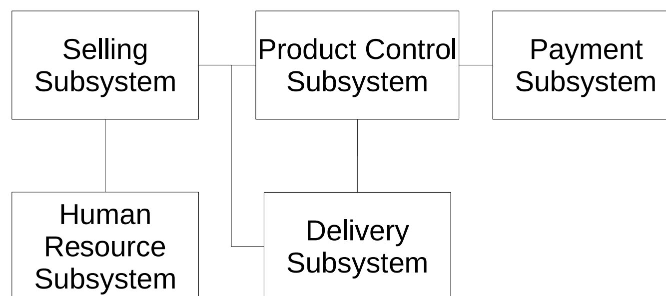
Chapitre 4 : Patrons d'architecture

« L'informatique n'est pas plus la science des ordinateurs que l'astronomie n'est celle des télescopes. » ***M. R. Fellows et I. Parberry***

1. Définition

La notion de patrons d'architecture se réfère à l'organisation structurelle de l'application. Elle est d'un niveau d'abstraction supérieur au patron de conception. L'architecture est la première préoccupation de la conception d'une solution logicielle. Elle définit le framework du logiciel. Voici un exemple d'une architecture. On peut y voir les différents composants et leurs relations :



1.1. Préoccupations

Pour élaborer son logiciel, l'architecte logiciel doit se poser les questions suivantes :

- Le système est-il interactif ?
- Nécessite-t-il de fréquents changements ?
- Le système est-il réparti sur le réseau ?
- Comment les fonctionnalités sont-elles décomposées entre les composants ?
- Y a-t-il une architecture générale à utiliser ?
- Quelles exigences non fonctionnelles sont importantes ?
    - La limitation des performances du matériel.
    - La haute disponibilité : les composants critiques doivent être tolérants aux pannes ou redondants.
    - La minimisation des risques d'entreprise (eg, non conformité à la législation, risques économiques et financiers, perte de compétitivité).
    - Sécurité : des fonctionnalités privées doivent être cachées, par exemple dans des couches internes.

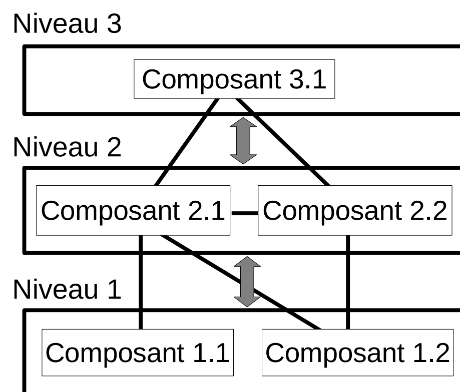
Ces questions vont permettre d'élaborer l'architecture du logiciel. Les enjeux de l'architecture sont les mêmes que pour la COO :

- Développabilité
- Maintenabilité
- Testabilité
- Réutilisabilité

Nous présentons ici les patrons d'architecture les plus utilisés.

2. Architecture en étages (n-tier)

- Organisation verticale



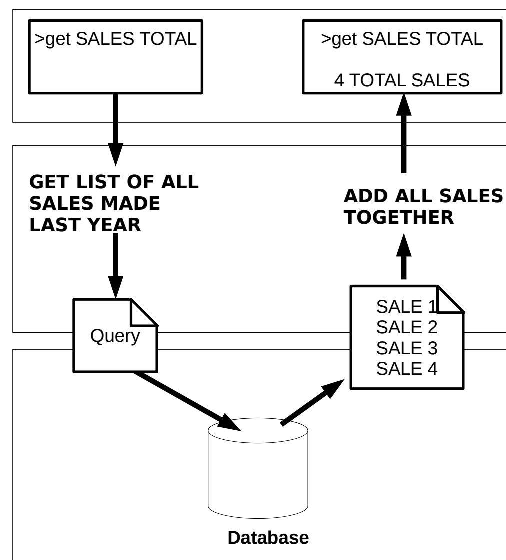
- Quand l'utiliser ?
  - Il y a ainsi plusieurs niveaux d'abstraction dans les responsabilités. Les couches supérieures correspondent aux interactions avec les utilisateurs. Les couches basses répondent aux requêtes.
- Exemple : *modèle OSI avec ses 7 couches.*

2.1. Caractéristiques

- Recommandation
  - Un étage est formé de composants **réutilisables dans les mêmes conditions**.
  - Les relations d'un étage à un autre sont décrites par des **interfaces**.
  - Aucun composant ne peut s'étaler sur deux étages. Le but est de créer des pare-feux contre les évolutions des étages en relation.
  - Les échanges sont limités entre deux étages consécutives.
- Avantages
  - Étages réutilisables et interchangeables.
  - Dépendances uniquement locales entre deux étages consécutifs.
  - Les développeurs et utilisateurs de chaque étage peuvent ignorer les autres. Tout passe par les interfaces.

2.2. Cas particulier d’une architecture Trois étages (Three-tier)

- C’est un cas particulier du n-tier avec des étages à responsabilités bien ciblées :
  - Niveau **présentation** (interface utilisateur)
    - Visualise les données.
  - Niveau **logique** (règles métier)
    - Coordonne les décisions et évaluations logiques, et effectue les calculs.
  - Niveau **données** (ORM)
    - Les données sont stockées et extraites de bases de données.



- Quand l’utiliser ?
  - Architecture de type client-serveur présentant une interface utilisateur.
- Exemple : *sites Web de commerce électronique*

2.3. Cas particulier d’une architecture deux étages (Réflexion (Reflection))

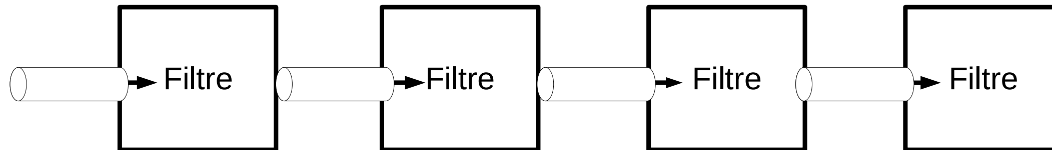
- Architecture avec les 2 étages suivants :
  - Niveau logique
    - Présentation des données aux utilisateurs.
  - Niveau physique
    - Implémentation des données.
- Quand l’utiliser ?
  - Pour des raisons d’efficacité, la représentation des données diffère de la logique métier.
  - Il n’y a pas ici de notion de hiérarchisation, mais une recherche d’efficacité.
- Exemple : *Système de gestion de bases de données*

- Recommandation
  - Nécessite une forte correspondance entre les niveaux.
  - La correspondance passe par des interfaces.
- Avantages

- Présente aux utilisateurs une interface conforme à la logique métier.
- Utilise la meilleure représentation des données.
- Cache la complexité de la représentation physique aux utilisateurs.

3. Tubes et Filtres (*Pipes & Filters*)

- Organisation horizontale



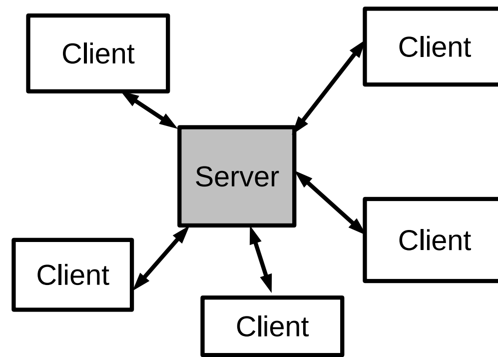
- Quand l'utiliser ?
    - Les fonctionnalités procèdent par traitement des données en flux. Les données passent d'un filtre à un autre par des tubes. Chaque étape de traitement est encapsulée dans un composant filtre.
- Exemples : *Compilateur*, *Chaîne de traitement d'images*

3.1. Caractéristiques

- Recommandation
    - Les tubes sont classiquement des pipes ou des fichiers.
- Caractéristiques
    - Il n'y a plus de relation hiérarchique.
    - Le comportement général du système est une succession de comportements individuels.
- Avantages
    - Les contraintes de communication sont limitées à deux composants.
    - Il est facile de remplacer un filtre.
    - Il est facile d'implanter la concurrence entre filtres.

4. Client-Serveur (Broker)

- Organisation centralisée



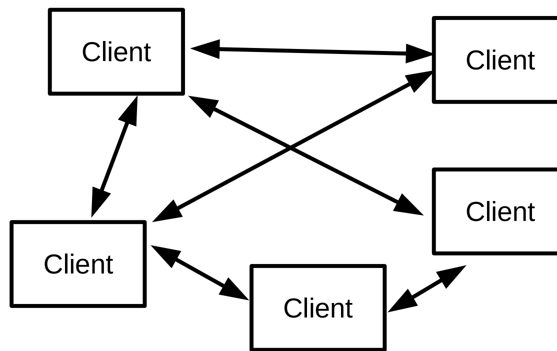
- Quand l'utiliser ?
    - Système distribué qui interagit par invocation de services localisés. Le serveur est un fournisseur de services et de ressources. Il est généralement composé d'une base de données. Le client interagit avec l'utilisateur et formule des requêtes au serveur.
- Exemple : *Guichet Automatique Bancaire*

4.1. Caractéristiques

- Variations
    - Clients lourds / Clients légers. Le client le plus léger ne fait que l'affichage des résultats de requête retournés par le serveur.
    - Multi-serveurs : un Broker (responsable de la communication) fait l'intermédiaire pour localiser le serveur qui possède le service. Les clients envoient les requêtes au broker qui les redirigent vers le bon serveur.
    - Le serveur et les clients sont généralement distribués sur des nœuds différents et la communication utilise le réseau mais tout peut très bien être localisé sur un même nœud et la communication utilise la mémoire.
- Avantages
    - Indépendance des composants pour le fonctionnement et le développement.
        - Côté serveur / Côté client.
    - Centraliser et sécuriser les accès aux données.
- Inconvénients
    - Faible tolérance aux erreurs. Tout repose sur la disponibilité du serveur.
    - Sensible à la montée en charge.
    - Difficile à tester.

5. Pair-à-Pair (*Peer to peer*)

- Organisation décentralisée



- Quand l'utiliser ?

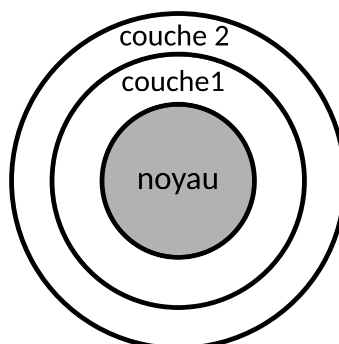
  Constat : plus une donnée est demandée moins elle est disponible avec une architecture client-serveur. L'architecture pair-à-pair inverse la tendance : chaque client qui a récupéré la donnée devient un serveur. Un « broker » se charge du routage.
- Exemple : *BitTorrent*

5.1. Caractéristiques

- Recommandations
  - Si la donnée est volumineuse, elle est divisée en segments. Un nœud du réseau distribue alors une liste de ces segments.
- Avantages
  - Grande tolérance aux fautes.
  - Rapidité de communication.
  - Autorise le parallélisme.
  - Plus une donnée est demandée plus elle est disponible.

6. Micro-noyau (Microkernel)

- Organisation en couches



  - Noyau : Il encapsule tous les services fondamentaux.
  - Services externes : ils utilisent les couches inférieures comme interface.
- Quand l'utiliser ?

    - Le système doit être adaptable au remplacement ou au changement de l'environnement d'accueil (p. ex. architecture matériel, OS).
- Exemples : *Windows NT*, *JVM*

6.1. Caractéristiques

- Recommandation
    - Minimiser la taille du noyau et déporter la plupart du code vers les couches externes.
- Avantages
    - Extensibilité.
    - Portabilité : seul le noyau a besoin d'être changé.

6.2. Microservices

- Dans la même veine, le microservice est un style d'architecture logicielle à partir duquel un ensemble complexe d'applications est décomposé en plusieurs processus indépendants et faiblement couplés, souvent spécialisés dans une seule tâche.
- Les processus indépendants communiquent les uns avec les autres en utilisant des API indépendantes du langage de programmation.

7. Entité – Composant – Système

Cette architecture induit un nouveau paradigme : la programmation orientée données. Elle passe d'une représentation par un tableau de structures à une structure de tableaux.

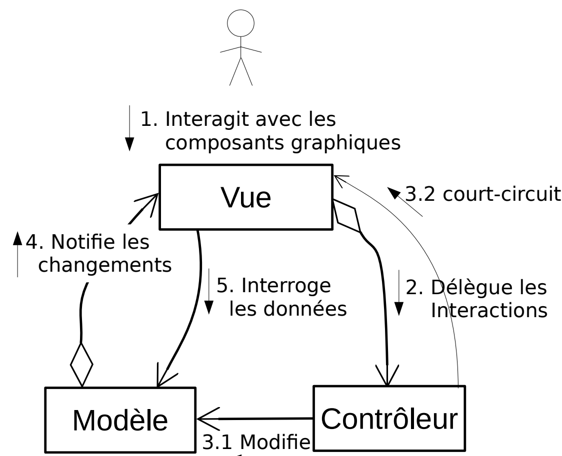
- Entité : la référence à un objet métier. Les implémentations utilisent généralement un entier simple pour cela.
- Composant : les données brutes pour un aspect de l'objet, et comment il interagit avec le monde (eg. la position). Les implémentations utilisent généralement des structures, des classes ou des tableaux associatifs.
- Système : les algorithmes portant sur les composants. Chaque système fonctionne en continu (comme si chaque système avait son propre thread privé) et effectue des actions globales sur chaque entité qui possède un composant du même aspect que ce système.

Organisation : globalement un tableau où les lignes sont les identificateurs, les colonnes les systèmes et l'intersection un composant.

- Quand l'utiliser ?
    - Quand la vitesse d'accès aux données est une limite. Cette architecture supprime la notion d'associations entre les objets.
- Exemple d'utilisation : *moteur de jeux vidéos* où les entités sont par exemple, les combattants ou les murs.

8. Modèle-Vue-Contrôleur (MVC)

- Organisation en boucle



- **Modèle : backend**
    - Stocke les données et l'état de l'interface.
    - Détient le modèle du domaine et la logique métier.
- **Vue : front end**
    - Affiche les informations aux utilisateurs.
    - Détient la logique de présentation.
- **Contrôleur : front end**
    - Gère les interactions avec les utilisateurs.
- Quand
    - Une interface graphique multi-fenêtrée (GUI).

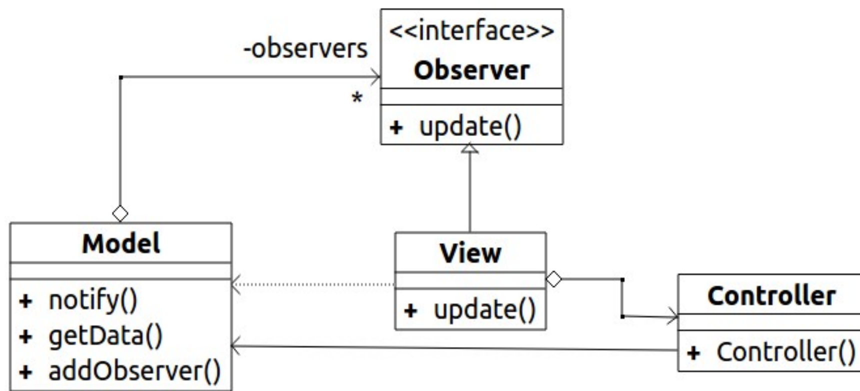
8.1. Les logiques dans une interface graphique

- Logique métier
    - Code de l'application qui crée, stocke et modifie les données et qui leur donne un sens. Elle est spécifique de l'application.
- Logique de présentation
    - Code relatif à la façon de réagir aux interactions avec l'utilisateur et de présenter les informations.
- Exemple
    - Soit une application qui gère l'évolution d'une température ambiante. Une règle indique que si la valeur de la température dépasse un certain seuil alors elle doit être affichée en couleur rouge sinon en noir. Nous divisons cette règle en trois parties :
        - 1/ Si la valeur dépasse un certain seuil, elle est trop élevée.
        - 2/ Si elle est trop élevée, elle devrait être affichée de manière spéciale.
        - 3/ Pour marquer une valeur spéciale, afficher en rouge sinon en noir.
    - La première partie se réfère à la logique métier. Une classe du domaine représente et gère la valeur.
    - La deuxième règle correspond à la logique de présentation. La présentation sait du

modèle quand la valeur est trop élevée et, par conséquent, la transmet à la représentation graphique avec une information indiquant qu'elle est spéciale.
- C'est le travail de la représentation graphique que de traduire cette information par une couleur comme l'indique la troisième règle. Cela pourrait être autre chose qu'une couleur, comme un signe à côté de la valeur. Tout dépend du choix du concepteur et des possibilités de la bibliothèque graphique support (GUI framework). Cela correspond à des choix de représentation graphique et il n'y a pas de logique à ce niveau.

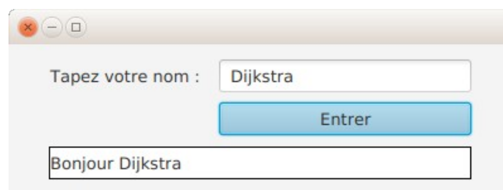
8.2. Implémentation

- Découpler les vues du modèle : Observateur. Il faut ajouter un protocole de souscription / notification pour les vues au modèle.

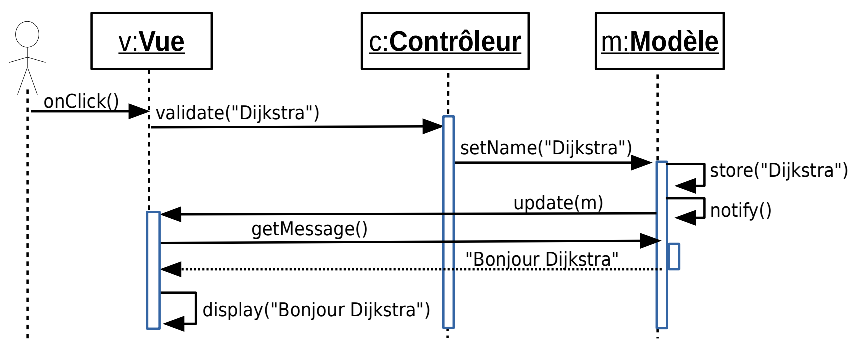


8.3. Exemple d'un écran de connexion

L'utilisateur renseigne le nom puis appuie sur le bouton « Entrer » et l'écran affiche un message de bienvenue.



Le diagramme de séquence résultant de l'implémentation avec une architecture MVC :

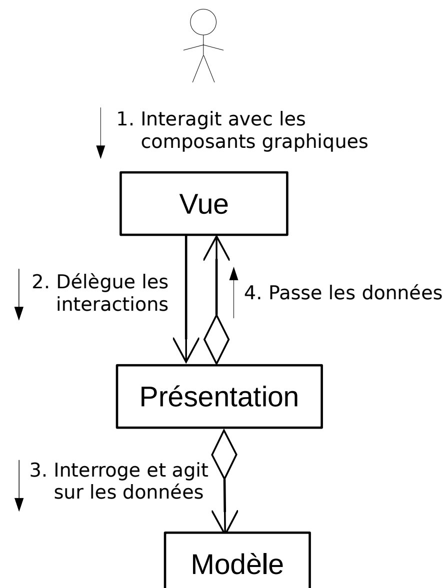


8.4. Caractéristiques

- Avantages
    - Découplage entre les vues et le modèle, qui peuvent évoluer indépendamment.
    - Plusieurs vues sur le même modèle.
    - Une vue peut être ajoutée facilement.
- Inconvénients
    - Mauvaise séparation des responsabilités. La vue a 2 responsabilités : affichage et récupération des données.
        - Qui détient l'état courant de l'interface : contrôleur, modèle ou vue ? De plus, le fait que l'état courant du dialogue avec l'utilisateur soit stocké dans le modèle rend ce modèle dépendant des problèmes d'interfaçage ce qui est contraire à l'ambition initiale du découplage. Prenons l'exemple d'un boîte de texte où l'utilisateur peut sélectionner une partie du texte. Qui garde cette information ? La vue ? Elle l'utilise pour afficher en surligné la sélection ou le point d'insertion. Le contrôleur : il l'utilise si l'on veut faire une insertion ou une complétion. Ou le modèle : le problème est que la sélection est temporaire. Que fait-on si on a plusieurs vues ?
    - Difficilement testable (inter-dépendance des unités).
- Conclusion
    - Ce patron est peu utilisé en pratique pour les GUI. On lui préfère des patrons dérivés : MVP et MVVC.

9. Modèle-Vue-Présentation (MVP)

- Retour à une organisation en étages : cette architecture permet de réduit le couplage Vue – Modèle.

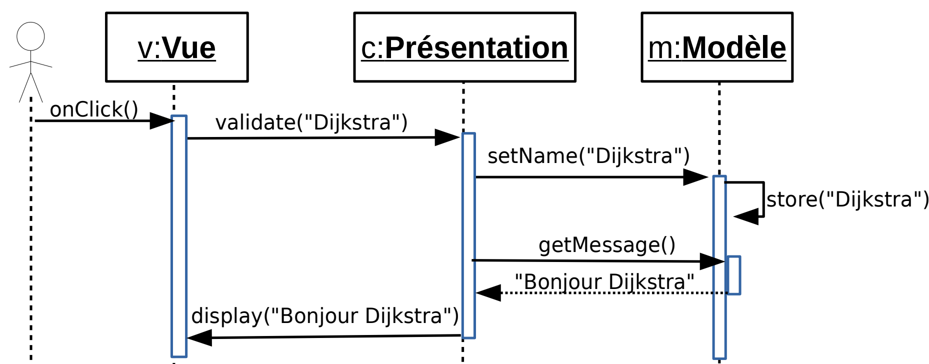


    - Modèle
        - Il est centré sur la logique métier. Il n'y a aucun lien avec les autres parties

(différent du MVC où il y a un lien avec les vues).
    - Il contient le modèle du domaine et la logique métier.
    - Il stocke les données.
- Vue
    - La vue ne concerne que la gestion graphique. En particulier, c'est la seule partie à inclure les paquets de la bibliothèque graphique. Elle ne contient aucune logique interne.
- Présentation
    - Elle contient la logique de présentation. Elle n'a aucun lien avec la bibliothèque graphique (aucun import). Elle réagit aux événements sur les vues en modifiant les données et en répercutant les effets sur les vues Elle détient l'état de l'interface et la logique de présentation.

9.1. Exemple de l'écran de connexion

Le diagramme de séquence avec une architecture MVP est cette fois :



9.2. Caractéristiques

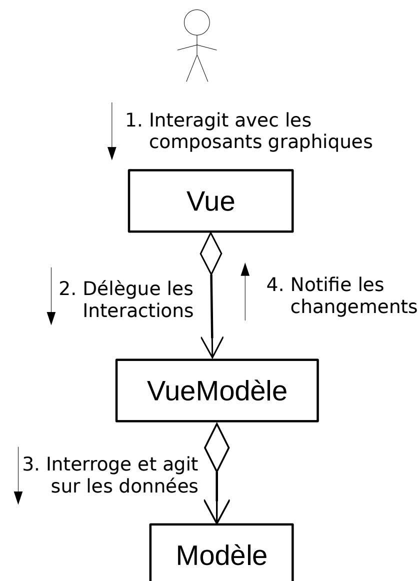
- Avantages
    - Élimine l'interaction entre la vue et le modèle.
    - L'interaction est faite par le biais de la présentation, qui organise les données à afficher dans la vue.
    - Tout est testable sauf la vue, mais elle est dépourvue de logique.
    - Le modèle est indépendant des autres composants.
- Inconvénients
    - Beaucoup de code redondant (boilerplate code) dans la présentation pour mettre à jour les données dans la vue, du genre :

```
• displayText1(text), displayText2(text)
• setButton1Enabled(true), setButton2Enabled(true).
```

10. Modèle-Vue-VueModèle (MVVM)

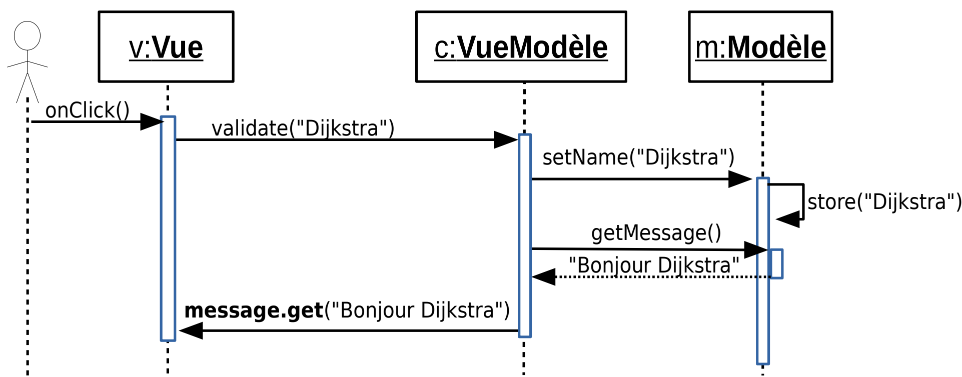
C'est encore une organisation en étages. Elle vise à supprimer le code redondant dans la présentation. Pour cela, l'unique différence avec MVP réside dans la mise à jour de la vue qui est faite par un mécanisme de liaison (binder) entre les composants de la vue et les attributs de vue modèle. Quand un attribut de la présentation est modifié, le composant de la vue est mis à jour automatiquement.
Le binder est réalisé avec le patron Observateur de MVC qui est réintroduit mais au niveau de chaque attribut. Exemple de « binding » en JavaFX :

```
label.textProperty().bindBidirectional(viewModel.input);
circle.scaleXProperty().bind(slider.valueProperty());
```



10.1. Exemple de l'écran de connexion

Le digramme de séquence pour une architecture MVVM est :



- La méthode message.get() est appelée par le mécanisme de liaison (binder). Le label de Vue qui affiche le message est lié à l'attribut message de VueModèle.

10.2. Caractéristiques

- Avantages
    - Exactement les mêmes que pour le patron MVP.
    - Élimination du surcoût de code redondant (*boilerplate code*).
- Inconvénients
    - Généraliser le modèle de vue pour de grande application est difficile.
    - La liaison de données dans de grandes applications peut aboutir à une consommation considérable de ressources.
    - À n'utiliser que si la bibliothèque graphique permet ces mécanismes de *binder*.
- Modèle de base pour .NET (avec XAML)

11. Conclusion

- L'architecture générale doit être choisie très tôt.
    - Premiers temps de l'analyse.
- Elle doit être relativement stable.
    - Elle définit l'organisation du projet.
- Mais elle évoluera pour s'adapter aux particularités de l'application en cours de conception.
    - Une bonne architecture doit permettre de différer les décisions majeures le plus tard possible.
- Utiliser des interfaces comme des pare-feux entre les composants de l'architecture.
- Plusieurs architectures peuvent être combinées pour un même projet.