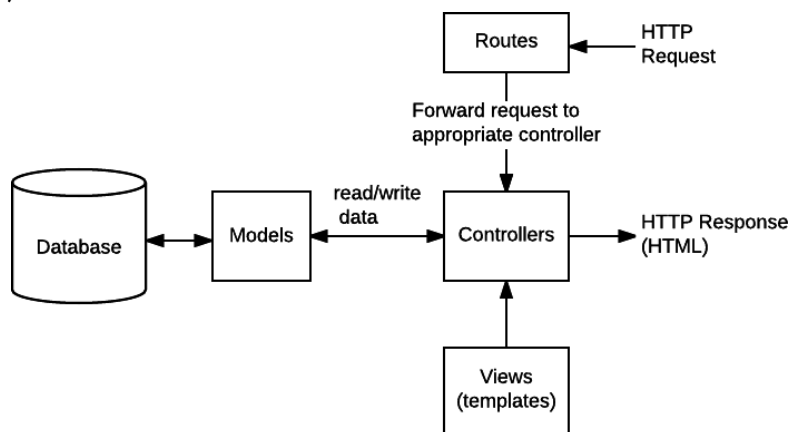


Chapitre 4 : Patrons d'architecture

« 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 à la notion de patron de conception. L'architecture est la première préoccupation de la conception d'une solution logicielle. Elle définit le framework du logiciel. On peut y voir les différents systèmes et leurs relations, comme dans l'exemple ci-dessous (une architecture typique de NodeJ.js) :



Les enjeux de l'architecture sont les mêmes que pour la conception orientée objet :

- développabilité,
- maintenabilité,
- testabilité,
- réutilisabilité..

2. « Clean Architecture »

Une architecture logicielle épurée repose sur plusieurs principes fondamentaux visant à **limiter la complexité, réduire le couplage et faciliter l'évolution du système**.

2.1. Séparer les différentes responsabilités

Les différentes préoccupations d'une application doivent être séparées en **composants indépendants** :

- **Objets métier** : représentation des concepts du domaine.
- **Règles métier** : logique et contraintes propres au domaine.

- **Interactions** : échanges avec les utilisateurs ou avec d'autres systèmes.
- **Infrastructure** : interactions avec les outils et frameworks externes, tels que les bases de données, les capteurs ou les services tiers.

Cette séparation correspond à l'application du **principe de responsabilité unique** à l'échelle de l'architecture : chaque composant doit avoir une responsabilité clairement définie.

2.2. Réduire le couplage

Les dépendances entre les composants doivent être organisées de manière à préserver la stabilité du système.

Les composants **stables**, notamment ceux qui portent les règles métier, ne doivent pas dépendre directement de composants **évolutifs ou externes**, tels que les interfaces utilisateur, les bases de données ou les frameworks.

Les dépendances doivent donc être orientées **vers les composants les plus stables**, et non l'inverse.

Il s'agit de l'application du **principe d'inversion des dépendances** à l'échelle de l'architecture.

2.3. Protéger l'isolation des composants

L'isolation entre les composants peut être assurée à l'aide **d'interfaces ou de ports**.

Ces interfaces jouent un rôle comparable à celui de **pare-feux** : elles limitent la propagation des modifications d'un composant vers les autres parties du système.

Ainsi, il devient possible de faire évoluer ou de remplacer un composant sans remettre en cause l'ensemble de l'application.

2.4. Une architecture nécessite de la discipline

Une architecture propre demande une certaine **discipline de développement** : les règles d'architecture doivent être respectées par l'ensemble de l'équipe.

Cet effort peut sembler important au début du projet, mais il constitue un **investissement** qui permet ensuite de maintenir plus facilement le système et de limiter le coût des évolutions.

3. Choix d'une architecture

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

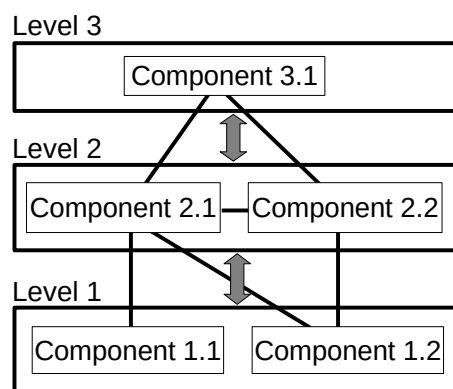
- Le système est-il interactif ?
- Nécessite-t-il de fréquents changements ?
- Le système est-il réparti sur le réseau ?
- Comment les fonctionnalités sont-elles décomposées entre les composants ?
- Y a-t-il une architecture générale à utiliser ?
- Quelles exigences non fonctionnelles sont importantes ?
 - Limitation des performances du matériel : mémoire, puissance de calcul, capacité de communication.

- Haute disponibilité (ie, taux de disponibilité d'un service).
- Minimisation des risques d'entreprise (eg, non conformité à la législation, risques économiques et financiers, perte de compétitivité).
- Sécurité : des fonctionnalités privées doivent-elles être cachées, par exemple dans des couches internes.

Ces questions vont permettre de choisir un patron d'architecture afin d'élaborer l'architecture du logiciel. Nous présentons dans ce chapitre les patrons d'architecture générale les plus utilisés. Il faut noter que la réalisation d'un logiciel peut mixer plusieurs architectures et sous-architectures.

4. Architecture en étages (N-tier)

4.1. Une organisation verticale



4.2. Quand l'utiliser ?

Il existe plusieurs niveaux d'abstraction dans les responsabilités. Les couches supérieures correspondent aux interactions avec les utilisateurs. Les couches basses se chargent de la construction de la réponse aux requêtes de l'utilisateur.

Exemple : *modèle OSI avec ses 7 couches*.

4.3. Caractéristiques

Recommandation d'implémentation

- 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 consécutifs.

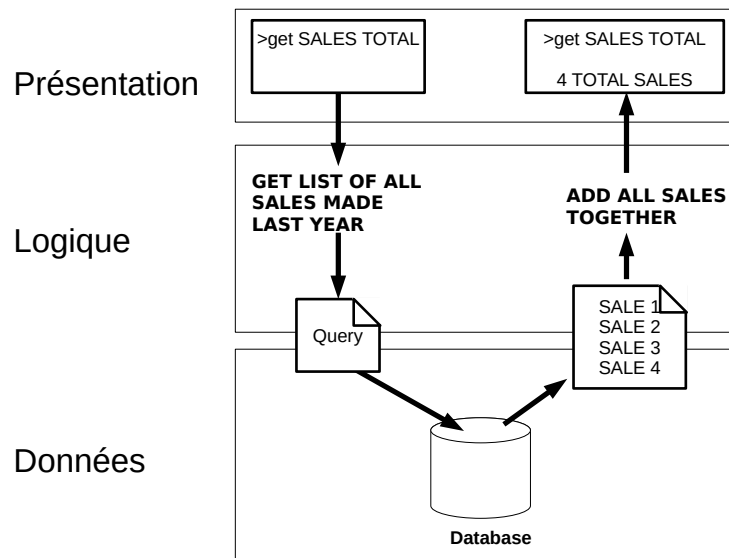
Avantages

- Étages réutilisables et interchangeables.
- Dépendances uniquement locales entre deux étages consécutifs.
- Les développeurs et utilisateurs de chaque étage peuvent s'ignorer les uns les autres. Tout passe par les interfaces.

4.4. Exemple d'une architecture à trois étages (*Three-tier*)

Le cas typique est un site web de commerce électronique. C'est un logiciel de type client-serveur présentant une interface utilisateur. L'architecture est :

- Niveau 3 : **présentation** (interface utilisateur)
 - Visualise les données.
- Niveau 2 : **logique** (règles métier)
 - Coordonne les décisions et les évaluations logiques, et effectue les calculs.
- Niveau 1 : **données** (ORM)
 - Les données sont stockées et extraites de bases de données.



4.5. Exemple d'une architecture deux étages (*Reflection*)

L'exemple typique est celui des systèmes de gestion de bases de données où l'on distingue le niveau physique dans lequel les données sont codées sous forme d'enregistrement du niveau logique dans lequel ces mêmes données sont présentées sous la forme de table.

Pour des raisons d'efficacité, la représentation des données diffère de la présentation qui en est faite à l'utilisateur. Il n'y a pas ici de notion de hiérarchisation, mais une recherche d'efficacité.

L'architecture distingue deux étages avec une sémantique bien précise :

- Niveau 2 : **logique**
 - Présentation des données aux utilisateurs.
- Niveau 1 : **physique**
 - Implémentation des données.

4.5.1. Caractéristiques

Recommandation d'implémentation

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

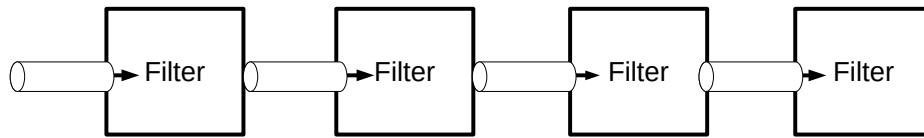
Avantages

- Présente aux utilisateurs une interface conforme à la logique métier.

- Utilise la meilleure représentation interne des données.
- Cache la complexité de la représentation physique aux utilisateurs.

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

5.1. Une organisation horizontale



5.2. 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*

5.3. Caractéristiques

- Il n'y a plus de relation hiérarchique.
- Le comportement général du système est une succession de comportements individuels.

Avantages

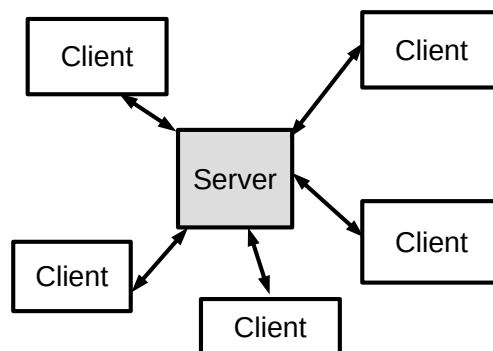
- Les contraintes de communication sont limitées à deux composants.
- Il est facile de remplacer un filtre.
- Il est facile d'implanter la concurrence entre filtres. L'entrée d'un filtre peut alors être choisie comme la sortie du filtre précédent la plus rapide ou la meilleure au sens d'un critère.

Recommandation d'implémentation

- Les tubes sont classiquement des « pipes » ou des fichiers.
- Le lien entre deux filtres passe par des interfaces.

6. Client-Serveur (Broker)

6.1. Une organisation centralisée



6.2. Quand l'utiliser ?

Le cas typique est celui d'un logiciel 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 avec des terminaux bancaires et un serveur centralisé.*

6.3. Caractéristiques

Avantages

- Indépendance des composants pour le fonctionnement et le développement, avec un côté serveur et un côté client.
- Centralisation et sécurité de l'accès aux données.

Inconvénients

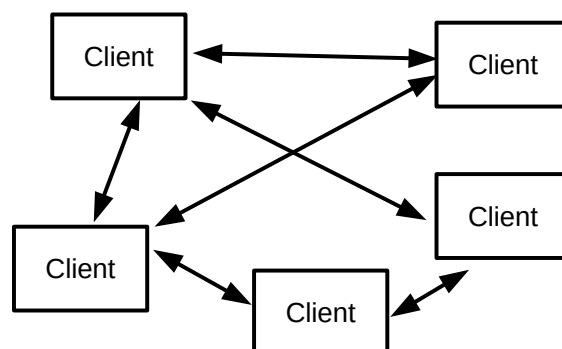
- Faible tolérance aux erreurs. Tout repose sur la disponibilité du serveur.
- Sensible à la montée en charge.
- Difficile à tester.

6.4. Variations

- Clients lourds / Clients légers. Le client le plus léger ne fait que l'affichage des résultats 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.

7. Pair-à-Pair (*Peer to peer*)

7.1. Une organisation décentralisée



7.2. Quand l'utiliser ?

Avec une architecture client-serveur, plus une donnée est demandée moins elle est disponible. 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*

7.3. Caractéristiques

Avantages

- Grande tolérance aux fautes.
- Rapidité de communication.
- Autorise le parallélisme.
- Plus une donnée est demandée plus elle est disponible.

Inconvénients

- Difficile à tester.
- Maintenance distribuée
- Insécurité des données puisqu'elles sont distribuées chez le client

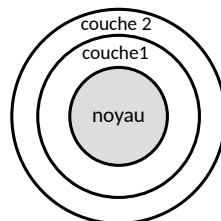
Recommandation d'implémentation

- Si la donnée est volumineuse, elle est divisée en segments. Un nœud du réseau distribue alors une liste de ces segments.

8. Micro-noyau (Microkernel)

8.1. Une organisation en couches

- Le noyau encapsule tous les services de base.
- Les couches contiennent des services de plus haut niveau qui sont bâtis sur les services des couches inférieures.



8.2. Quand l'utiliser ?

Le système doit être adaptable au remplacement ou au changement de l'environnement d'accueil tel que l'architecture matériel ou l'OS. Dans ce cas, seul le noyau est à changer.

Exemples : *Windows NT, JVM*

8.3. Caractéristiques

Recommandation d'implémentation

- Minimiser la taille du noyau et déporter la plupart du code vers les couches externes.
- Les services proposés par le noyau doivent être optimisés.

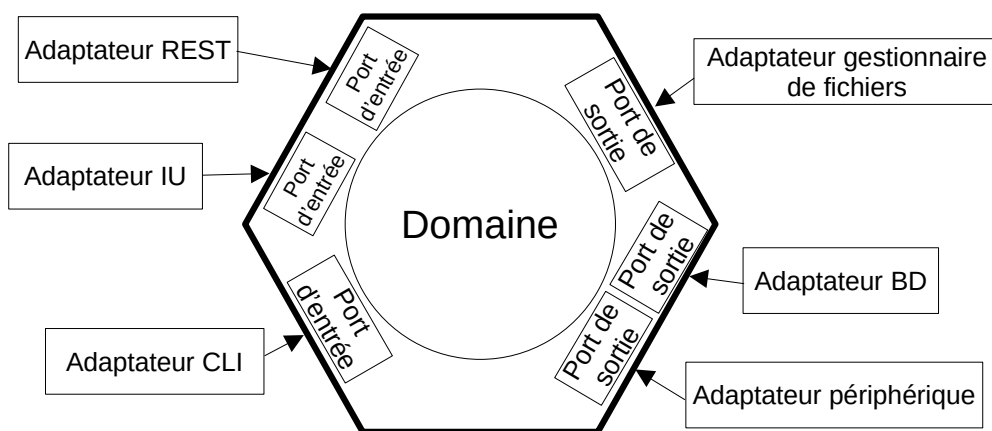
Avantages

- Extensibilité.
- Portabilité : seul le noyau a besoin d'être changé.

9. Architecture hexagonale¹ (*Ports and Adapters*)

9.1. Une organisation avec un cœur métier entouré de ports et d'adaptateurs

La logique métier est placée au cœur d'une application pour l'isoler des détails techniques et des dépendances externes. Les interactions vers l'extérieur passe par des ports. Les adaptateurs assurent la connexion avec les composants extérieurs. Ainsi, tout dépend du cœur et passe par le cœur. Rien d'autre ne dépend les uns des autres.



- Le **domaine** : logique métier pure de l'application. Il ne doit dépendre d'aucun framework, base de données ou interface utilisateur en particulier.
- Les **ports** : interfaces (ou contrats) qui définissent comment interagir avec le domaine (représenté concrètement par des interfaces Java). Il y a des ports d'entrée (généralement nommés API : *Application Programming Interface*), par exemple UI, CLI ou capteurs. Il y a des ports de sortie généralement nommés SPI (*Service Provider Interface*), par exemple base de données ou système de gestion de fichiers.
- Les **adaptateurs** : implémentations concrètes qui connectent le monde extérieur aux ports (le mot adaptateur correspond bien au sens du patron de conception).

Les ports se réfèrent essentiellement aux interfaces par lesquelles l'inversion de dépendance est mise en œuvre.

9.2. Quand l'utiliser ?

L'architecture hexagonale est particulièrement bien adaptée aux contextes suivants :

- Des domaines complexes et évolutifs : lorsque la logique métier est complexe ou change fréquemment, cette architecture permet de protéger le cœur des dépendances externes et

¹ Le terme « hexagonal » fait référence à la figure géométrique utilisée pour représenter le schéma de l'architecture, qui permet de connecter beaucoup d'adaptateurs. Il n'y a aucune autre raison à cette appellation.

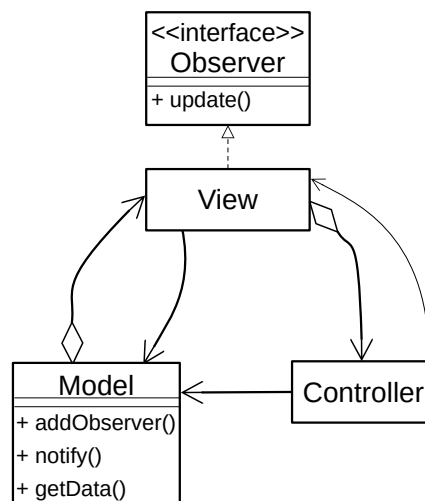
simplifie la maintenance à long terme.

- Exigences élevées en matière de testabilité : en isolant le domaine des frameworks et de l'infrastructure, il est plus facile d'écrire des tests automatisés robustes sans s'appuyer sur des bases de données ou des services tiers.
- Interfaces multiples ou points d'intégration : si la même logique de base doit être utilisée avec des API REST, des WebSockets, des outils CLI ou des tâches par lots, l'architecture hexagonale permet un développement d'adaptateur propre avec un minimum de duplication. On pense aussi ici à l'utilisation de micro-services.
- Projets de longue durée ou grandes équipes : la séparation claire des préoccupations favorise l'évolutivité de l'équipe et l'intégration en douceur des nouveaux développeurs.

Exemple classique : *site Web marchand*.

10. Modèle-Vue-Contrôleur (MVC)

10.1. Une organisation en boucle



Cette architecture est composée de trois entités :

- **Modèle :**
 - Il stocke les données et l'état de l'interface. Il détient le modèle du domaine et la logique métier. Par exemple, pour une application de banque, le modèle représente les comptes clients et les opérations de retrait et de dépôt. Il vérifie la conformité de ces opérations.
- **Vue :**
 - Elle affiche les données provenant du modèle aux utilisateurs. Elle détient la logique de présentation. Une vue peut être aussi simple qu'un bouton ou un formulaire voire aussi complexe qu'un écran combinant plusieurs fenêtres.
- **Contrôleur :**
 - Il gère les interactions avec les utilisateurs. C'est une partie organisationnelle de l'interface utilisateur qui présente et coordonne plusieurs vues à l'écran, et qui reçoit les entrées des utilisateurs et envoie les messages appropriés aux vues correspondantes.

10.2. Quand l'utiliser ?

L'utilisation typique est un logiciel disposant d'une interface graphique multi-fenêtrée (GUI).

10.3. Les logiques dans une interface graphique

On distingue deux logiques distinctes :

1. Logique métier : c'est le code de l'application qui crée, stocke et modifie les données et qui leur donne un sens. Elle est spécifique du domaine d'application.
2. Logique de présentation : le code relatif à la façon de réagir aux interactions avec l'utilisateur et de présenter les informations.

Prenons un exemple pour illustrer ces deux logiques :

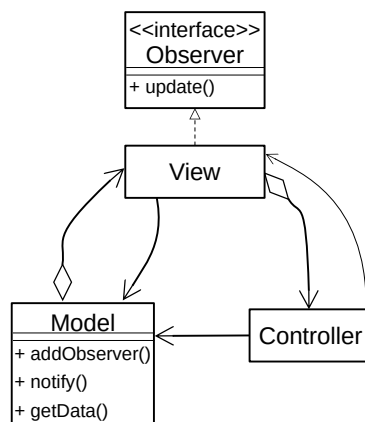
Soit une application qui gère l'évolution d'une température ambiante. Une règle indique que si la valeur de la température dépasse un certain seuil alors elle doit être affichée avec la couleur rouge sinon avec la couleur noire. Nous divisons cette règle en trois parties :

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, elle est affichée 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 partie correspond à la logique de présentation. La présentation sait du modèle quand la valeur est trop élevée et, par conséquent, elle transmet la valeur 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. Tout dépend des choix de représentation graphique et il n'y a pas de logique à ce niveau.

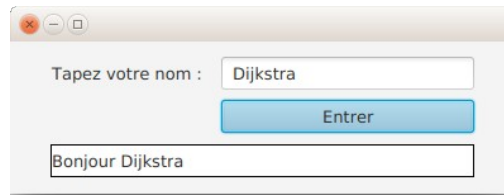
10.4. Implémentation

Pour découpler les vues du modèle, on utilise le patron de conception Observateur. Il faut ajouter un protocole de souscription / notification pour les vues au modèle.

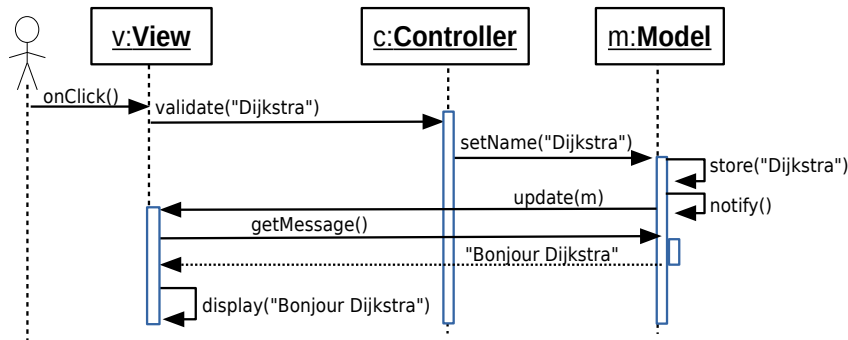


10.5. Exemple d'un écran de connexion

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



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



10.6. Caractéristiques

Avantages

- Découplage entre les vues et le modèle, qui peuvent ainsi évoluer indépendamment.
- Plusieurs vues peuvent être associées à un même modèle.
- Une vue peut être ajoutée facilement.

Inconvénients

- Cette architecture n'offre pas une bonne séparation des responsabilités.
 - La vue a 2 responsabilités : affichage et récupération des données, ce qui est contraire au principe de responsabilité unique.
 - Qui détient l'état courant de l'interface : contrôleur, modèle ou vue ? Prenons l'exemple d'une boîte de texte où l'utilisateur peut sélectionner une partie du texte. Qui garde l'information de la partie de texte sélectionnée ? La vue ? Elle l'utilise pour afficher en surligné la sélection ou le point d'insertion. Il faut donc faire passer cette sélection dans la boucle de contrôle à chaque utilisation ce qui est très peu efficace. Le contrôleur ? Il l'utilise si l'on veut faire une insertion ou une complétion. Que faire s'il y a plusieurs vues ? Le modèle ? On rend le modèle dépendant des problèmes d'interfaçage ce qui est contraire à l'ambition initiale du découplage. Que fait-on si on a plusieurs vues ? La répartition des responsabilités n'est donc pas claire dans ce patron.
- Difficilement testable à cause de l'inter-dépendance des unités.

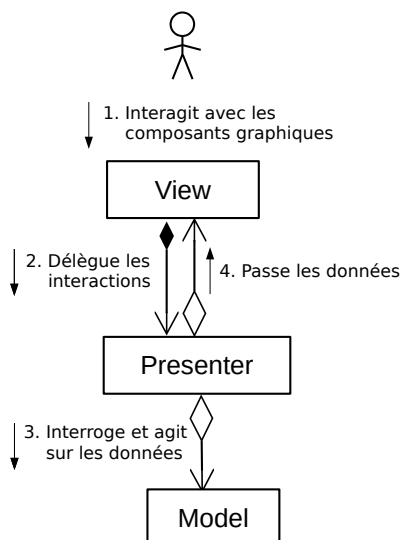
10.7. Conclusion sur le patron MVC

À cause de la mauvaise répartition des responsabilités et de la lourdeur de la boucle d'interaction, ce

patron est peu utilisé en pratique pour les GUI. On lui préfère des patrons organisés en étages comme MVP et MVVC.

11. Modèle-Vue-Présentation (MVP)

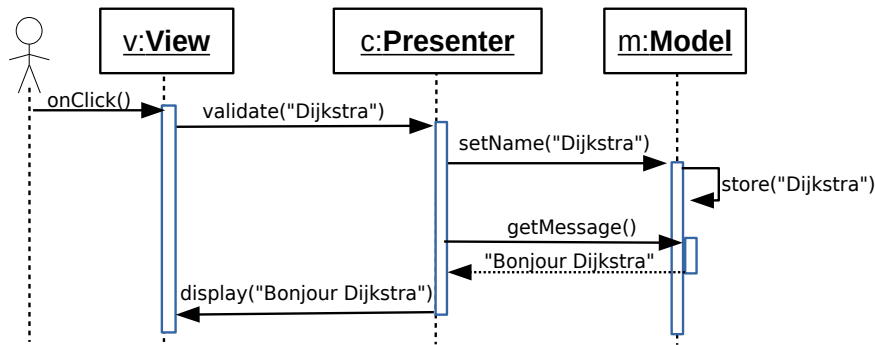
L'architecture MVP est une évolution de MVC. On revient à une organisation en 3 étages (3-tier). Le but est de supprimer le couplage entre la Vue et le Modèle.



- **Modèle**
 - Il est centré sur la logique métier. Il n'y a aucun lien avec les autres parties.
 - 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 importer 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 fait l'intermédiaire entre la vue et le modèle. 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. En général, c'est la vue qui crée la présentation et lui est associée par une interface.

11.1. Exemple de l'écran de connexion

En reprenant, l'exemple de l'écran de connexion, le diagramme de séquence est :



11.2. Caractéristiques

Avantages

- Cette architecture élimine l'interaction entre la vue et le modèle. L'interaction est faite par la présentation, qui organise les données à afficher dans la vue.
- Tout est testable sauf la vue. Mais, dépourvue de logique, la vue n'a pas besoin d'être testée.
- Le modèle est indépendant des autres composants.
- On peut mettre autant de présentateurs que de vues.

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(false).

```

12. Modèle-Vue-Vue Modèle (MVVM)

L'architecture MVVM est une variante de MVP qui 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 la présentation. Quand un attribut de la présentation est modifié, le composant de la vue qui lui est lié est mis à jour automatiquement. La présentation est maintenant nommée Vue-Modèle.

Le binder est réalisé avec le patron Observateur de MVC qui est réintroduit mais au niveau de chaque attribut.

Exemple de « binding » en JavaFX :

- Au niveau de la Vue, le composant graphique output est un label récupéré de la construction de l'interface à partir du fichier FXML :

```
@FXML private Label _output;
```

- Au niveau de la Vue-Modèle, on définit l'attribut message comme une propriété JavaFX de type String.

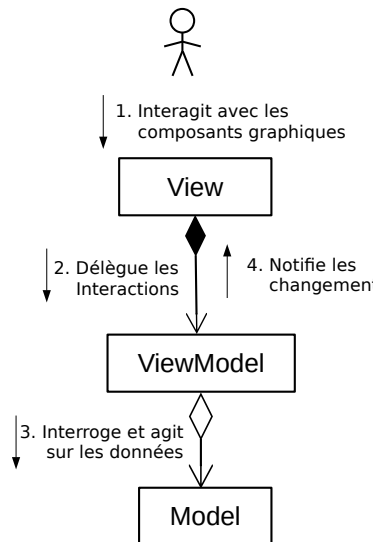
```
private StringProperty message = new SimpleStringProperty(this, "message");
```

- Le composant graphique de la vue est lié à la propriété de la Vue-Modèle ce qui fait qu'à chaque fois que l'on modifie la propriété message l'interface affiche automatiquement le nouveau message dans le label.

```
_output.textProperty().bindBidirectional(viewModel.message);
```

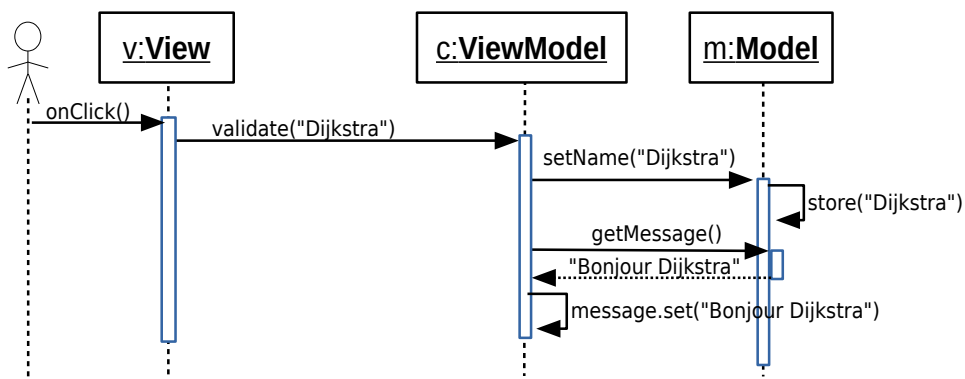
- Pour l'utiliser, il suffit de modifier la valeur de la propriété dans la Vue-Modèle pour afficher automatiquement le texte dans la vue :

```
message.set("un texte");
```



12.1. Exemple de l'écran de connexion

Le diagramme de séquence pour une architecture MVVM est :



La méthode `message.set()` utilise le mécanisme de liaison (binder) pour afficher automatiquement le message sur la vue.

12.2. Caractéristiques

Avantages

- Exactement les mêmes que pour le patron MVP.

- Par rapport à MVP, il y a élimination de la dépendance entre la Vue-Modèle et la Vue et donc élimination du surcoût de code redondant (*boilerplate code*).

Inconvénients

- Le lien entre la valeur d'un attribut et le composant graphique n'est pas toujours direct et peut nécessiter du calcul intermédiaire complexe qui n'est pas possible avec ce mécanisme. On peut donc être amené à remettre un lien entre la Vue-Modèle et la Vue.
- À n'utiliser que si la bibliothèque graphique permet ces mécanismes de *bind*. Sinon, il faut créer autant d'observateurs que de composants graphiques utilisés.

À noter que c'est l'architecture de base pour .NET (avec XAML).

12.3. Remarque générale

Il existe une réelle confusion entre ces trois modèles MVC, MVP et MVVM dans la documentation des frameworks pour les applications Web comme Spring, Symfony, Laravel, Ruby on Rails ou Django. Les documentations de ces frameworks indiquent qu'ils reposent sur une architecture MVC que l'on croirait donc en boucle, alors qu'en réalité, elles décrivent plutôt une architecture multi-étage type MVP, et pas une architecture en boucle type MVC originelle. La différence avec le patron MVP est qu'avec le patron MVC c'est le contrôleur qui crée la vue, alors qu'avec le patron MVP, c'est la vue qui crée le présentateur. Certaines documentations parlent alors de MVC-2 pour limiter la confusion.

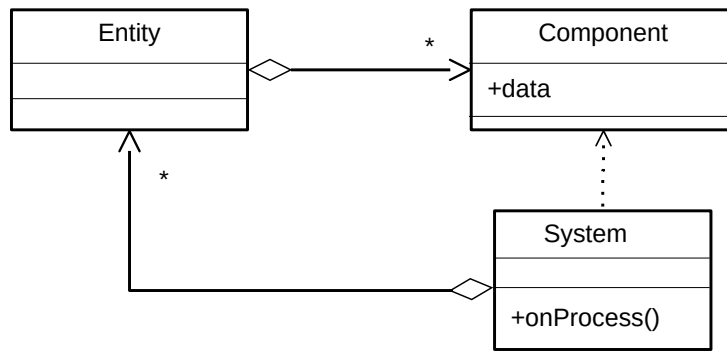
13. Entité - Composant - Système

La conception orientée objet facilite la conception par le fait qu'elle permet de décomposer un problème complexe en un ensemble de sous-problèmes simples. Mais elle est très consommatrice en temps d'exécution. Les objets ont tendance à être connectés avec beaucoup d'indirections (eg., associations, tables virtuelles pour les méthodes) et une distribution des objets dispersée sur la mémoire. Cela provoque un morcellement de la mémoire et des coûts d'accès élevés aux données.

Quand les problèmes de temps d'exécution sont cruciaux, l'architecture Entité-Composant-Système (ECS) propose de réarranger les objets en mémoire pour minimiser la localisation des données et produire du code qui utilise de larges bandes de mémoire. Par exemple, dans les jeux vidéos, on est amené à itérer sur un ensemble d'objets plusieurs fois par seconde, exécutant des méthodes sur eux à chaque fois. Un objet est chargé entièrement dans le cache alors que peu de ses données sont réellement utilisées à chaque instant, ce qui induit perte de temps et de place mémoire.

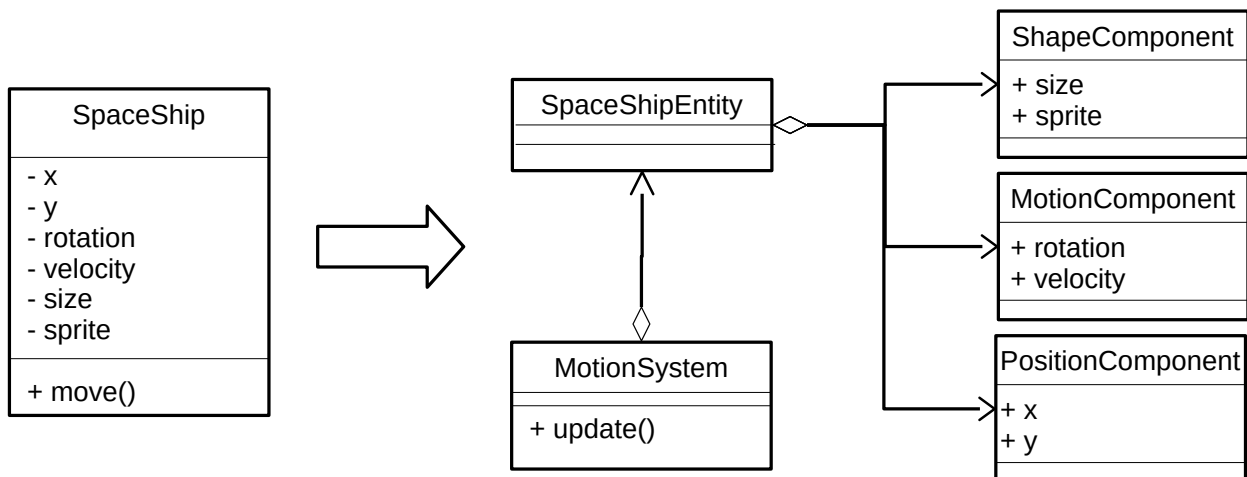
L'architecture ECS propose de complètement démanteler les classes et de supprimer la hiérarchie des objets métier. Les attributs et les méthodes sont éclatés dans des classes séparées qui ne sont plus, en réalité, que de pures structures. Ainsi, les données sont séparées de leur logique pour des raisons d'efficacité.

Cette architecture induit un nouveau paradigme : programmation orientée données (data-driven design).



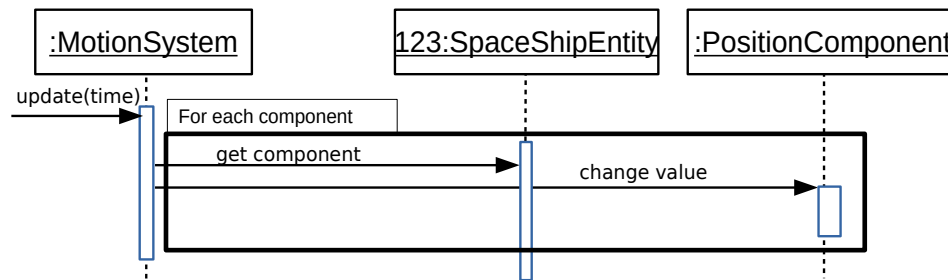
- Une **entité** référence un objet métier, c'est-à-dire n'importe quel élément du domaine, par exemple, un rocher ou une balle. Elle ne possède ni données, ni méthodes propres. Une entité n'est rien d'autre qu'un identifiant d'un objet du domaine.
- Un **composant** représente un aspect d'un objet, par exemple la vitesse, la position, le nombre de points de santé ou un sprite (image). Il n'y a pas de code dans les composants. Sa représentation est assimilée à une structure (ie, un Messenger). Une entité est caractérisée par un ensemble de composants, par exemple Rocher (position, sprite), Balle (position, vitesse, sprite).
- Un **système** est la représentation d'un algorithme portant sur les composants, par exemple le déplacement des objets mobiles ou les conséquences d'un combat. Concrètement, un système implémente la fonction `update()` qui sera appelée à chaque pas de simulation. Un système effectue des actions globales sur chaque entité qui possède les composants requis par le système. Les systèmes portent donc la logique métier.

Pour illustrer le changement de paradigme entre la conception orientée objet et la conception orientée données, prenons la classe `SpaceShip` qui possède des attributs de position et de vitesse et une méthode pour déplacer le vaisseau. Dans l'architecture ECS, la classe est éclatée en trois classes : l'entité `SpaceShip` (sans attribut ni méthode), les composants `Shape`, `Motion` et `Position` et le système `MotionSystem` qui détient le code pour faire bouger une entité formée des trois composants : `Position`, `Motion` et `Shape`.



- À chaque exécution de la méthode `update()` d'un système, le système recherche toutes les entités sur lesquelles il s'applique et vérifie que l'entité possède les bons composants. Puis, il

exécute son code sur l'entité. Cela se traduit par le diagramme de séquence suivant :



13.1. Caractéristiques

Ce patron d'architecture a des connexions avec le patron de conception Visiteur où les traitements sont aussi extraits des objets puis appliqués de façon externe sur les objets.

Recommandation d'implémentation

- Les entités, les composants et les systèmes sont rassemblés dans des tableaux.

Avantages

- Arrange les données en mémoire pour minimiser la location des données et produire du code qui utilise de larges bandes de mémoire plutôt que de morceler la mémoire.
- Élimine les problèmes d'ambiguïté des hiérarchies d'héritage profondes et larges qui sont difficiles à étendre. Dans une modélisation orientée objet, à mesure que le nombre d'entité augmente, il devient de plus en plus difficile de placer une nouvelle entité dans la hiérarchie, surtout si l'entité a besoin de nombreux types de fonctionnalités réparties dans la hiérarchie. Dans une modélisation orientée données, il n'y a plus ici de hiérarchie mais des compositions.

Désavantage

- La modélisation séparant les propriétés et de leurs traitements ne rend pas compte de la structure et du fonctionnement du logiciel. L'auto-documentation n'est plus possible.

13.2. Quand l'utiliser ?

- Quand la vitesse d'accès aux données est une limite. Cette architecture supprime la notion d'associations entre les objets.
- Quand une modélisation objet pose des problèmes d'arbre d'héritage profond.

14. Conclusion du chapitre

L'architecture générale d'un logiciel doit être choisie très tôt, dans les premiers temps de l'analyse. Elle doit être relativement stable puisqu'elle peut aussi conditionner 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.

Il faut utiliser des interfaces comme des pare-feux entre les composants de l'architecture. Plusieurs architectures sont en général combinées pour un même logiciel.