

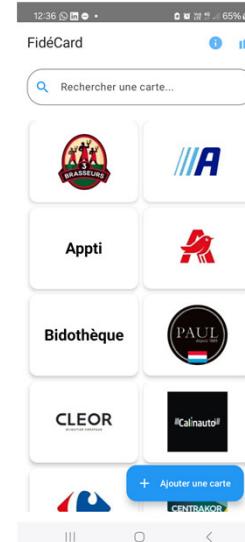
Avant de commencer le chapitre 2...

- Concernant la vidéo à réaliser pour le projet, dans le cahier des charges (CDC), j'ai ajouté :
 - *Pour réaliser cette vidéo dans les salles du M5, vous pouvez désormais utiliser vokoscreen-ng. Il suffit de taper le nom du logiciel dans la barre de recherche pour le trouver.*
Les vidéos sont enregistrées par défaut dans ~/Videos/ mais vous pouvez changer l'emplacement facilement. Notamment mettre dans /local si vous êtes toujours sur la même machine.

Questions sur le cahier des charges ?

Android, faites vos propres applis facilement !

- À la fin de l'UE, vous pourrez faire vos propres applis
 - Exactement ce que vous voulez (ex. avec prises de notes Remind You)
 - Pour vous rendre service (ex. Fidécard)
 - Pour vous faire connaître (boite à coucou, boite à meuh)



UE TAC

M1 Informatique

Cours : Jean-Claude TARBY

TDs : Mehdi BRAHMI, Cédric DUMOULIN, Jean-Claude TARBY

Avant « d'attaquer » le cours...

Jetpack ?

- On a parlé de Jetpack, mais késako ?

Jetpack compose ? (ChatGPT)



Jetpack Compose est un framework moderne pour construire des interfaces utilisateur (UI) en Android. Développé par Google, il est conçu pour simplifier et moderniser le développement d'UI en utilisant une approche déclarative. Cela signifie que, plutôt que de définir une hiérarchie complexe de vues (comme c'était le cas avec l'ancienne API XML d'Android), les développeurs décrivent ce que l'UI doit afficher en fonction de l'état de l'application, et Jetpack Compose se charge de mettre à jour l'interface utilisateur en conséquence.

Voici quelques points clés sur Jetpack Compose :

1. **Déclaratif** : Contrairement à la méthode impérative traditionnelle, Jetpack Compose utilise une approche déclarative. Le développeur décrit simplement ce que doit être l'interface utilisateur, et Compose se charge de la rendre à l'écran et de la mettre à jour en réponse aux changements d'état.
2. **Intégration avec Kotlin** : Jetpack Compose est conçu pour fonctionner parfaitement avec le langage Kotlin. Grâce aux fonctionnalités de Kotlin, comme les lambdas et les coroutines, Compose offre un code plus concis et lisible.
3. **Composants réutilisables** : Avec Compose, les UI sont construites à partir de composants (appelés "composables"). Ces composants sont facilement réutilisables. Ces

Pourquoi l'utiliser ? (ChatGPT)



Jetpack Compose est une technologie moderne pour développer des interfaces utilisateur (UI) en natif sur Android, introduite par Google. Voici pourquoi les développeurs choisissent de l'utiliser :

1. Simplicité et Clarté

- **Déclaratif** : Contrairement à l'approche impérative traditionnelle (XML + Java/Kotlin), Jetpack Compose utilise un modèle déclaratif. Vous définissez **quoi** afficher plutôt que **comment** l'afficher, ce qui rend le code plus simple et lisible.
- **Moins de code** : Compose réduit la quantité de code nécessaire pour créer des interfaces complexes, ce qui accélère le développement et réduit le risque d'erreurs.

2. Flexibilité et Réactivité

- **Dynamisme** : Les UI en Compose s'adaptent facilement aux changements d'état grâce au modèle de réactivité intégré. Lorsqu'un état change, Compose réévalue et met à jour l'UI automatiquement.
- **Personnalisation** : Jetpack Compose offre une grande flexibilité pour personnaliser les composants UI, ce qui permet de créer des designs uniques et adaptés aux besoins

Jetpack: 109 librairies (dont Compose et Room)

<https://developer.android.com/jetpack>

The screenshot shows a web browser window displaying the Android Jetpack documentation at <https://developer.android.com/jetpack>. The page is in French, as indicated by the 'Français' button in the top right. The navigation bar at the top includes links for 'Developers', 'Plate-forme', 'Jetpack' (which is underlined in green), 'Plus', 'Recherche', 'Français', 'Android Studio', and 'Connexion'. Below the navigation bar, there's a sub-navigation menu for 'JETPACK' with tabs for 'Présentation' (selected), 'Premiers pas', 'Bibliothèques' (circled in yellow), 'Compose', and 'Communauté'. The main content area features a large title 'Android Jetpack' and a paragraph describing Jetpack as a suite of libraries for developers. To the right, there's a cartoon illustration of an Android character floating in space. At the bottom, there are three cards: one for 'Version Compose du 22 octobre', one for 'Paging 3.1', and one for 'Wear Compose 1.0'.

Bibliothèques

Android Jetpack

Jetpack est une suite de bibliothèques qui permet aux développeurs de suivre les bonnes pratiques, de réduire le code récurrent et d'écrire du code qui fonctionne de manière cohérente sur toutes les versions d'Android et sur tous les appareils. Ainsi, ils peuvent se concentrer sur le code qui les intéresse.

Premiers pas avec Jetpack Regarder la vidéo

Version Compose du 22 octobre

Inclut de nouvelles fonctionnalités telles que VariableFonts et SwipeRefreshLayout pour faciliter le développement de l'UI. Nous avons

Paging 3.1

Paging 3 inclut une prise en charge de premier ordre des coroutines Kotlin et de Flow, et prépare le terrain pour l'intégration de Compose. Cette version a pour objectif de réduire le

Wear Compose 1.0

Exploiter Compose sur Wear OS. Compose pour Wear OS s'appuie sur les bibliothèques principales de Compose pour fournir des composants supplémentaires spécifiques aux

Jetpack: 109 librairies (dont Compose et Room)

compose.animation	Créer des animations dans les applications Jetpack Compose afin d'enrichir l'expérience utilisateur.
compose.compiler	Transformation de fonctions <code>@Composable</code> et prise en charge des optimisations avec un plug-in de compilation en Kotlin.
compose.foundation	Conception d'applications Jetpack Compose avec des éléments de base prêts à l'emploi et extension de la base pour créer vos propres éléments de système de conception.
compose.material	Création d'UI Jetpack Compose avec des composants Material Design prêts à l'emploi. Il s'agit du point d'entrée de niveau supérieur de Compose, conçu pour fournir des composants qui correspondent à ceux décrits sur www.material.io .
compose.material3	Création d'UI Jetpack Compose grâce aux composants Material Design 3, la nouvelle évolution de Material Design. Material 3 inclut des thématisations et des composants mis à jour ainsi que des fonctionnalités de personnalisation Material You telles que les couleurs dynamiques. Material 3 est conçu pour correspondre au nouveau style visuel et à l'UI du système Android 12.
compose.runtime	Composants fondamentaux du modèle de programmation et de la gestion d'état de Compose, et environnement d'exécution principal du plug-in Compose Compiler à cibler.
compose.ui	Les composants fondamentaux de l'interface utilisateur de Compose nécessaires pour interagir avec l'appareil, y compris la mise en page, le dessin et la saisie.

Jetpack : les librairies

The screenshot shows the 'Libraries' section of the Jetpack page on the Android Developers website. The top navigation bar includes links for Platform, Android Studio, Google Play, Jetpack (which is underlined), Kotlin, Docs, and Games. A search bar and language selection (English) are also present. The left sidebar has sections for Overview, Get Started, Libraries (which is selected and highlighted in blue), Community, and Compose. Under 'Explore the libraries', there are links for Library Releases, Release Notes, Release Note Archive, and Jetpack Compose Roadmap. The main content area is titled 'Explore the Jetpack libraries by type'. It features a 'Select a category:' section with several buttons: All (checked), Popular (checked), Beyond phones, Data, Graphics, Lifecycle, Media, Navigation, Security, Performance/Test, and UI (which is currently selected). Below this is a 'Filter by keyword or use case' input field. A note states '* Popular and often-used library'. A table lists the following libraries:

Library	Description
activity *	Access composable APIs built on top of Activity.
ads	Get an advertising ID with or without Play Services.
annotation	Expose metadata that helps tools and other developers understand your app's code.
appcompat *	Allows access to new APIs on older API versions of the platform (many using Material Design).
appsearch *	Build custom in-app search capabilities for your users.

M1 Info - <https://developer.android.com/jetpack/androidx/explorer>

Jetpack Compose

The screenshot shows the official Jetpack Compose documentation page on the developer.android.com website. The top navigation bar includes links for developers, Platform, Android Studio, Google Play, Jetpack (which is underlined), Kotlin, Docs, and Games. A search bar and language selection (English) are also present. Below the navigation, a secondary navigation bar for 'JETPACK' includes links for Overview, Get Started, Libraries, Community, and Compose (which is underlined). The main content features a large, bold title: "Build better apps faster with Jetpack Compose". A subtitle explains: "Jetpack Compose is Android's modern toolkit for building native UI. It simplifies and accelerates UI development on Android. Quickly bring your app to life with less code, powerful tools, and intuitive Kotlin APIs." To the left of the title is a code snippet:

```
@Composable
fun JetpackCompose() {
    Card {
        var expanded by remember { mutableStateOf(false) }
        Column(Modifier.clickable { expanded = !expanded }) {
            Image(painterResource(R.drawable.jetpack_compose))
            AnimatedVisibility(expanded) {
                Text(
                    text = "Jetpack Compose",
                    style = MaterialTheme.typography.h2,
                )
            }
        }
    }
}
```

To the right of the code is a preview card showing a blue hexagonal icon with a white center, the text "Jetpack Compose", and a "Card" label at the bottom.

M1 Info - l
<https://developer.android.com/jetpack/compose>

[View tutorial](#) [View docs](#)

16

Pourquoi utiliser Jetpack ?

Why use Android Jetpack?



Follow best practices

Built around modern design practices, Android Jetpack Libraries enable fewer crashes and less memory leaks with backwards-compatibility baked in.

[REVIEW CASE STUDY](#)



Eliminate boilerplate code

Android Jetpack manages tedious activities like background tasks, navigation, and lifecycle management, so you can focus on what makes your app great.

[REVIEW CASE STUDY](#)

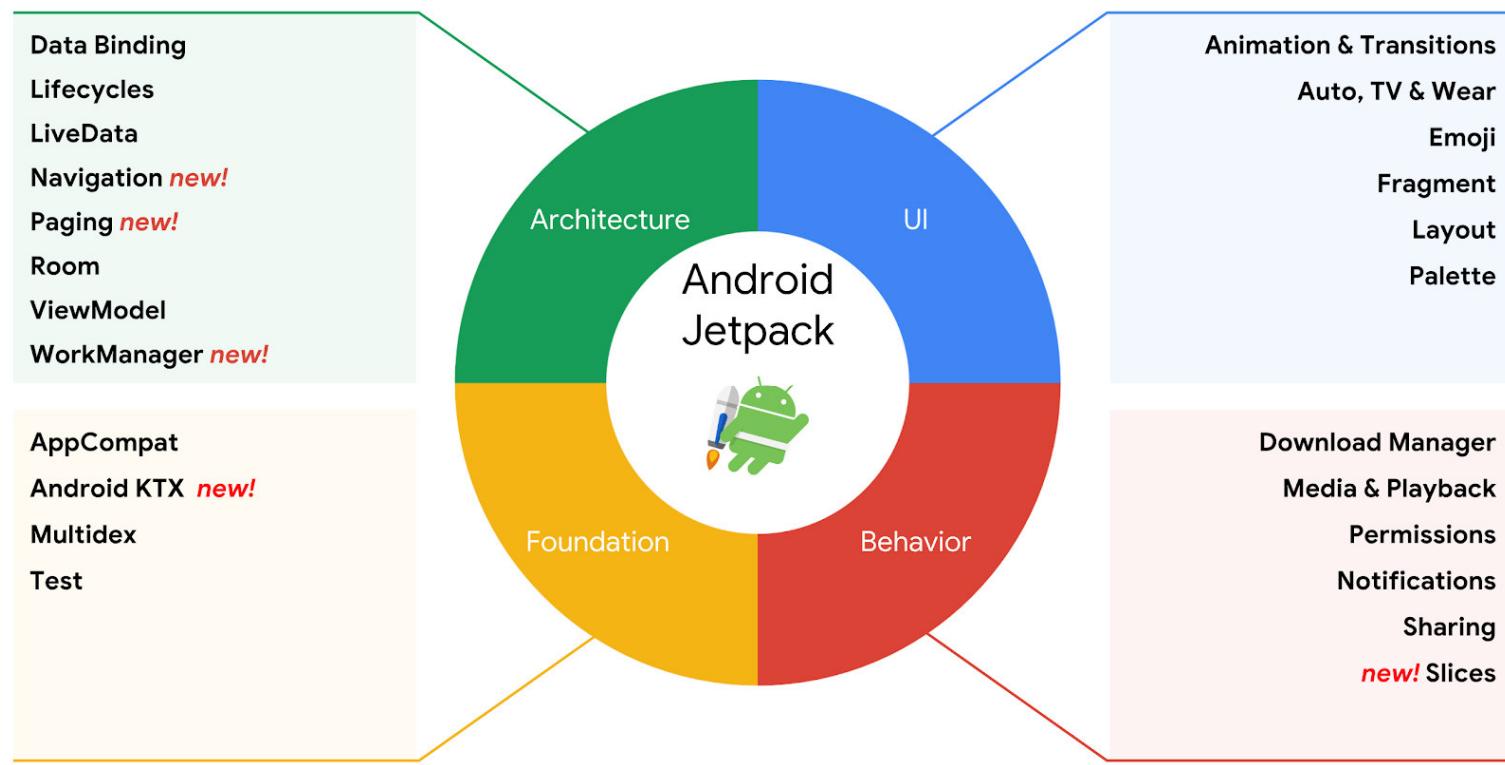


Reduce fragmentation

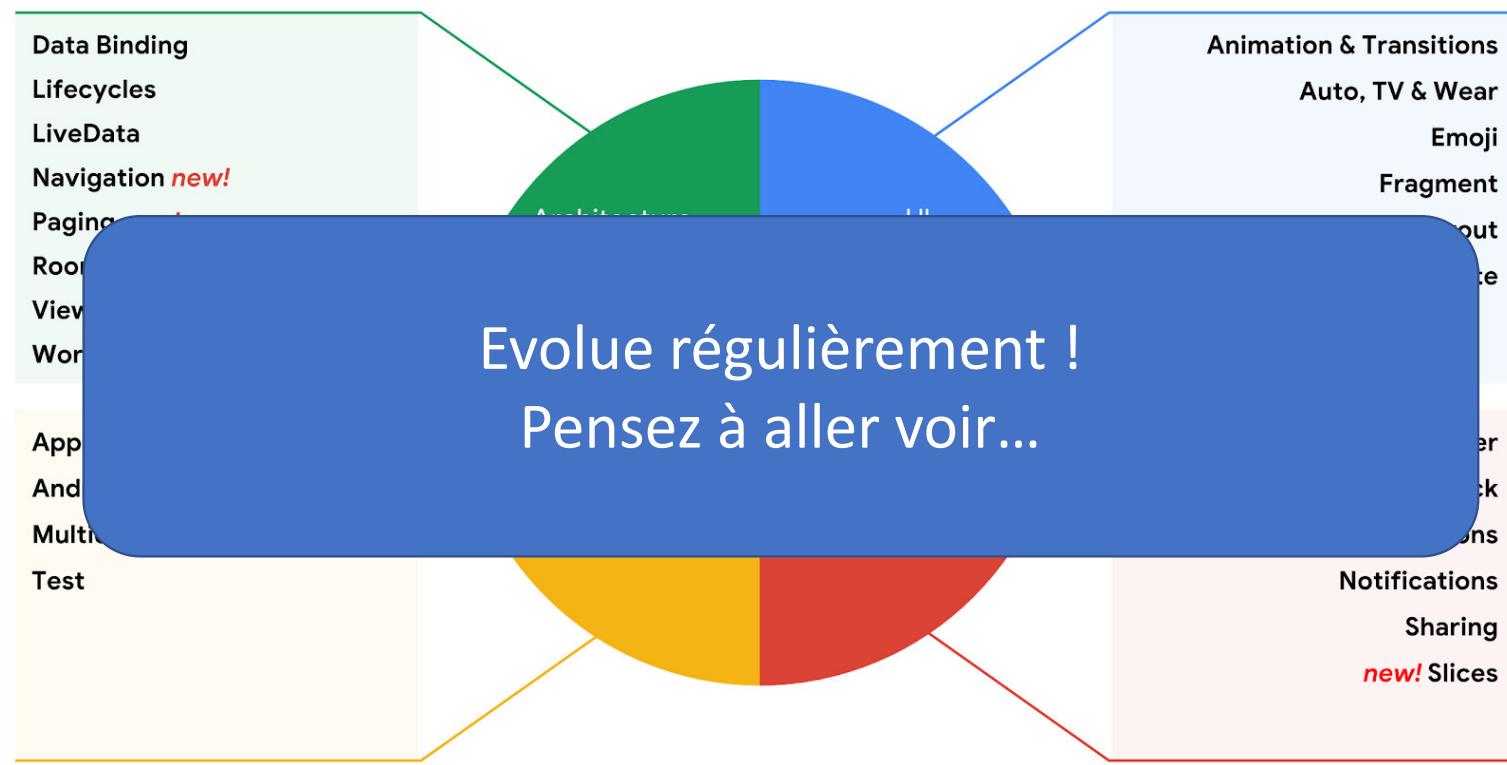
Reduce complexity with libraries that work consistently across Android versions and devices.

[READ TESTIMONIALS](#)

Jetpack



Jetpack



Revenons au cours...

Cours 2

« Architectures » Android

Les architectures

- A quoi ça sert ?
- Vous en utilisez ?
- Lesquelles?
- Pourquoi celles-là ?

Les architectures

- A quoi ça sert ?
 - Maintenabilité
 - Testabilité
 - Séparation des préoccupations
 - Lisibilité

Les architectures...et les Design Patterns ?

Les architectures...et les Design Patterns ?

- A quoi ça sert ?
- Et c'est quoi un Design Pattern ?
- MVC = Architecture ou Design Pattern ?

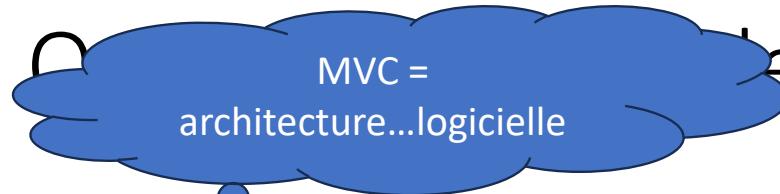
Rappel (?) sur les Design Patterns

Qu'est-ce qu'un design pattern ?

- Beaucoup de définitions différentes ont été données.
 - "Chaque pattern décrit un problème qui se présente souvent dans notre environnement, et décrit le cœur d'une solution à ce problème sur un mode qui permet d'utiliser cette solution des millions de fois, sans jamais avoir à refaire deux fois la même chose" (Christophe Alexander cité par Gamma, *et al.*)
 - « un design pattern est une structure de réflexion qui permet de trouver une solution à un problème qui survient maintes et maintes fois sans pour autant être jamais identique ». JC. Tarby
- Qu'est-ce que cela veut dire ?
→ détails ci-après.....

Qu'est-ce qu'un design pattern ?

- MVC utilise des designs patterns de façon implicite
 - par exemple « souscription/abonnement ».
- Le concept d'abonnement (souscription/notification) est un problème tout à fait général.
 - Il peut être appliqué à beaucoup d'autres cas, par exemple :
 - un processus à l'écoute d'un autre processus,
 - un programme à l'écoute d'un périphérique
 - Une IHM à l'écoute du NF...



Design pattern ?

- MVC utilise des designs patterns de façon implicite
 - par exemple « souscription/abonnement ».
- Le concept d'abonnement (souscription/notification) est un problème tout à fait général.
 - Il peut être appliqué à beaucoup d'autres cas, par exemple :
 - un processus à l'écoute d'un autre processus,
 - un programme à l'écoute d'un périphérique
 - Une IHM à l'écoute du NF...

Qu'est-ce qu'un design pattern ?

- MVC utilise des designs patterns de façon implicite
 - par exemple « **souscription/abonnement** ».
- Le concept d'abonnement (souscription/notification) est un problème tout à fait général.
 - Il peut être appliqué à beaucoup d'autres cas, par exemple :
 - un processus à l'écoute d'un autre processus,
 - un programme à l'écoute d'un périphérique
 - Une IHM à l'écoute du NF...

Qu'est-ce qu'un design pattern ?

- Derrière ce protocole de communication, il y a toujours les mêmes primitives
 - primitive qui permet de demander à s'abonner,
 - primitive qui notifie les abonnés en cas de changement,
 - primitive qui permet de changer d'abonnement,
 - etc.
- Mettre en pratique ce protocole dans un cas particulier → développer le code spécifique à ce problème.
→ Le **design pattern** permet de structurer votre pensée et éventuellement de vous **donner des pistes pour structurer votre code**, mais en aucun cas il ne vous donne une solution complète et immédiate.

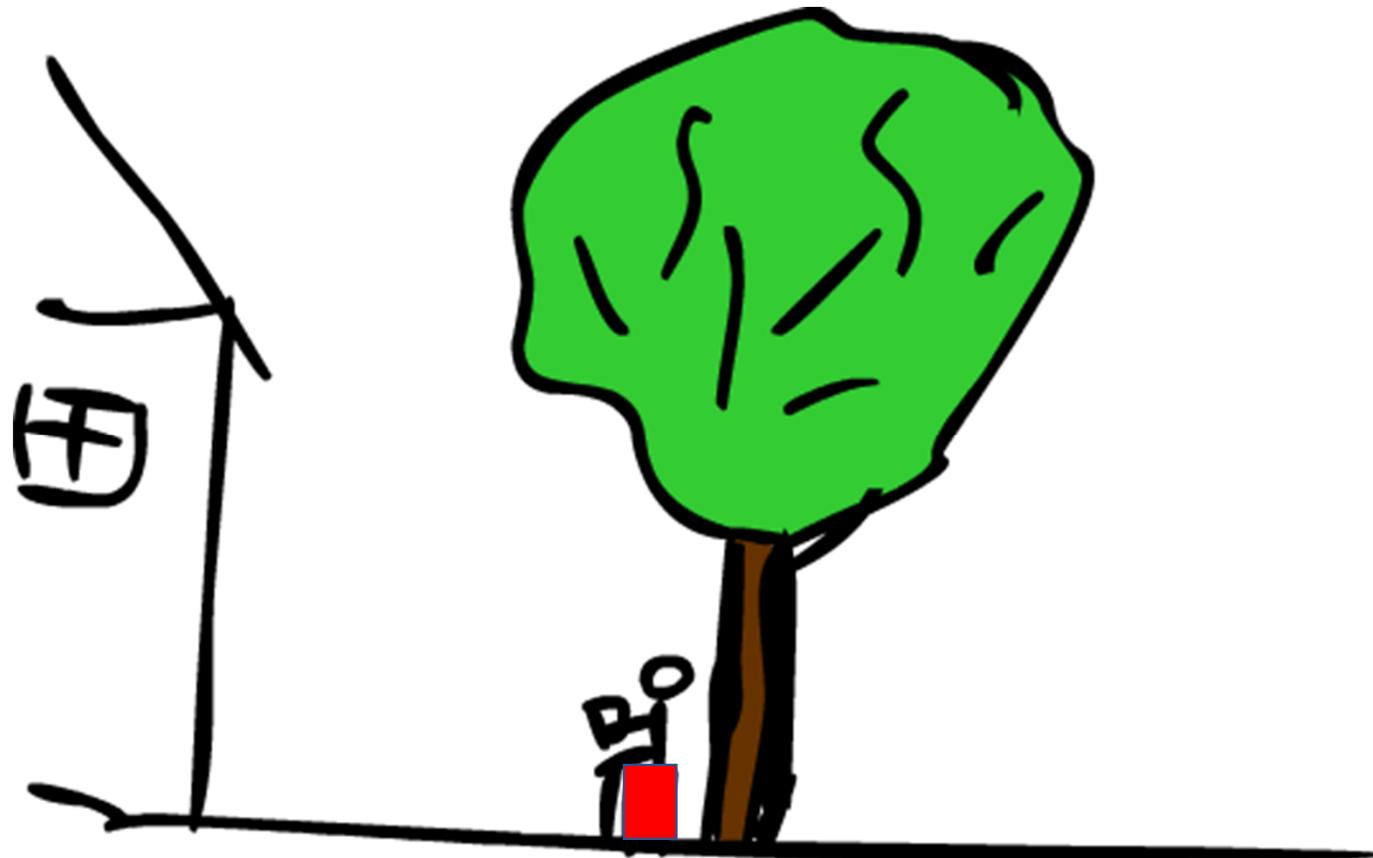
Origine des Design Patterns : l'Architecture !

- Depuis très longtemps déjà, les architectes se sont rendus compte qu'ils construisaient plus ou moins toujours les mêmes choses.
 - une maison du Nord ressemble à une autre maison du Nord, une maison du Sud ressemble à une autre maison du Sud,
 - et pourtant chaque maison du Nord est différente des autres maisons du Nord, même chose pour les maisons du Sud.
- Les architectes se sont donc posés la question de savoir s'il était possible d'établir des cadres de réflexion pour des plans de maison, de quartier, de ville.
(Christopher Alexander, 1970)
 - Grâce à cela, il serait possible d'avoir des structures de villes (de ou quartiers) de repos, de villes (ou quartiers) de travail, de villes (ou quartiers) de loisirs,...
 - Christopher Alexander a créé 253 modèles (patterns) pour décrire la conception d'architectures de bâtiments

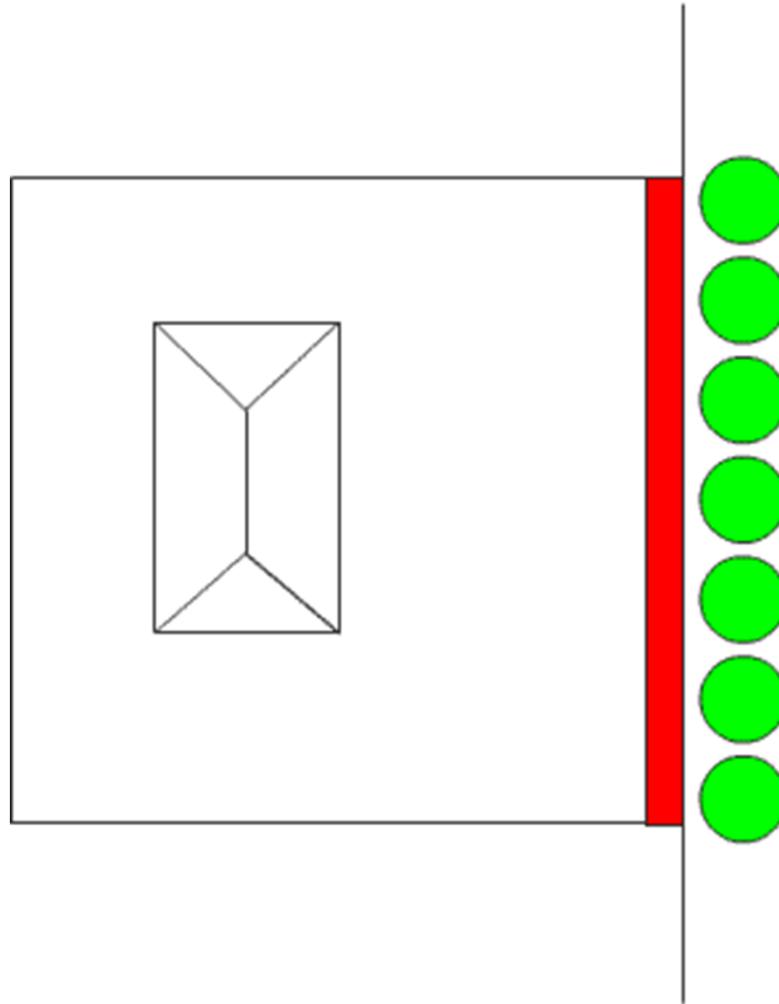
L'exemple du jardin

- Devant votre maison, vous avez un espace vert qui vous appartient mais qui n'est pas délimité.
- Cet espace vert se poursuit par un parc public.
- Les gens peuvent donc entrer dans votre jardin sans le savoir, et vous voulez remédier à ce désagrément.
- Le problème est que :
 - vous voulez délimiter votre jardin sans pour autant couper de l'espace vert public (pas de mur de 3 mètres de haut, ni de grillage électrifié !),
 - vous voulez que les gens, qui sont dans l'espace public, profitent de la profondeur donnée par votre jardin sans pour autant y pénétrer par inadvertance,
 - vous voulez profiter de la perspective du jardin public.
- Quelle solution ???

LA solution



LA solution



L'exemple du jardin

- Le pattern conçu par les architectes consiste à construire un muret à la frontière votre jardin et à planter une rangée d'arbres juste après ce muret, à la frontière avec l'espace public. Ce muret à une hauteur qui vous permet de vous asseoir dessus (environ 50 cm) et une largeur qui vous permet d'être assis de façon confortable.
 - Ce pattern est donc une **solution** élégante au problème, et il **peut être décliné d'un millier de façons différentes** : murs en briques, murs en béton, murs en bois, rangée de sapins, rangée de chênes, rangée de hêtres, ... Cette déclinaison à l'infini à partir d'une structure de réflexion se retrouve également dans le domaine de l'informatique. C'est ce qu'on appelle les **designs patterns**.

Le GoF, LA référence !



Design Patterns: Elements of Reusable Object-Oriented Software



Relié – 31 octobre 1994

Édition en Anglais | de [Erich Gamma \(Auteur\)](#), [Richard Helm \(Auteur\)](#), [Ralph Johnson \(Auteur\)](#), [John Vlissides \(Auteur\)](#)

4,7 ★★★★★ 2 371 évaluations

[Afficher tous les formats et éditions](#)

Capturing a wealth of experience about the design of object-oriented software, four top-notch designers present a catalog of simple and succinct solutions to commonly occurring design problems. Previously undocumented, these 23 patterns allow designers to create more flexible, elegant, and ultimately reusable designs without having to rediscover the design solutions themselves.

The authors begin by describing what patterns are and how they can help you design object-oriented software. They then go on to systematically name, explain, evaluate, and catalog recurring designs in object-oriented systems. With Design Patterns as your guide, you will learn how these important patterns fit into the software development process, and how you can leverage them to solve your own design problems most efficiently.

Each pattern describes the circumstances in which it is applicable, when it can be applied in view of other design constraints, and the consequences and trade-offs of using the pattern within a larger design. All patterns are compiled from real systems and are based on real-world examples. Each pattern also includes code that demonstrates how it may be implemented in object-oriented programming languages like C++ or Smalltalk.

 [Signaler un problème avec ce produit](#)

Un excellent ouvrage en français



Design patterns Broché – 22 septembre 2005

de Eric Freeman (Auteur), Elisabeth Freeman (Auteur), Kathy Sierra (Auteur), Bert Bates (Auteur), & 1 plus

5,0 ★★★★★ ▾ 9 évaluations

Afficher tous les formats et éditions

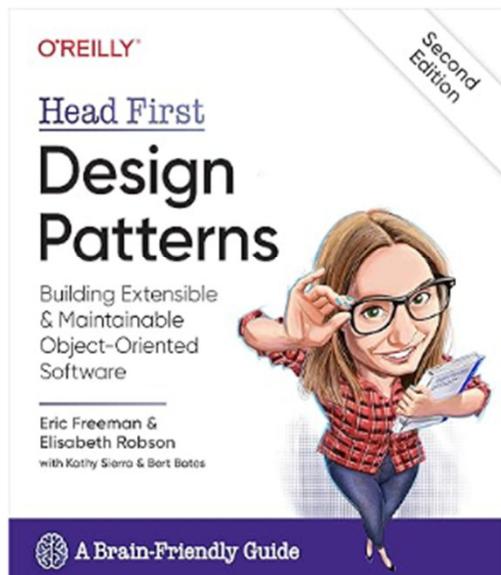


Si vous souhaitez vous distinguer des développeurs ordinaires et devenir une force de proposition pour les architectures d'applications, vous devez de maîtriser les design patterns. Pour cela, rien ne vaut l'expérience de ceux qui ont déjà été confrontés aux mêmes problèmes. Vous profiterez à la fois de leur savoir-faire et de ce qui se fait de mieux en matière de conception. Qui plus est, vous gagnerez un temps précieux que vous occuperez bien mieux à autre chose de plus complexe et de plus amusant, ce qui fera le bonheur de vos chefs. Le plus important demeure que vous ne voulez pas vous endormir en apprenant les design patterns. Si vous connaissez déjà la collection Tête la première, vous savez déjà à quoi vous en tenir. Voici les principaux bénéfices que vous tirerez de cet ouvrage : Distinguer les principaux patterns (Fabrique, Singleton, Adaptateur, Itérateur, Composite, etc.) ; Apprendre à les utiliser : quand et comment les appliquer dans vos conceptions, sans pour autant vous laisser gagner par la patternite ; Contrôler l'accès aux objets avec des proxies ; Maîtriser les mécanismes MVC et de son acolyte Web : MVC 2 ; Ne plus se sentir désœuvré face à des problèmes de conception en apprenant à réaliser des conceptions souples qui s'adaptent à vos besoins ; Arriver à reconnaître les design patterns qui vous entourent au quotidien dans les API Java, les frameworks ou autres applications, et comprendre enfin comment ils fonctionnent ; Réaliser des conceptions bien pensées et faciles à maintenir. Sous des dehors amusants et ludiques, cet

▼ En lire plus

Signaler un problème avec ce produit

Le même en anglais (l'original)



Head First Design Patterns: Building Extensible and Maintainable Object-Oriented Software Broché – 31 décembre 2020

Édition en Anglais | de Eric Freeman (Auteur), Elisabeth Robson (Auteur)

4,7 ★★★★★ 691 évaluations

Fait partie de : Head First (44 livres) [Afficher tous les formats et éditions](#)

You know you don't want to reinvent the wheel, so you look to design patterns—the lessons learned by those who've faced the same software design problems. With design patterns, you get to take advantage of the best practices and experience of others, so that you can spend your time on something more challenging. Something more fun. This book shows you the patterns that matter, when to use them and why, how to apply them to your own designs, and the object-oriented design principles on which patterns are based. Most importantly, you want to learn design patterns in a way that won't put you to sleep. If you've read a Head First book, you know what to expect—a visually rich format designed for the way your brain works. Newly updated and expanded, Head First Design Patterns, 2E, will load patterns into your brain in a way that sticks. In a way that makes you better at solving software design problems and better at speaking the language of patterns with others on your team. Join the hundreds of thousands of developers who have already improved their object-oriented design skills through Head First Design Patterns

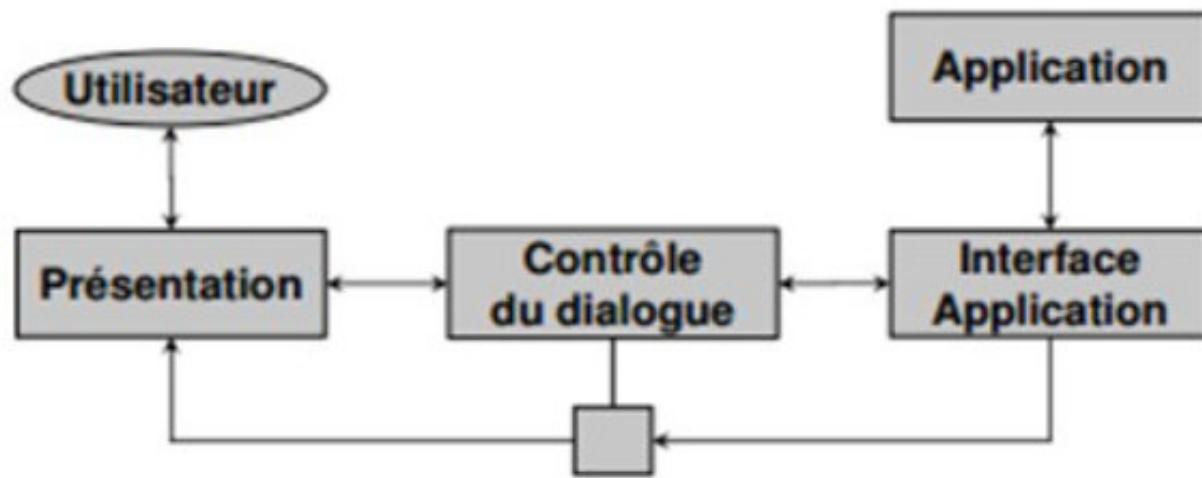
[Signaler un problème avec ce produit](#)

Retour au cours sur les Architectures...

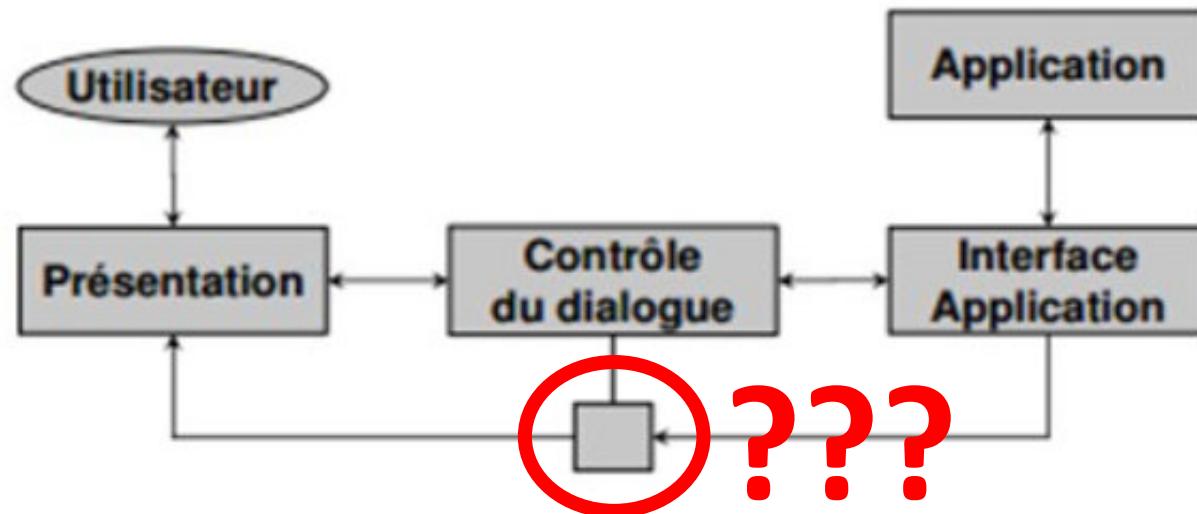
Un rapide historique

- Il était une fois...

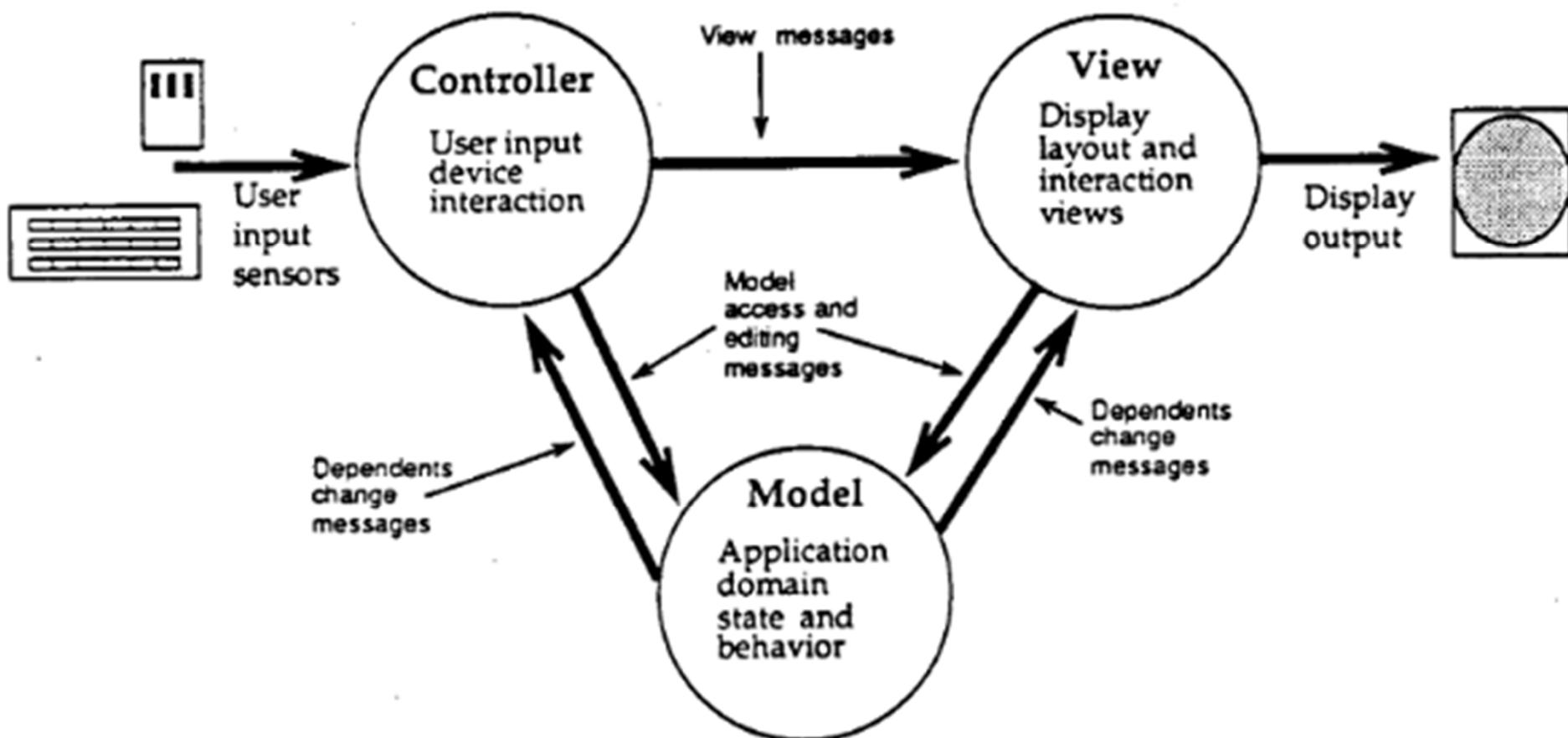
Modèle de Seeheim (1985)



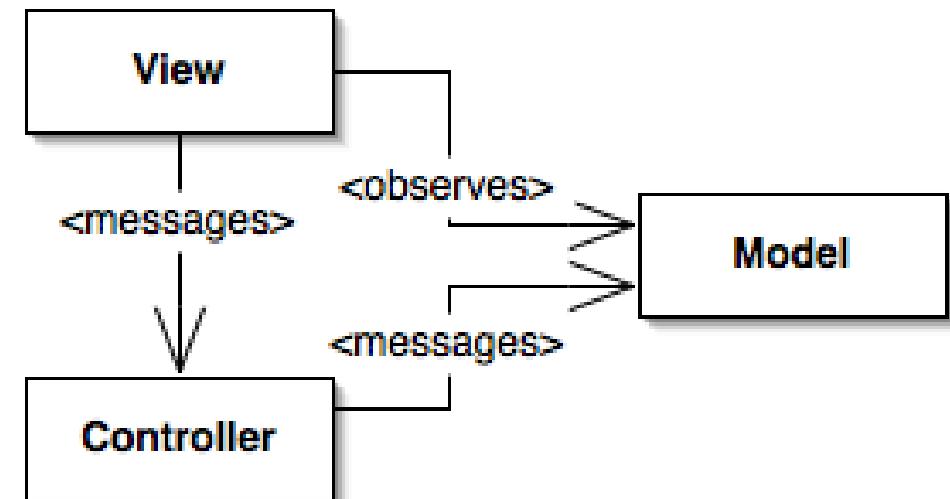
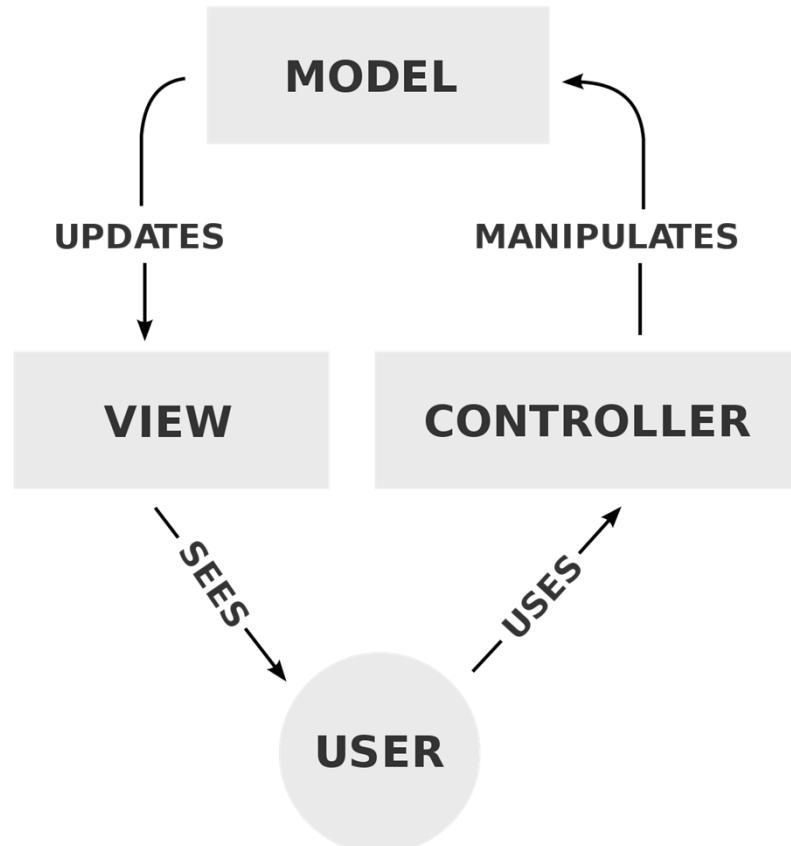
Modèle de Seeheim (1985)



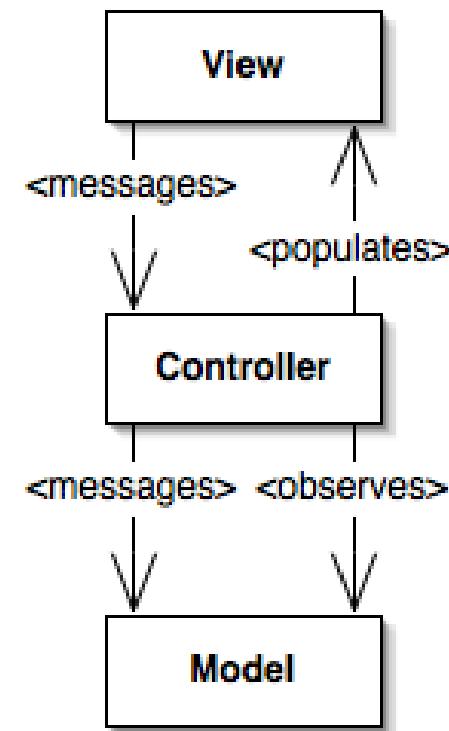
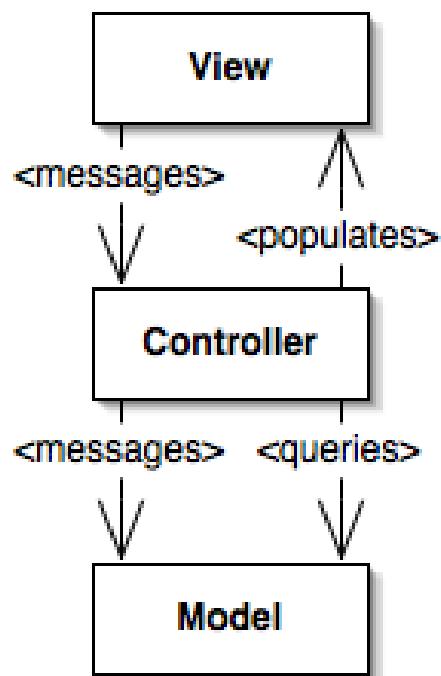
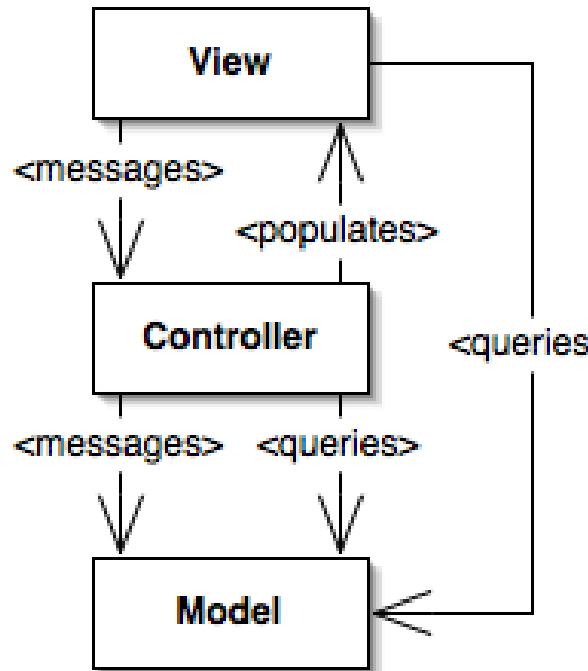
MVC Smalltalk 80 (1972)



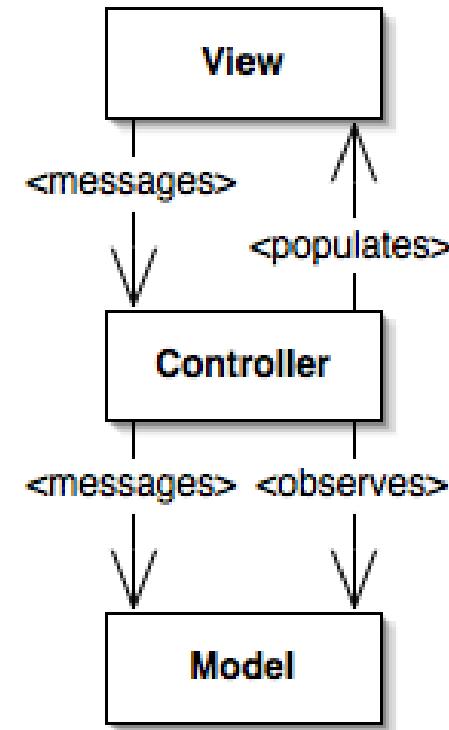
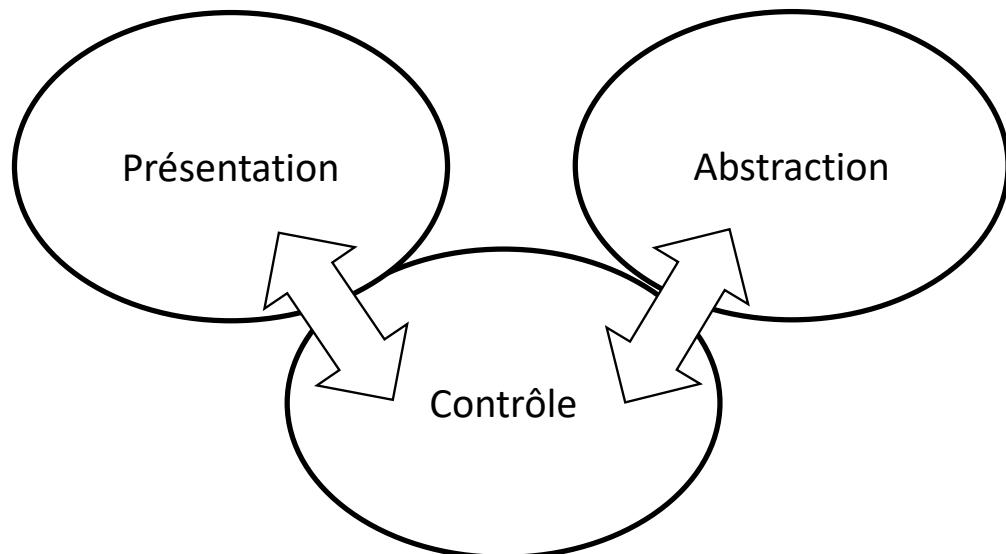
Evolutions du MVC durant les années 90 et 2000...



Encore des évolutions...pour en « revenir » à ...



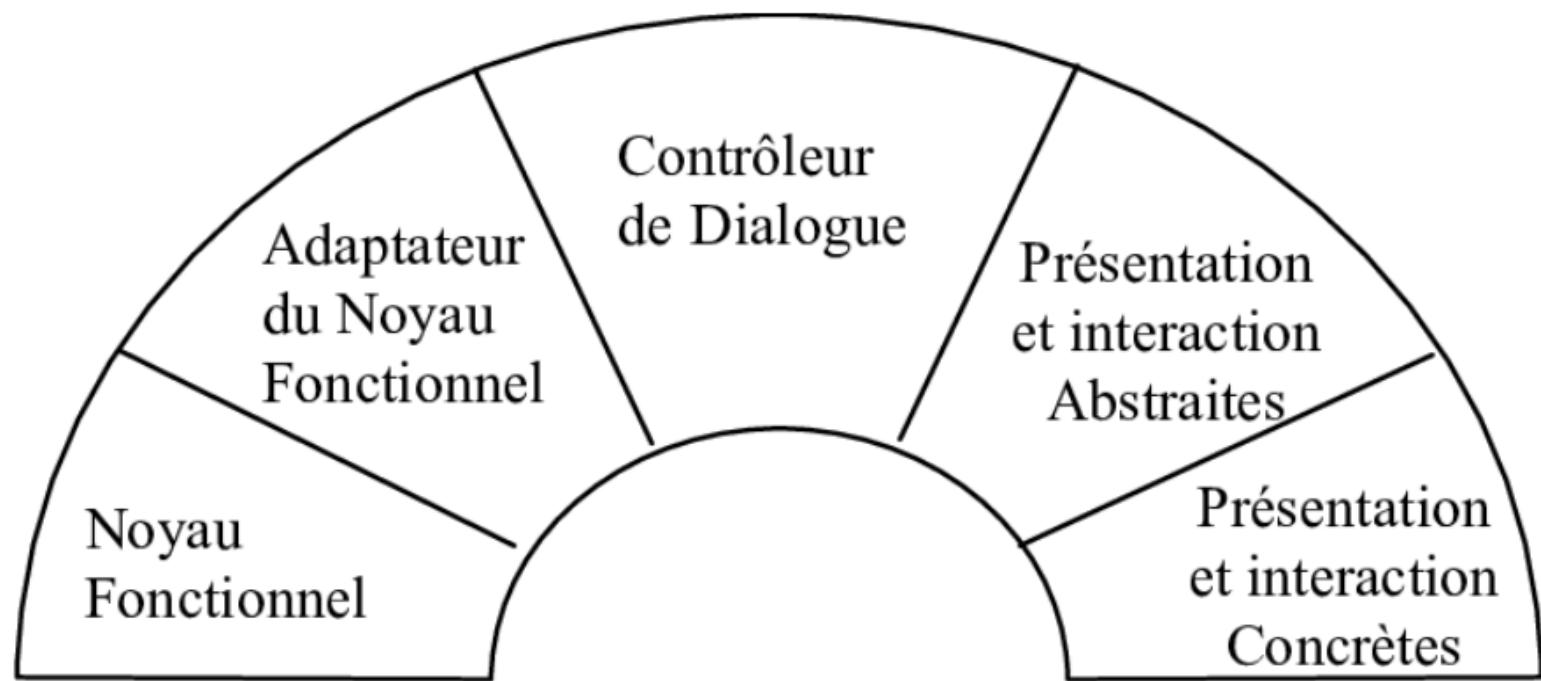
Le modèle PAC (1987)



Plein d'autres encore...

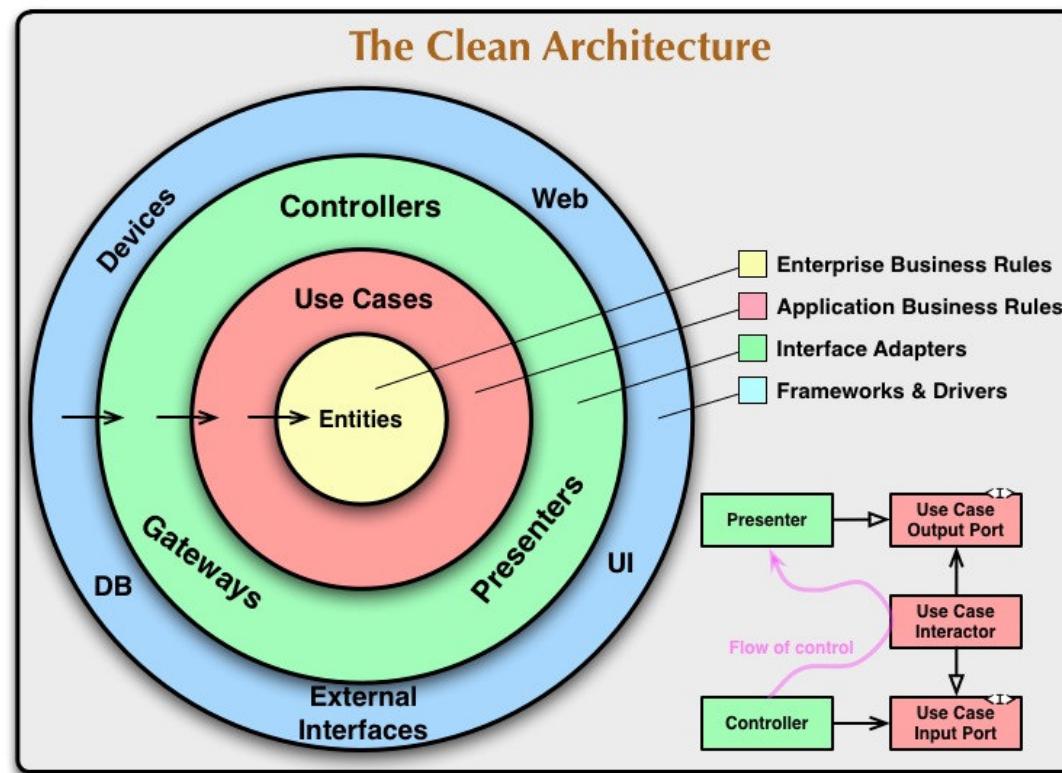
- PAC-Amodeus
- AMF
- MVC2
- M2VC
- ...

Le modèle Arch (1992)



Modèle Arch = Clean Architecture ?

- <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>



Parenthèse dans le cours...

- Toujours des nouveautés, veille techno permanente !
 - Icônes adaptatives (obligatoire)
 - https://developer.android.com/develop/ui/views/launch/icon_design_adaptive?hl=fr

The screenshot shows a web browser displaying the Android Developers website. The URL in the address bar is https://developer.android.com/develop/ui/views/launch/icon_design_adaptive?hl=fr. The page title is "Icônes adaptatives". The left sidebar has a section titled "Utiliser des icônes de lanceur adaptatives" which is currently selected. The main content area displays a grid of various adaptive icons, each consisting of a central Android robot head inside a shield-like shape, set against a background of different colors and patterns. A caption below the grid reads: "Une icône adaptative, ou `AdaptiveIconDrawable`, peut s'afficher différemment selon les fonctionnalités de l'appareil et le thème de l'utilisateur. Les icônes adaptatives sont principalement utilisées par le lanceur d'applications sur". The right sidebar contains links for "Concevoir des icônes adaptatives" and "Ajouter votre icône adaptative à votre application".

Parenthèse dans le cours...

- Toujours des nouveautés, veille techno permanente !
 - Pages mémoires de 16 Ko (obligatoire)
 - <https://android-developers.googleblog.com/2025/07/transition-to-16-kb-page-sizes-android-apps-games-android-studio.html>

The screenshot shows a blog post from the Android Developers Blog. The header includes the Android logo, the title 'Android Developers Blog', a search bar, and navigation links for Platform, Android Studio, Google Play, Jetpack, Kotlin, Docs, and News. Below the header, there's a share section with icons for LinkedIn, Twitter, Facebook, Email, and Copy link. The main content is a post titled 'Transition to using 16 KB page sizes for Android apps and games using Android Studio', dated 10 July 2025. The post text encourages upgrading app performance by embracing 16 KB memory page sizes. At the bottom, it mentions 'Android's transition to 16 KB Page size'.

Revenons à Android...

MVC - MVP - MVVM

- Qu'est-ce que c'est ?

MVC - MVP - MVVM

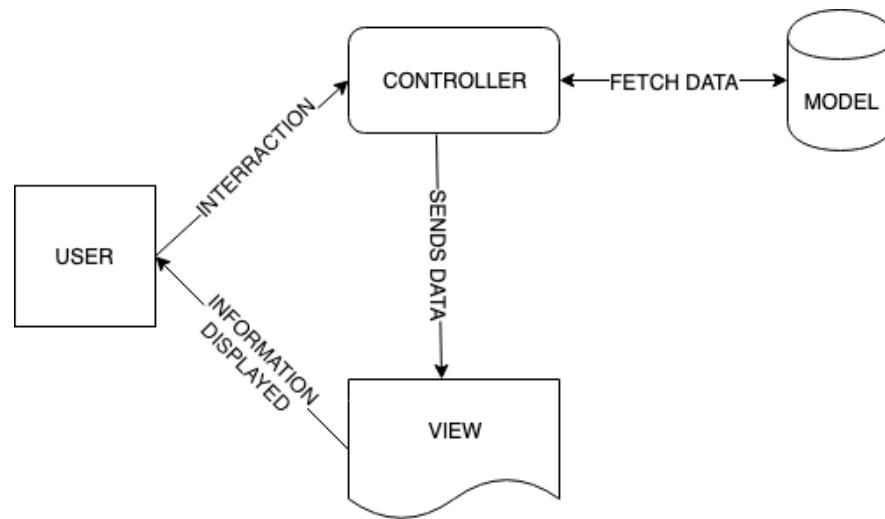
- Qu'est-ce que c'est ?
 - Model View Controller
 - Model View Presenter
 - Model View ViewModel
- Mais pas que (MVI...), vous pouvez lire cet article pour mieux comprendre :
 - <https://dev.to/adammc331/mvwtf-demystifying-architecture-patterns-ap1>

MVC

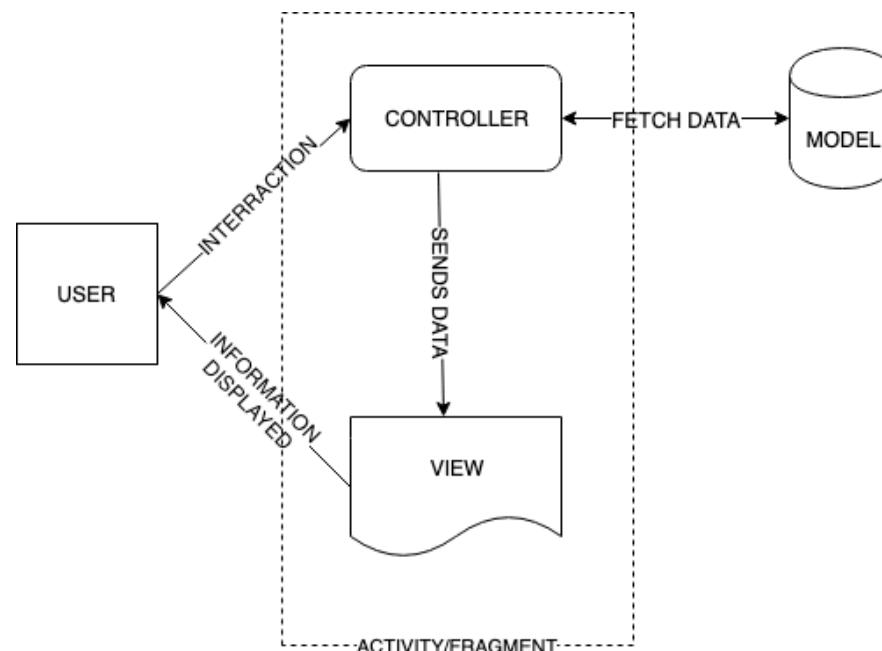
- Modèle
 - Pas connecté à la Vue et au Contrôleur
- Vue
 - Représentation du Modèle (son IHM)
 - Intercepte les interactions et les envoie au Contrôleur
 - Ne connaît pas le Modèle, son état et sa logique métier
- Contrôleur
 - La « colle » qui tient l'ensemble
 - Décide de ce qui se passe dans l'application
 - Gère la logique de l'application selon les interactions de la vue.
 - Fortement lié à la vue et au SDK Android

Très proche de PAC !

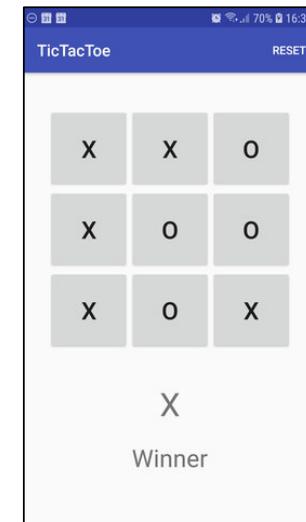
MVC et Android



MVC et Android

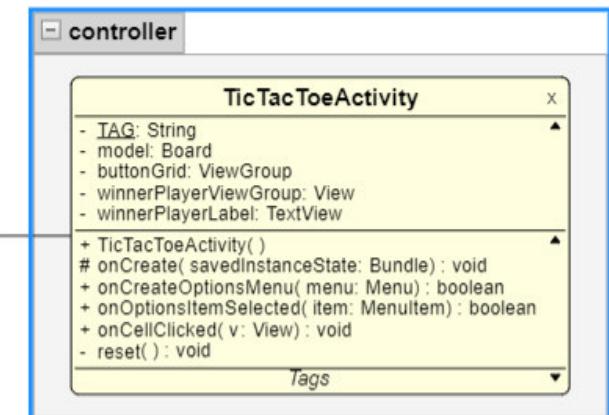
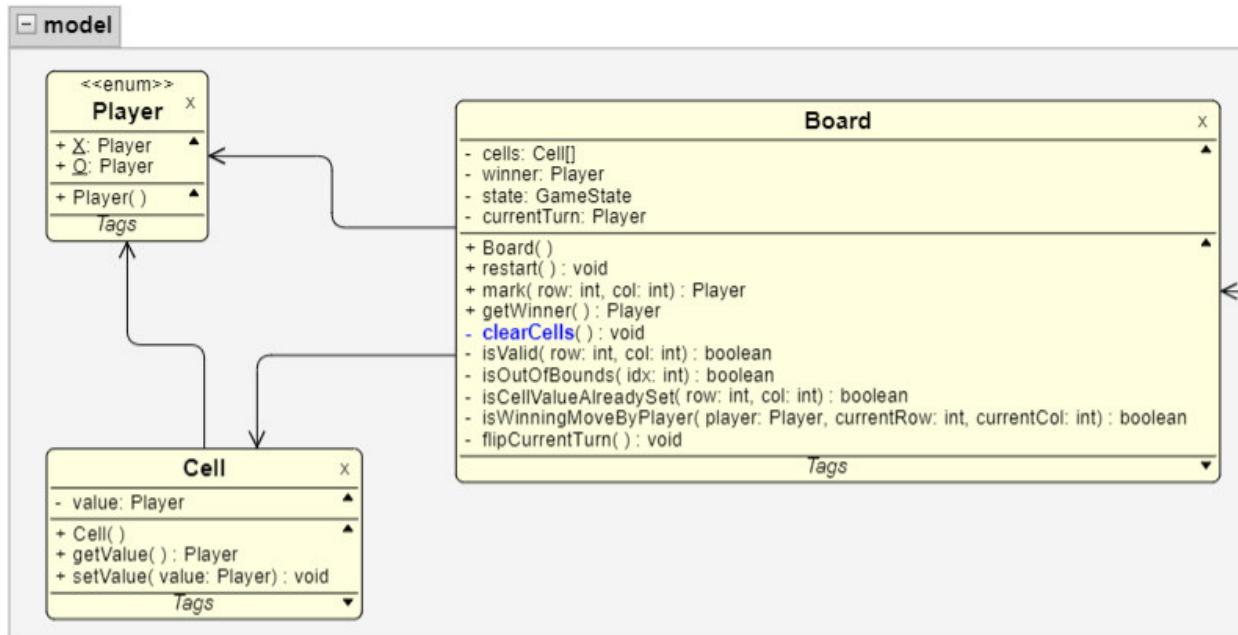


MVC : exemple du Tic Tac Toe

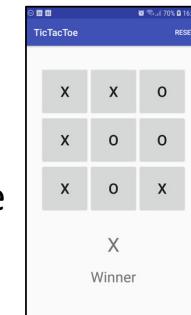


MVC

Diagrams de classe fait avec [VisUML](#)



+ Layout pour la Vue



MVC

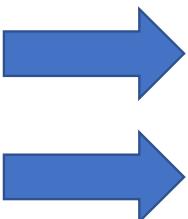
- À la création du Contrôleur (c'est-à-dire l'activité !), celui-ci se « connecte » au Modèle (qu'il crée !), et se connecte à la Vue (qu'il initialise !)

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.tictactoe);  
    winnerPlayerLabel = (TextView) findViewById(R.id.winnerPlayerLabel);  
    winnerPlayerViewGroup = findViewById(R.id.winnerPlayerViewGroup);  
    buttonGrid = (ViewGroup) findViewById(R.id.buttonGrid);  
  
    model = new Board();  
}
```

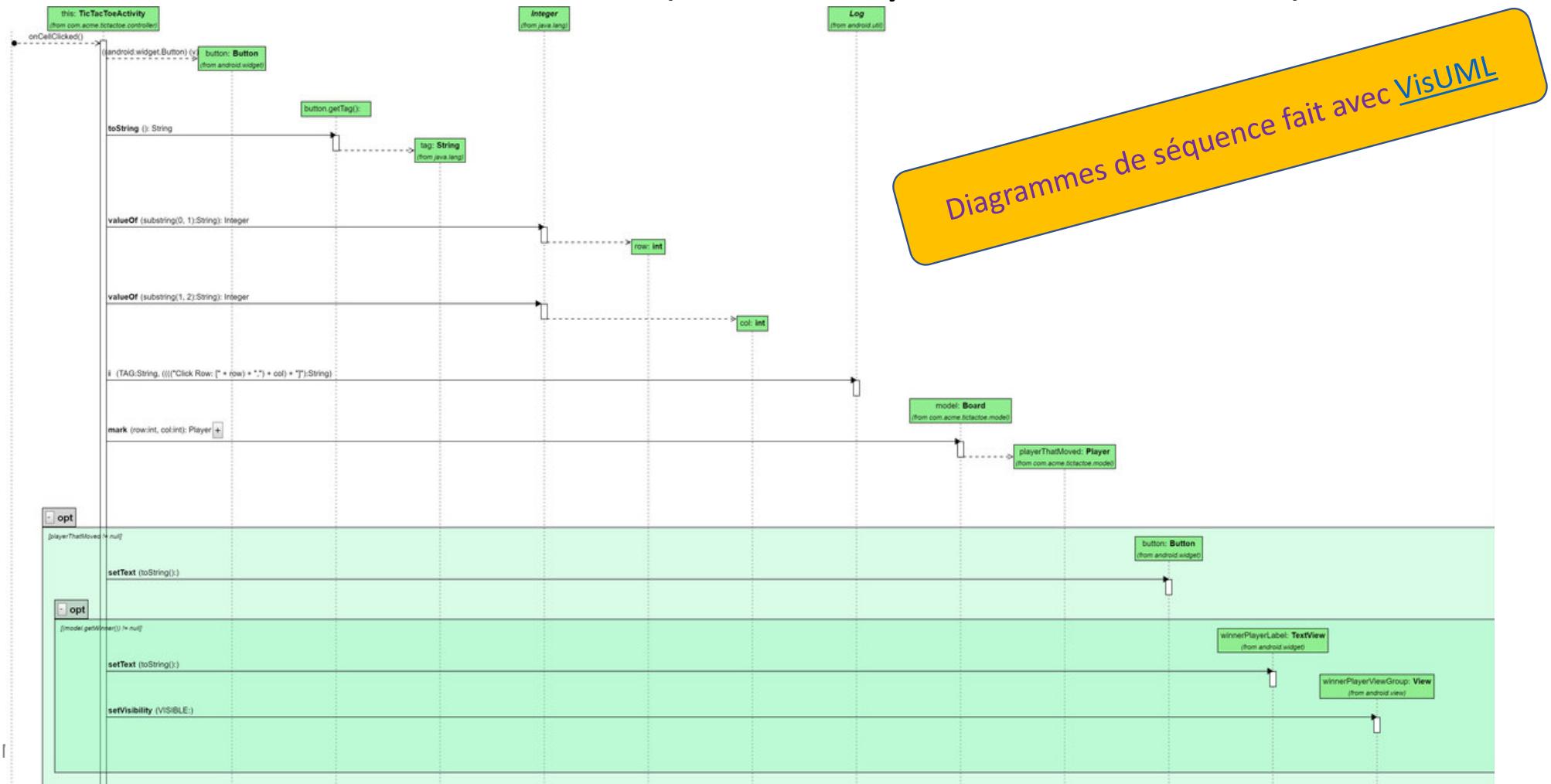
MVC : onCellClicked (Activity/Contrôleur)

- Quand l'utilisateur clique sur une cellule,
 - le contrôleur met à jour le modèle
 - Puis interroge le modèle pour connaître son état pour savoir par exemple si le joueur a gagné ou non, et met à jour la vue

```
public void onCellClicked(View v) {  
  
    Button button = (Button) v;  
  
    String tag = button.getTag().toString();  
    int row = Integer.valueOf(tag.substring(0,1));  
    int col = Integer.valueOf(tag.substring(1,2));  
    Log.i(TAG, "Click Row: [" + row + "," + col + "]");  
  
    Player playerThatMoved = model.mark(row, col);  
  
    if(playerThatMoved != null) {  
        button.setText(playerThatMoved.toString());  
        if (model.getWinner() != null) {  
            winnerPlayerLabel.setText(playerThatMoved.toString());  
            winnerPlayerViewGroup.setVisibility(View.VISIBLE);  
        }  
    }  
}
```



MVC : onCellClicked (Activity/Contrôleur)



MVC : Au final

- Bonne séparation, mais peut mieux faire car :
 - *Testabilité* : le Contrôleur est fortement basé sur l'API Android, donc difficile de le tester séparément.
 - *Modularity & Flexibility* : le Contrôleur est fortement couplé à la Vue (ou aux Vues). Si on change la Vue, on doit modifier le code du Contrôleur.
 - *Maintenance* : si les Modèles sont simples, les développeurs les intègrent souvent directement au Contrôleur 😞

MVP . . .



- The Model
 - It represents the layer, which **holds the business logic** as well as **controls how data is created, stored and modified**. *In Android, it is a data access layer, for example, database API or Remote server API.*
 - The Model consists of **components that are responsible for functionalities** like for generating, storing, exposing and fetching the data.
 - All these functionalities **usually perform in the background thread**. Because, these functionalities could be time-consuming, and they can potentially block the main thread UI.
- The View
 - It is a **passive interface**, which **displays data**, and the **routes user actions to Presenter**.
 - *In Android, the View is represented by Activity, Fragment or View.*
 - The View component contains a visual part of the application.
 - The View contains the UI and it does **not contain any logic or knowledge of the displayed data**.
- The Presenter
 - The Presenter is in **between Model and the View**. And, it **triggers the business logic**, and lets to know 'the View' when to **update**.
 - It **recovers data received from the Model and shows it in the View**.
 - It **interacts with the Model**, then fetches and **transforms the data from the Model to update the view**.

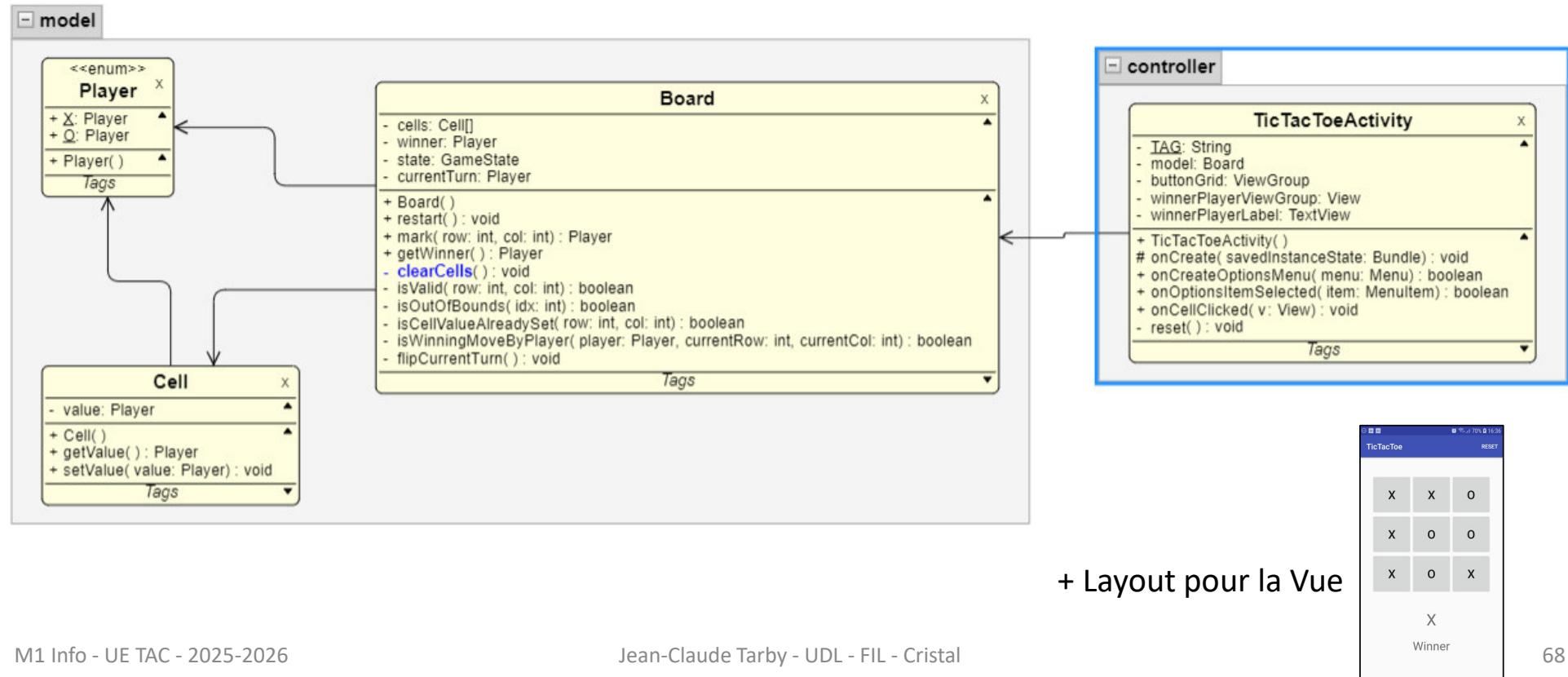
MVC (rappel du Tic Tac Toe)



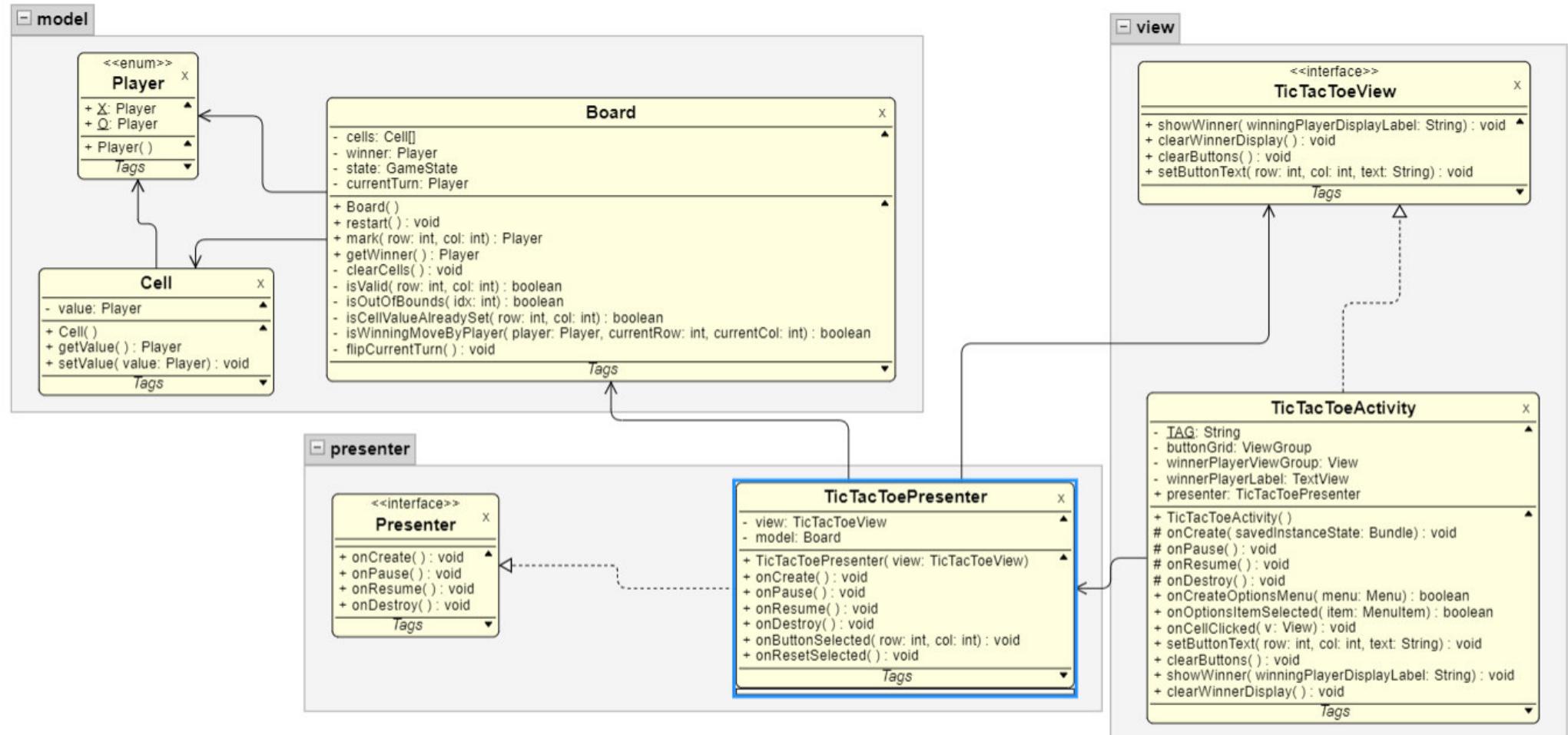
MVP



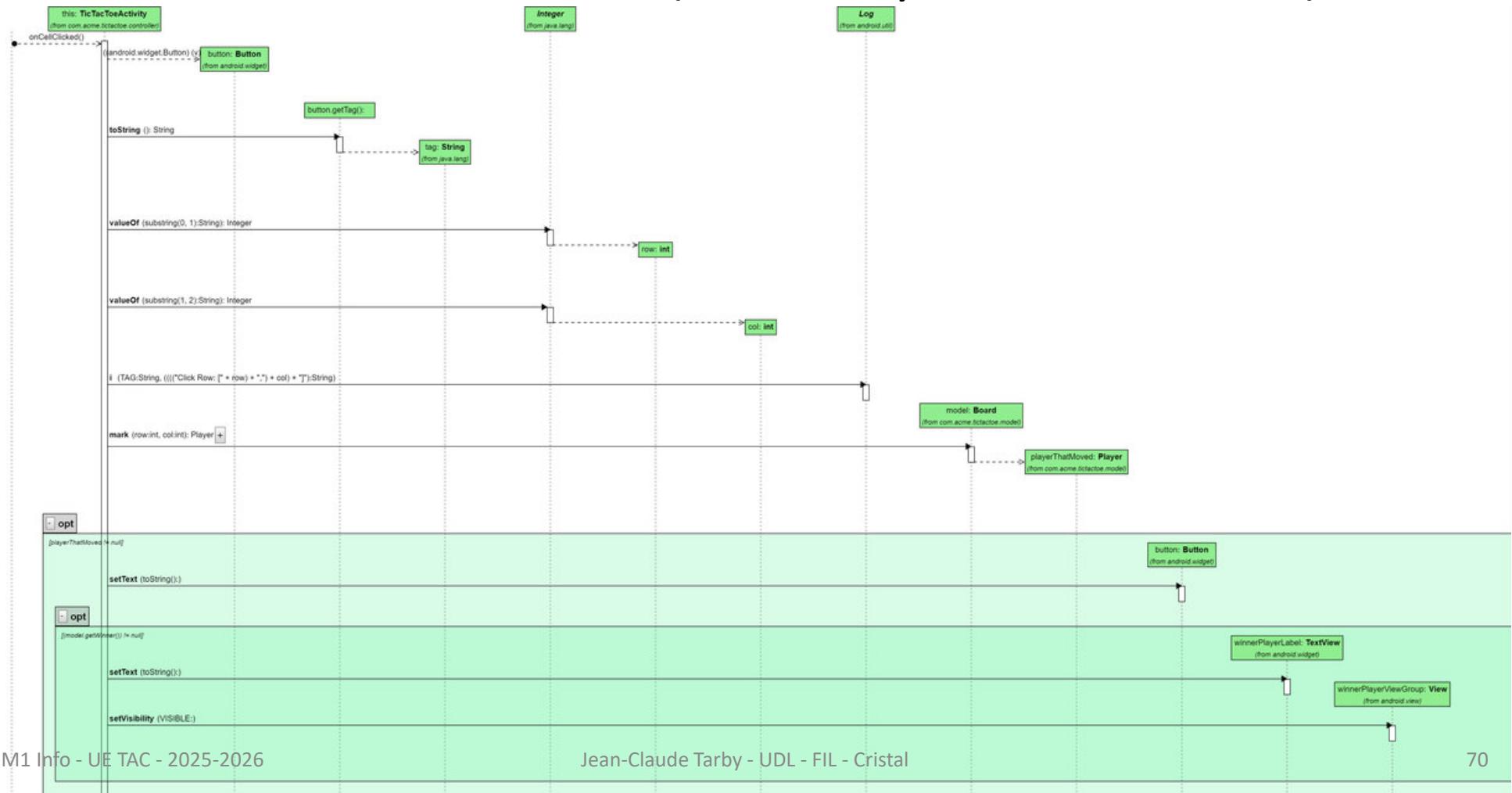
MVC (rappel)



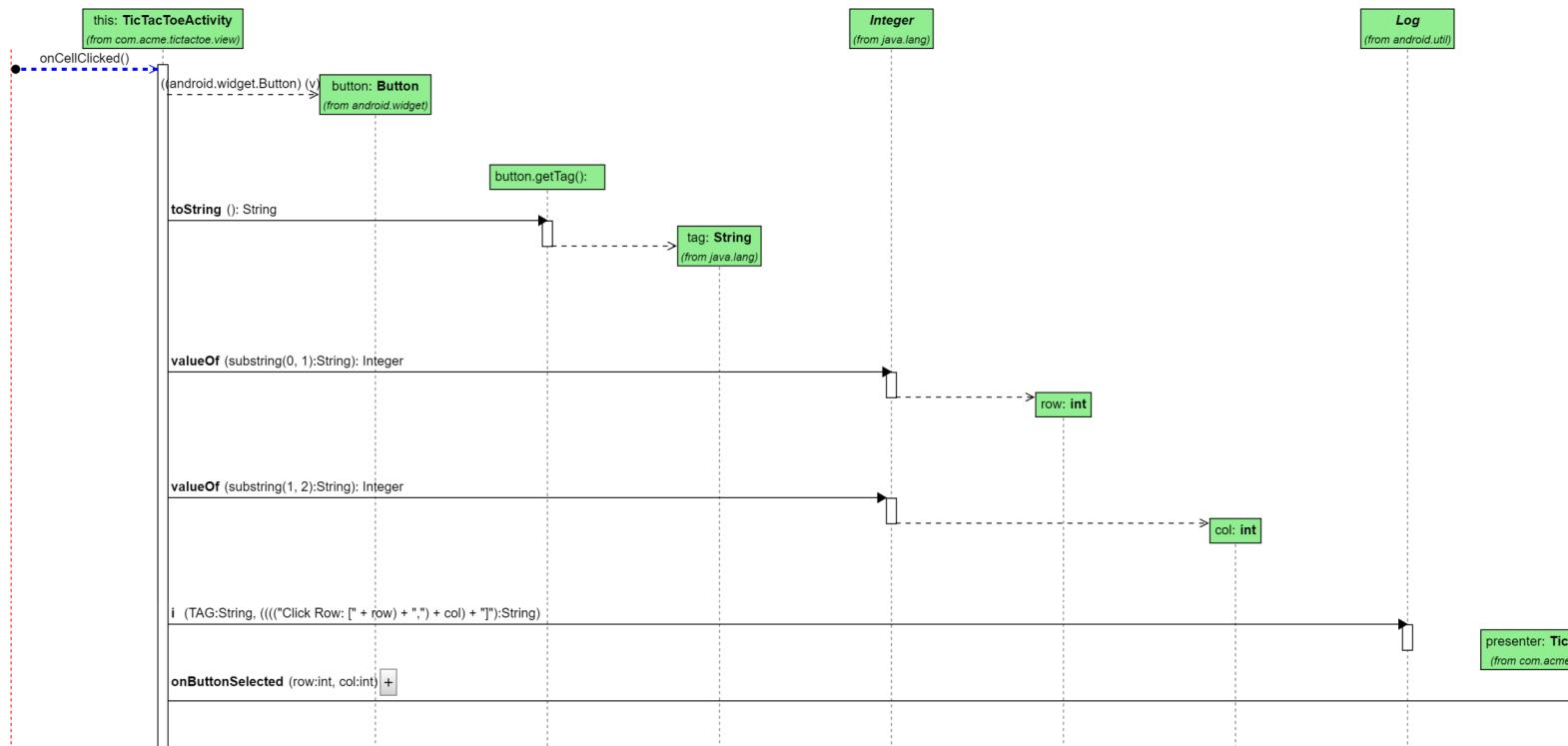
MVP



MVC : onCellClicked (Activity/Contrôleur)



MVP : onCellClicked (Activity/View)



+ simple que celle du MVC

Présentateur

- l'intention de chaque action est plus simple et plus claire.
- Plutôt que de dire à la Vue « comment afficher » quelque chose, il lui dit simplement « quoi afficher ».

MVP

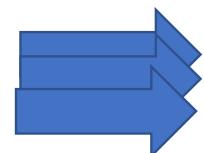
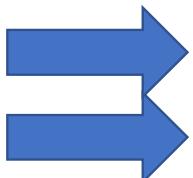
```
public class TicTacToePresenter implements Presenter {  
  
    private TicTacToeView view;  
    private Board model;  
  
    public TicTacToePresenter(TicTacToeView view) {  
        this.view = view;  
        this.model = new Board();  
    }  
  
    // Here we implement delegate methods for the standard Android Activity Lifecycle.  
    // These methods are defined in the Presenter interface that we are implementing.  
    public void onCreate() { model = new Board(); }  
    public void onPause() { }  
    public void onResume() { }  
    public void onDestroy() { }  
}
```

MVP

```
public class TicTacToePresenter implements Presenter {  
  
    private TicTacToeView view;  
    private Board model;  
  
    public TicTacToePresenter(TicTacToeView view) {  
        this.view = view;  
        this.model = new Board();  
    }  
  
    // Here we implement delegate methods for the standard Android Activity Lifecycle.  
    // These methods are defined in the Presenter interface that we are implementing.  
    public void onCreate() { model = new Board(); }  
    public void onPause() { }  
    public void onResume() { }  
    public void onDestroy() { }  
}
```



MVP



```
/**  
 * When the user selects a cell, our presenter only hears about  
 * what was (row, col) pressed, it's up to the view now to determine that from  
 * the Button that was pressed.  
 */  
public void onButtonSelected(int row, int col) {  
    Player playerThatMoved = model.mark(row, col);  
  
    if(playerThatMoved != null) {  
        view.setButtonText(row, col, playerThatMoved.toString());  
  
        if (model.getWinner() != null) {  
            view.showWinner(playerThatMoved.toString());  
        }  
    }  
}  
  
/**  
 * When we need to reset, we just dictate what to do.  
 */  
public void onResetSelected() {  
    view.clearWinnerDisplay();  
    view.clearButtons();  
    model.restart();  
}
```

MVP

- L'activité n'est pas liée au présentateur
 - l'activité implémente une interface.
 - pour tester les interactions avec la vue du présentateur, il suffit de créer une simulation basée sur cette interface

MVP : Au final

- Évaluation
 - Structure plus propre.
 - Facile de tester la logique du présentateur parce qu'elle n'est pas liée à des vues et APIs spécifiques à Android
 - Permet également de travailler avec n'importe quelle autre vue tant que la vue implémente l'interface TicTacToeView.
- Préoccupations du présentateur
 - *Maintenance* : les développeurs se retrouvent souvent avec de gros présentateurs encombrants et difficiles à maintenir.
→ MVVM peut aider à résoudre ce problème en faisant moins pour commencer.

MVC vs MVP

MVC Pattern	MVP Pattern
<ul style="list-style-type: none">The ‘Controller’ is based on behaviors. It can be shared across views.The ‘Controller’ can be responsible in order to determine which view to display first.High maintenance cost	<ul style="list-style-type: none">The ‘View’ is loosely coupled to the ‘Model’.The ‘Presenter’ is responsible for binding the ‘Model’ to the ‘View’.In MVP, it is much easier to do Unit testing.Low Maintenance cost

MVVM

- Modèle
 - Le même que pour MVC
- Vue
 - La vue est liée aux variables *observables* et aux actions exposées par le « ViewModel » de manière flexible.
- Vue-Modèle
 - Le ViewModel est chargé de ‘wrapper’ le modèle et de préparer les données *observables* nécessaires à la Vue. Il fournit également des ‘hooks’ pour que la vue puisse passer les événements au modèle.
 - Le ViewModel n'est cependant pas lié à la Vue.

MVVM ...



L'actuel en
Android → TAC

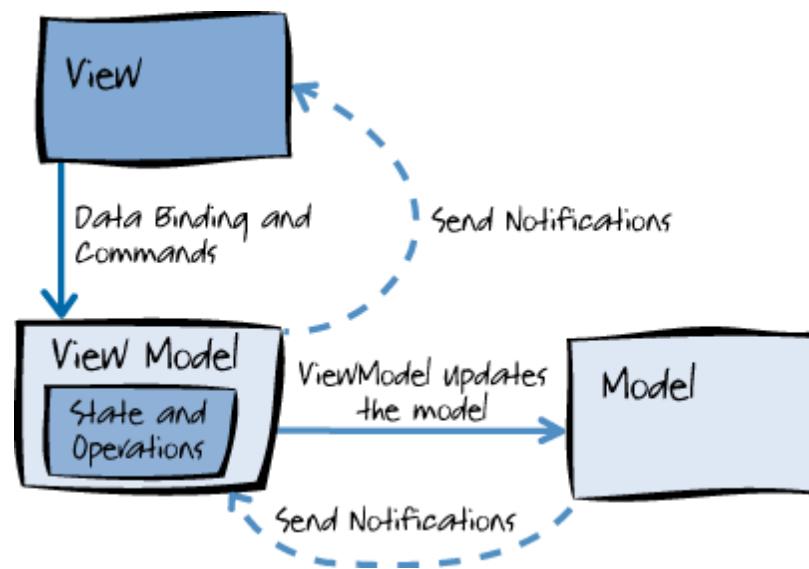
- Modèle
 - Le même que pour MVC
- Vue
 - La Vue est liée aux variables *observables* et aux actions exposées par le « ViewModel » de manière flexible.
- Vue-Modèle
 - Le ViewModel est chargé de ‘wrapper’ le modèle et de préparer les données *observables* nécessaires à la Vue. Il fournit également des ‘hooks’ pour que la vue puisse passer les événements au modèle.
 - Le ViewModel n'est cependant pas lié à la Vue.

MVVM

- Poussé fortement par Google, mais pas nouveau en soi !
 - Existe depuis longtemps dans .NET avec C# par exemple (basé sur le DataBinding)
 - Google : ~~DataBinding, puis LiveData... puis Flow (Kotlin)~~
 - ~~DataBinding, LiveData : on en parlera dans un prochain cours~~
 - ~~Choisir entre LiveData et Flow ? Une réponse à <https://medium.com/android-dev-hacks/exploring-livedata-and-kotlin-flow-7c8d8e706324#> (Exploring LiveData and Kotlin Flow)~~
 - [...] it's a continuous data flow between ViewModel and UI, so I prefer to use **Kotlin flow**.
 - ~~Kotlin flow primarily focuses on seamless data integration, so it makes sense. So when you're dealing with constant data flow which might include complex async operations try to do it with **Kotlin flow** and use **LiveData** for other things like service calls, state changes and stuff like that.~~
 - ViewModel : <https://developer.android.com/topic/libraries/architecture/viewmodel>
 - LiveData : <https://developer.android.com/topic/libraries/architecture/livedata>
 - DataBinding : <https://developer.android.com/topic/libraries/data-binding>
 - Flow : <https://developer.android.com/kotlin/flow>

MVVM Microsoft (2004 !)

- [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246\(v=pandp.10\)?redirectedfrom=MSDN](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/hh848246(v=pandp.10)?redirectedfrom=MSDN)



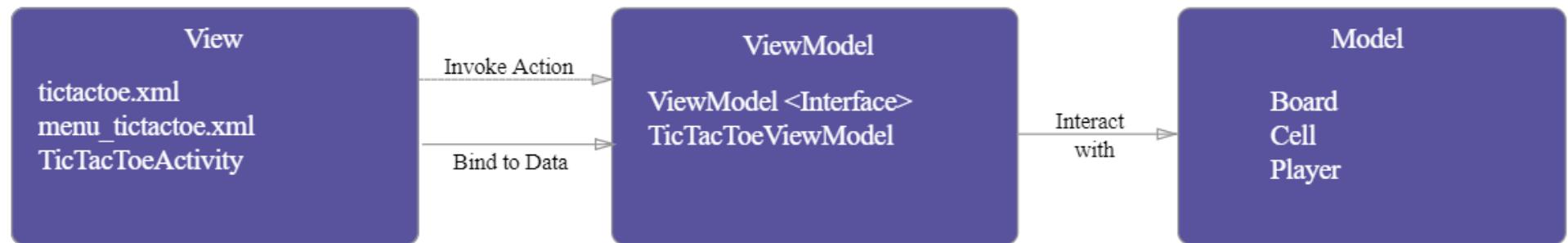
MVVM (rappel)

- Modèle
 - Le même que pour MVC
- Vue
 - La vue est liée aux variables *observables* et aux actions exposées par le « ViewModel » de manière flexible.
- Vue-Modèle
 - Le ViewModel est chargé de ‘wrapper’ le modèle et de préparer les données *observables* nécessaires à la vue. Il fournit également des ‘hooks’ pour que la vue puisse passer les événements au modèle.
 - Le ViewModel n'est cependant pas lié à la vue.

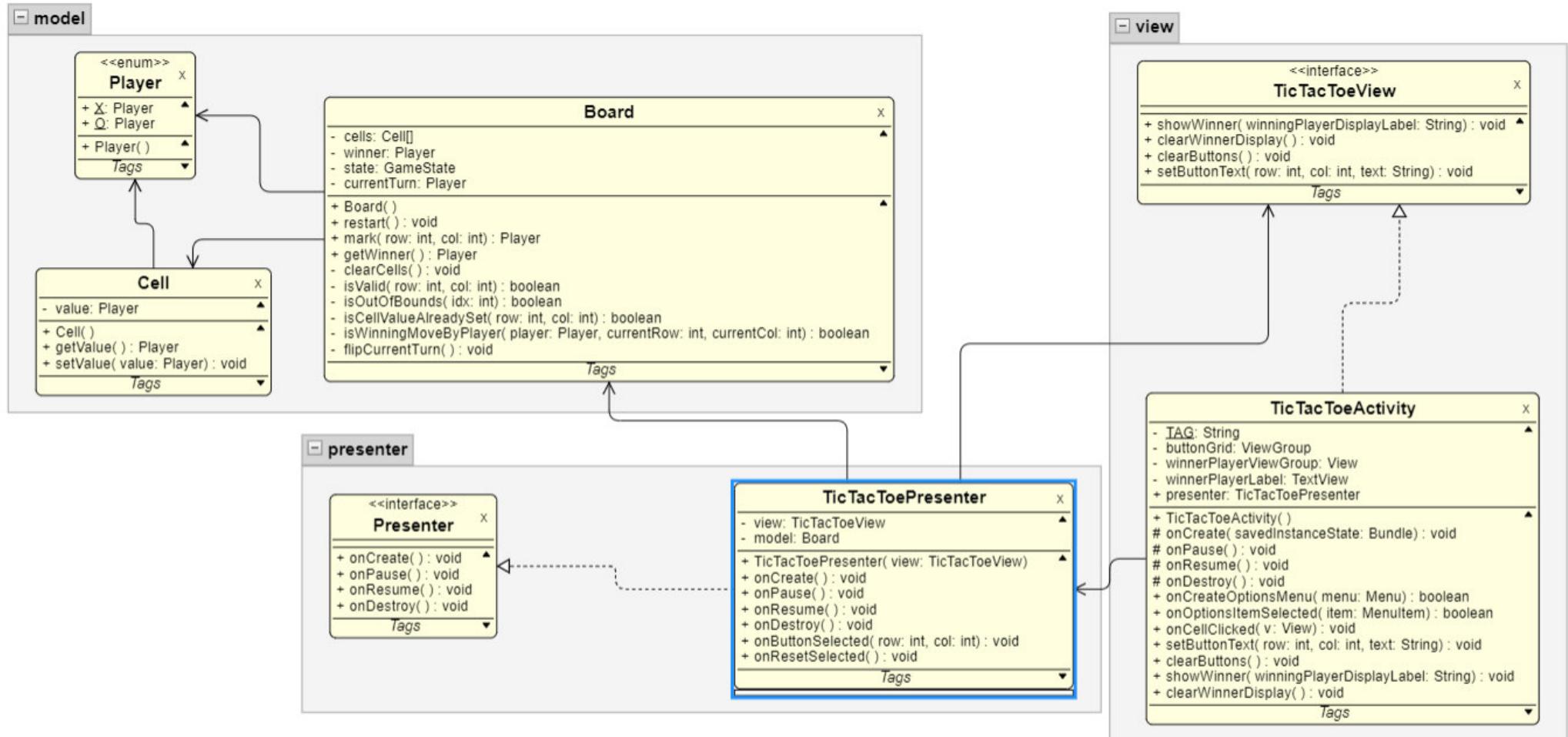
MVP (rappel exemple du Tic Tac Toe)



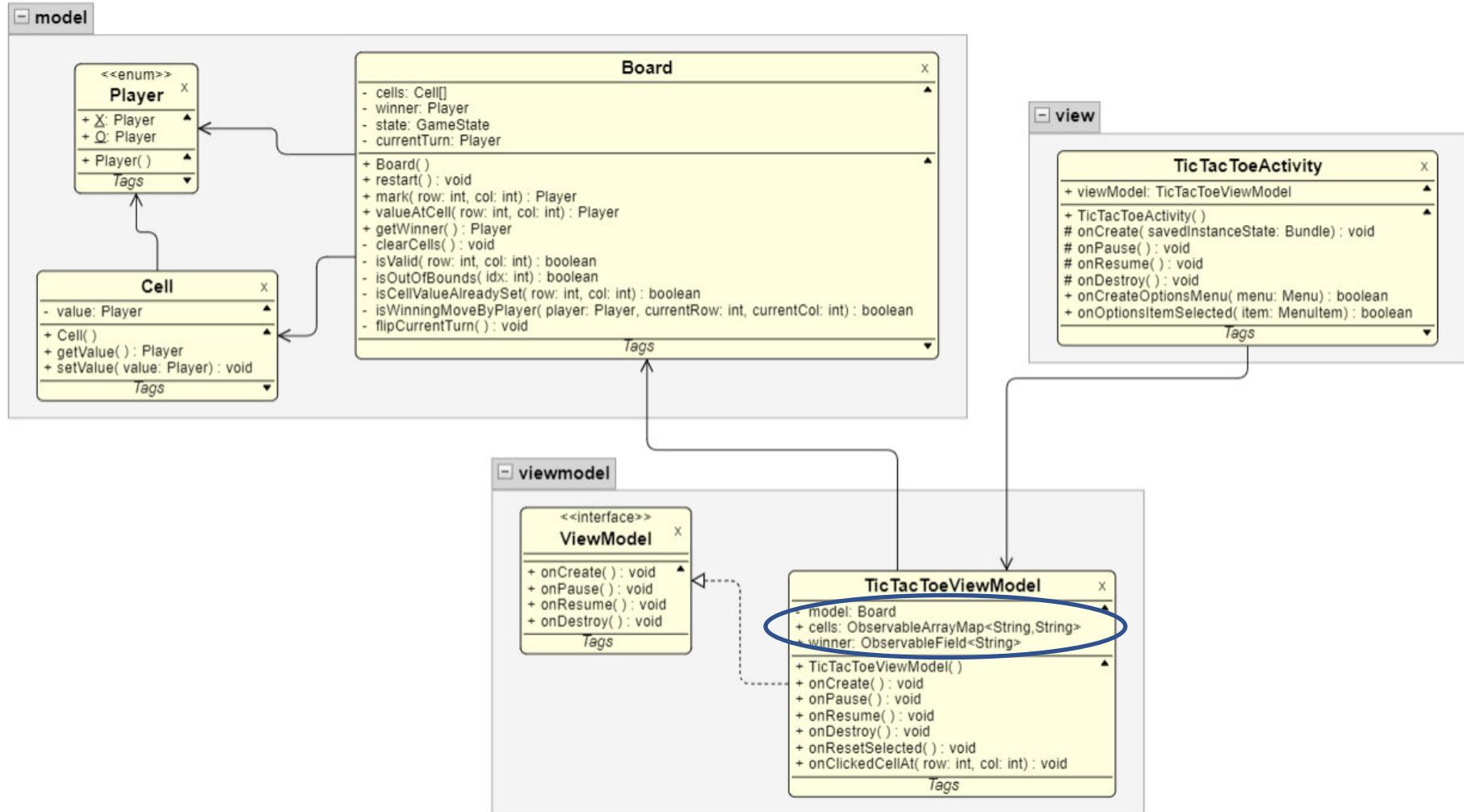
MVVM



MVP (rappel)



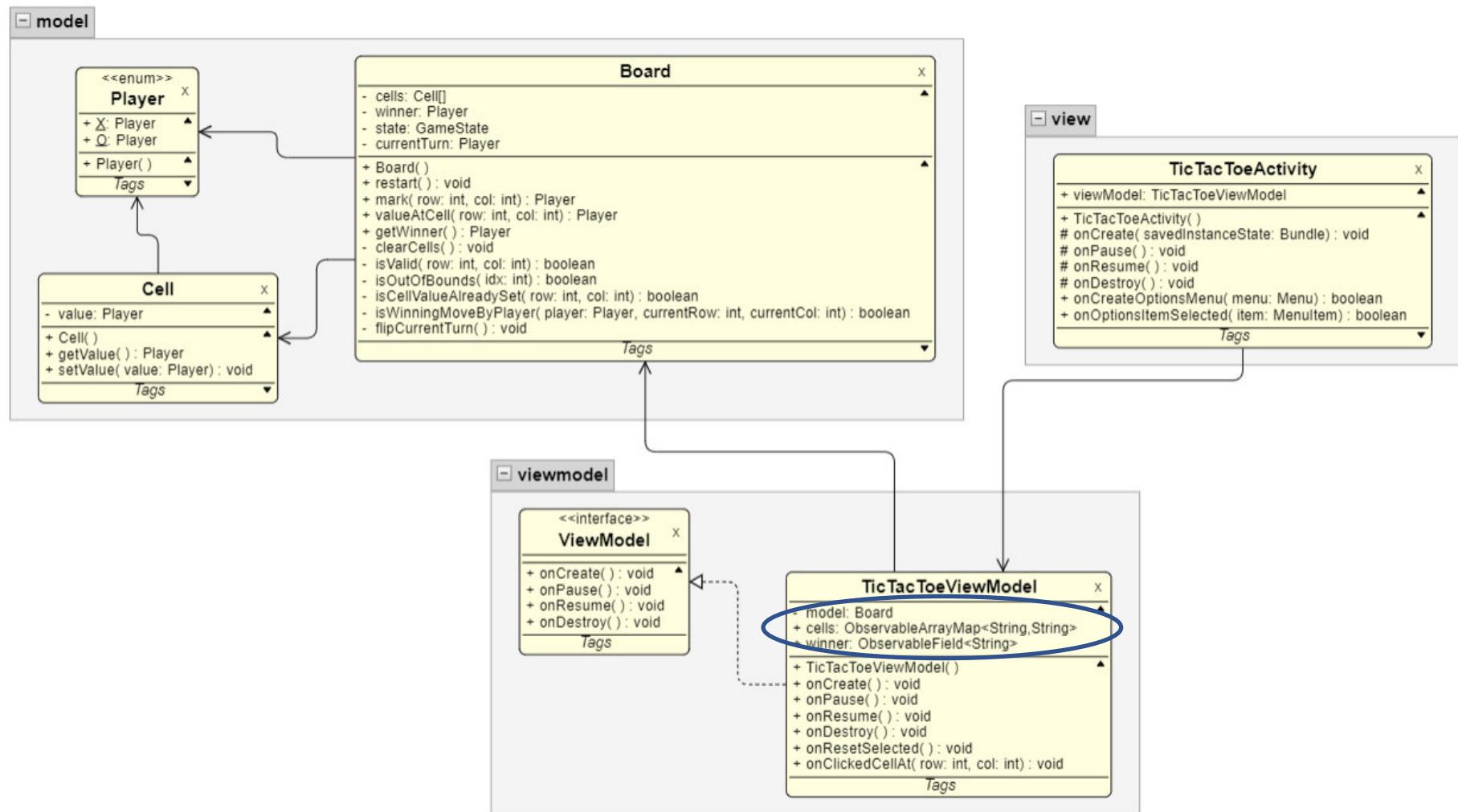
MVVM



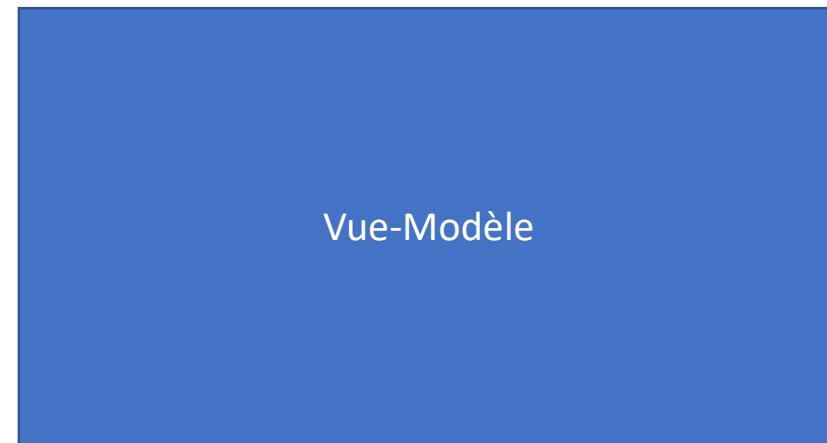
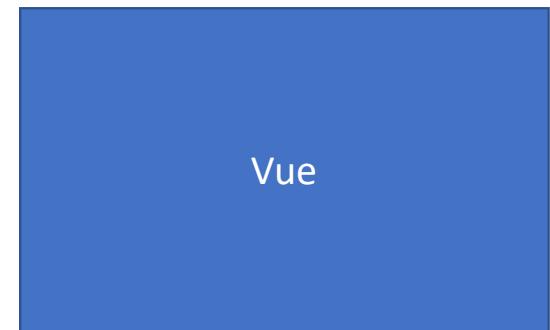
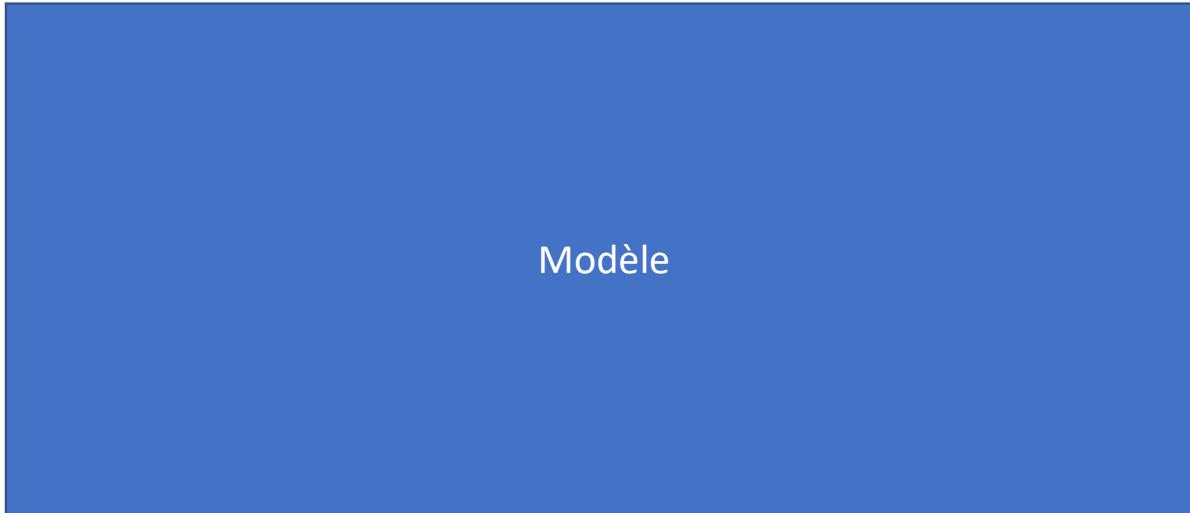
MVVM

- Ce n'est qu'un exemple d'archi...
- Ici, on a « directement » le modèle en lien avec la vue-modèle...
- Mais on peut avoir « autre chose »...
 - Observable
 - State
 - ...

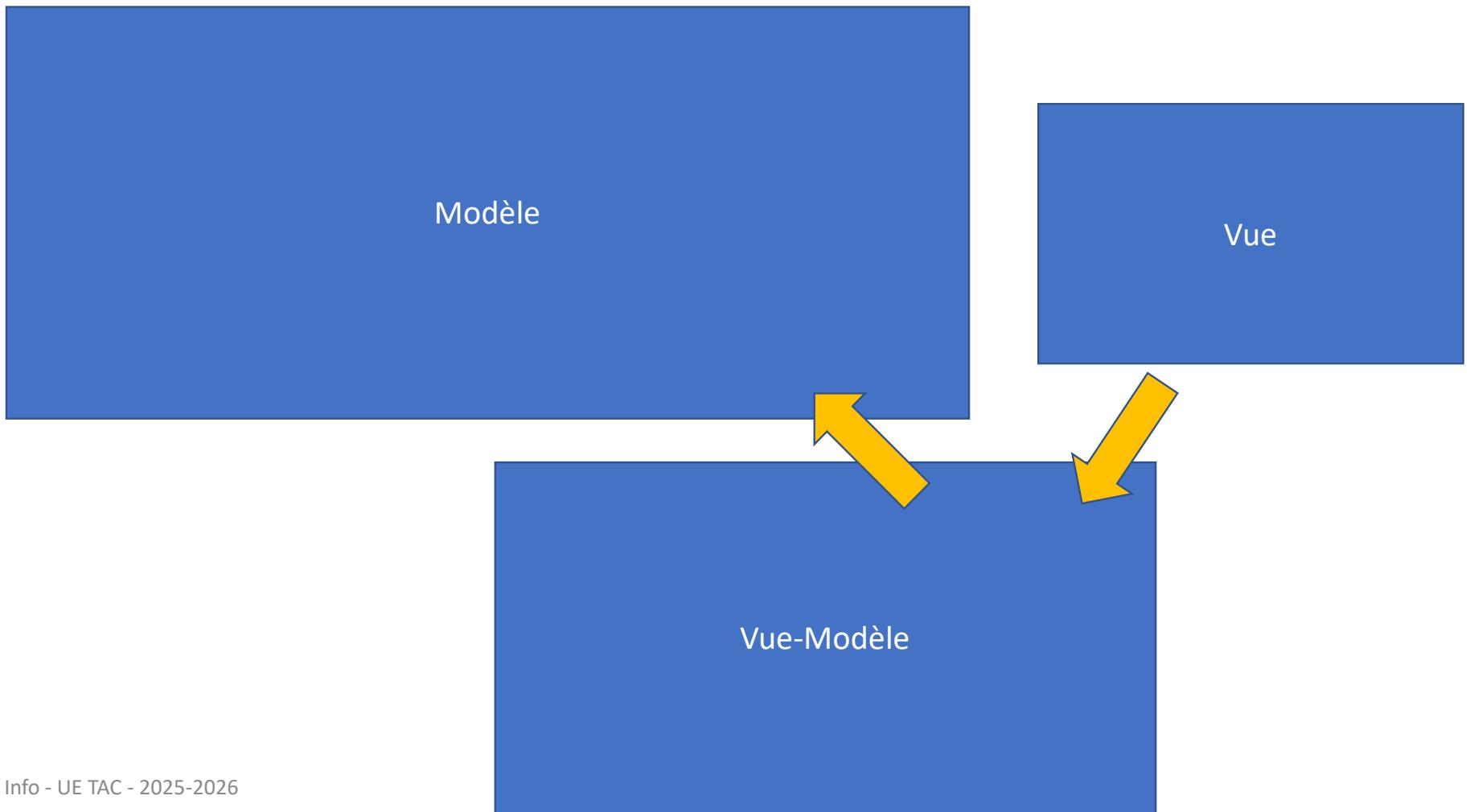
MVVM



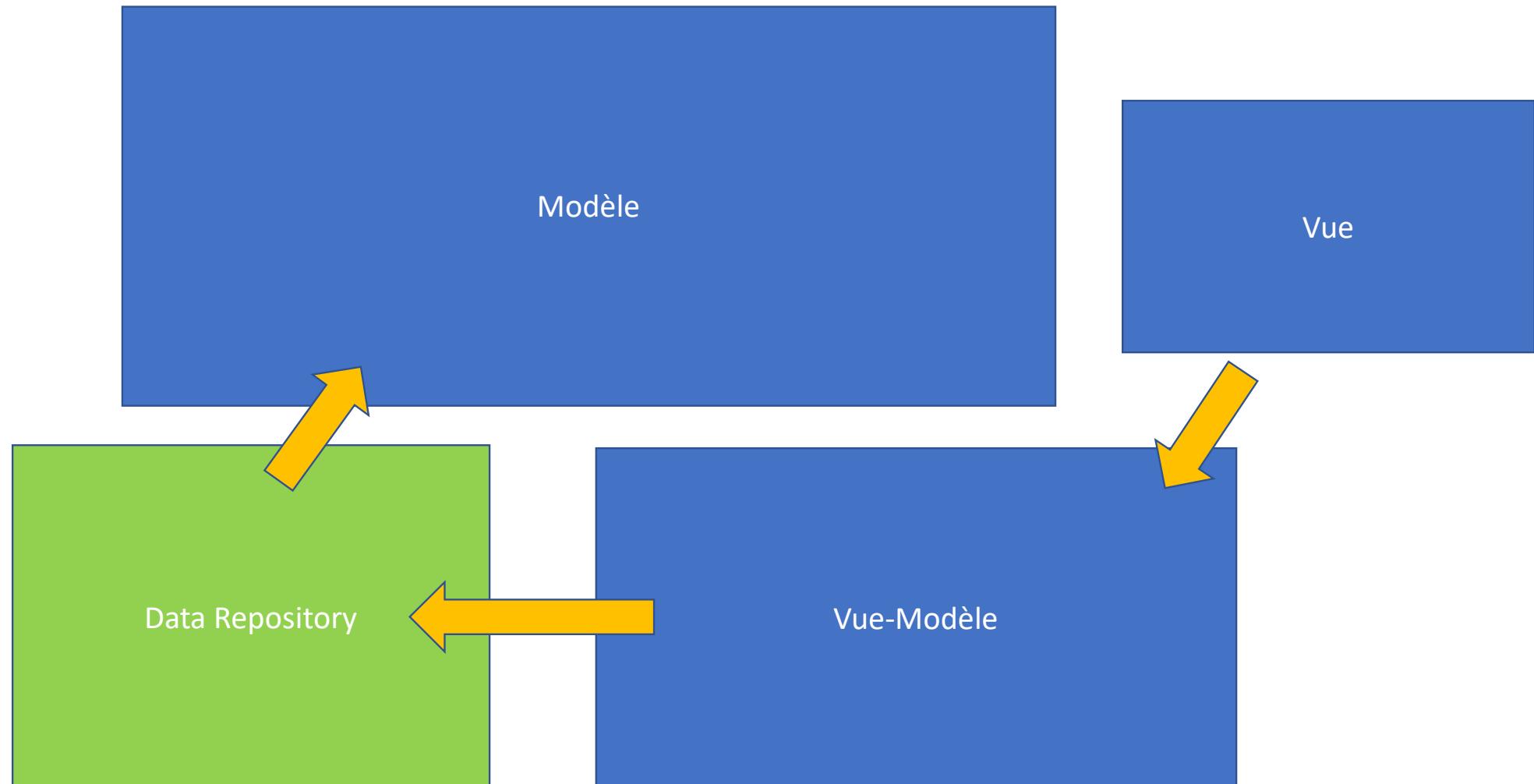
MVVM



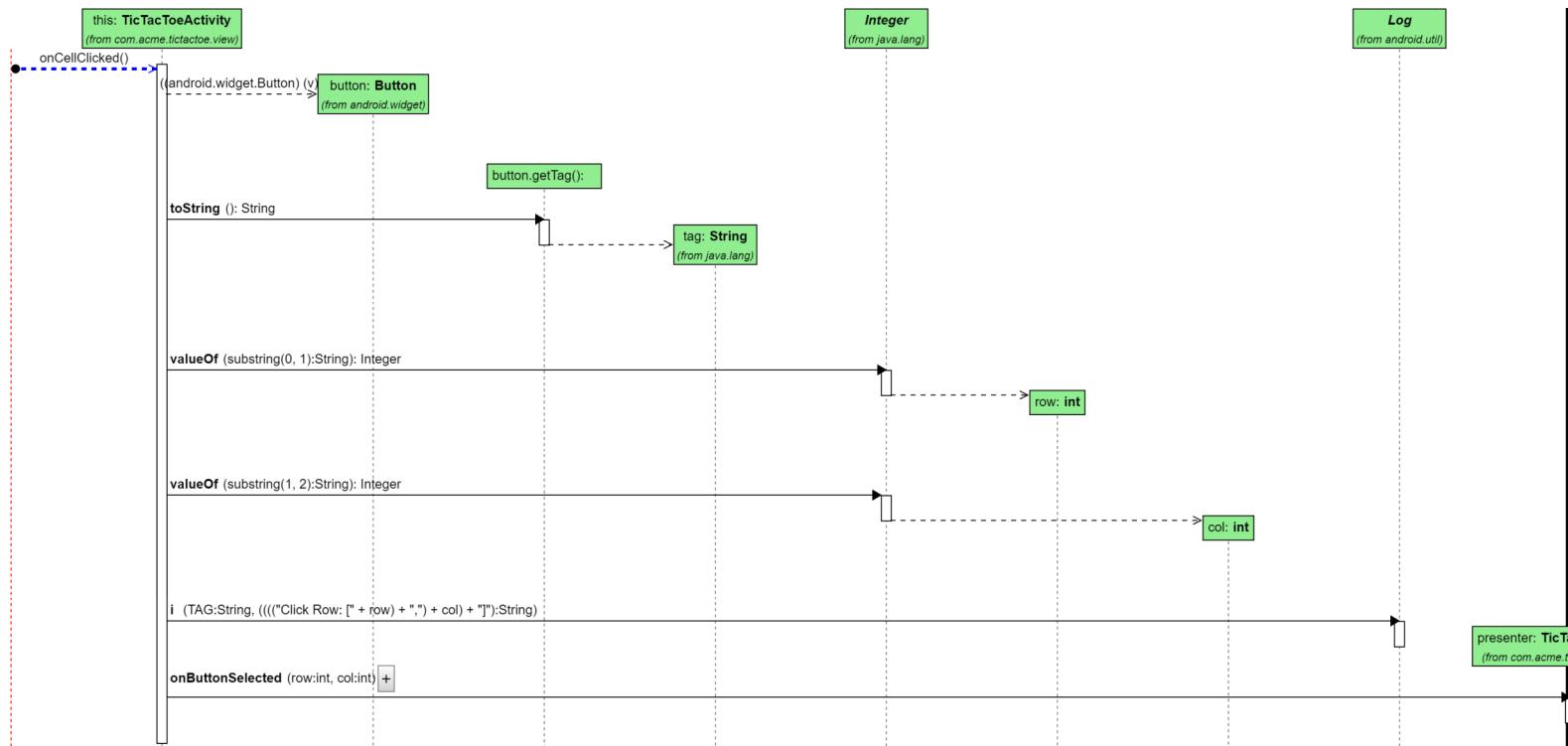
MVVM



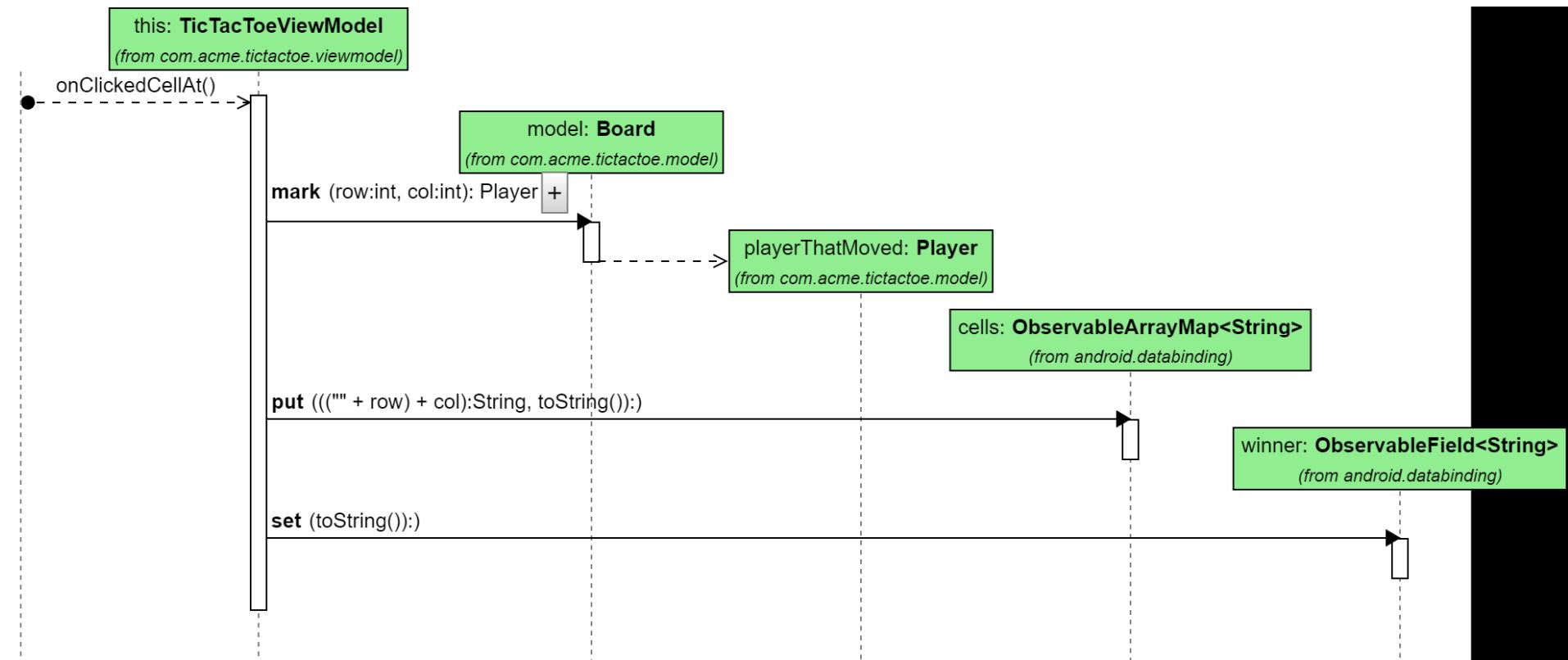
MVVM



MVP : onCellClicked (Activity/View)



MVVM : onClickedCellAt (ViewModel)



MVVM : onClickedCellAt (ViewModel)

```
public void onClickedCellAt(int row, int col) {  
    Player playerThatMoved = model.mark(row, col);  
    cells.put("" + row + col, playerThatMoved == null ? null : playerThatMoved.toString());  
    winner.set(model.getWinner() == null ? null : model.getWinner().toString());  
}
```

MVVM : Au final

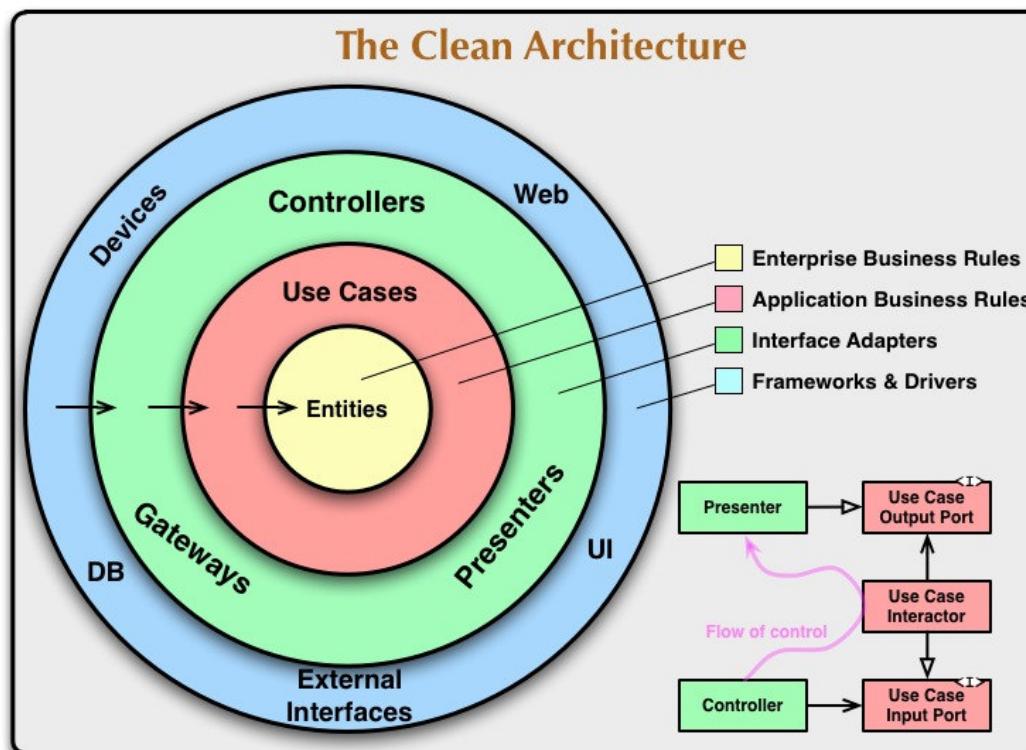
- Le test unitaire est encore plus facile (aucune dépendance à l'égard de la vue).
- Lors des tests, vérifier seulement que les variables observables sont correctement définies lorsque le modèle change. Il n'est pas nécessaire de maquetter la vue pour les tests comme c'était le cas avec le modèle MVP.
- Maintenance
 - Puisque les vues peuvent se lier à la fois aux variables et aux expressions, une logique de présentation étrangère peut s'infiltrer avec le temps, ajoutant ainsi du code à notre XML. Pour éviter cela, **obtenez toujours les valeurs directement à partir du ViewModel plutôt que d'essayer de les calculer ou de les dériver dans l'expression de liaison des vues**. De cette façon, le calcul peut être testé à l'unité de façon appropriée.

Conclusion

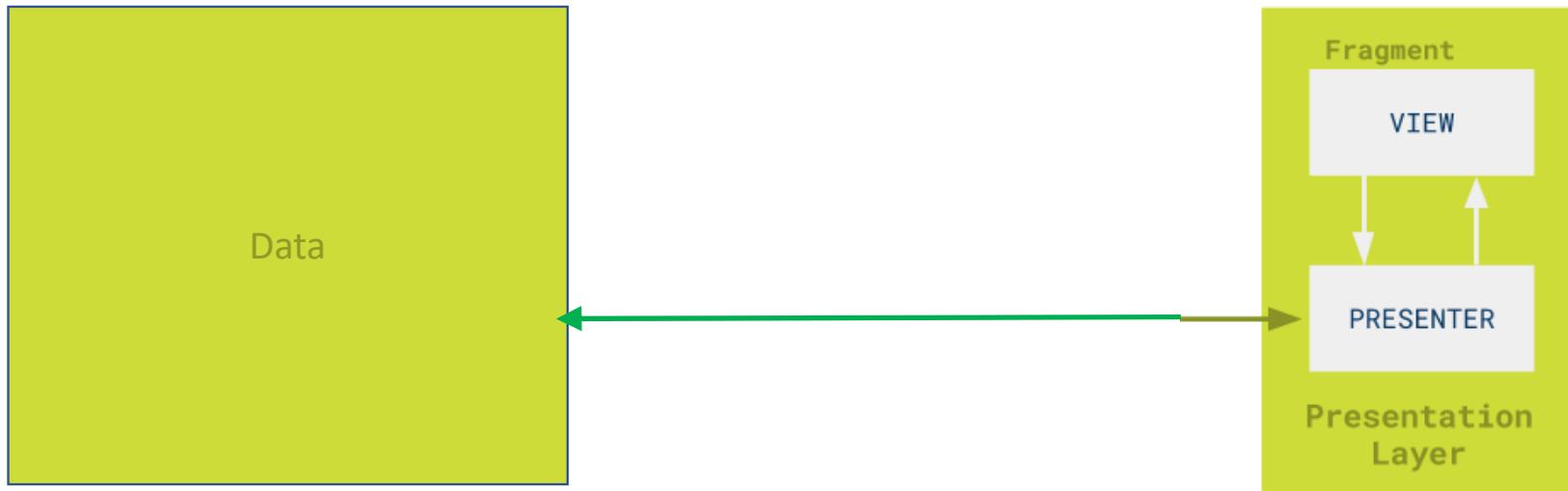
- MVP et MVVM font un **meilleur travail** que MVC en décomposant votre application en composants modulaires à usage unique, mais ils **ajoutent aussi plus de complexité** à votre application.
- Pour une **application** très simple avec seulement **un ou deux écrans**, **MVC peut très bien fonctionner**.
- Le **MVVM avec liaison de données** est attrayant car il suit un modèle de programmation plus réactif et **produit moins de code**.
- Alors, quel modèle vous convient le mieux ? Si vous choisissez entre MVP et MVVM, une grande partie de la décision revient à votre préférence personnelle, mais les voir en action vous aidera à comprendre les avantages et les compromis.
- **RAPPEL : projet TAC = MVVM !**

Architecture ? Clean Architecture ? Clean Code ?

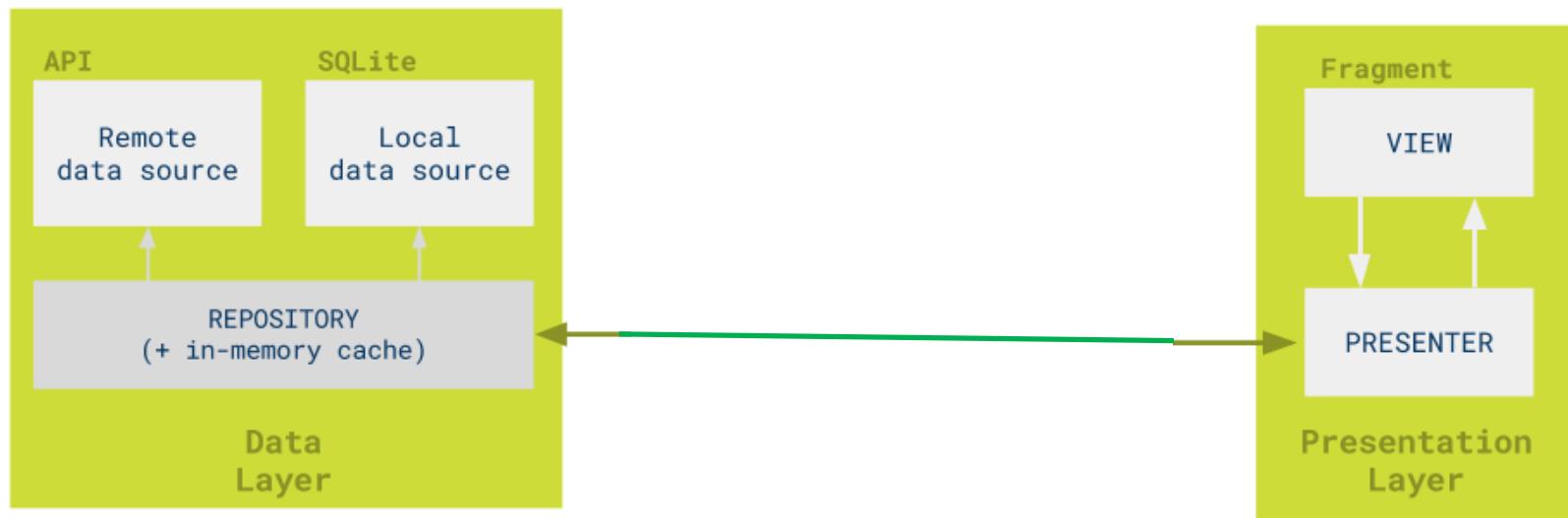
2014, Google parle de Clean Architecture



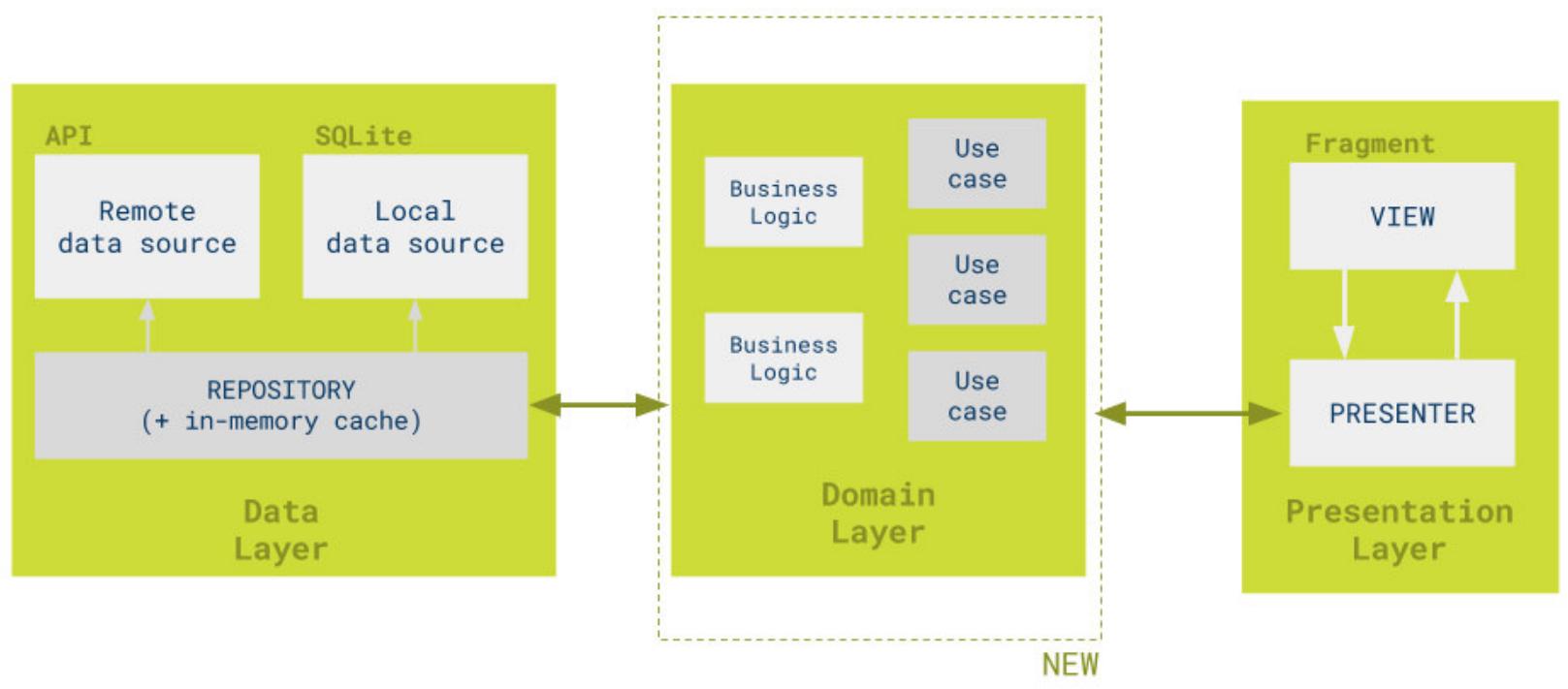
En 2016, Google « poussait » le MVP



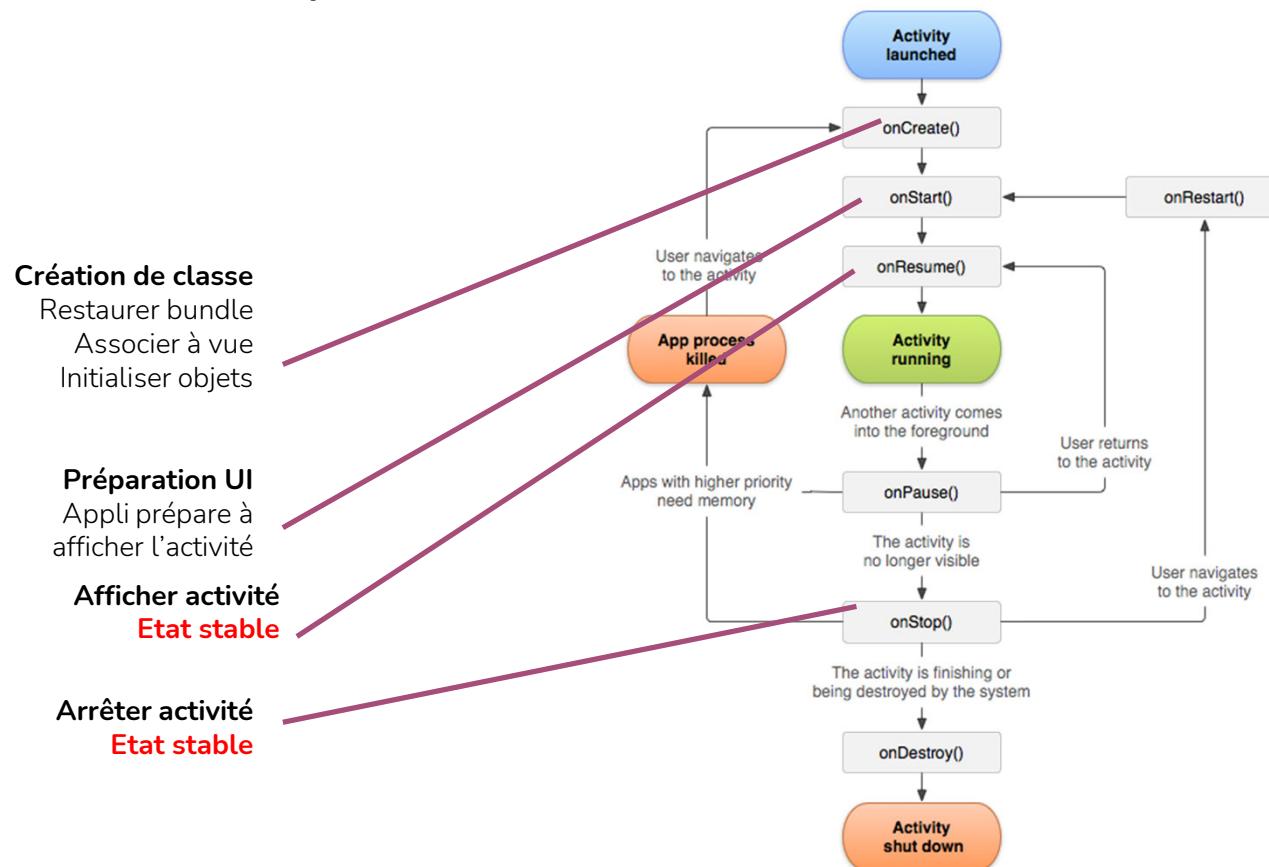
En 2016, Google « poussait » le MVP



En 2017, Google poussait le « nouveau » MVVM

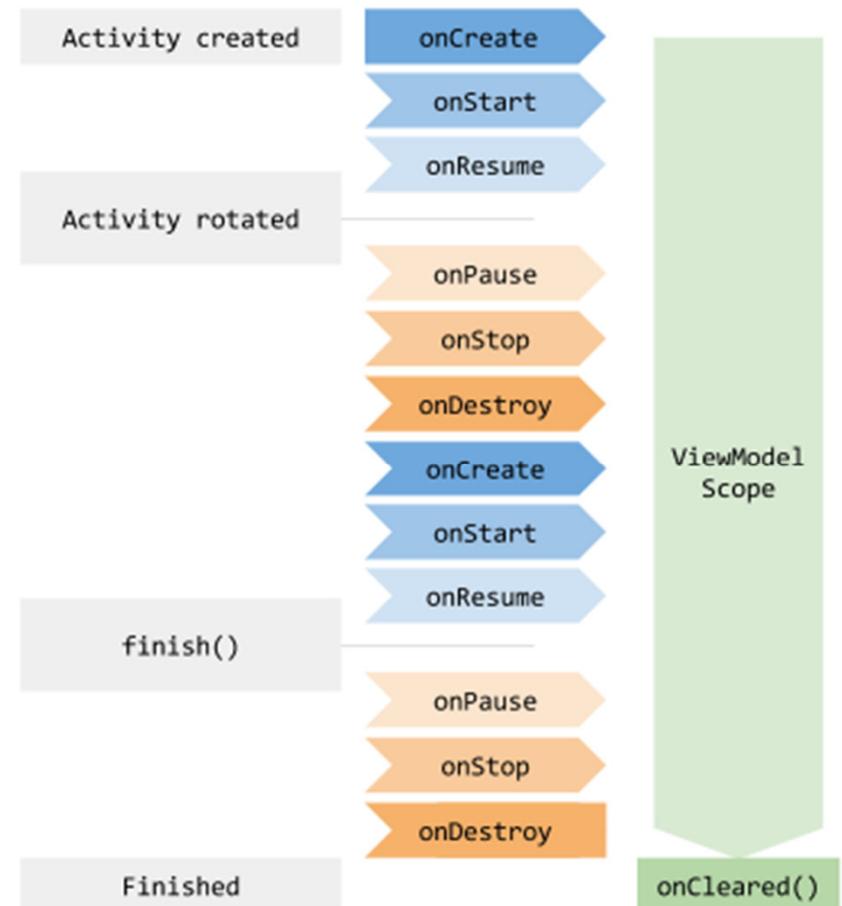


Cycle de vie d'une activité (on en reparlera en TD)



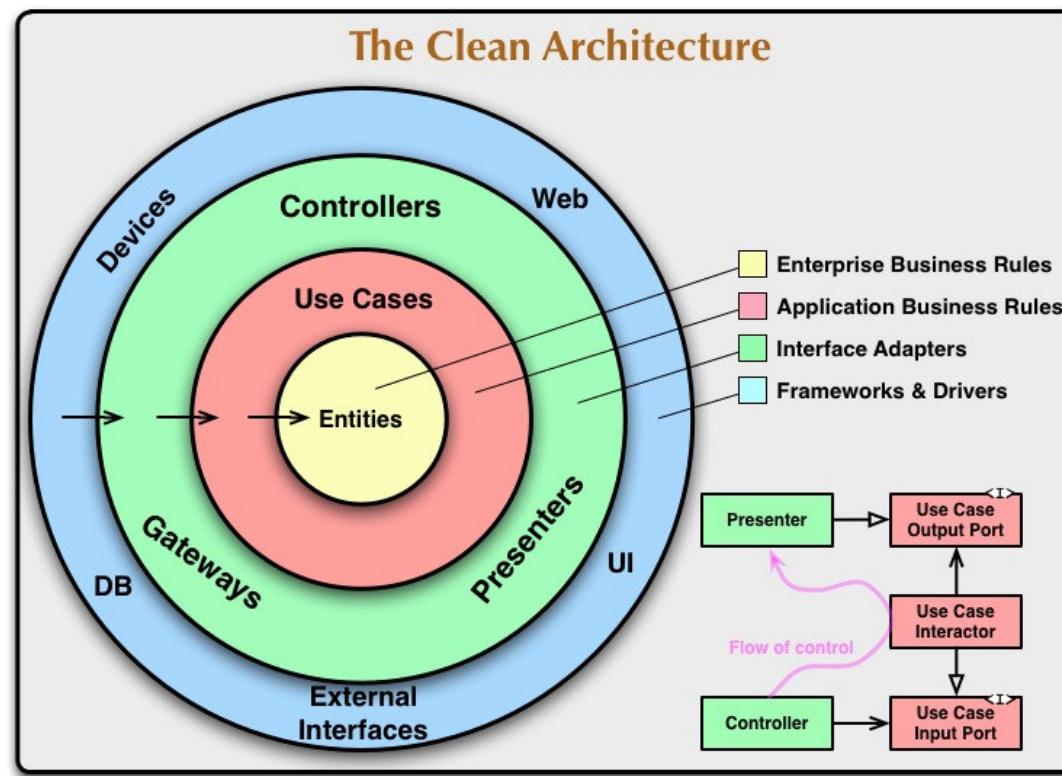
MVVM et cycle de vie

- <https://developer.android.com/topic/libraries/architecture/viewmodel>
- Le ViewModel existe tant que l'activité n'est pas détruite définitivement → ses données restent accessibles



Clean Architecture / Clean Code

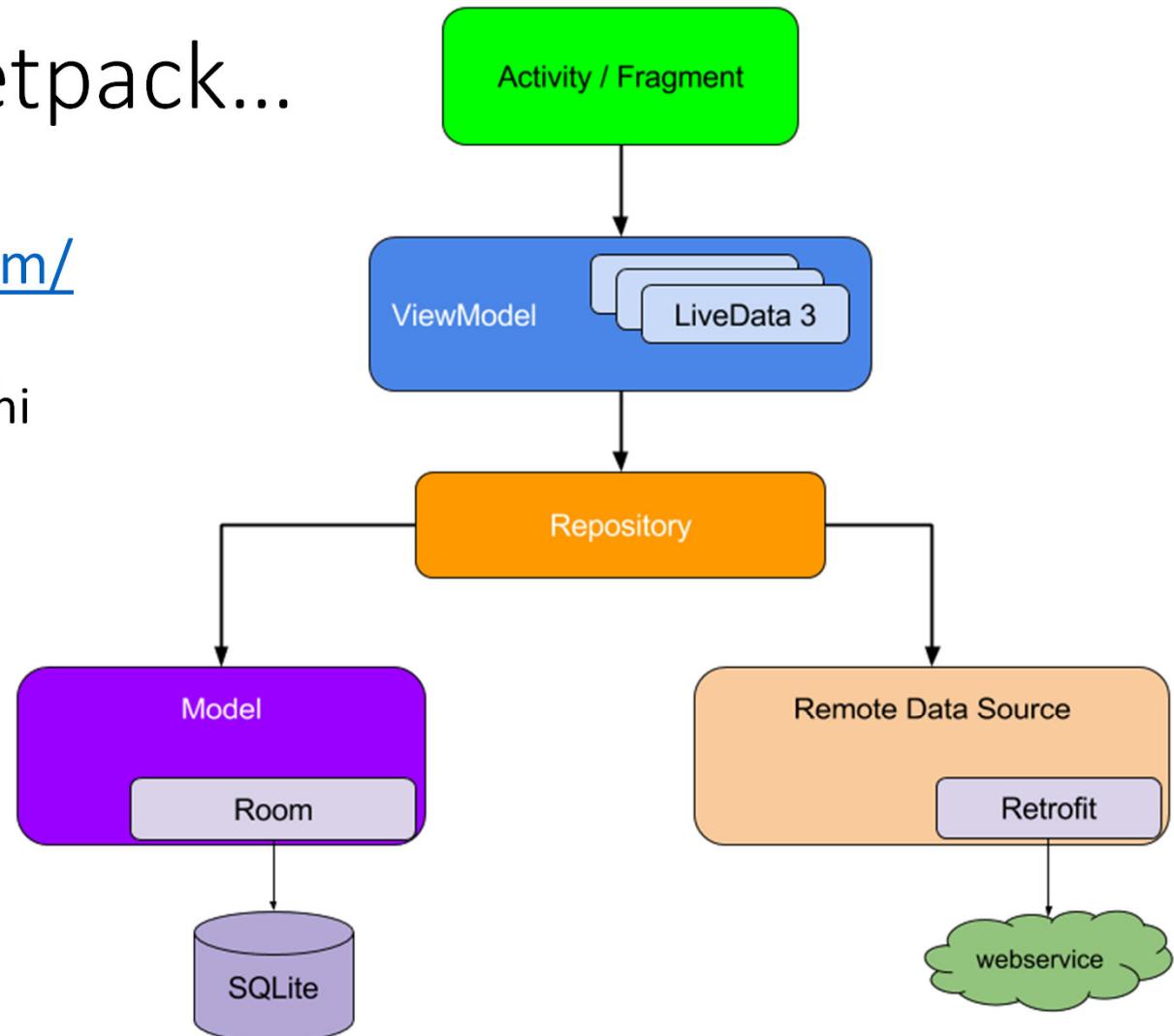
- <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>



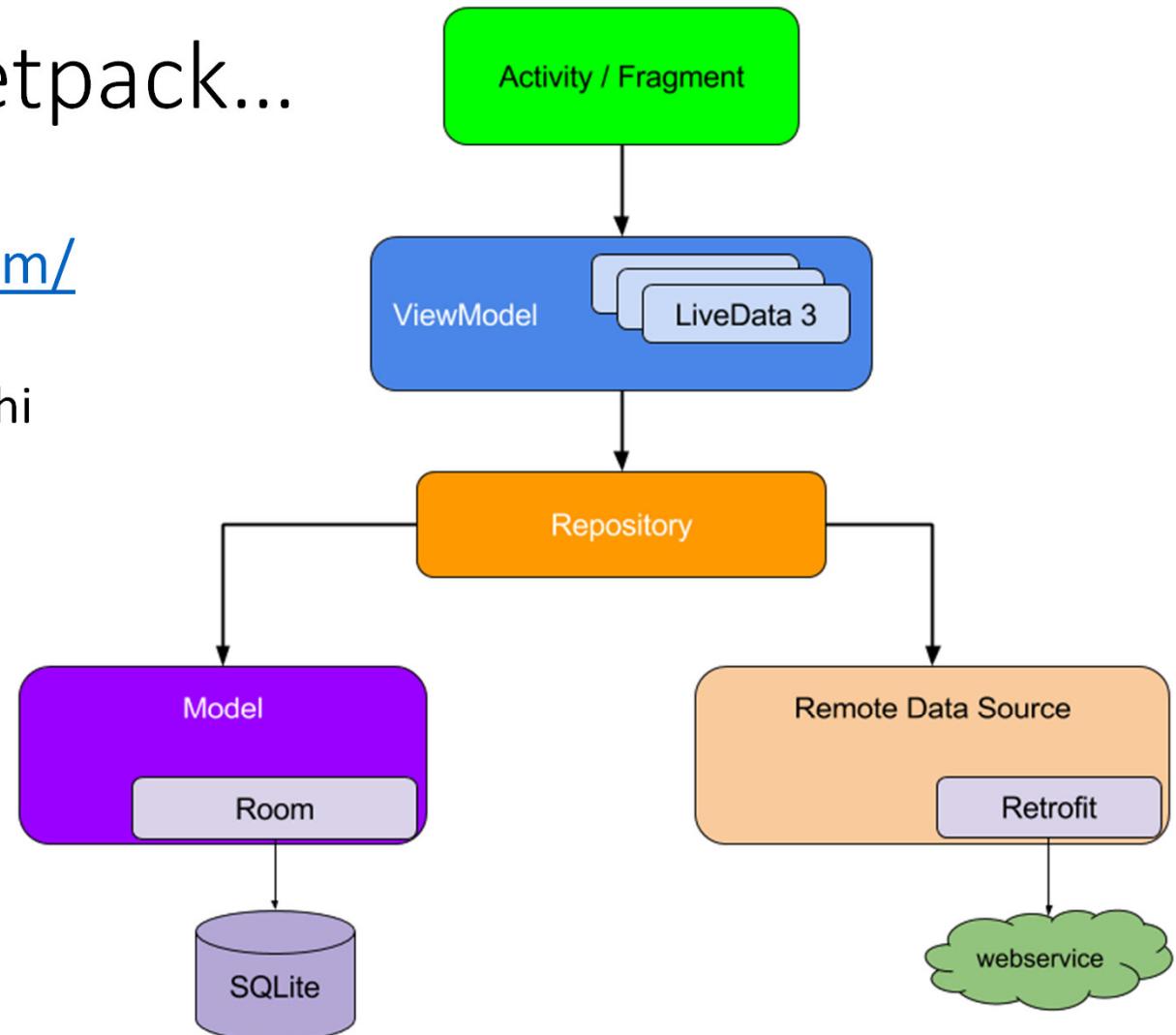
Clean archi 'new', jetpack...

- <https://developer.android.com/jetpack/guide>

- Explications détaillées de l'archi avec exemples de code (en Kotlin)
- Cf. aussi les « best practices »



Clean archi 'new', jetpack...

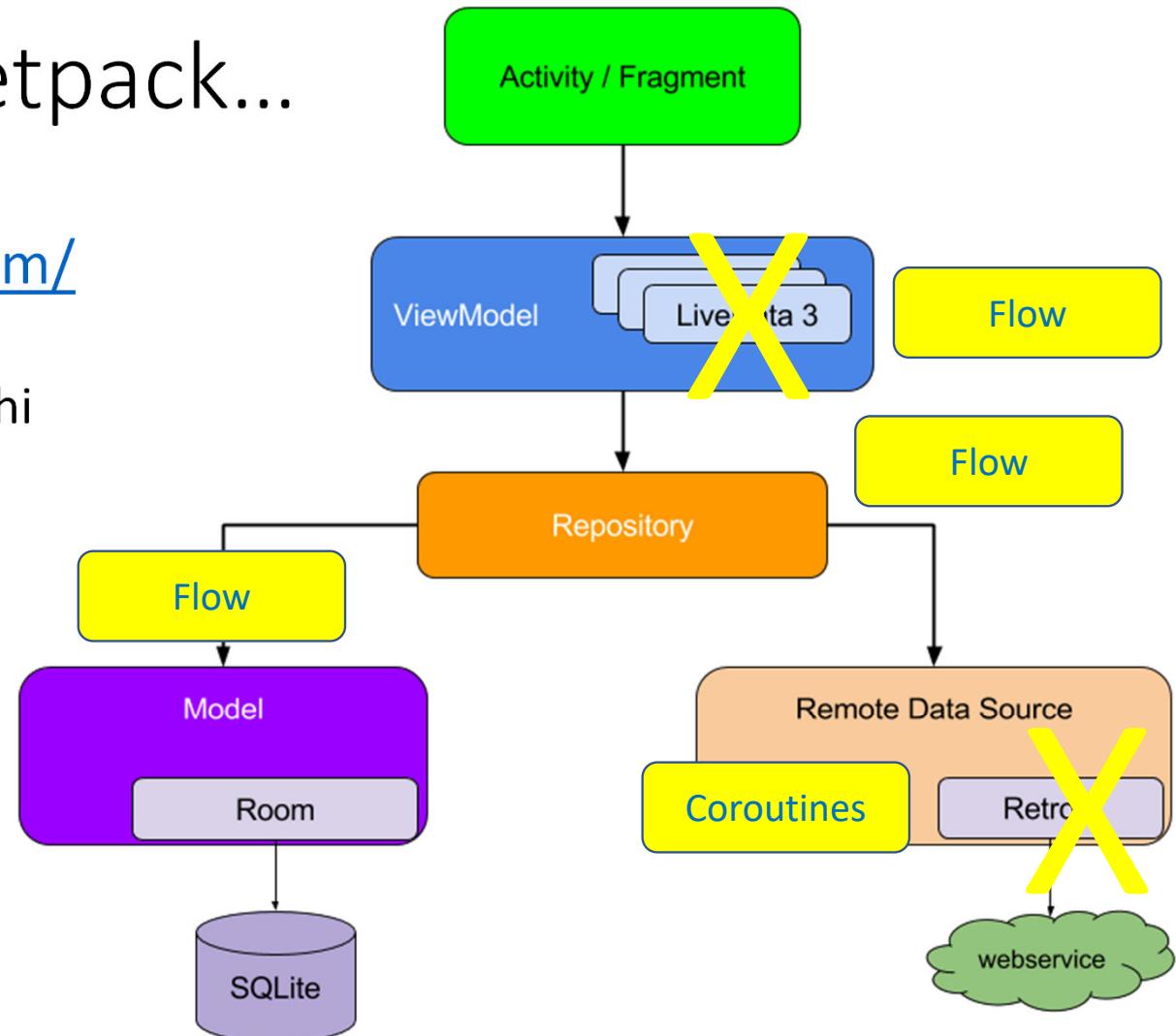


Clean archi 'new', jetpack...

- <https://developer.android.com/>

Maintenant on va plutôt sur ça...

même si **Retrofit** est toujours intéressant, tout comme les flux Rx, les LiveData, le DataBinding, etc.



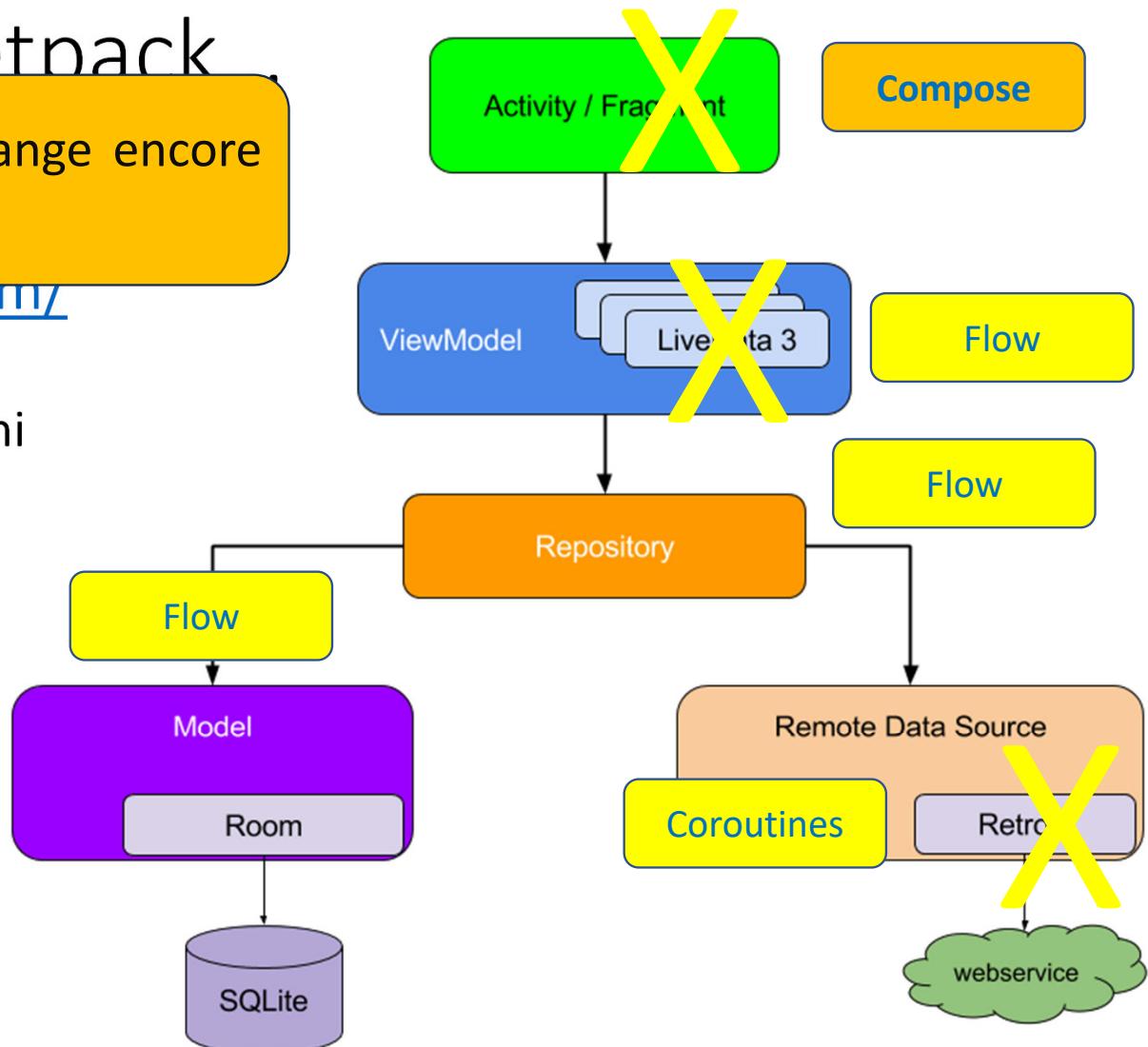
Clean archi 'new' jetpack

+ Compose

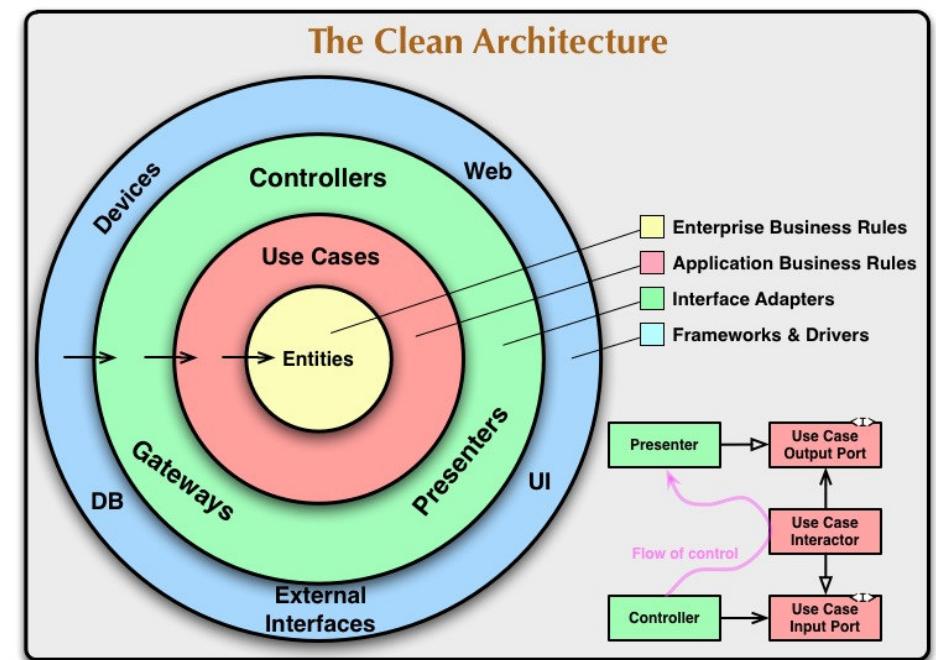
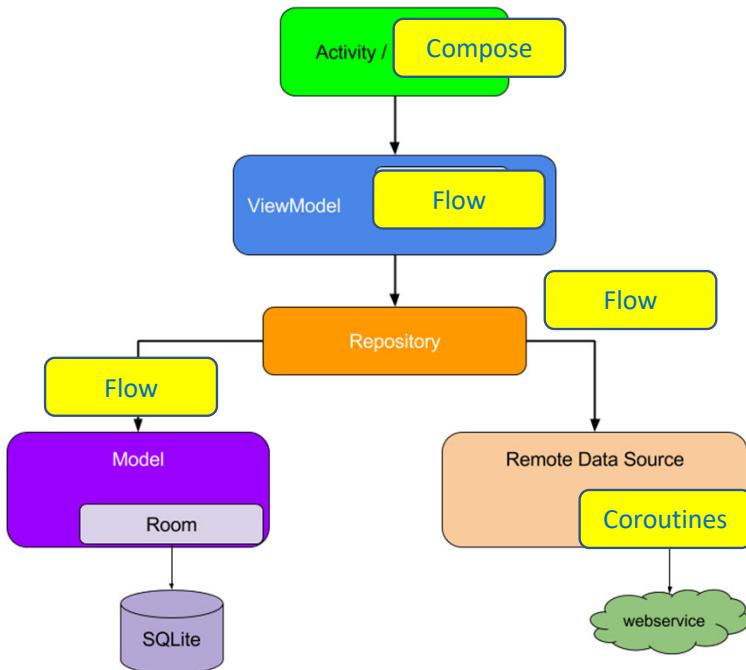
Et avec Compose on change encore la donne côté IHM.

- <https://developer.android.com/jetpack/guide>

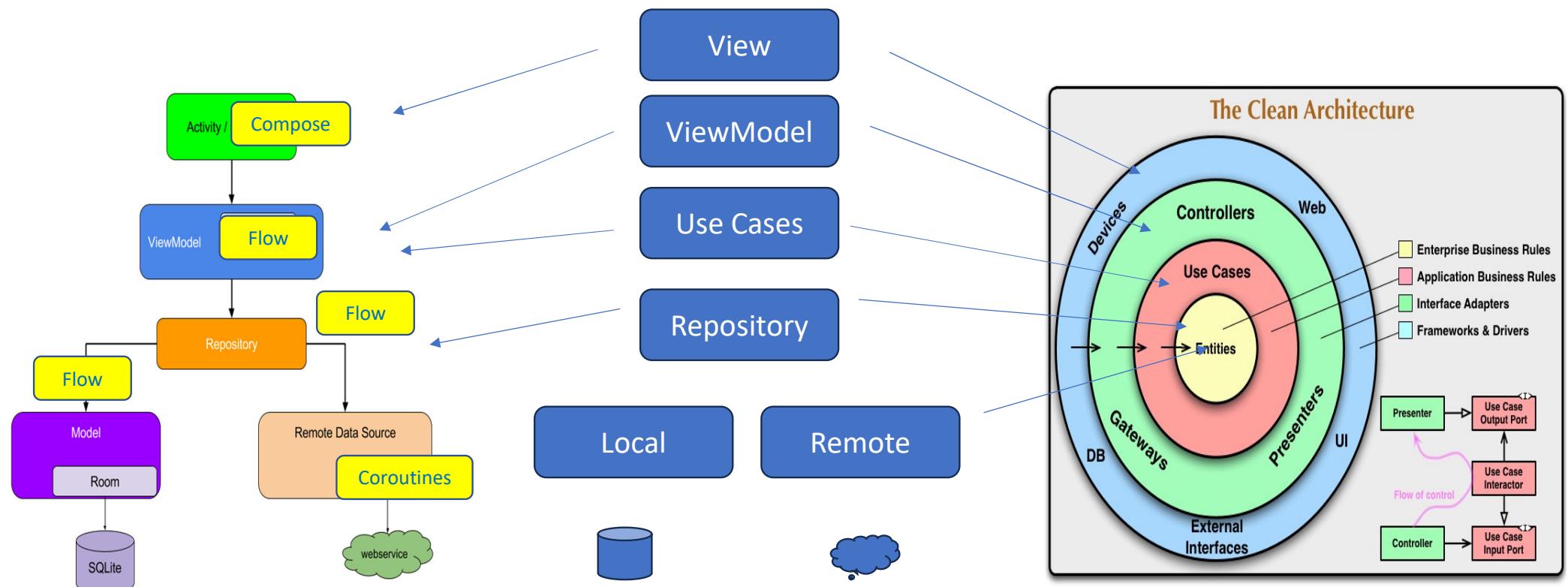
- Explications détaillées de l'archi avec exemples de code (en Kotlin)
- Cf. aussi les « best practices »



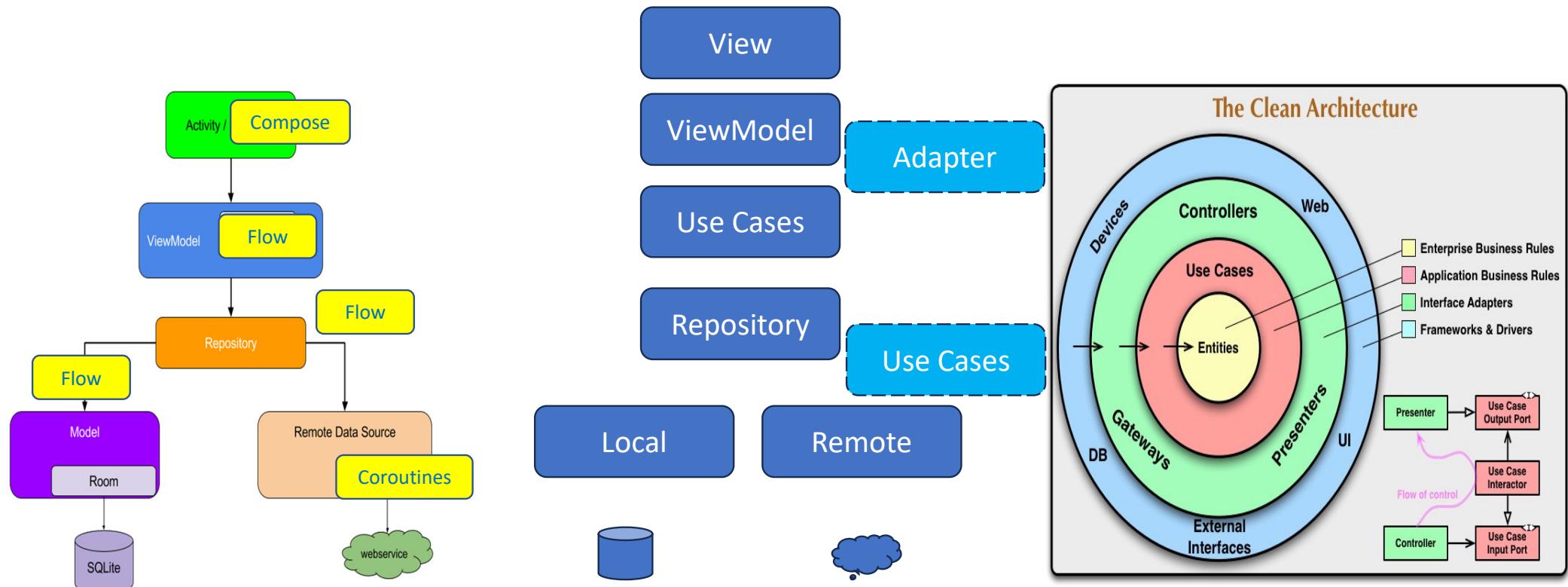
Quel est le lien entre les deux ?



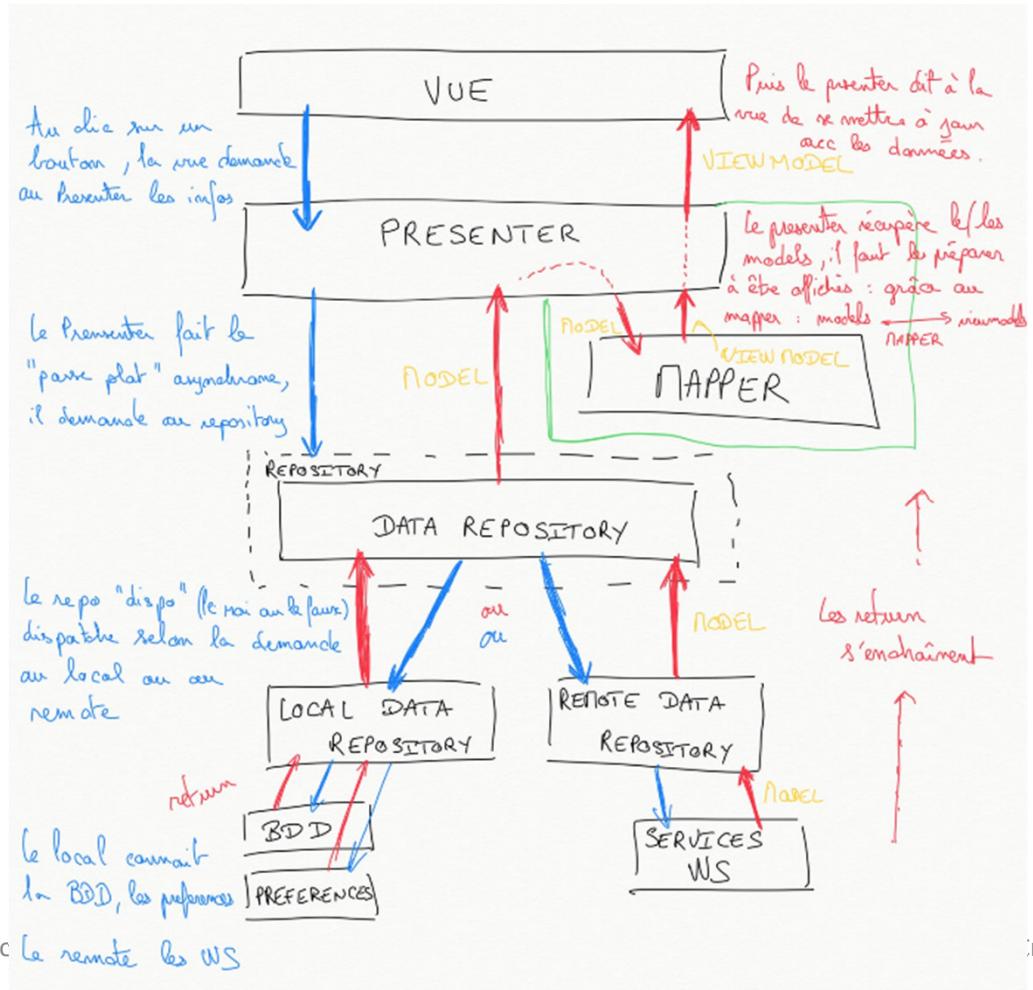
C'est la clean architecture avec MVVM



C'est la clean architecture avec MVVM (autre version)



Autre exemple de Clean Architecture (MVP)



Par couche ou par feature, à réfléchir avant, au choix,

presentation

feature1

adapter

Code de la feature 1

feature2

common

data

feature1

local

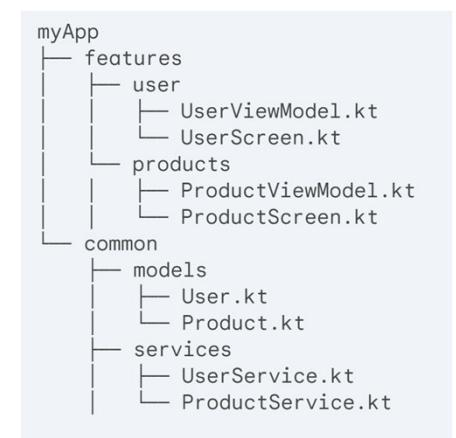
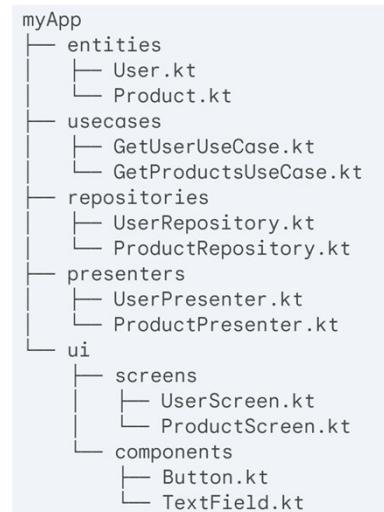
remote

feature2

common

Approche par couche ou par feature ?

- Demandez à l'IA (ChatGPT, Gemini...) « peux-tu me donner un exemple de structure de projet avec une approche par couche de la clean architecture, et avec une approche par feature ? »
 - Je vous laisse tester par vous-même hors cours 😊
 - Laquelle est la « meilleure » ?
 - Peut-on les combiner ?
- Choisissez pour projet TAC.



Avantages et inconvénients de la Clean Architecture

- Advantages of Using Clean Architecture

- Your code is even more easily testable than with plain MVVM.
- Your code is further decoupled (the biggest advantage.)
- The package structure is even easier to navigate.
- The project is even easier to maintain.
- Your team can add new features even more quickly.

- Disadvantages of Clean Architecture

- It has a slightly steep learning curve. How all the layers work together may take some time to understand, especially if you are coming from patterns like simple MVVM or MVP.
- It adds a lot of extra classes, so it's not ideal for low-complexity projects.

<https://www.toptal.com/android/android-apps-mvvm-with-clean-architecture>

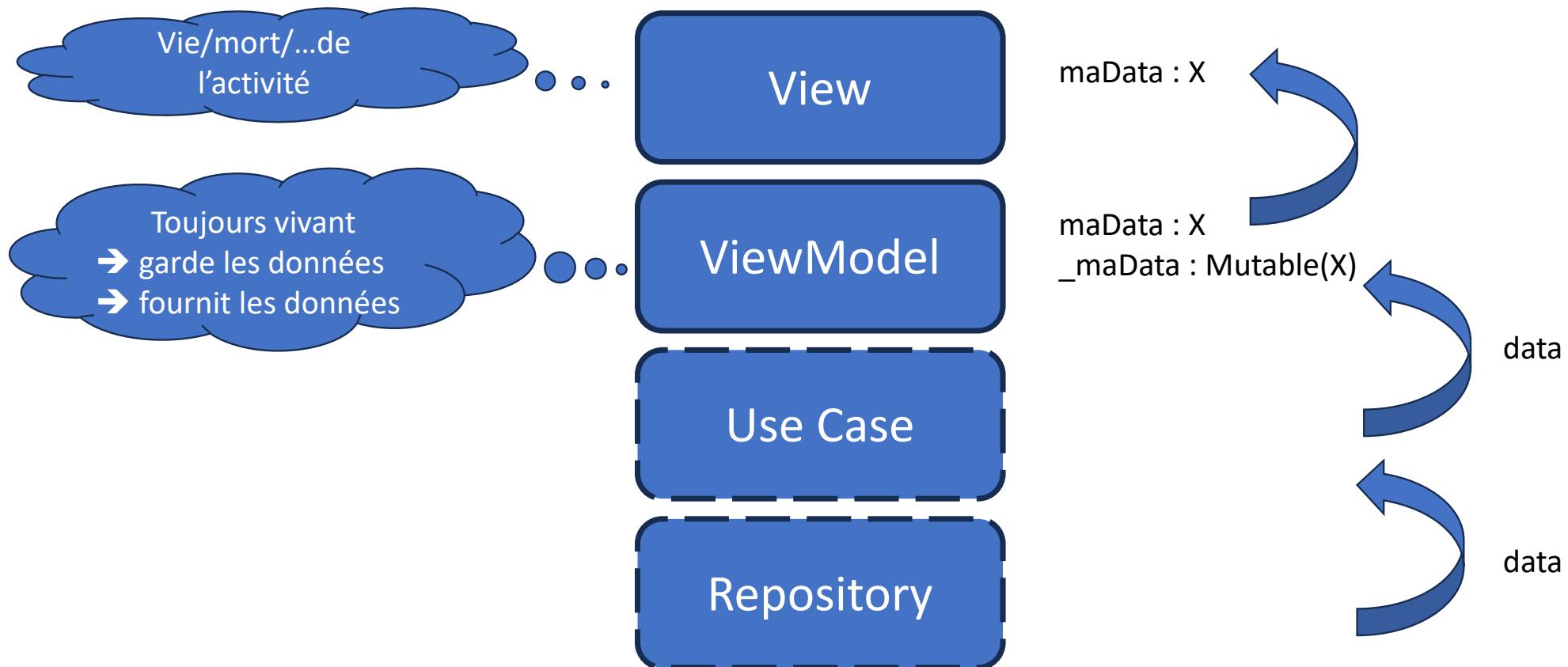
Conseils pour les architectures...

- Vous en trouverez « à la pelle » !
- Chacun y va de sa solution !
- En entreprise, prenez la leur...
- Gardez toujours un œil sur les évolutions de Google (Jetpack, Compose, Flow, Coroutine...), les propositions sur le Net, ...

Revenons sur le ViewModel

- Vous souvenez-vous de ce que c'est ?
- A quoi ça sert ?
- Comment ça marche ?

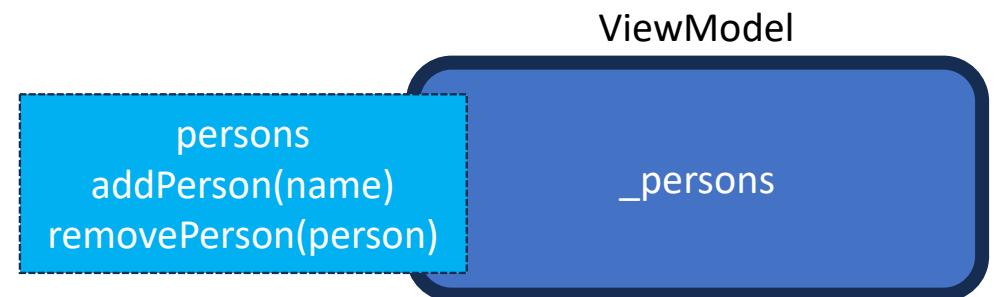
ViewModel (de façon simpliste)



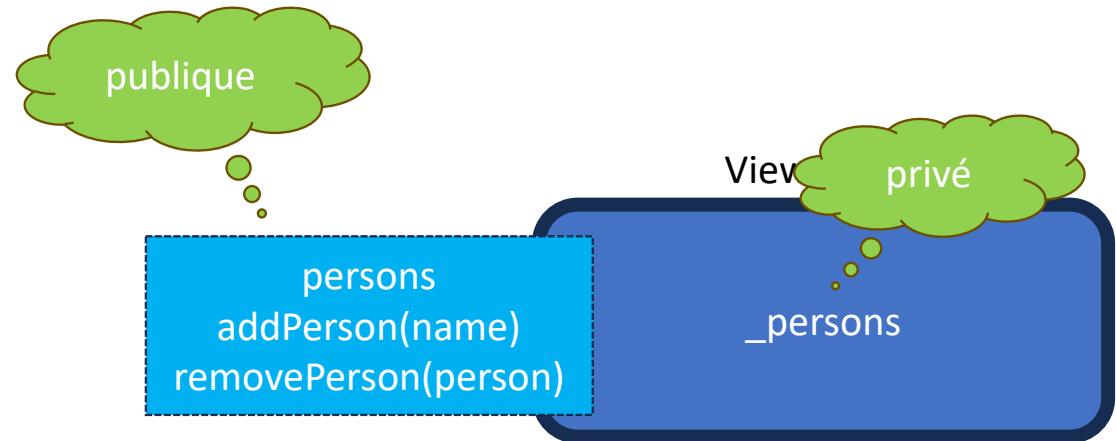
Voyons un peu de code...

- Pour le moment, que le ViewModel
- Plus tard nous verrons les Repository, Use Case, et peut-être Koin (vers la fin des cours).

Le ViewModel

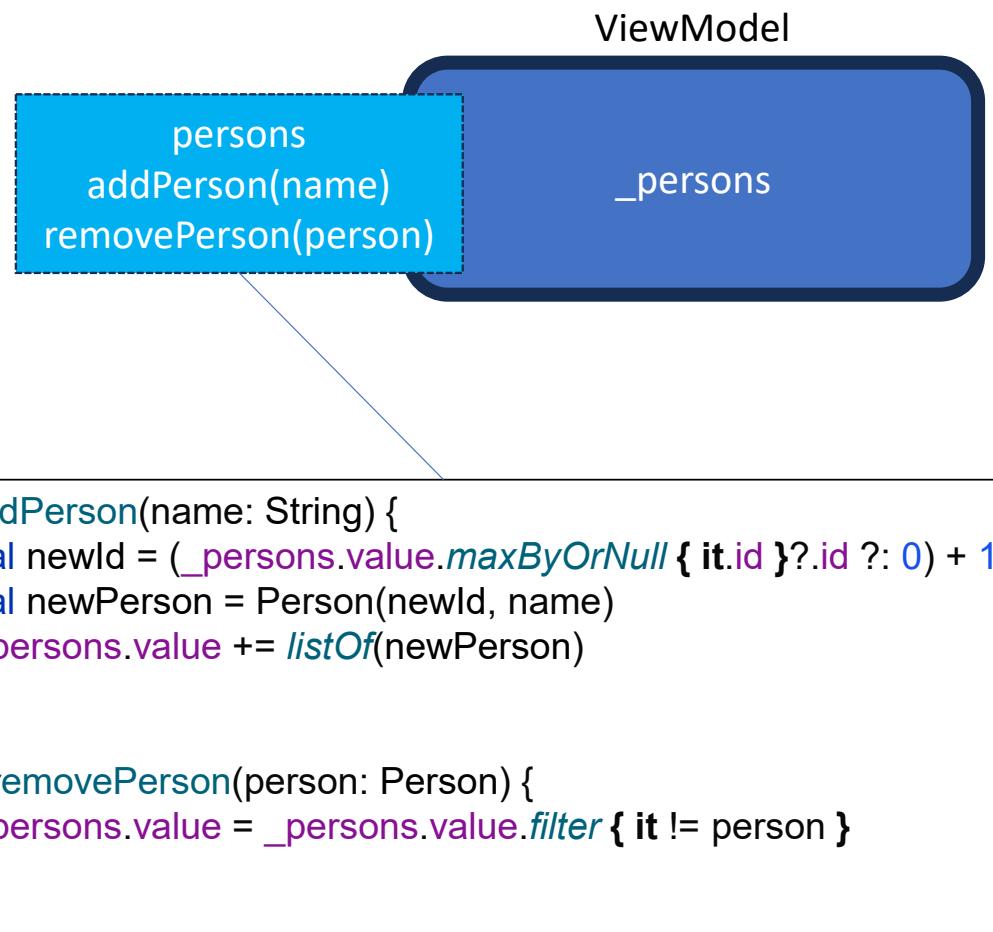


Le ViewModel



Le ViewModel

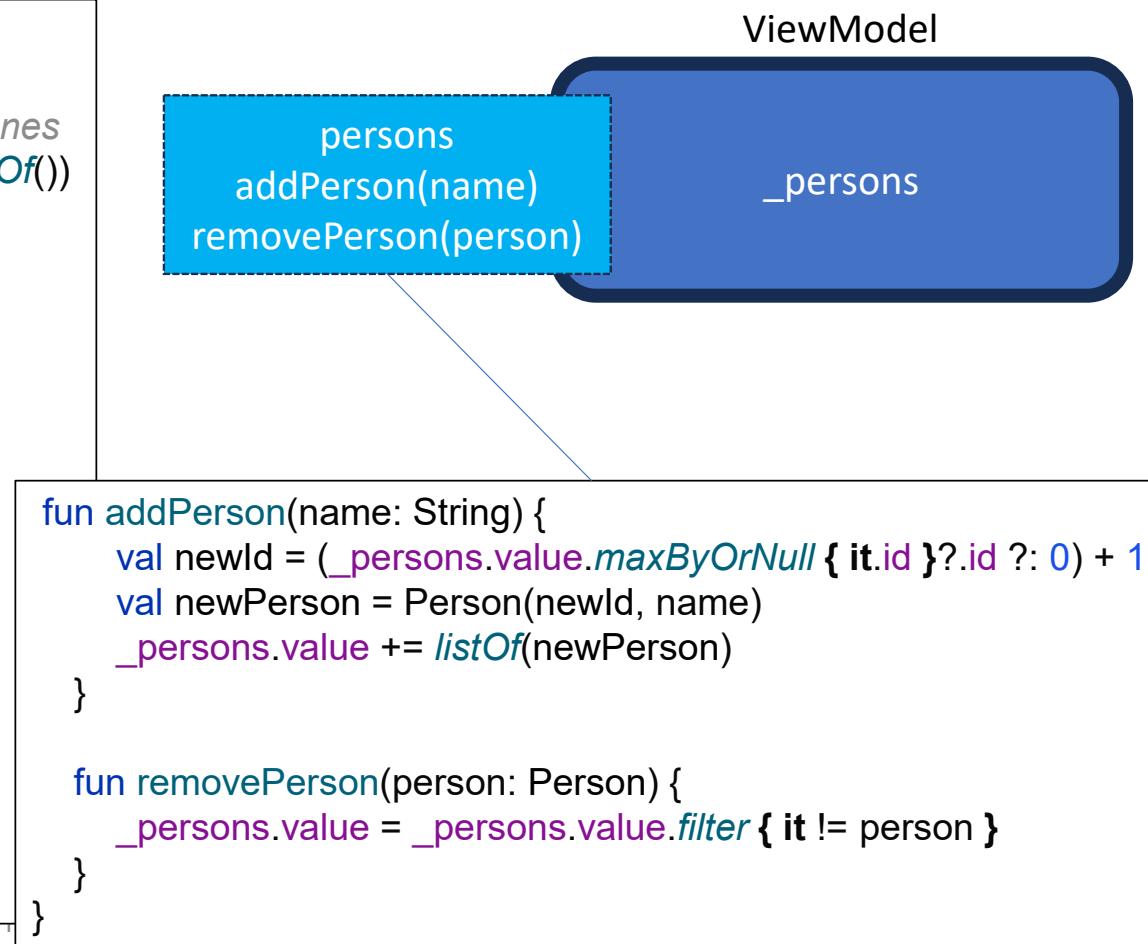
```
class PersonListViewModel() : ViewModel() {  
  
    // Utilisation de MutableState pour gérer la liste des personnes  
    private var _persons = mutableStateOf<List<Person>>(listOf())  
  
    // Exposer la liste en tant que State pour la lecture seule  
    val persons: State<List<Person>> = _persons  
  
    init {  
        Log.d("JC.ViewModel", "création du viewmodel $this")  
  
        // ici on construit en dur la liste. En général, ces valeurs  
        // proviennent de couches inférieures telles que Use Case,  
        // Repository,...  
        _persons.value = mutableListOf(  
            Person(1, "Alice"),  
            Person(2, "Bob"),  
            Person(3, "Charlie")  
        )  
    }  
}
```



Le ViewModel

```
class PersonListViewModel() : ViewModel() {  
  
    // Utilisation de MutableState pour gérer la liste des personnes  
    private var _persons = mutableStateOf<List<Person>>(listOf())  
  
    // Exposer la liste en tant que State pour la lecture seule  
    val persons: State<List<Person>> = _persons  
  
    init {  
        Log.d("JC_ViewModel", "création du viewmodel $this")  
  
        // ici on construit en dur la liste. En général, ces valeurs  
        // proviennent de couches inférieures telles que Use Case,  
        // Repository,...  
        _persons.value = mutableListOf(  
            Person(1, "Alice"),  
            Person(2, "Bob"),  
            Person(3, "Charlie")  
        )  
    }  
}
```

MutableStateOf<List<T>> → Approche réactive, c'est-à-dire mise à jour automatique de l'IHM
MutableListOf<T> → simple liste mutable, vous devrez demander la mise à jour vous-même depuis l'IHM.



MutableStateOf<List> ou
MutableStateList<> ?

MutableStateOf<List<T>> ➔ Approche réactive, c'est-à-dire mise à jour automatique de l'IHM

```
// Utilisation de MutableState pour gérer la liste des personnes
private var _persons = mutableStateOf<List<Person>>(listOf())
// Exposer la liste en tant que State pour la lecture seule
val persons: State<List<Person>> = _persons
```

```
init {
    Log.d("JC.ViewModel", "création du viewmodel $this")
    // ici on construit en dur la liste. En général, ces valeurs
    // proviennent de couches inférieures telles que Use Case,
    // Repository, ...
    _persons.value = mutableListOf(
        Person(1, "Alice"),
        Person(2, "Bob"),
        Person(3, "Charlie")
    )
}
```

ViewModel

```
persons
addPerson(name)
removePerson(person)
```

_persons

```
fun addPerson(name: String) {
    val newId = (_persons.value.maxByOrNull { it.id }?.id ?: 0) + 1
    val newPerson = Person(newId, name)
    _persons.value += listOf(newPerson)
}

fun removePerson(person: Person) {
    _persons.value = _persons.value.filter { it != person }
}
```

mutableStateOf<List<T>> ou mutableStateListOf<T> ?

« Ça dépend » 😊

mutableStateOf<List<T>>

- **Encapsule une liste immuable.**
- Chaque fois que la liste est modifiée, une nouvelle instance de la liste doit être créée et assignée à l'état.
- Les observateurs sont notifiés uniquement lorsque l'instance de la liste change.

```
private val _notes = mutableStateOf<List<Note>>(emptyList())
val notes: State<List<Note>> = _notes
```

```
fun addNote(note: Note) {
    _notes.value = _notes.value + note
}
```

mutableStateListOf<T>

- **Crée une liste mutable qui est directement observable.**
- Les modifications à la liste (ajout, suppression, modification d'éléments) notifient automatiquement les observateurs.
- Plus efficace pour les opérations fréquentes sur la liste car il n'est pas nécessaire de créer une nouvelle instance de la liste.

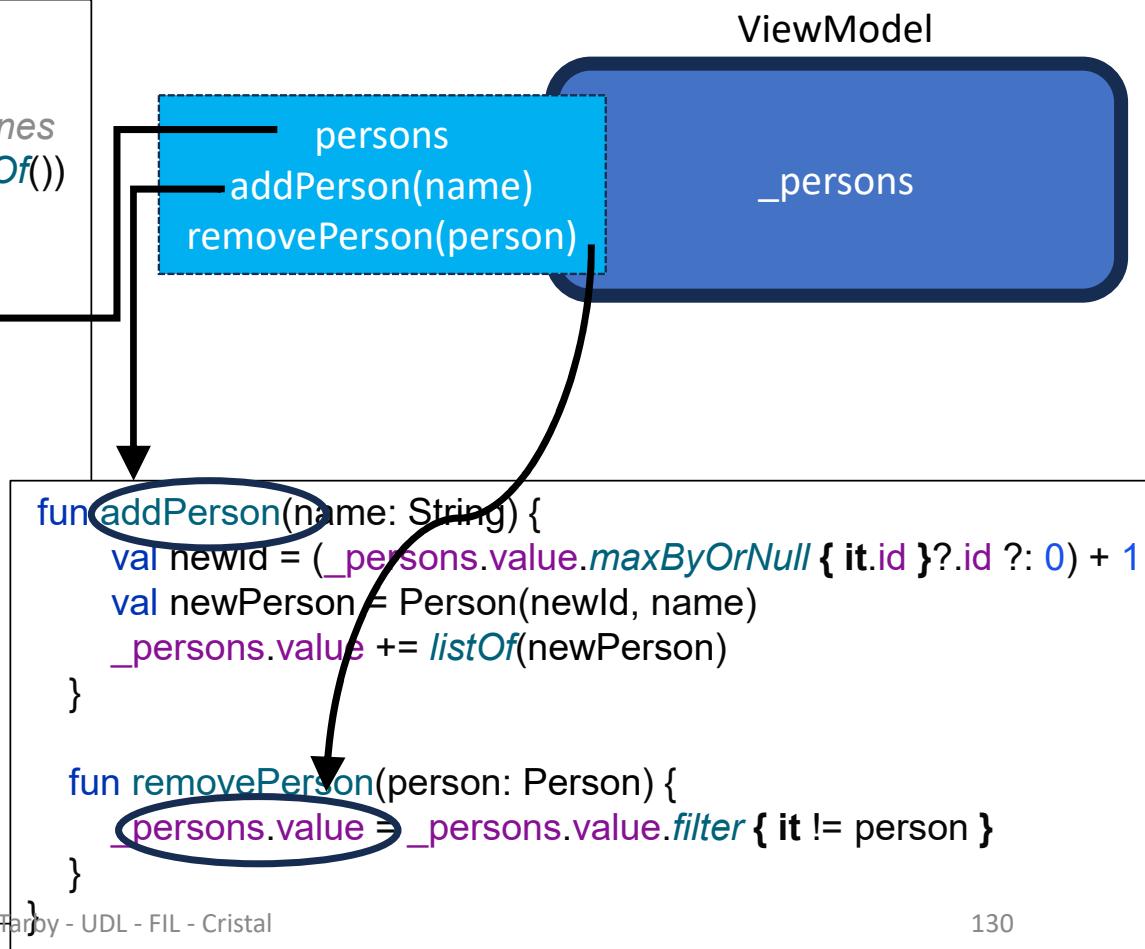
```
private val _notes = mutableStateListOf<Note>()
val notes: SnapshotStateList<Note> = _notes
```

```
fun addNote(note: Note) {
    _notes.add(note)
}
```

En résumé, mutableStateOf<List<T>> est utile lorsque vous travaillez avec des listes immuables et que vous souhaitez notifier les observateurs uniquement lorsque l'instance de la liste change. mutableStateListOf<T> est plus approprié pour les listes mutables où les modifications fréquentes doivent être observées directement.

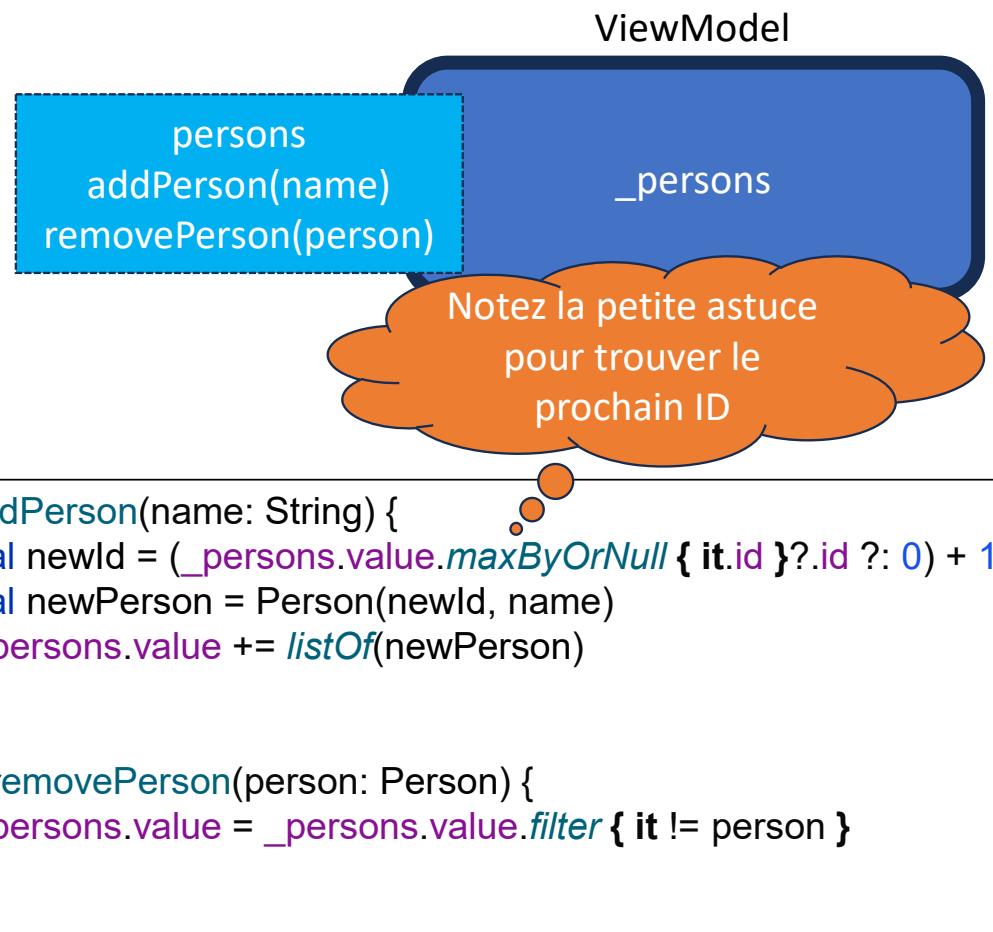
Le ViewModel

```
class PersonListViewModel() : ViewModel() {  
  
    // Utilisation de MutableState pour gérer la liste des personnes  
    private var _persons = mutableStateOf<List<Person>>(listOf())  
  
    // Exposer la liste en tant que State pour la lecture seule  
    val persons: State<List<Person>> = _persons  
  
    init {  
        Log.d("JC_ViewModel", "création du viewmodel $this")  
  
        // ici on construit en dur la liste. En général, ces valeurs  
        // proviennent de couches inférieures telles que Use Case,  
        // Repository,...  
        _persons.value = mutableListOf(  
            Person(1, "Alice"),  
            Person(2, "Bob"),  
            Person(3, "Charlie")  
        )  
    }  
}
```

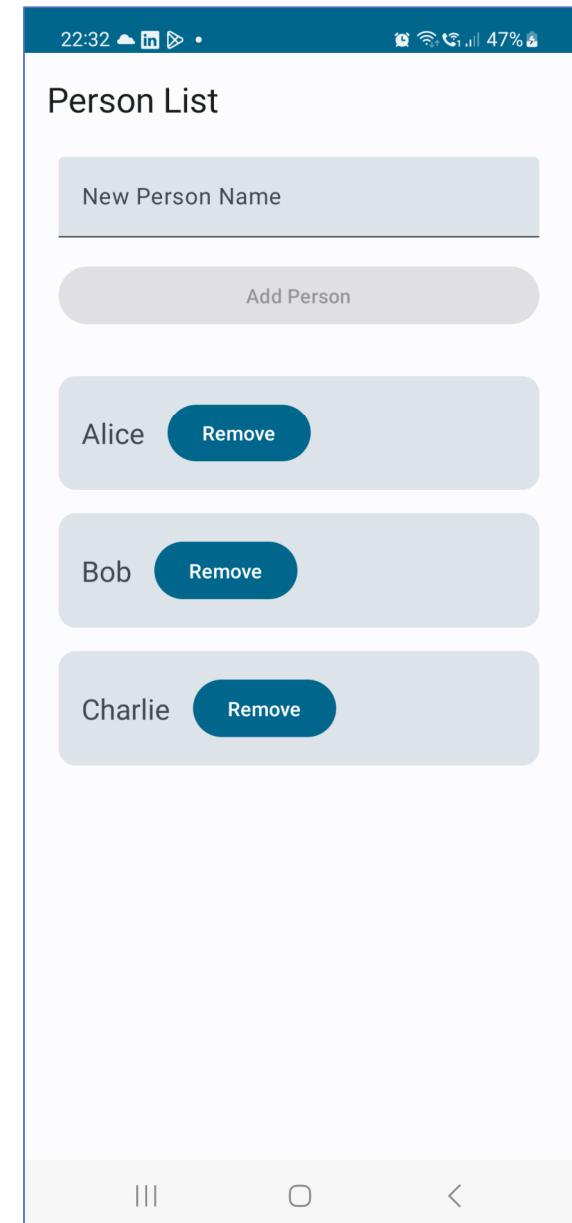


Le ViewModel

```
class PersonListViewModel() : ViewModel() {  
  
    // Utilisation de MutableState pour gérer la liste des personnes  
    private var _persons = mutableStateOf<List<Person>>(listOf())  
  
    // Exposer la liste en tant que State pour la lecture seule  
    val persons: State<List<Person>> = _persons  
  
    init {  
        Log.d("JC_ViewModel", "création du viewmodel $this")  
  
        // ici on construit en dur la liste. En général, ces valeurs  
        // proviennent de couches inférieures telles que Use Case,  
        // Repository,...  
        _persons.value = mutableListOf(  
            Person(1, "Alice"),  
            Person(2, "Bob"),  
            Person(3, "Charlie")  
        )  
    }  
}
```

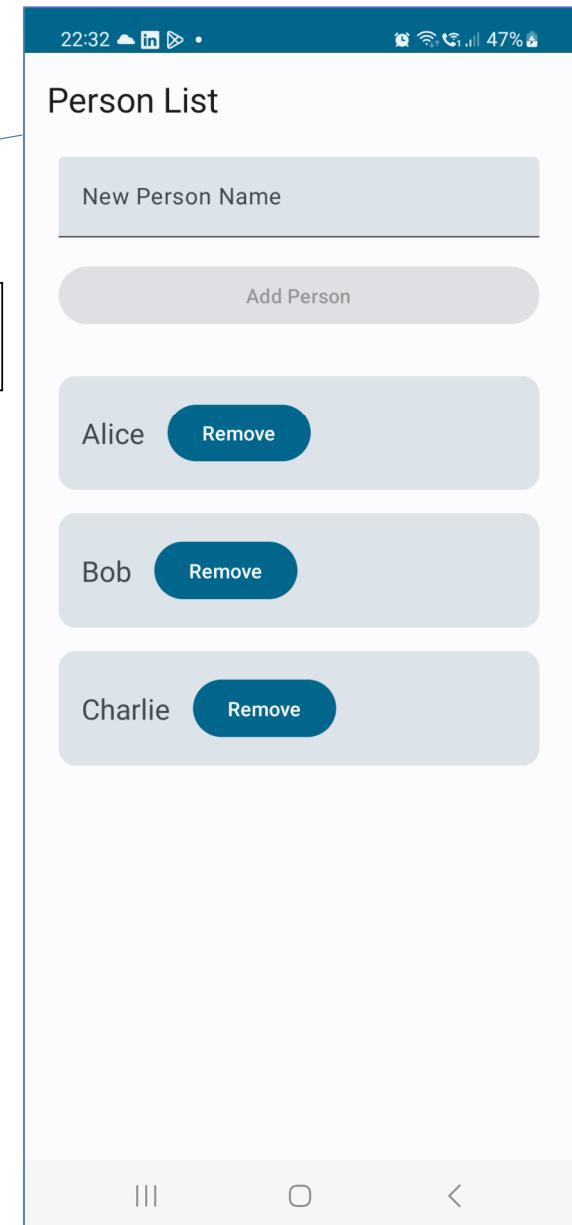


L'activité principale



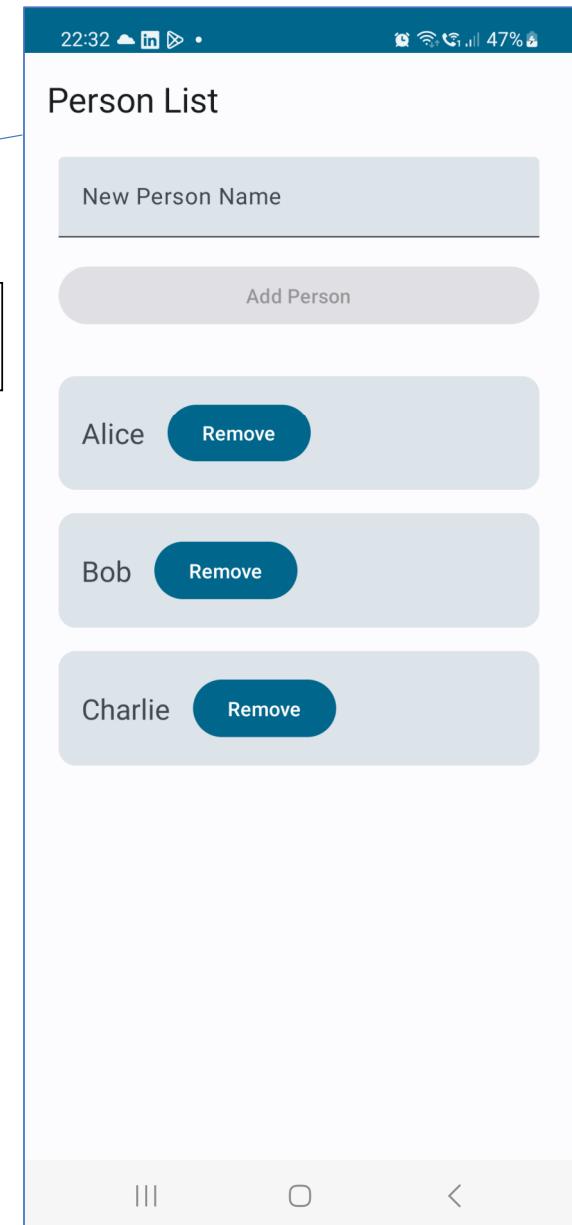
L'activité principale

```
// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique  
val persons = viewModel.persons
```



L'activité principale

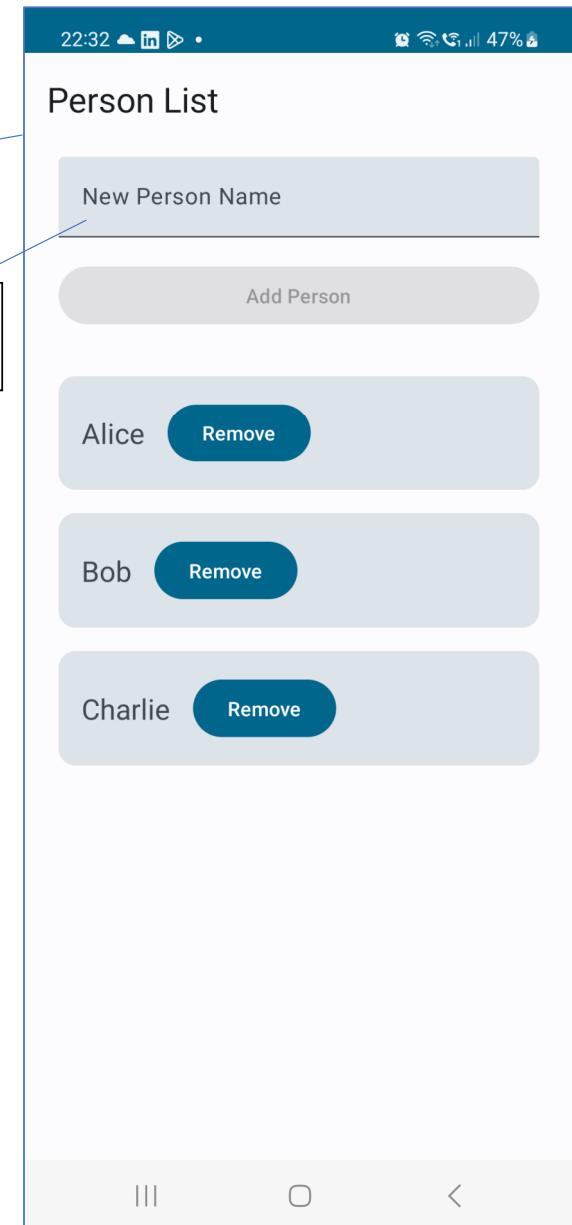
```
// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique  
val persons = viewModel.persons
```



L'activité principale

```
// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique  
val persons = viewModel.persons
```

```
// Ajout d'une nouvelle personne  
var newPersonName by remember { mutableStateOf("") }  
TextField(  
    value = newPersonName,  
    ...)
```

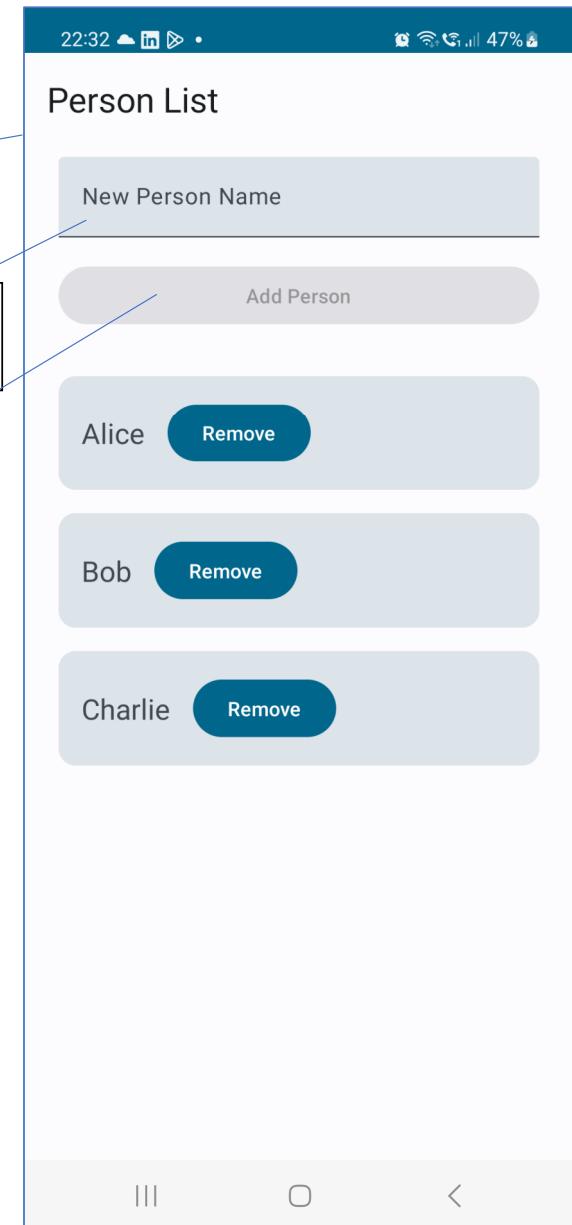


L'activité principale

```
// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique  
val persons = viewModel.persons
```

```
// Ajout d'une nouvelle personne  
var newPersonName by remember { mutableStateOf("") }  
TextField(  
    value = newPersonName,  
    ...)
```

```
Button(onClick = {  
    viewModel.addPerson(newPersonName)},  
    ...) {...}
```

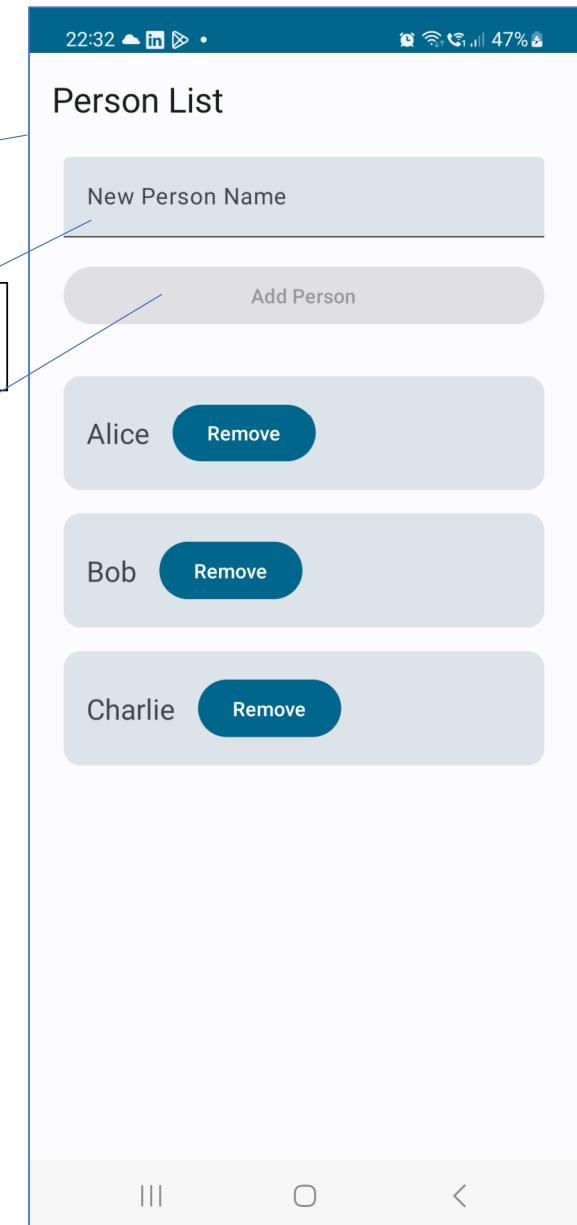


L'activité principale

```
// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique  
val persons = viewModel.persons
```

```
// Ajout d'une nouvelle personne  
var newPersonName by remember { mutableStateOf("") }  
TextField(  
    value = newPersonName,  
    ...)
```

```
Button(onClick = {  
    viewModel.addPerson(newPersonName)},  
    ...) {...}
```



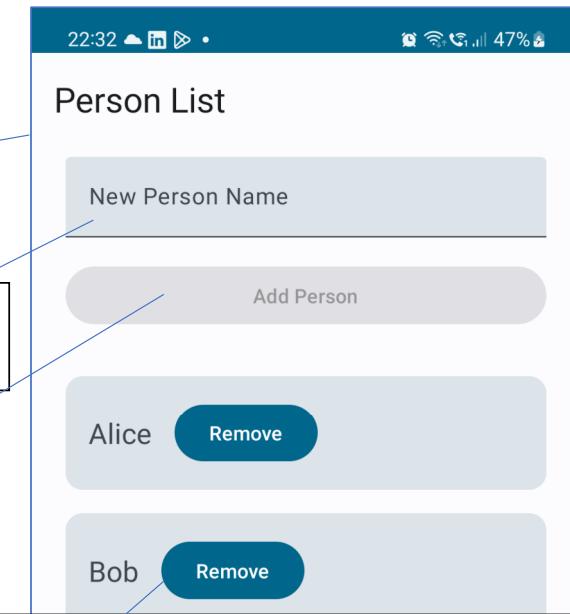
L'activité principale

```
// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique  
val persons = viewModel.persons
```

```
// Ajout d'une nouvelle personne  
var newPersonName by remember { mutableStateOf("") }  
TextField(  
    value = newPersonName,  
    ...)
```

```
Button(onClick = {  
    viewModel.addPerson(newPersonName),  
    ...} {...})
```

```
LazyColumn {  
    items(persons.value) { person ->  
        Card(...)  
        {  
            Text(text = person.name, style = TextStyle(fontSize = 20.sp))  
            Spacer(modifier = Modifier.width(16.dp))  
            Button(  
                onClick = { viewModel.removePerson(person) },  
                content = {  
                    Text(text = "Remove")  
                }  
            )  
        }  
    }  
}
```



L'activité principale

```
// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique  
val persons = viewModel.persons
```

```
// Ajout d'une nouvelle personne  
var newPersonName by remember { mutableStateOf("") }  
TextField(  
    value = newPersonName,  
    ...)
```

```
Button(onClick = {  
    viewModel.addPerson(newPersonName),  
    ...} {...})
```

```
LazyColumn {  
    items(persons.value) { person ->  
        Card(...)  
        {  
            Text(text = person.name, style = TextStyle(fontSize = 20.sp))  
            Spacer(modifier = Modifier.width(16.dp))  
            Button(  
                onClick = { viewModel.removePerson(person) },  
                content = {  
                    Text(text = "Remove")  
                }  
            )  
        }  
    }  
}
```

L'activité principale

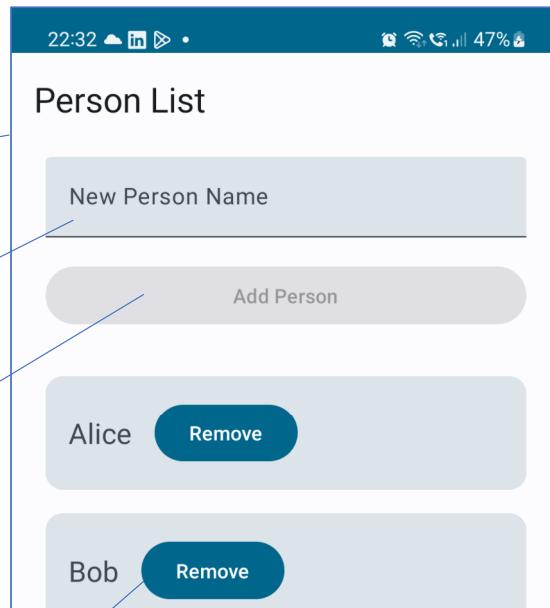
```
// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique  
val persons = viewModel.persons
```

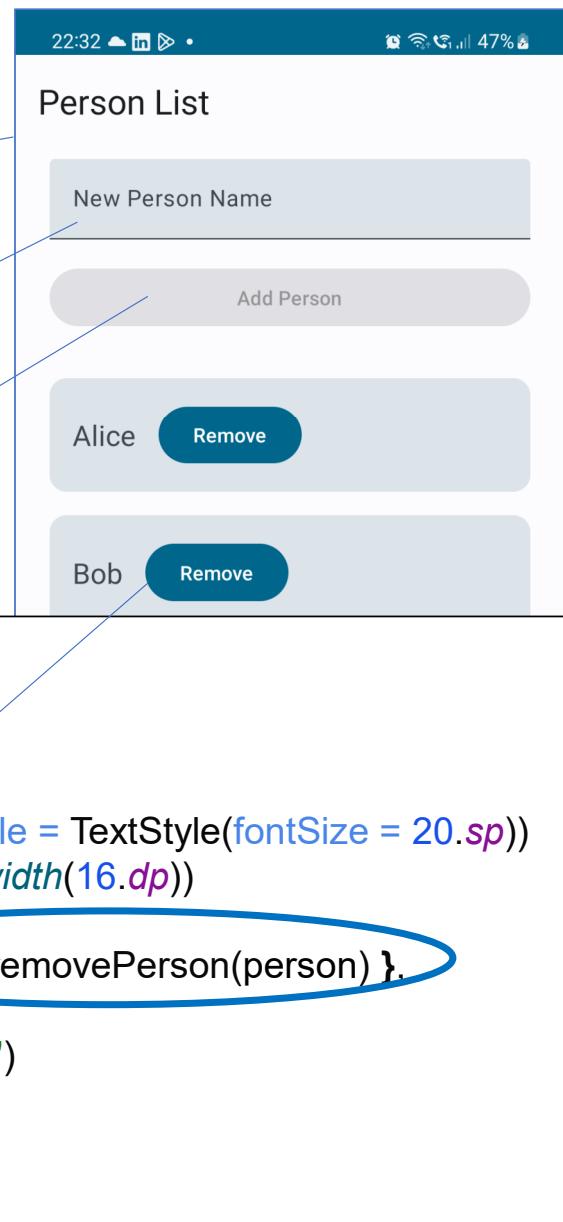
```
// Ajout d'une nouvelle personne  
var newPersonName by remember { mutableStateOf("")}  
TextField(  
    value = newPersonName,  
    ...)
```

```
Button(onClick = {  
    viewModel.addPerson(newPersonName),  
    ...} {...})
```

Modifie la valeur du state « persons » du ViewModel

```
LazyColumn {  
    items(persons.value) { person ->  
        Card(...) {  
            Text(text = person.name, style = TextStyle(fontSize = 20.sp))  
            Spacer(modifier = Modifier.width(16.dp))  
            Button(  
                onClick = { viewModel.removePerson(person) },  
                content = {  
                    Text(text = "Remove")  
                }  
            )  
        }  
    }  
}
```





Donc mise à jour automatique ici de « persons »,
et donc mise à jour de la liste affichée

// on récupère les données depuis le viewmodel. C'est un State donc recomposition automatique
val persons = viewModel.persons

// Ajout d'une nouvelle personne
var newPersonName by remember { mutableStateOf("")}
TextField(
 value = newPersonName,
 ...)

Button(onClick = {
 viewModel.addPerson(newPersonName)},
 ...) {...}

Modifie la valeur du state
« persons » du ViewModel

```
LazyColumn {  
    items(persons.value) { person ->  
        Card(...)  
        {  
            Text(text = person.name, style = TextStyle(fontSize = 20.sp))  
            Spacer(modifier = Modifier.width(16.dp))  
            Button(  
                onClick = { viewModel.removePerson(person) },  
                content = {  
                    Text(text = "Remove")  
                }  
            )  
        }  
    }  
}
```

OK pour le schéma « général »

- Mais comment créer le ViewModel ? (sans parler du Repository, etc.)
 - Plusieurs solutions... : 7 montrées ici

Contrairement à l'an passé, nous ne reverrons pas ces exemples de cours en TD.
➔ Travaillez-les chez vous pour qu'on attaque directement le TD.

Création de ViewModel : solution 1

```
// Solution 1.  
// On crée le viewmodel et on l'initialise de suite  
val viewModel = ViewModelProvider(this).get(PersonListViewModel::class.java)
```

Code 1

démo

Création de ViewModel : solution 2

// Solution 2

// lateinit var ==> on initialise la variable lors de son premier usage, et pas lors de sa déclaration.

```
private lateinit var viewModel : PersonListViewModel
```

...code de l'application...

// on initialise ICI la variable (premier usage), et pas lors de sa déclaration.

```
viewModel = ViewModelProvider(this).get(PersonListViewModel::class.java)
```

Code 1

démo

Création de ViewModel : solution 3

```
// Solution 3.  
// Avec val et by lazy --> création du ViewModel uniquement si nécessaire. Cela peut être  
// utile pour retarder la création d'objets coûteux jusqu'à ce qu'ils soient réellement nécessaires  
val viewModel: PersonListViewModel by lazy {  
    ViewModelProvider(this).get(PersonListViewModel::class.java)  
}
```

Code 1

démo

Création de ViewModel : solution 4

// Solution 4.

// même principe que la solution 3 précédente. Le code fait la même chose que 'by lazy' sans avoir
// besoin de spécifier le ViewModelProvider. Il le détermine automatiquement pour vous.

```
val viewModel: PersonListViewModel by viewModels()
```

démo



Solution la +
simple si pas de
paramètres

Création de ViewModel avec paramètre(s)

- Exemple : création d'un Repository
 - Autre exemple : passage de l'application comme référence pour Room (+ tard)
- ➔ Modification du ViewModel + usage d'une Factory

Le ViewModel (sans paramètre)

```
class PersonListViewModel() : ViewModel {
```

```
    // Utilisation de MutableState pour gérer la liste des personnes  
    private var _persons = mutableStateOf<List<Person>>(listOf())
```

```
    // Exposer la liste en tant que State pour la lecture seule  
    val persons: State<List<Person>> = _persons
```

```
    init {  
        Log.d("JC.ViewModel", "création du viewmodel $this")  
    }
```

// ici on construit en dur la liste. En général, ces valeurs proviennent de couches inférieures telles que Use Case, Repository,...

```
    _persons.value = mutableListOf(  
        Person(1, "Alice"),  
        Person(2, "Bob"),  
        Person(3, "Charlie")  
    )  
}
```

ViewModel

persons
addPerson(name)
removePerson(person)

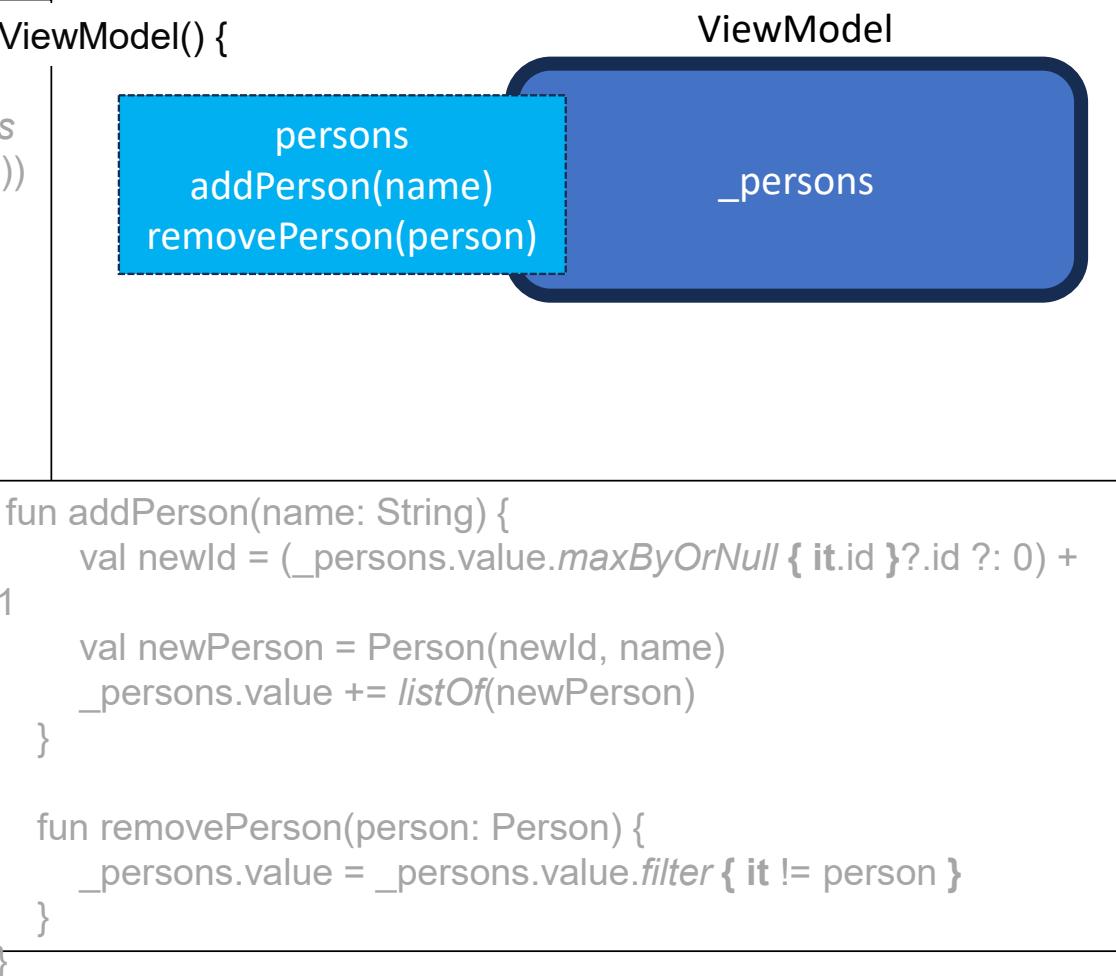
_persons

```
    fun addPerson(name: String) {  
        val newId = (_persons.value.maxByOrNull { it.id }?.id ?: 0) +  
        1  
        val newPerson = Person(newId, name)  
        _persons.value += listOf(newPerson)  
    }  
  
    fun removePerson(person: Person) {  
        _persons.value = _persons.value.filter { it != person }  
    }
```

Le ViewModel (avec paramètre)

```
class PersonListViewModel(private val repository: Repository) : ViewModel() {  
  
    // Utilisation de MutableState pour gérer la liste des personnes  
    private var _persons = mutableStateOf<List<Person>>(listOf())  
  
    // Exposer la liste en utilisant une State pour la lecture seule  
    val persons: State<List<Person>> = _persons  
  
    init {  
        Log.d("ViewModel", "création du viewmodel $this")  
  
        // construit en dur la liste. En général, ces valeurs  
        // proviennent de couches inférieures telles que Use Case,  
        // Repository,...  
        _persons.value = mutableListOf(  
            Person(1, "Alice"),  
            Person(2, "Bob"),  
            Person(3, "Charlie")  
    }  
}
```

Paramètre = propriété d'instance !



Création de ViewModel : solution 5.1

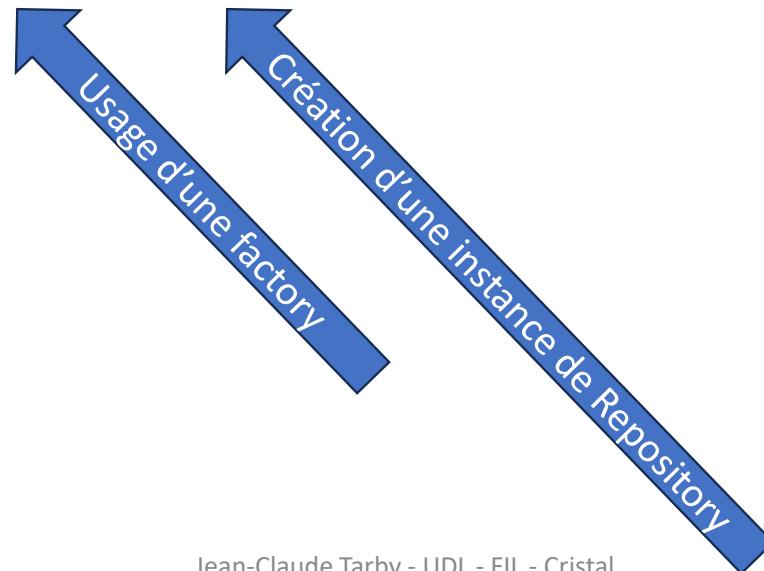
```
onCreate(...){
```

```
...
```

```
// Solution 5.1.
```

```
// Le souci est qu'on crée toujours des nouvelles instances de Repository, à chaque rotation d'écran...
```

```
    val viewModel: PersonListViewModel by viewModels {
        PersonListViewModelFactory(Repository())
    }
```



Création du Repository

```
// pour solution 5.1
// Si vous tournez le téléphone, vous lancez onCreate de l'activité, et donc vous recréez un
// Repository. Vous pourrez le voir dans le log, le n° de l'objet change à chaque fois.
class Repository {
    init {
        Log.d("JC.ViewModel", "création du Repository $this")
    }
}
```

Création de la factory

```
// pour solution 5.1 et 5.2 uniquement
class PersonListViewModelFactory(private val repository: Repository) : ViewModelProvider.Factory {
    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {

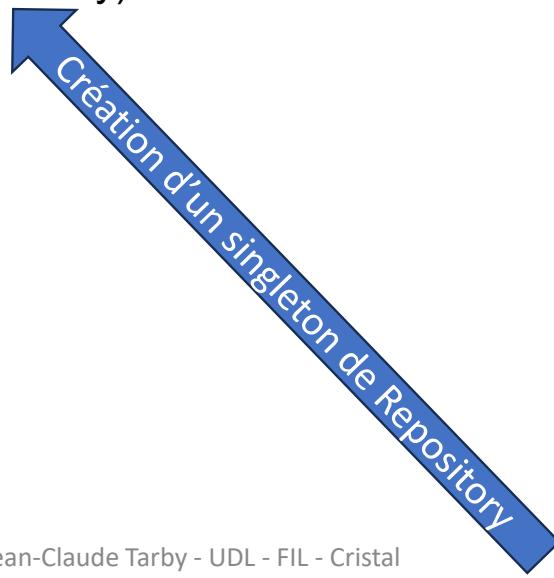
        if (modelClass.isAssignableFrom(PersonListViewModel::class.java)) {
            return PersonListViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

démo

Ne vous préoccupez pas du code, utilisez-le directement

Création de ViewModel : solution 5.2

```
// Solution 5.2.  
// Idem que 5.1, mais Repository en Singleton, même si rotation d'écran...  
val viewModel: PersonListViewModel by viewModels {  
    PersonListViewModelFactory(Repository)  
}
```



Création du Repository

```
// pour solution 5.2 et 5.3
// Le Repository est maintenant un Singleton 'à la Kotlin', c-à-d un object et non plus une classe.
object Repository {
    init {
        Log.d("ViewModel", "création du Repository this")
    }

    fun initialise(uneChaine: String): Repository {
        Log.d("ViewModel", "je suis LE Repository this, et je suis initialisé avec " + uneChaine)
        return this
    }
}
```

Création de la factory

```
// pour solution 5.1 et 5.2 uniquement
class PersonListViewModelFactory(private val repository: Repository) : ViewModelProvider.Factory {
    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {

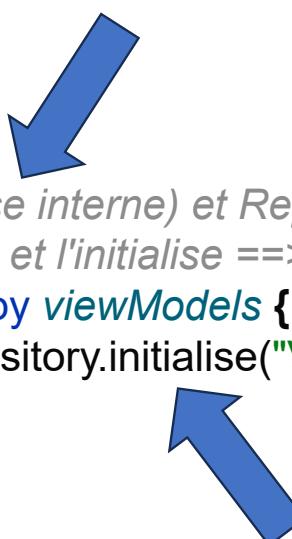
        if (modelClass.isAssignableFrom(PersonListViewModel::class.java)) {
            return PersonListViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

démo

Le même code que précédemment...

Création de ViewModel : solution 5.3

```
// Solution 5.3.  
// Code + actuel pour les Factory (en classe interne) et Repository en Singleton.  
// Ici, le ViewModel instancie le Repository et l'initialise ==> couplage fort ! -> solution : injection de dépendances  
val viewModel: PersonListViewModel by viewModels {  
    PersonListViewModel.Factory(Repository.initialise("ViewModel"))  
}
```



Le ViewModel paramétré

```
class PersonListViewModel(private val repository: Repository) : ViewModel() {  
    // Utilisation de MutableLiveData  
    private var _persons = mutableStateList<Person>()  
  
    // Exposer la liste en tant que StateList  
    val persons: StateList<Person> get() = _persons  
  
    // pour solution 5.3 uniquement  
    // Intégration de la Factory dans la classe associée (recommandé par Google,  
    // cf. https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-factories?hl=fr#creationextras)  
  
    class Factory(private val repository: Repository) : ViewModelProvider.Factory {  
        @Suppress("UNCHECKED_CAST")  
        override fun <T : ViewModel> create(modelClass: Class<T>): T {  
            if (modelClass.isAssignableFrom(PersonListViewModel::class.java)) {  
                return PersonListViewModel(repository) as T  
            }  
            throw IllegalArgumentException("Unknown ViewModel class")  
        }  
    }  
}  
Même code que  
précédemment auquel on  
ajoute le code ci-contre.  
La précédente classe  
Factory est obsolète.  
  
_persons.value = mutableListOf(  
    Person(1, "Alice"),  
    Person(2, "Bob"),  
    Person(3, "Charlie")  
)  
}  
}  
```

ViewModel

Même code que précédemment auquel on ajoute le code ci-contre. La précédente classe Factory est obsolète.

// pour solution 5.3 uniquement

// Intégration de la Factory dans la classe associée (recommandé par Google, cf. <https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-factories?hl=fr#creationextras>)

class Factory(private val repository: Repository) : ViewModelProvider.Factory {

@Suppress("UNCHECKED_CAST")

override fun <T : ViewModel> create(modelClass: Class<T>): T {

if (modelClass.isAssignableFrom(PersonListViewModel::class.java)) {

return PersonListViewModel(repository) as T

}

throw IllegalArgumentException("Unknown ViewModel class")

}

}

}

MI MO - 01 MAI 2025 2020

Jean-Claude Ro

Consultez le code à l'occasion

- <https://gitlab.com/m1-ue-tac/compose/creation-de-viewmodel>

RAPPEL ! Nous ne reverrons pas ces exemples de cours en TD.

➔ Travaillez-les chez vous pour qu'on attaque directement le TD.