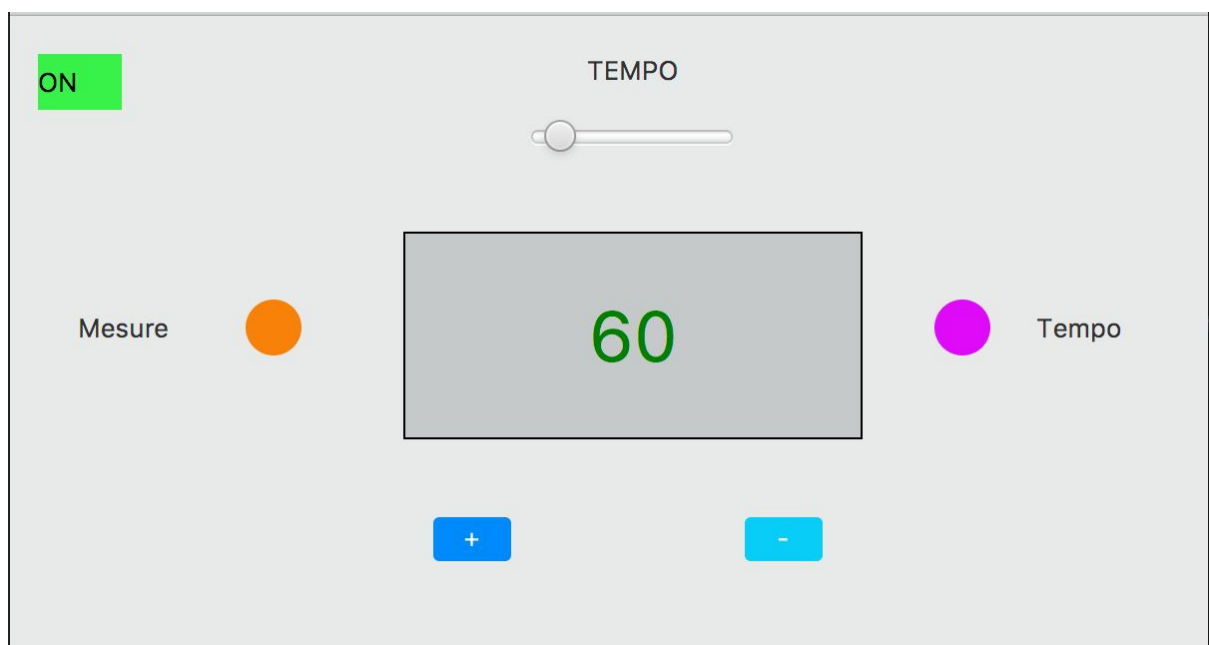


Abdallah Samy  
Coliaux Matthieu

## Projet Métronome



Master 2 MIAGE

Introduction :

Nous présenterons tout d'abord la version 1, puis la version 2.

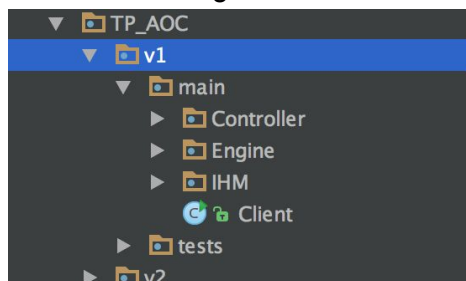


Les "+" de notre solution sont :

- Un son émis au tempo et un son émis à la mesure qui sont très agréables à écouter
- Une arborescence du code claire
- Un design coloré et simple (flat design)

Nous avons réalisé les deux versions demandées ainsi qu'une série de tests pour chacune de ces versions.

L'arborescence globale :



Chacune des versions est structurée de la même manière :

Un package **main** et un package **tests**.

Le main est composée des packages :

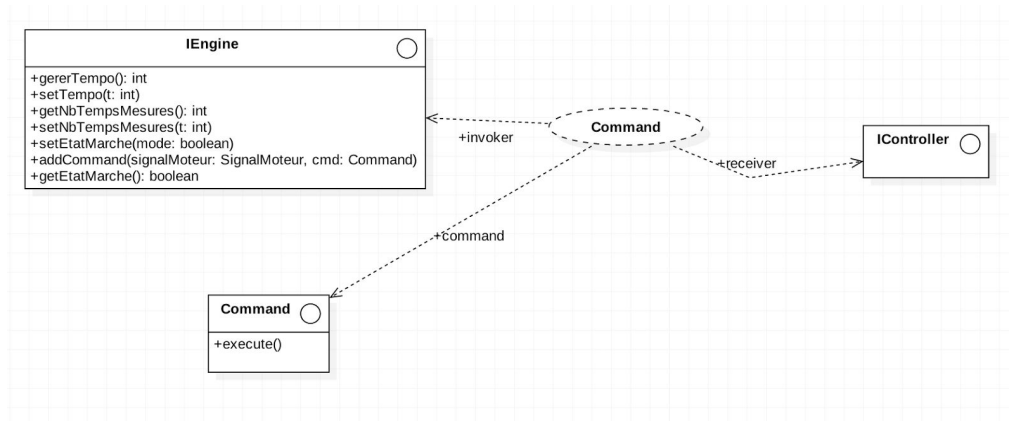
- **Controller** : pour la gestion de la partie Controller
- **Engine** : pour tout ce qui concerne le moteur
- **IHM** : pour tout ce qui réfère à l'interface graphique et **Materiel**

# 1 - Version 1

## 1.1 Les modèles :

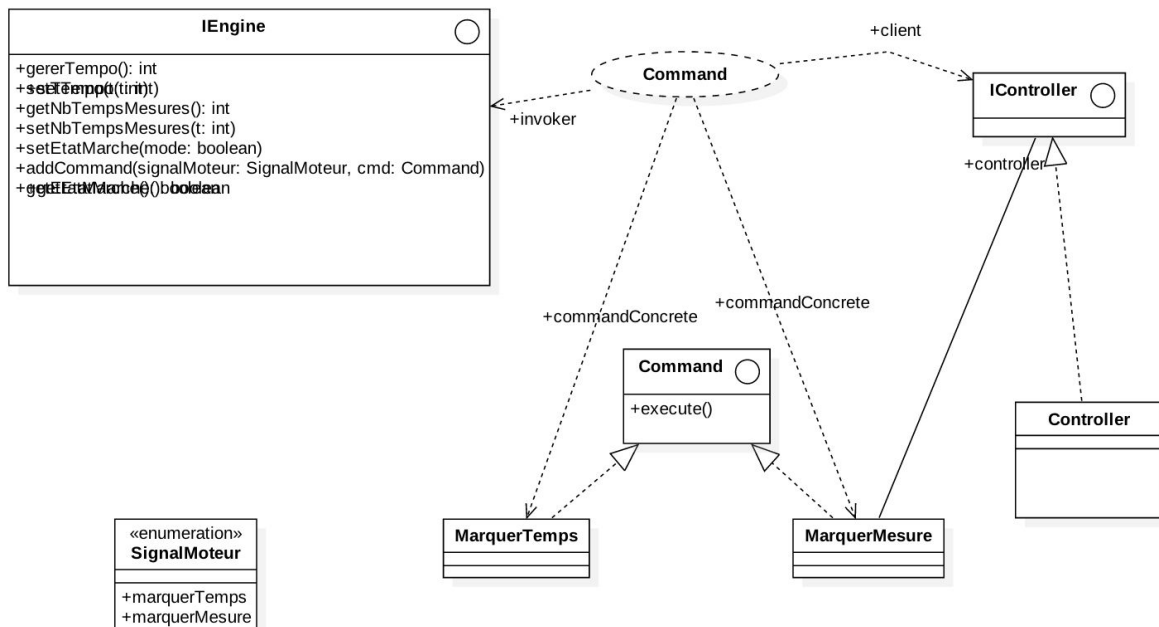
### 1.1.1 La relation Controller - Engine

Utilisation d'un pattern Command



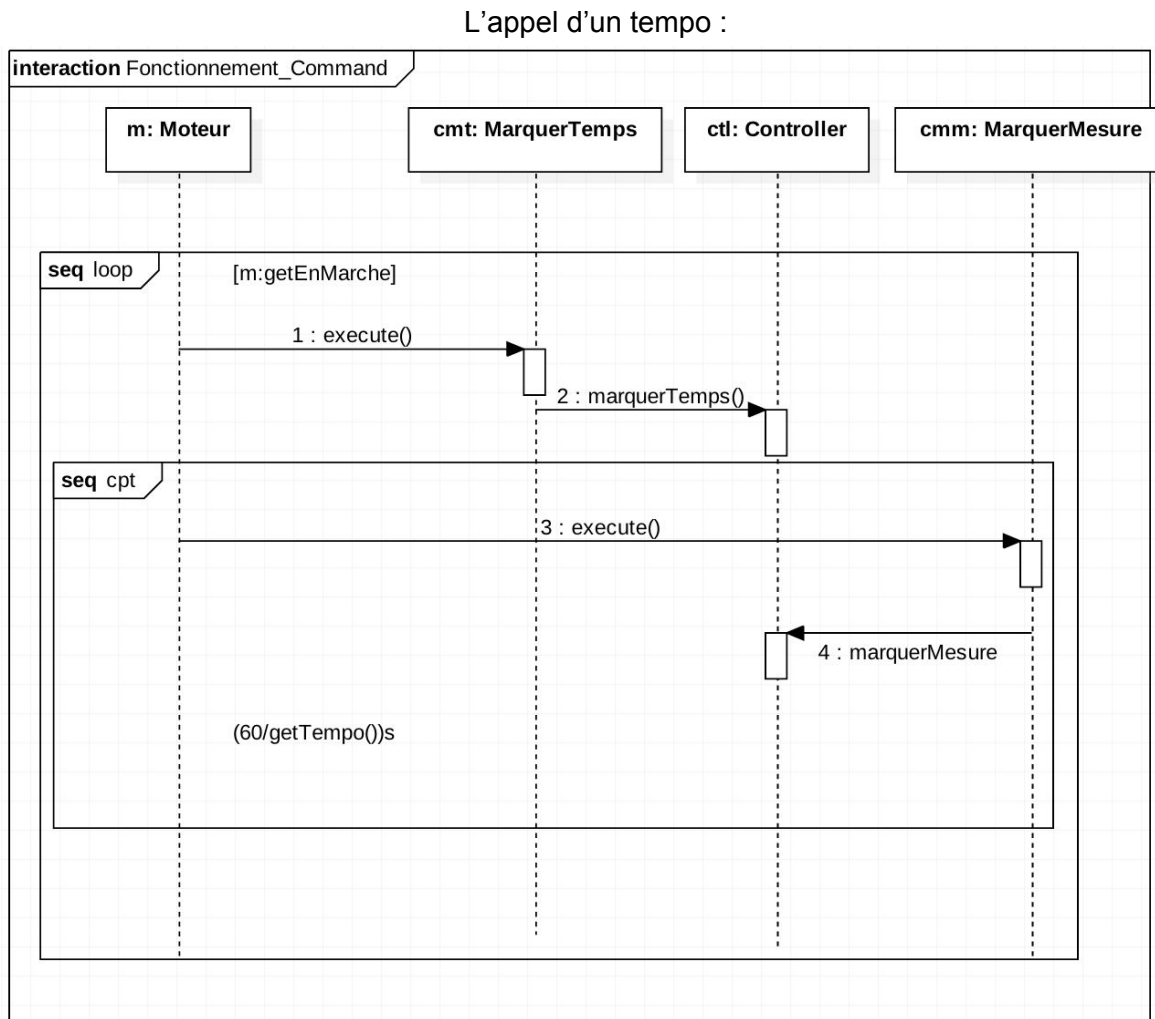
La relation entre l'Engine et le controller se fait grâce à un pattern Observer implémenté par un Pattern Command.

Les commandes MarquerTemps et MarquerMesure :



Voici une implémentation du pattern Command avec les commandes MarquerTemps et MarquerMesure.

Voici le diagramme de séquence relatif à l'appel d'un tempo.

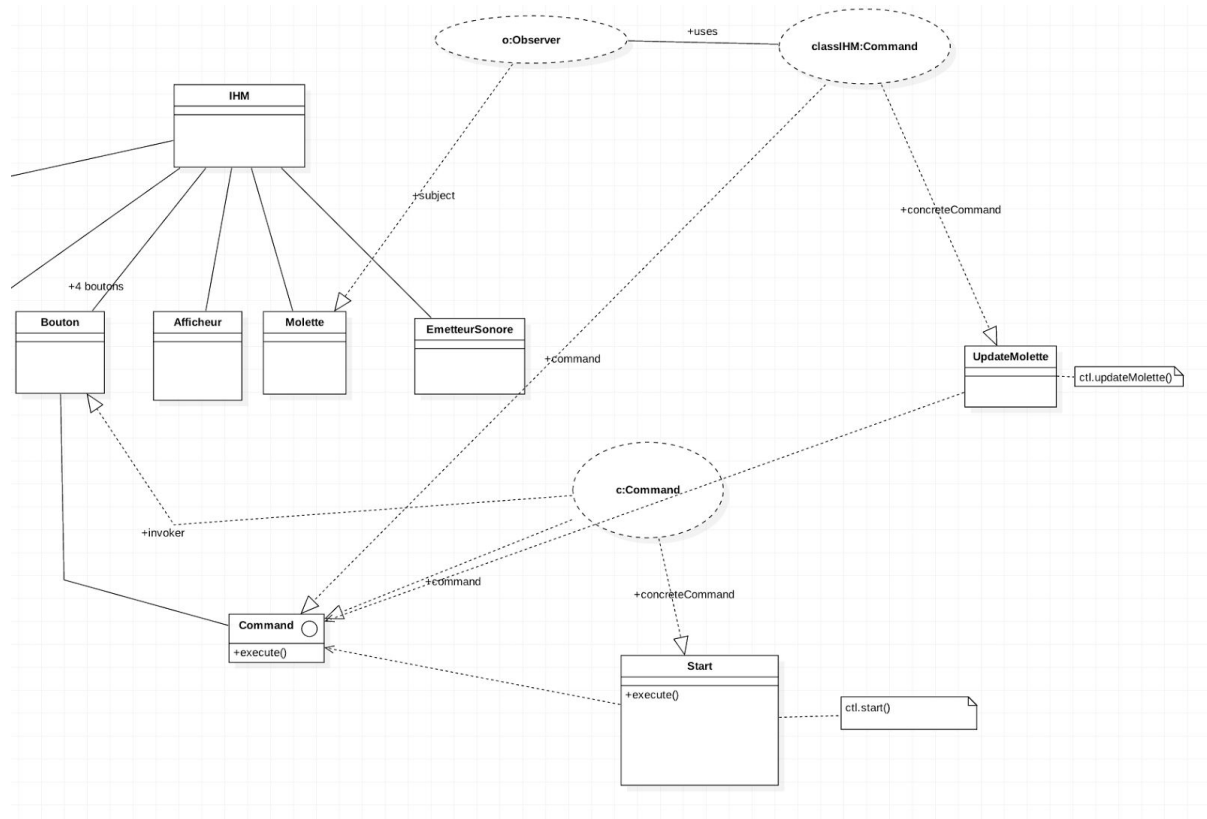


Le moteur reçoit un appel de l'horloge indiquant qu'un tempo se fait. Il appelle alors la commande MarquerTemps pour faire un execute(). Ce qui va indiquer au controller qu'un tempo à eu lieu.

Afin d'éviter un effet zip, nous avons un compteur qui permet de savoir quand à lieu la mesure. Si une mesure à lieu, alors le moteur appelle à la fonction MarquerMesure.execute() sans appeler à la fonction MarquerTemps.execute()

### 1.1.2 La relation entre IHM et le controller

### Utilisation d'un pattern Observer/command :

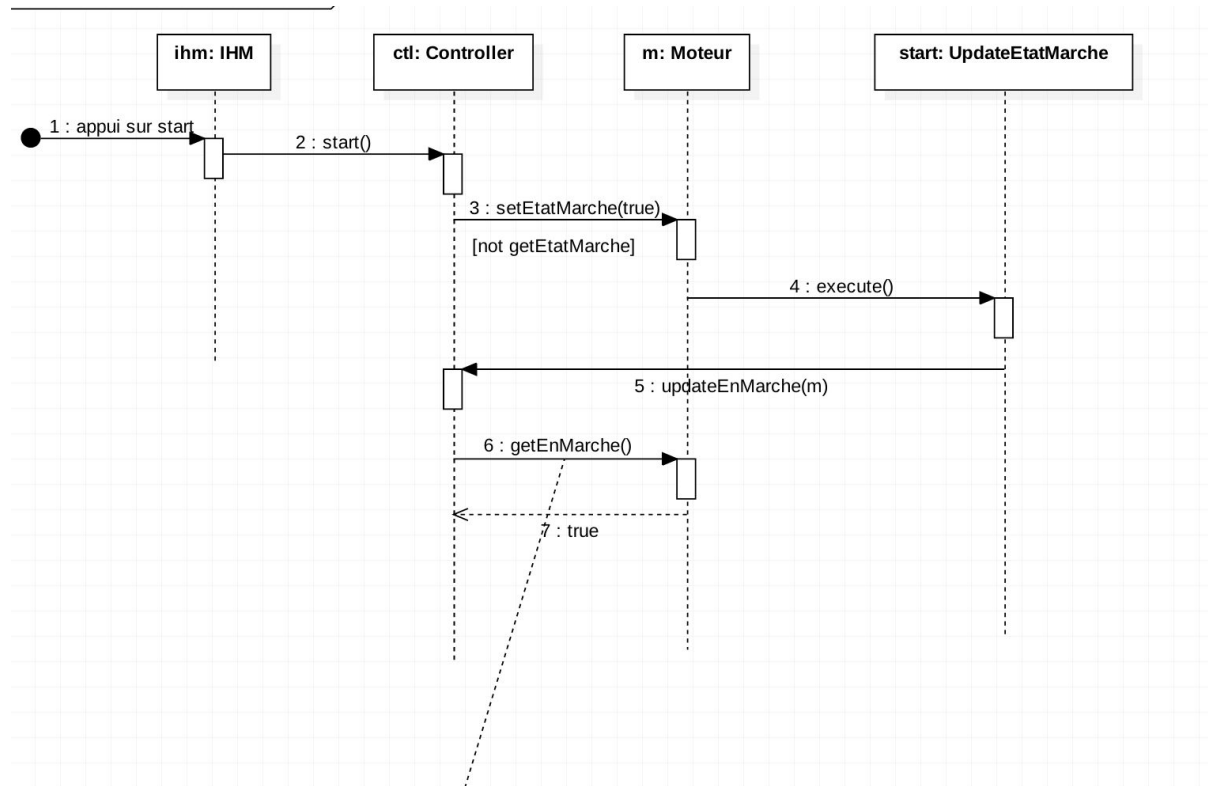


Pour la molette, le bouton “+” et “-” de la mesure ainsi que le “on”/“off” nous avons des commandes.

Ces commandes appellent des fonctions du controller.

Ci-dessus une partie du diagramme de classe avec la commande Start.

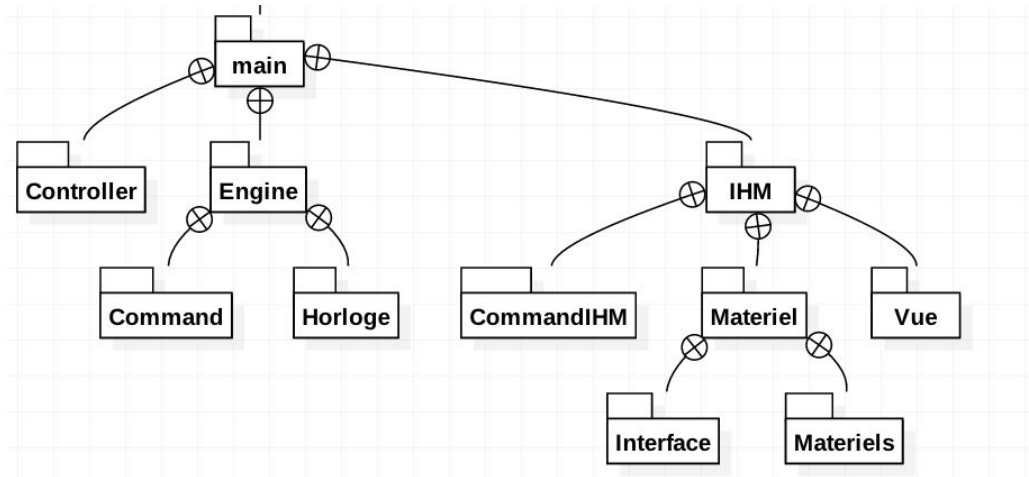
### 1.1.3 La relation entre les trois entités



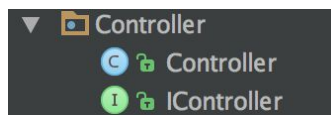
Le 6: `getEnMarche()` est fait parce qu'on ne sait pas comment pourra être implémenté l'architecture par la suite. Ce serait très utile si on place une LED qui sera allumée par la fonction `getEnMarche()`. Si on ne fait pas le contrôle on se dit que ça pourrait marcher tout le temps, peut être que le moteur va changer d'état sans que le controller en change.

## 1.2 La structure du code

Voici la hiérarchie des packages :

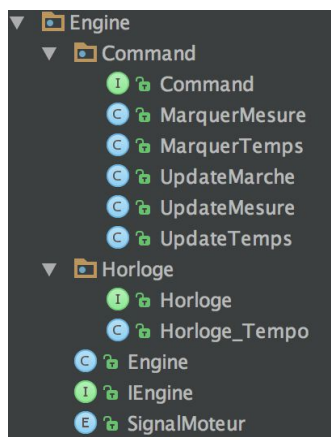


### 1.2.1 Le package Controller



Ce package contient le Controller et son interface

### 1.2.2 Le package Engine



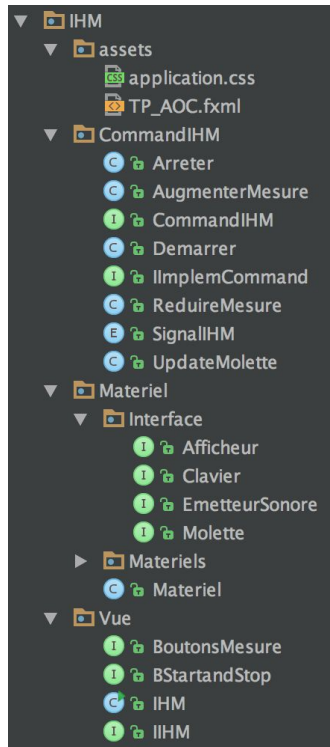
Les packages suivants :

- **Command** : les commandes permettant d'appeler le controller depuis le moteur
- **Horloge** : L'horloge initialisée par le moteur pour faire des appels tempo.

Utilisation d'un type énuméré SignalMoteur pour l'ajout des commandes.

### 1.2.3 Le package IHM :

Pour l'interface graphique, le choix de JavaFX a été retenu même s'il n'a pas été vu en cours.



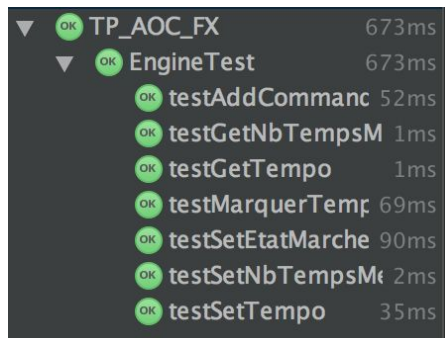
Il est composé des packages suivants :

- **assets** : ce qui est propre au design, css et placement des boutons.
- **CommandIHM** : les commandes permettant d'appeler le controller depuis l'interface
- **Materiel** : Les interfaces (package **Interface**) et classes (package **Materiels**) donné dans l'énoncé du TP
- **Vue** : Les interfaces utiles au fonctionnement GUI ainsi que le Controller FX :  
NB : La classe IHM est le Controller au sens JavaFX



## 1.3 Les Tests :

Voici le rapport des tests :



▼ OK TP_AOC_FX	673ms
▼ OK EngineTest	673ms
OK testAddCommanc	52ms
OK testGetNbTempsM	1ms
OK testGetTempo	1ms
OK testMarquerTemp	69ms
OK testSetEtatMarche	90ms
OK testSetNbTempsMe	2ms
OK testSetTempo	35ms

Seul la classe Engine est concernée par les tests. Cependant nous avons fait en sorte d'illustrer que nous étions capable de faire ces tests sur toutes les autres.

Pour ces tests Junit4 et Mockito ont utilisés.

Lorsque nous testions des fonctions, des appels à d'autres classes pouvaient être fait. De ce fait nous avons utilisé un mockito qui permet de bloquer ces appels.

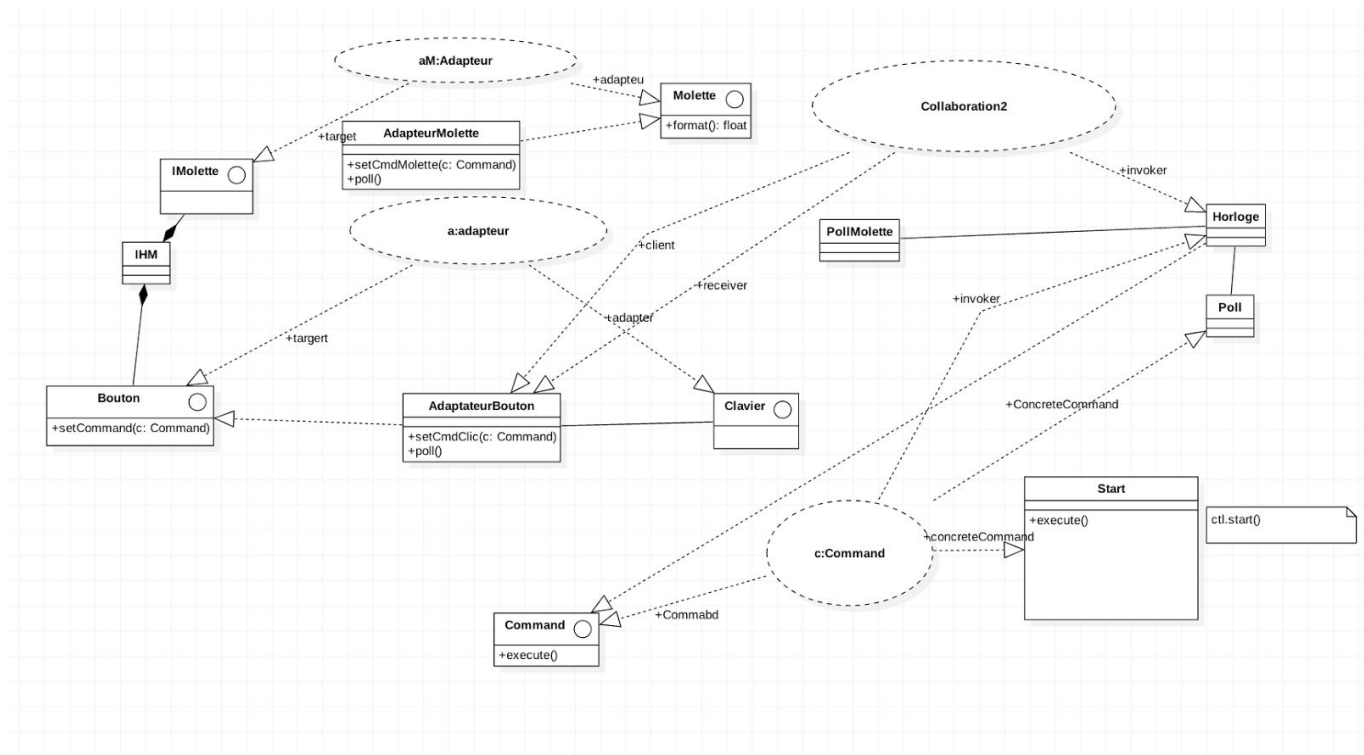
En voici un exemple :

```
UpdateTemps updt = Mockito.spy(new UpdateTemps(controller));
Mockito.doNothing().when(updt).execute();
engine.addCommand(SignalMoteur.UpdateTemps, updt);
Mockito.doNothing().when(engine).marquerTempo();
```

Nous avons fait en sorte de tester les posts et préconditions de chacune des fonctions.

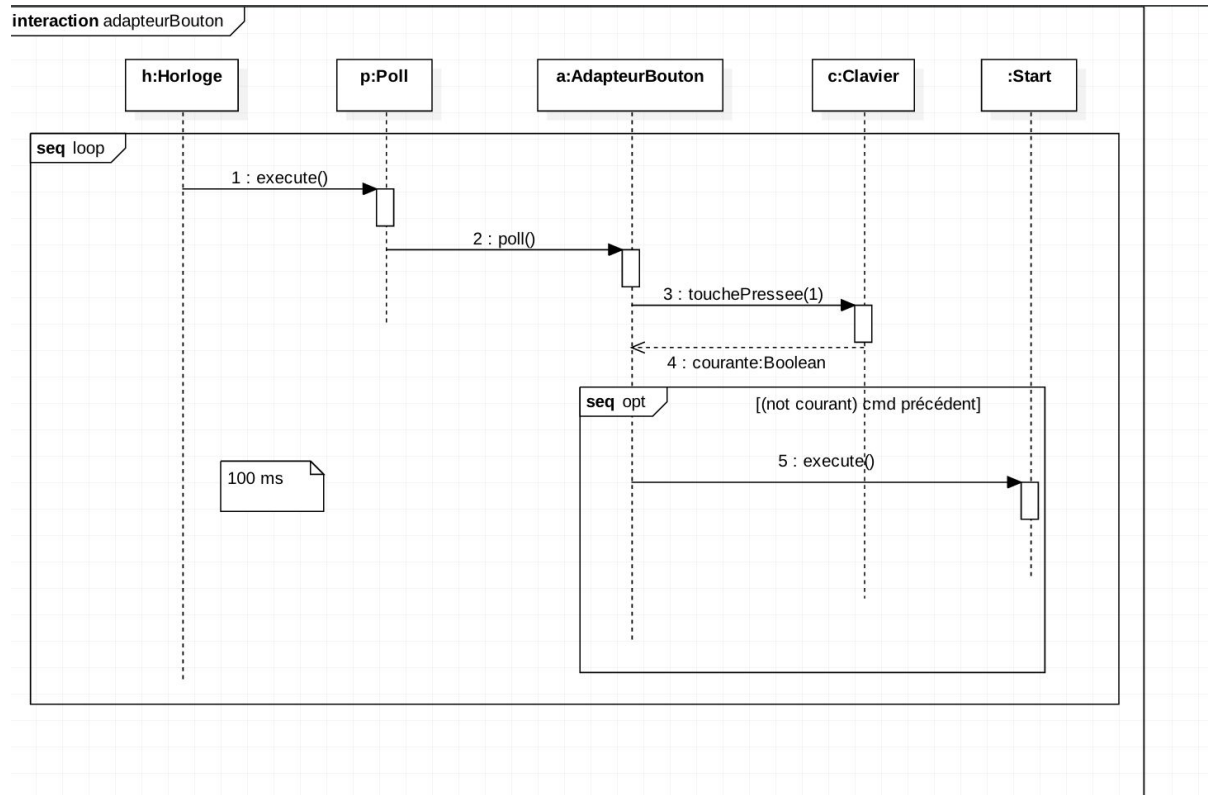
# La version 2

## 2.1 Les modèles :



Voici un diagramme de classe illustrant le fonctionnement de l'adaptateur.

Pour mieux comprendre voici un diagramme de séquence :



4: donne l'état courant du bouton 1

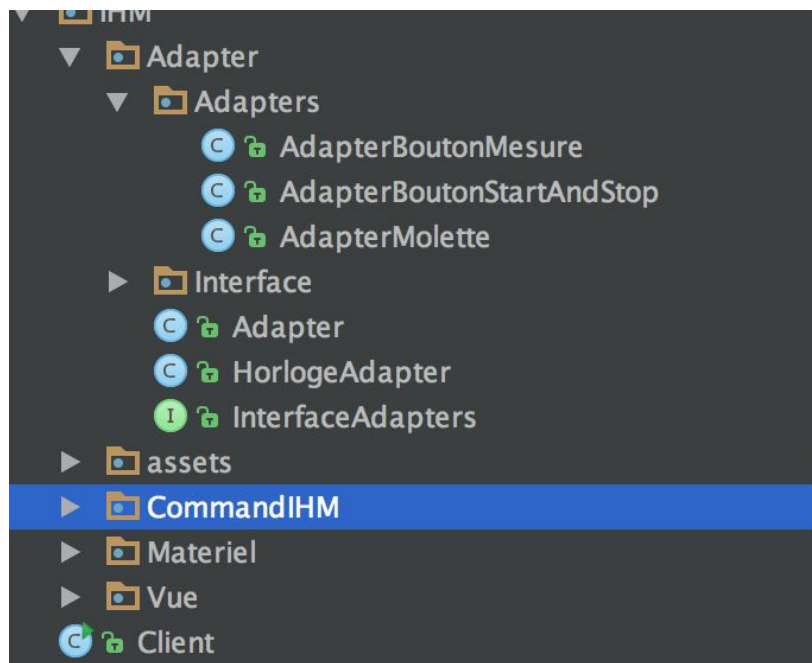
Adaptateur doit détecter l'état, il doit aller lire périodiquement le bouton, comparer l'état par rapport à celui d'avant et prendre une décision. On appelle donc la commande que le controller a installé.

Pour vérifier les changements d'états, on se sert de l'horloge. Le principe de l'adaptateur de BoutonMesure :

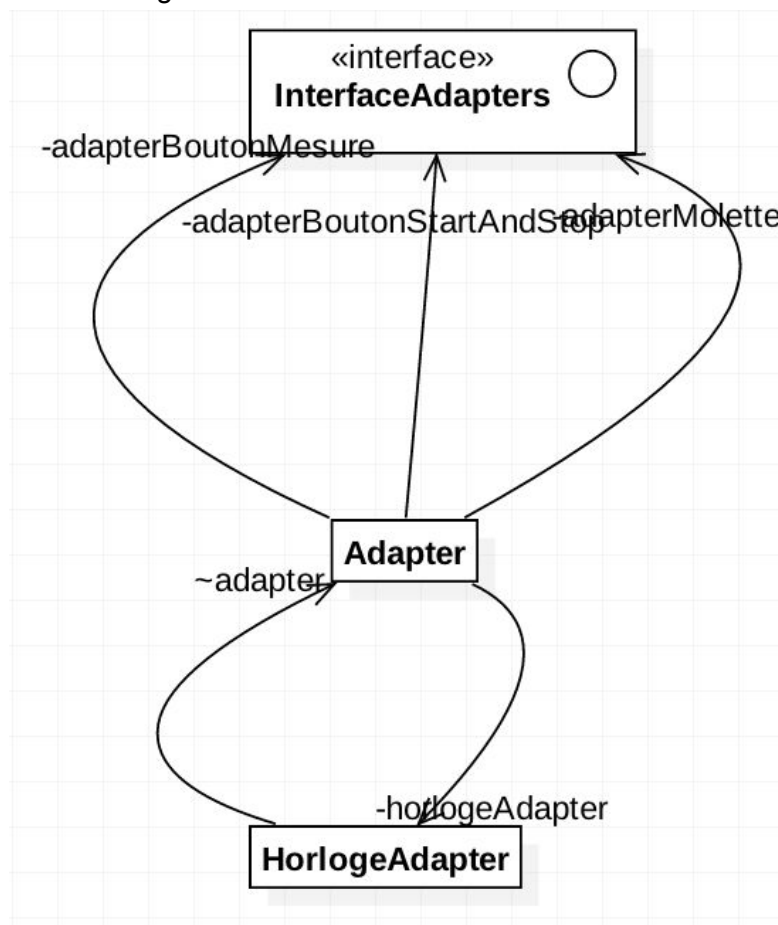
- lecture périodique de l'IHM du bouton
- comparaison avec l'état précédent et détection d'événement
- appel de la commande stockée par `addCommand(SignallIHM.AugmenterMesure, new AugmenterMesure(this))` à l'initialisation
- la lecture périodique est faite par réaction à un événement d'horloge

## 1.2 La structure du code

Nous avons ajouté un package Adapter pour développer les nouveautés de la version 2.



Voici un diagramme de classes



Nous avons enlevé les commandesIHM de l'IHM pour les mettre dans l'adapter.  
 Les adaptateurs : AdapterBoutonMesure, AdapterBoutonStartandStop et AdapterMolette  
 héritent de l'interface InterfaceAdapters.  
 Ils sont reliés à Adapter.

```
public interface InterfaceAdapters {  
    void verification();  
}
```

Ce qui permet à Adapter lors de l'appel de l'horloge de faire cette fonction :

```
public void appelerAdapters(){  
    adapterBoutonMesure.verification();  
    adapterBoutonStartAndStop.verification();  
    adapterMolette.verification();  
}
```

De cette manière les appels des adaptateurs se font grâce à une seule horloge.  
Prenons l'exemple de de AdapterBoutonMesure.verification()

```
@Override  
public void verification() {  
    if (adapter.getMatériel().getClavier().touchePressee(3))  
        augmenterMesure();  
    if (adapter.getMatériel().getClavier().touchePressee(4))  
        reduireMesure();  
}
```

Cette fonction fait appel au materiel et regarde si les boutons sont pressés, si c'est le cas, alors les commandes liées sont appelées.