'en cherchais justement un, merci J'ai cependant une petite question: lors d'un changement d'activité, par exemple une liste de recettes, vous tapez sur l'une d'elle et vous passez à une autre activité qui vous affiche les détails de cette recette. Comment organiseriez-vous votre (vos) mvc('s) ? Un contrôleur, un modèle pour les 2 activités et 2 vues ? Comment gérez-vous le fait de passer d'une vue à l'autre ? Par dans ce cas-ci lorsque le contrôleur est le même et lorsque le contrôleur est différent ?

Merci.

RÉPONDRE ↓

**NICOLAS**
*8 octobre 2012 à 16 h 35 min*

Bonjour Gaspoute,

Désolé pour ma réponse tardive…

Dans votre cas, vous souhaitez afficher des recettes et uniquement des recettes. C'est donc le même modèle et le même contrôleur. En revanche il y aura 2 vues: ListView.java et ShowView.java par exemple.

Le modèle va s'occuper de faire des requêtes en Base de données et de valider les données d'un utilisateur qui souhaite créer une nouvelle recette par exemple. Les vues vont décrire l'affichage de la données en positionnant des widgets par exemple. Le contrôleur va se

charger de lier toutes les entités (interactions entre modèle et vues) et de gérer la logique applicative.

Concernant les activités, je ne peux pas trop vous répondre de manière affirmative. Je n'ai pas eu l'opportunité de poursuivre le développement Android comme je l'aurais voulu, donc je n'ai pas assez de recul. Voici quand même quelques éléments de réponse.

Si une vue est de type menu qui lance d'autres vues, la réponse est qu'il faut utiliser une autre activité (lien de parenté). Par exemple, un menu de liste de recettes donne naissance à une vue détaillée de la recette. Je vous renvoi à un autre tuto que j'ai écris et dans lequel on utilise 2 activités: http://code-weblog.com/appli-android-pour-enquete-de-terrain/.

En revanche, si on souhaite passer d'une recette à une autre, il y a 2 solutions. Soit lancer une autre activité (autre thread = gourmand et je pense inutile ici), soit mieux (je pense), actualiser la vue avec les informations de la nouvelle recette.

Pour conclure, cela dépend des cas… Il faut se poser la question du lien. Est-ce que ma vue doit vivre en même temps que l'autre? Oui => nouvelle activité. Non => les 2 solutions sont possibles. Y a-t-il intérêt de garder une activité en background (échange de données, système de tabs qui permettrait de passer de l'une à l'autre, etc). Etc, etc…

Vous l'aurez compris, en pratique, il faut toujours peser le pour et le contre et ne jamais appliquer des conseils « tout fait ». Il n'y a pas souvent de solutions magiques mais des solutions plus ou moins coûteuses et plus ou moins facile à mettre en place, etc.

En espérant vous avoir aidé… Bon courage! 😉

RÉPONDRE ↓

**GASPOUTE**

*9 octobre 2012 à 22 h 57 min*

Je dirais plutôt que vous avez répondu assez vite 😉 merci.

Je suppose que c'est le même schéma de vues avec un même contrôleur et un même modèle si dans la ShowView (vue d'une recette en détails) on peut modifier la recette en question ? Il n'y aura qu'à ajouter quelques méthodes de modifier au contrôleur et au modèle. Quel « genre » de modèle utiliseriez-vous ? Une RecipesList par exemple ? Dernière question: est-ce une bonne idée de ma part de considérer une activité comme une vue à part entière, donc sans objet CalculatorView si on prend comme ex ce tuto ? J'instancie alors le modèle et le contrôleur dans la vue en assignant ce qu'il faut à chacun d'eux.

RÉPONDRE ↓

**NICOLAS**

*10 octobre 2012 à 15 h 45 min*

Pour la première question, on peut soit faire une vue RecipeShow et une vue RecipeEdit ou utiliser la même pour les 2 actions. Qu'est-ce que vous entendez par « genre » de modèle?? Le modèle devra être chargé des responsabilités suivantes: valider les données de l'utilisateur qui modifierait une recette (par exemple s'assurer que les masses soient bien des données

numériques, etc), lire et écrire en base de données, il devra également notifier les vues de ses changements.

Pour la 2ème question, j'en ai discuté avec un collègue qui fait beaucoup de dév sur Android, et il assimile très souvent vue/activité. Je pense effectivement que ce doit être plus facile à écrire et à maintenir. Par contre il peut y avoir des cas où on ne change pas d'activité si on doit uniquement modifier un tout petit morceau d'une vue. Par exemple dans le cas de la vue RecipeEdit qui pourrait servir à la fois à la visualisation et à l'édition, on pourrait simplement changer un widget de type « simple texte » en « textfield ».

Effectivement il reste à savoir où instancier le MVC… L'application possède nécessairement une activité principale, c'est son point d'entrée. Je pense que tous les MVC de l'application devraient être instanciés le plus souvent dans cette activité principale. En effet si on les instancie dans des activités secondaires, en cas de destruction de ces activités, on perdra les instances et on devra alors tout ré-instancier à nouveau. Ou alors il faut a minima conserver des références dans l'activité principale afin que le garbage collector ne détruise pas tout. Évidemment tout dépend de l'ampleur de l'application… Si elle possède des centaines de MVC, on ne pourra pas tout allouer en même temps…

Dans tous les cas, il faut faire des choix et les valider par expérimentations. Je ne suis pas du tout certain de ne pas avoir dit de bêtises… Il faut tester… Peut être que d'autres développeurs pourront apporter plus d'expertise et rectifier mes propos si j'ai dis des bêtises…

RÉPONDRE ↓

**GASPOUTE**

*10 octobre 2012 à 16 h 11 min*

Merci

J'entends par « genre » de modèle dans ce cas-ci un modèle nommé « RecipesList » et il gèrerait l'affiche des recettes et la modification de chaque recette.Ou peut-être que je dois être plus global dans le modèle ? Et est-ce que selon vous je dois différencier les entités des modèles ? Pour moi le modèle est en quelque sorte un calque de la vue contrairement à une entité qui ne possède que des setters/getters et est utilisée par un modèle. Je me trompe peut-être ?

**NICOLAS**

*10 octobre 2012 à 17 h 54 min*

Je pense qu'il vous faut revoir la notion de modèle. Le modèle:
– caractérise la donnée
– va chercher de la donnée (base de données, service, …)

– valide la donnée
– retravaille la donnée

Dans votre cas, il ne peut y avoir qu'un seul modèle et il s'appellerait « Recipe ».

Il peut y avoir plusieurs vues qui peuvent présenter la même donnée mais présentée de diverses façons.

**GASPOUTE**
*10 octobre 2012 à 19 h 21 min*

Donc dans le cas, si le tout est stocké dans une base de données, le modèle manipule directement l'objet retourné par la classe DAO qui se charge de faire correspondre un objet à une entrée de la bdd ? Et est-ce que dans le modèle Recipe sont comprises les méthodes qui renvoyent une liste de Recipe ? Un modèle représente alors un « domaine » ?

Merci de votre aide précieuse 

RÉPONDRE ↓

## Laisser un commentaire

Votre adresse de messagerie ne sera pas publiée. Les champs obligatoires sont indiqués avec *

Nom *

Adresse de contact *

Site web

Commentaire

Vous pouvez utiliser ces balises et attributs HTML :

```
<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code> <del datetime=""> <em> <i> <q cite=""> <strike> <strong>
```

Laisser un commentaire