Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

ATL – Ateliers logiciels

Architectural pattern

Trois-tiers, MVC, MVP et MVVM

Consignes

Ce TD présente quelques architectures logicielles courantes qui sont fortement utilisées dans les frameworks que nous étudieront par la suite (Spring, Django, Android,...). L'objectif est de se familiariser avec ces architectures et d'en comprendre les grandes différences.
Le temps de travail estimé se compose de 4h de travail en classe et de 4h de travail à domicile.

1 Une architecture

Wikipedia

> En informatique, un patron d'architecture est une solution générale et réutilisable à un problème d'architecture récurrent. Les patrons d'architecture sont semblables aux patrons de conception mais ont une portée plus large. Ils servent de modèle de référence et de source d'inspiration lors de la conception de l'architecture d'un système ou d'un logiciel informatique, pour décomposer celui-ci en éléments plus simples.

Ce TD présente 4 de ces patrons. D'autres façons d'organiser son code existent mais la compréhension de ces 4 patrons permet de comprendre la logique générale derrière ces architectures.

Afin d'illustrer ces patrons, consultez durant ce TD les implémentations proposées en téléchargeant les ressources via le lien `https://git.esi-bru.be/ATL/architectural/repository/archive.zip`

Vous y trouverez une implémentation des patrons :

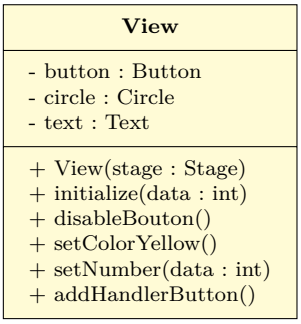
- Trois-tiers : projet `ThreeLayers`
- Modèle-vue-contrôleur : projet `MVC`
- Modèle-vue-présentation : projet `MVP`
- Modèle-vue-vue modèle : projet `MVVM`

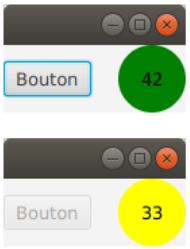
Vous remarquerez que ces architectures ont tendance à multiplier le nombre de classes à définir et elles semblent alourdir la conception d'application. Cependant le découplage ainsi obtenu assure une maintenance facilitée des applications.

Version du document
2 mars 2020

2 Exemple simplifié : les constituants

Afin de se concentrer sur la différence entre ces architectures, l'application implémentée est simple. Il s'agit d'un logiciel qui génère un nombre entier aléatoire entre 0 et 50.





La vue

La vue utilisée dans ce TD est composée de trois éléments :

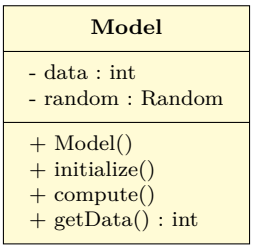
- ▷ un bouton à usage unique ;
- ▷ un cercle de couleur ;
- ▷ un nombre entier.

La valeur et la couleur affichée change après une pression sur le bouton de l'interface. Cette vue peut se décrire via le diagramme de classe ci-dessus. Elle est développée en utilisant la librairie `JavaFX`.

Le modèle

Le modèle de l'application est composé d'un nombre entier `data` qui prend la valeur 42 lors de l'`initialisation` du modèle.

Une méthode `compute` permet de demander au modèle de générer un nombre aléatoire entre 0 et 50 pour remplacer la valeur initiale.

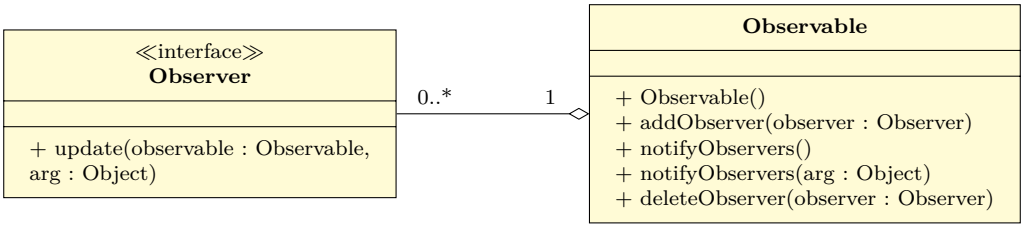


Objet de transfert de données

Le transfert des données est effectué dans les différentes architectures par des `DTO`. Afin de garder l'exemple le plus simple possible, nous utiliserons des entiers et des `String` pour jouer le rôle de `DTO`.

Observateur-Observé

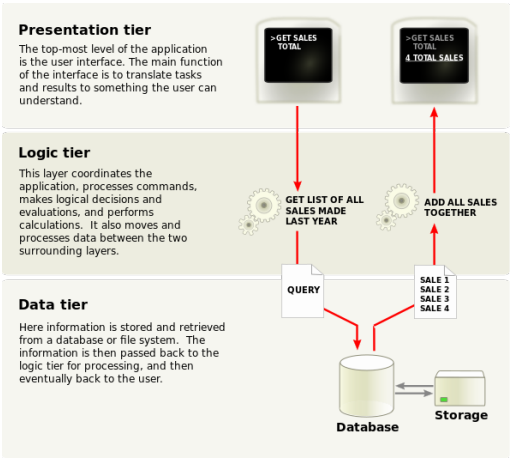
Le patron de conception `Observateur-Observé` est utilisé fréquemment durant ce TD. Voici la version du diagramme de classe associé à ce patron que nous utilisons dans ce TD.



3 Architecture 3-tiers

L'objectif de cette architecture est de diviser l'application en trois couches distinctes. Chaque couche a une responsabilité définie :

- ▷ **couche présentation** : gère l'affichage des données pour l'utilisateur et capte les interactions avec celui-ci ;
- ▷ **couche métier** : traite les données de l'application, cette couche contient la logique de l'application ;
- ▷ **couche d'accès aux données** : gère l'accès et la persistance des données.



Lorsqu'un utilisateur clique sur un bouton de l'interface, la couche présentation appelle une méthode la couche métier. La couche métier si elle a besoin d'accéder à des données persistées appelle des méthodes de la couche d'accès aux données. Chaque couche communique avec sa couche inférieure par le biais de méthodes qui échangent des DTO (Data Transfer Object).

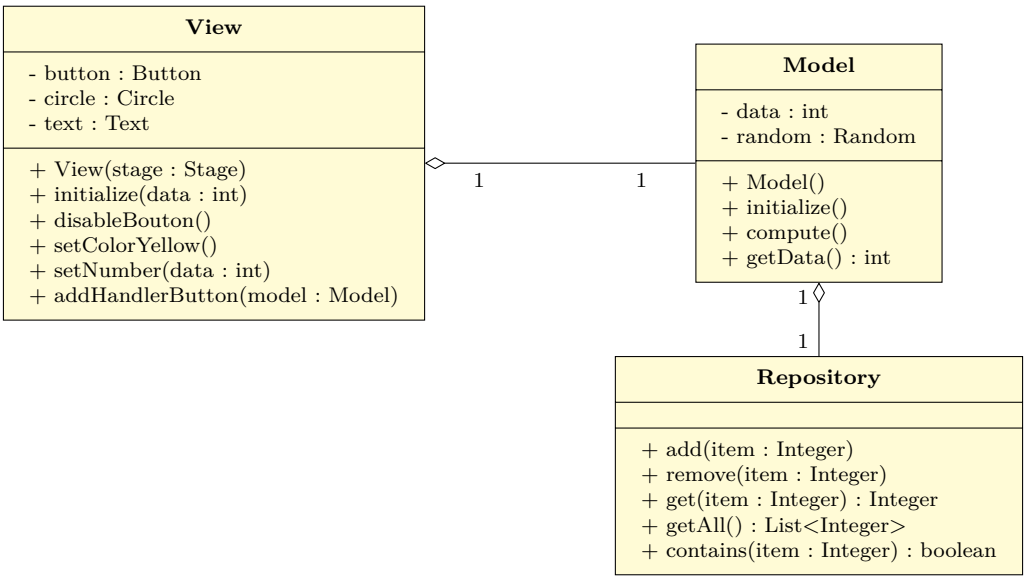
Le rôle de chacune des couches et leur interface de communication étant bien définis, les fonctionnalités de chacune d'entre elles peuvent évoluer sans induire de changement dans les autres couches.

Une implémentation de la couche d'accès aux données peut utiliser le patron `Repository`, étudié lors des TDs précédent. Ce patron communique justement via des `DTOs`.

Dans cette architecture, on constate que la vue est **active** : la vue donne des ordres au modèle.

Diagramme de classe

Afin de comprendre la mise en place de cette architecture, consultez le projet `ThreeLayers`. Ci-dessous vous pouvez analyser le diagramme de classe de ses composants principaux.



Remarques

- ▷ la Vue a une dépendance forte avec le Modèle ;
- ▷ la Vue donne des ordres au Modèle.

Tests unitaires

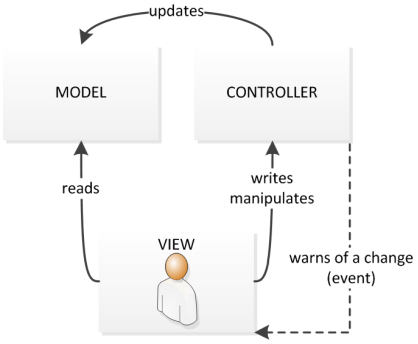
Déterminons les parties du logiciel qui doivent être testées unitairement :

- ▷ le Modèle contient la logique métier et doit être testé unitairement ;
- ▷ la Vue contient les algorithmes qui interprètent les actions de l'utilisateur et doit être testé unitairement. Ce n'est malheureusement pas possible de manière simple.

4 Modèle-vue-contrôleur

Modèle-vue-contrôleur est une architecture divisée en trois parties :

- ▷ **le modèle** : contient la logique métier, elle traite les données de l'application ;
- ▷ **les vues** : présentation visuelle de l'état du modèle ;
- ▷ **les contrôleurs** : déclencheurs d'actions à effectuer sur le modèle et/ou sur une vue.



Ce découpage permet d'isoler les différentes parties du logiciel. Des modifications apportée sur une partie n'auront, idéalement, aucune conséquence sur les autres.

Dans une architecture trois-tiers, l'implémentation de plusieurs vues peut devenir complexe. Cette complexité provient du statut actif d'une vue, c'est elle qui dirige le modèle. Si une vue demande au modèle de mettre à jour une donnée, elle doit ensuite appeler les autres vues afin de mettre à jour chacune des interfaces. Ce lien entre les différentes vues peut vite devenir contraignant.

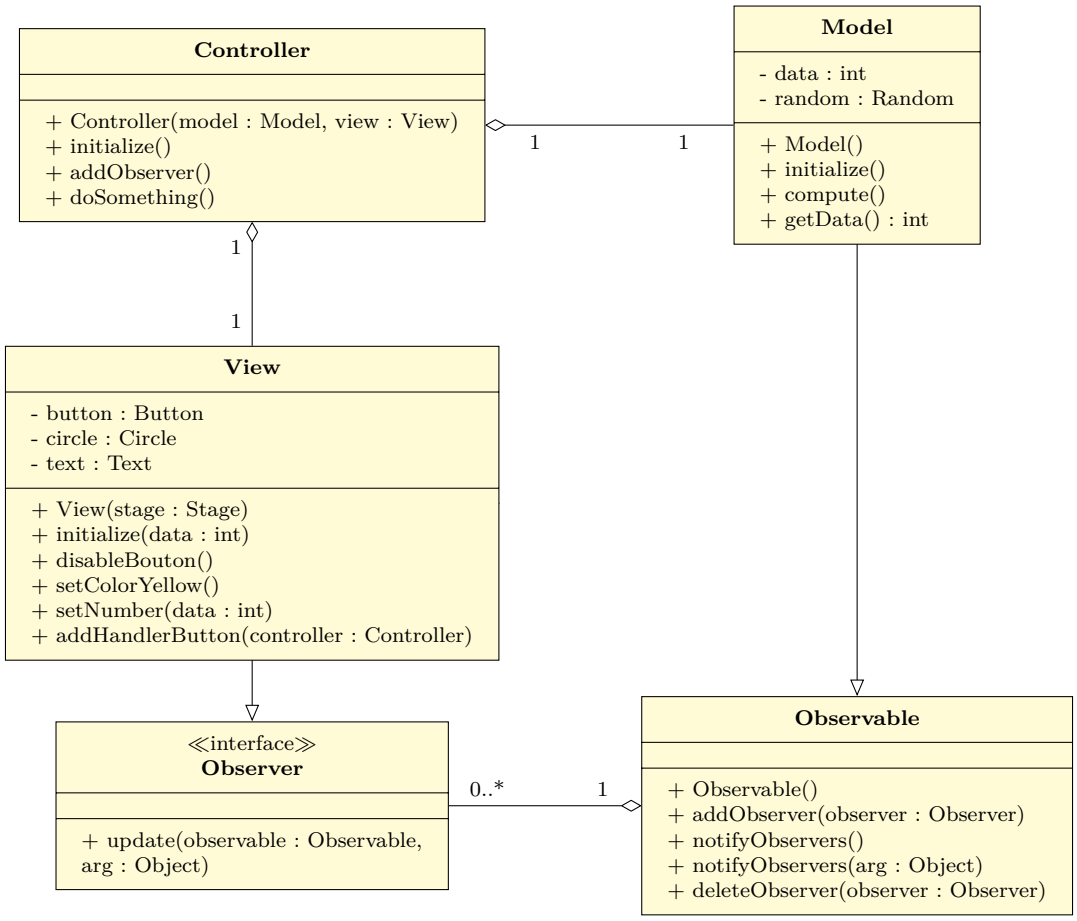
Pour résoudre ce soucis, le `MVC` utilise le patron de conception *Observateur-Observé* pour lier le modèle et les vues. Dès que le modèle met à jour une donnée, le modèle a la responsabilité d'envoyer une notification à **toutes** les vues.

Aucune mention d'accès aux données n'est présentée dans ce patron. Il se cache en fait au sein du modèle qui est responsable des données. Le modèle peut accéder aux données via un `Repository` comme nous l'avons vu dans les Tds précédents. Par soucis de simplicité nous omettrons ce `Repository` dans la suite de ce TD.

Le patron d'architecture `MVC` est assez populaire. On peut notamment le rencontrer dans des applications développées via les frameworks Ruby on Rails, Spring, Struts, Symfony, Laravel, ou AngularJs.

Diagramme de classe

Afin de comprendre la mise en place de cette architecture, consultez le projet `MVC`. Ci-dessous vous pouvez analyser le diagramme de classe de ses composants principaux.



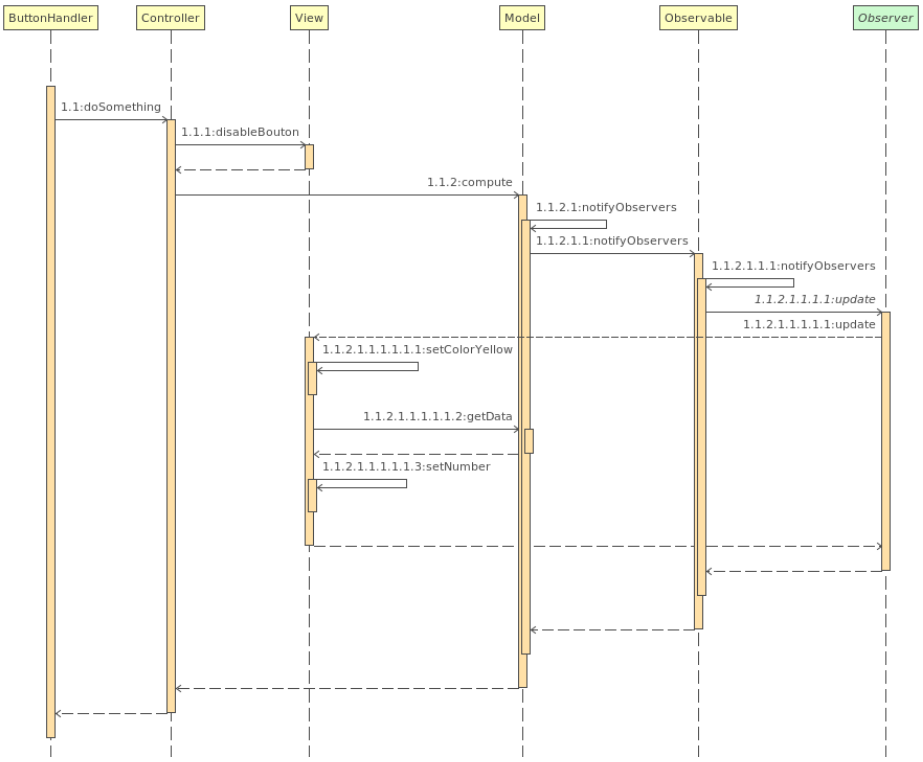
Remarques

- ▷ la vue dépend faiblement du modèle via l'`Observateur-Observé` ;
- ▷ le contrôleur est fortement dépendant de la vue et du modèle
- ▷ la vue met à disposition du contrôleur des méthodes pour mettre à jour l'affichage (`disableBouton`, `setColorYellow`,`setNumber`)
- ▷ le contrôleur est responsable de lier le modèle à la vue via la méthode `addObserver`

C'est le contrôleur qui va choisir à quoi la vue va ressembler. Donc, dans la vue, les méthodes sont publiques

Séquence des appels d'une action utilisateur

Le plus compliqué dans cette architecture est souvent d'avoir une vision claire des méthodes appelées lorsqu'un utilisateur interagit avec la vue. Pour ce faire analysons le diagramme de séquence qui suit le clic sur le bouton de notre application.



1. l'utilisateur clique sur le bouton ce qui déclenche un événement ;
2. l'événement appelle le Contrôleur ;
3. le Contrôleur demande à la Vue de mettre à jour l'interface en désactivant le bouton ;
4. le Contrôleur demande au Modèle de commencer un calcul ;
5. le Modèle traite les données ;
6. lorsque le Modèle a terminé son calcul il notifie la Vue ;
7. la Vue demande au Modèle les nouvelles données ;
8. en se basant sur les données reçues, la Vue décide de mettre à jour les composants qu'elle juge nécessaire via sa méthode `update`.

Exercice 1 Imaginons que la vue contienne un champs de texte contenant la valeur maximale de l'entier que le modèle peut générer. Quel composant de votre architecture est responsable de récupérer cette valeur et de la transmettre au modèle ? A quelle étape cette demande est-elle effectuée ? Le controller. Il le demande avant le compute

Tests unitaires

Déterminons les parties du logiciel qui doivent être testées unitairement :

- ▷ le Modèle contient la logique métier et doit être testé unitairement ;
- ▷ le Contrôleur contient les algorithmes qui interprètent les actions de l'utilisateur et doit être testé unitairement. Il faut recourir au `Mock` de la Vue et du Modèle ;
- ▷ la méthode `update` de la Vue contient la logique de mise à jour et doit être testée unitairement. Ce qui n'est malheureusement pas possible de manière simple.

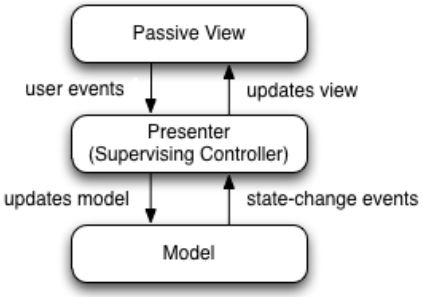
Exercice 2 Ajoutez une nouvelle vue qui affiche dans une seconde fenêtre la valeur binaire de l'entier proposé. Le code de cette seconde vue est disponible sur PoESI via le fichier `ViewBinary.java`. Quelles parties de l'application ont dus être modifiées lors de l'ajout de cette nouvelle vue ?

5 Modèle-Vue-Présentation

Le Modèle-Vue-Présentation est un patron d'architecture, dérivé du `MVC`.

`MVP` est une architecture divisée en trois parties :

- ▷ **le Modèle** : contient la logique métier, elle traite les données de l'application ;
- ▷ **les Vues** : présentation visuelle de l'état du modèle ;
- ▷ **la Présentation** : déclencheurs d'actions à effectuer sur le modèle et/ou sur une vue.

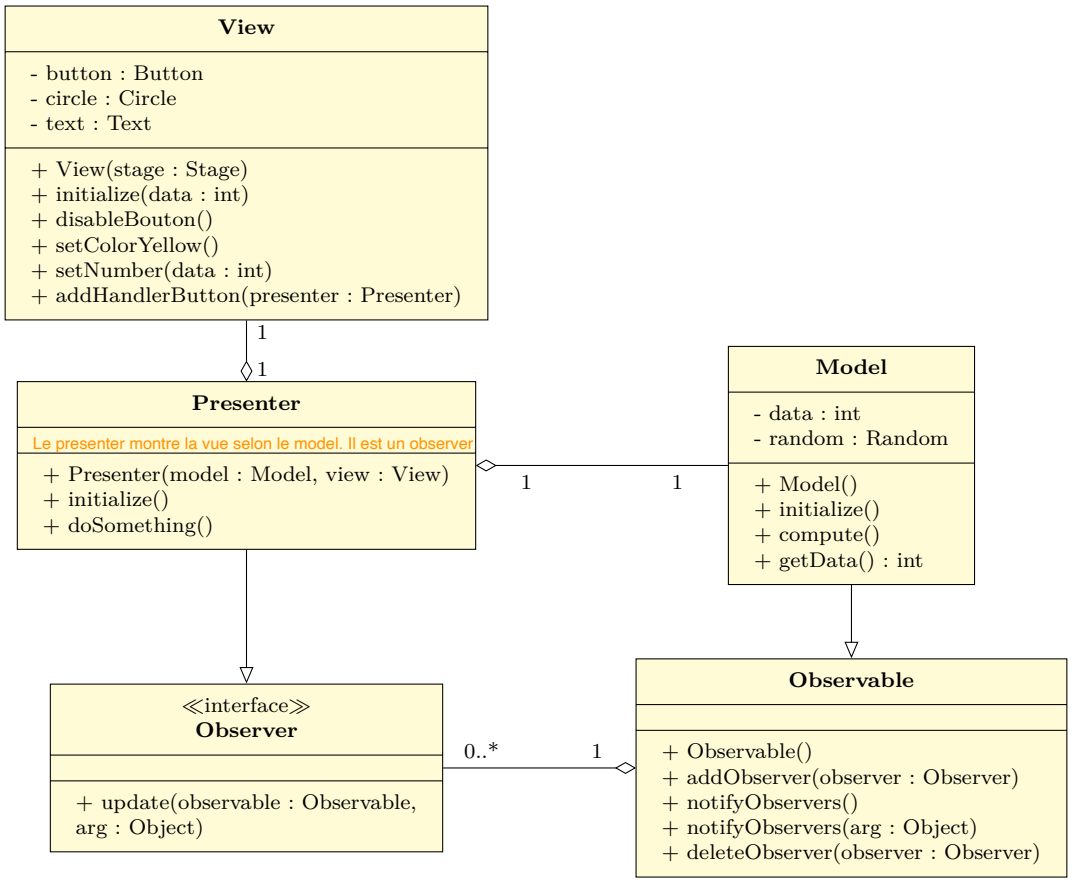


Le `MVP` élimine l'interaction entre la Vue et le Modèle. La Présentation a la responsabilité de gérer cette interaction. Dans cette architecture la Vue est considérée comme **passive**, elle ne donne plus aucun ordre. La Vue ne contient plus de logique.

> On retrouve fréquemment l'utilisation de ce patron d'architecture dans des applications Android.

Diagramme de classe

Afin de comprendre la mise en place de cette architecture, consultez le projet `MVP`. Ci-dessous vous pouvez analyser le diagramme de classe de ses composants principaux.

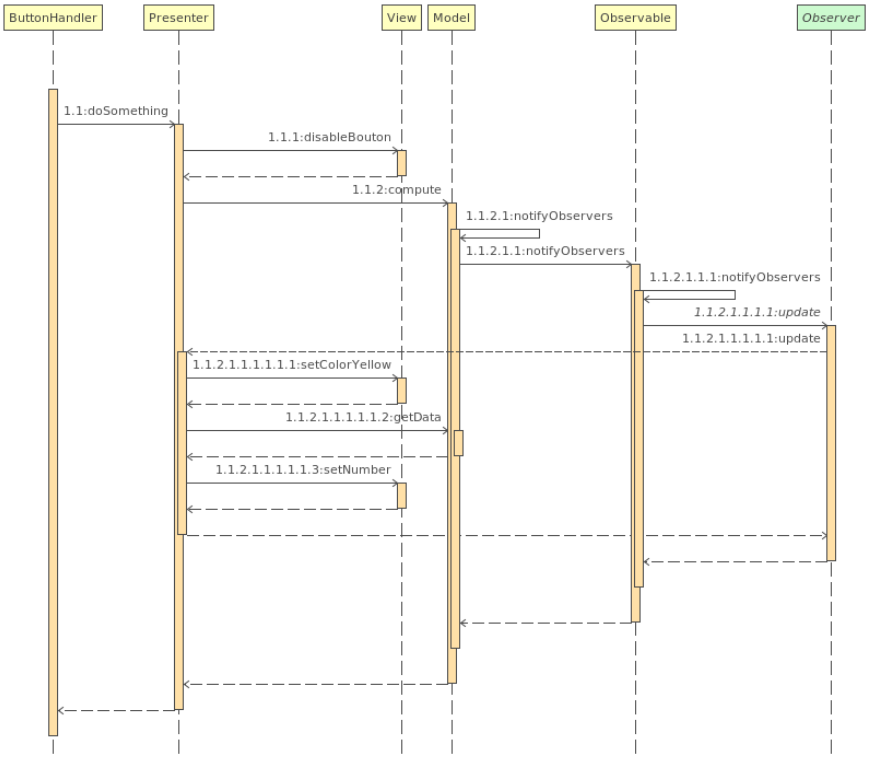


Remarques

- ▷ la Vue ne dépend pas du Modèle ;
- ▷ la Présentation dépend de la Vue et du Modèle ;
- ▷ la Présentation est abonnée aux notifications du Modèle via l'`Observateur-Observé` ;
- ▷ la Vue met à disposition de la Présentation des méthodes pour mettre à jour l'affichage (`disableBouton`, `setColorYellow`,`setNumber`).

Séquence des appels d'une action utilisateur

Déterminons la séquence des appels qui suit le clique du bouton de l'interface par un utilisateur.



1. l'utilisateur clique sur le bouton ce qui déclenche un événement ;
2. l'événement appelle la Présentation ;
3. la Présentation demande à la Vue de mettre à jour l'interface en désactivant le bouton ;
4. la Présentation demande au Modèle de commencer un calcul ;
5. le Modèle traite les données ;
6. lorsque le Modèle a terminé son calcul il notifie la Présentation ;
7. la Présentation demande au Modèle les nouvelles données ;
8. la Présentation demande à la Vue de mettre à jour les composants que la Présentation juge nécessaire.

Tests unitaires

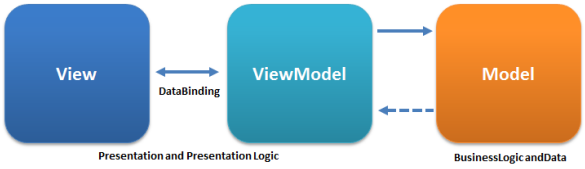
Déterminons les parties du logiciel qui doivent être testées unitairement :

- ▷ le modèle contient la logique métier et doit être testé unitairement ;
- ▷ la présentation contient les algorithmes qui interprètent les actions de l'utilisateur et doit être testé unitairement. Il faut recourir au `Mock` de la vue et du modèle ;
- ▷ la vue ne contient plus de logique et **ne doit pas** être testée unitairement.

Exercice 3 Ajoutez une nouvelle vue qui affiche dans une seconde fenêtre la valeur binaire de l'entier proposé. Le code de cette seconde vue est disponible sur PoESI via le fichier `ViewBinary.java`. Les modifications apportées au code sont-elles les mêmes que lors de l'exercice sur le `MVC` ?

6 Modèle-Vue-Vue-modèle

Le Modèle-Vue-Vue-modèle est une architecture divisée en trois parties :



- ▷ **Modèle** : contient la logique métier, elle traite les données de l'application ;
- ▷ **Vues** : présentation visuelle de l'état du modèle ;
- ▷ **Vue-modèle** : gère les actions de l'utilisateur et modifie l'affichage via le système de `Binding`. Certains de ses attributs sont des conteneurs des valeurs affichées dans les vues.

Si une vue affiche un nombre entier via l'utilisation de la classe `javafx.scene.text.Text`, la valeur de cet entier est contenue dans un attribut du **Vue-modèle**. Cet attribut a la particularité d'être un `Observable` et d'avoir comme `Observateur` l'instance de la classe `javafx.scene.text.Text` de la vue.

Binding

Lien entre un attribut `Observable` du **Vue-modèle** et un composant `Observateur` de la **Vue**.

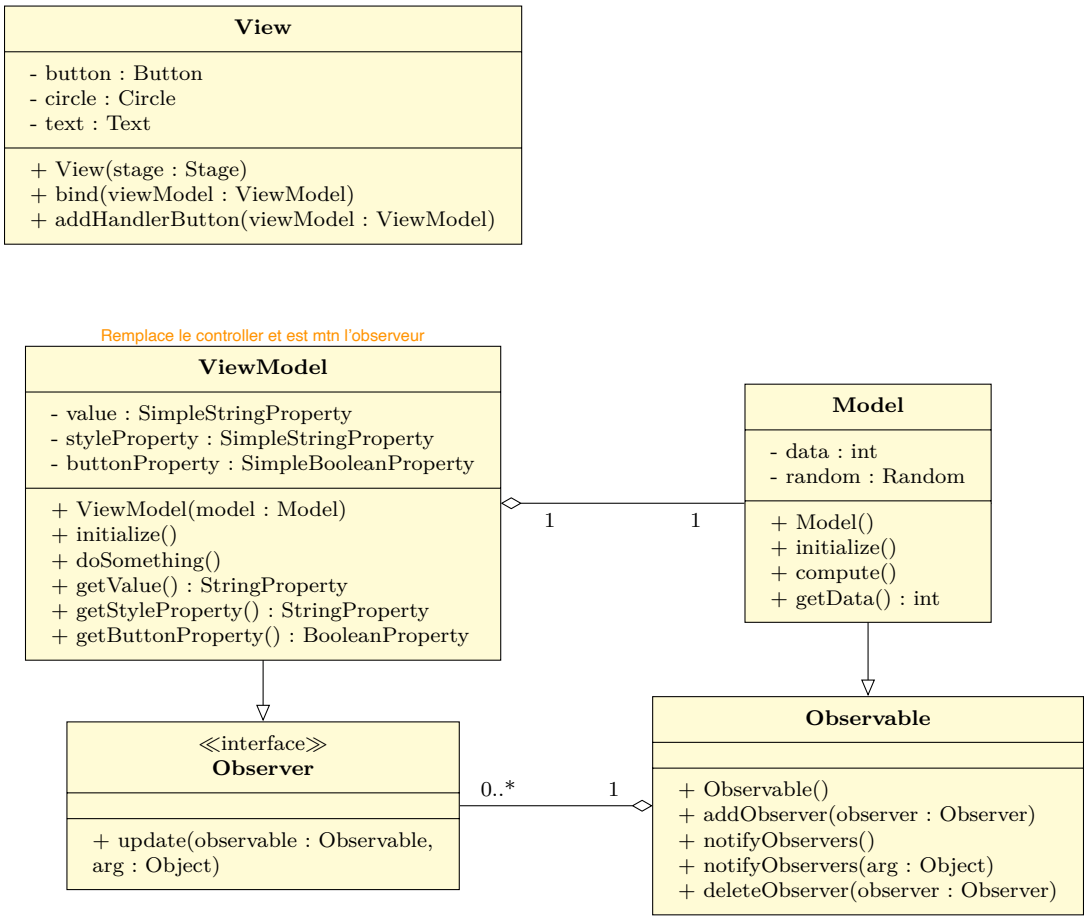
L'objectif est d'éviter de devoir coder les mises à jours de la vue explicitement. Dès qu'un attribut du **Vue-modèle** est mis à jour, le composant associé de la vue se met à jour.

Cette architecture dépend fortement de l'implémentation de la notion de `Binding` dans les libraires qui gèrent la vue. Dans le cas de `JavaFX` tout les composants standards (`Bouton`, `Label`, `TextField`,...) peuvent utiliser le `Binding` facilement. La difficulté survient lorsque des composants ne permettent pas d'utiliser le `Binding` nativement. Il faut alors développer ce code dans son application.

Actuellement on retrouve notamment l'utilisation de ce patron d'architecture dans les applications `VueJS` ou encore `Android`.

Diagramme de classe

Afin de comprendre la mise en place de cette architecture, consultez le projet `MVVM`. Ci-dessous vous pouvez analyser le diagramme de classe de ses composants principaux.

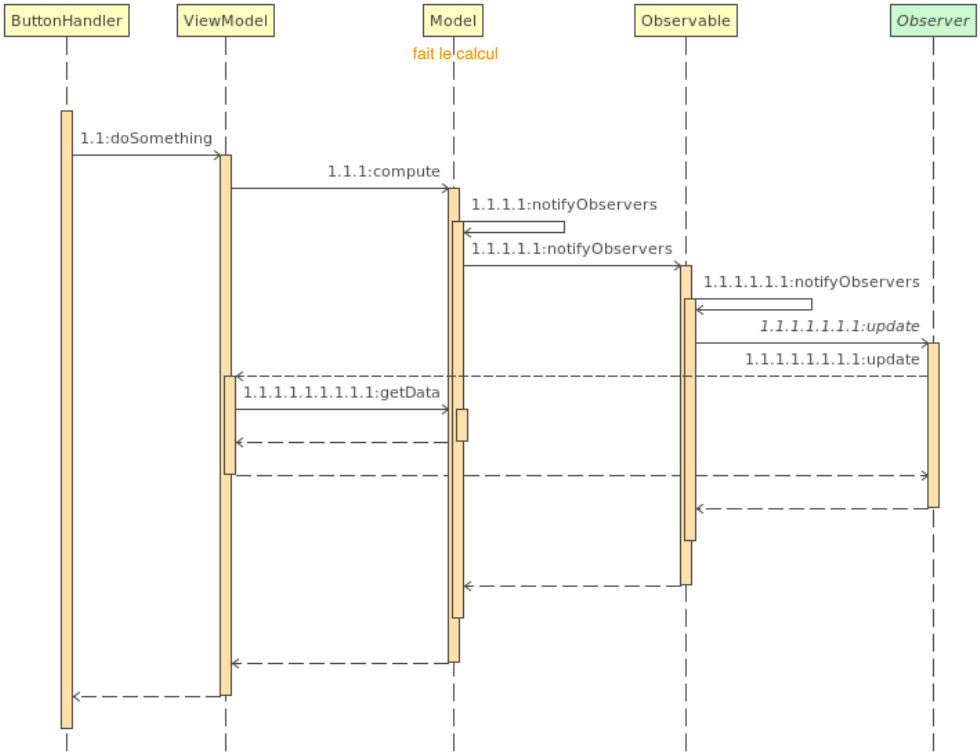


Remarques

- la Vue dépend faiblement du Vue-modèle via le `binding` ;
- le Vue-modèle ne dépend pas de la Vue ;
- le Vue-modèle dépend faiblement du Modèle ;
- le Vue-modèle possède des attributs qui reflètent la Vue : la valeur de l'entier, le style associé au cercle, la statut du bouton ;
- le Vue-modèle est abonné aux notifications du Modèle via l'`Observateur-Observé` ;
- le Vue-modèle dépend de l'existence d'attributs `Observable`, nommé `Property` dans le cas de `JavaFX`.

Séquence des appels d'une action utilisateur

Déterminons la séquence des appels qui suit le clique du bouton de l'interface par un utilisateur.



1. l'utilisateur clique sur le bouton ce qui déclenche un événement ;
2. l'événement appelle le Vue-modèle ;
3. le Vue-modèle met à jour un de ses attributs : le statut activé/désactivé du bouton ;
4. l'attribut étant lié à un composant de la Vue, la Vue se met à jour automatiquement ;
5. le Vue-modèle demande au modèle de commencer un calcul ;
6. le Modèle traite les données ;
7. lorsque le Modèle a terminé son calcul il notifie le Vue-modèle ;
8. le Vue-modèle demande au Modèle les nouvelles données ;
9. le Vue-modèle met à jour ses attributs si nécessaire et la Vue se met automatiquement à jour.

Tests unitaires

Déterminons les parties du logiciel qui doivent être testées unitairement :

- ▷ le Modèle contient la logique métier et doit être testé unitairement ;
- ▷ le Vue-modèle contient les algorithmes qui interprètent les actions de l'utilisateur et doit être testé unitairement. Il faut recourir au `Mock` du Modèle **uniquement** ;
- ▷ la Vue ne contient plus de logique et **ne doit pas** être testée unitairement.

Exercice 4 Ajoutez une nouvelle vue qui affiche dans une seconde fenêtre la valeur binaire de l'entier proposé. Le code de cette seconde vue est disponible sur PoESI via le fichier `ViewBinary.java`.

Remarques

Chaque composant de ces patrons d'architecture (modèle, vue, contrôleur, présentation, repository...) peuvent être isolés dans un package et leurs fonctionnalités rendues accessibles par une Facade [a].

a. https://en.wikipedia.org/wiki/Facade_pattern

7 Choix d'une architecture

Lors de l'implémentation d'une application, l'architecture a utiliser doit prendre en compte les différents critères énoncés durant ce TD :

- framework adapté ;
- popularité de l'architecture dans la technologie ;
- vue active ou passive ;
- évolution de l'application : le nombre de vues est fixe ou variable ;
- couverture de test souhaitée.

Il n'existe pas de classification absolue de ces différentes architectures et ses questions doivent être discutées au début de chaque nouvelle application.