

Module – AOC

—

Thibaud Destouches, Marceau Lacroix
2012 - 2013

Table des matières

1	Introduction	1
2	Architecture	1
2.1	Version 1	1
2.2	Version 2	1
3	Choix techniques et Ajouts	1
3.1	Choix techniques	1
3.2	Ajouts/Améliorations	1
4	tests	2
5	conclusion	2

1 Introduction

Le projet de métronome consiste en la conception et l'implémentation d'une architecture pour l'application métronome. Le métronome est décrit comme "un appareil qui émet un signal sonore ou lumineux à une fréquence donnée". La première version de cette application est d'une part un moteur de métronome et d'autre part une Interface Homme-Machine (IHM) pour ce moteur. La deuxième version vise au développement d'un adaptateur pour le métronome afin de "brancher" celui-ci sur une interface matérielle (simulée). La troisième version (que nous ne devons pas réaliser) est l'utilisation de l'interface matérielle sur le moteur logiciel défini dans la V1. La réalisation de ce projet nous a confronté à la mise en oeuvre de plusieurs patrons de conception de manière simultanée et coopérative (Command et adapter implémenté par nous et observer implémenté par swing pour les listeners).

2 Architecture

2.1 Version 1

Cette première version du métronome utilise une IHM java "standard" (swing). Il s'agit d'une interface active, c'est à dire que l'interface notifie le contrôleur lorsque l'un des composants impliquant un changement pour le métronome est utilisé. (changement de tempo, de taille de mesure, marche/arrêt)

2.2 Version 2

Pour cette deuxième version, l'interface devient complètement passive. Elle ne peut donc plus notifier le contrôleur des changements, c'est à lui de venir vérifier qu'un changement a eu lieu.

3 Choix techniques et Ajouts

3.1 Choix techniques

TODO

3.2 Ajouts/Améliorations

- **Visualisateur de mesure** : Ayant tous les deux déjà utilisé un métronome, nous avons décidé d'ajouter un "compteur" pour les mesures ; ce compteur permet à l'utilisateur de savoir où en est la mesure en cours grâce à une barre de progression des temps dans la mesure. Au niveau de l'architecture, cet ajout a été très simple, nous avons juste ajouté deux appels de méthode dans l'IHM "de base" : un dans le `toc temps` (pour remplir la barre d'un cran) et un dans `toc mesure` (pour vider la barre)

- **Desactivations des boutons** : Afin d'améliorer l'expérience utilisateur, nous avons choisi de désactiver les boutons lorsque l'action associée n'est pas disponible (par exemple, le bouton d'incrément de mesure est désactivé lorsque la mesure fait 7 temps).
- **Compteur de mesure** : toujours dans le but d'améliorer l'expérience utilisateur, nous avons choisi d'afficher le nombre de temps que comporte la mesure. L'ajout de cette fonctionnalité est quasiment "gratuite" en terme de temps de développement et apporte une information importante pour l'utilisateur.

4 tests

5 conclusion

L'utilisation d'une interface passive pour la V2 nous a obligé à renoncer aux fonctionnalités "bonus" que nous avons développé pour la première version (notamment l'afficheur de mesure) mais le sujet était clair à ce sujet. Contrairement au projet de Master 1 (mini éditeur de texte) le métronome nous aura permis de réaliser que Java est rapidement limité lorsque l'on essaye de jouer du son en "temps réel". On notera aussi que, contrairement à la plupart des autres projets que nous avons à réaliser cette année, les spécifications du métronome étaient très claires.

```

classDiagram
    class TocTemps {
        <<constructor>>+TocTemps(ctl : Controller)
        <<JavaElement>>+execute() : void{JavaAnnotations = "@Override"}
    }
    class Controller {
        <<constructor>>+Controller()
        <<setter>>+setMoteur(mot : Moteur)
        <<setter>>+setIHM(ihm : IHM)
        +tocMesure() : void
        +tocTemps() : void
        +start() : void
        +stop() : void
        +inc() : void
        +dec() : void
        <<setter>>+setTempo(tempo : int)
        <<getter>>+getMesure() : int
    }
    class TocMesure {
        <<constructor>>+TocMesure(ctl : Controller)
        <<JavaElement>>+execute() : void{JavaAnnotations = "@Override"}
    }
    class Toc {
        <<constructor>>+Toc(mot : Moteur)
        <<JavaElement>>+execute() : void{JavaAnnotations = "@Override"}
    }
    class Moteur {
        -enMarche : boolean
        -mesure : int
        -t : int
        -tempo : float
        <<constructor>>+Moteur()
        <<setter>>+setController(ctl : Controller) : void
        <<setter>>+setTocTemps(tocTemps : Command) : void
        <<setter>>+setTocMesure(tocMesure : Command) : void
        +toc() : void
        -calcMesure() : void
        <<getter>>+getEtatMarche() : boolean
        <<setter>>+setEnMarche(actif : boolean) : void
        +inc() : void
        +dec() : void
        <<setter>>+setTempo(tempo : float) : void
        <<getter>>+getTempo() : float
        <<setter>>+setMesure(mesure : int) : void
        <<getter>>+getMesure() : int
    }
    class Command {
        +execute() : void
    }
    class EteindreLed {
        <<constructor>>+EteindreLed(led : Led)
        <<JavaElement>>+execute() : void{JavaAnnotations = "@Override"}
    }
    class Led {
        +LEDTEMPS : int = 0{readOnly}
        +LEDMESURE : int = 1{readOnly}
        <<getter>>+getC() : Color
        <<setter>>+setC(c : Color) : void
        <<constructor>>+Led(c : Color, h : Horloge)
        +update(i : int) : void
    }
    class Horloge {
        <<const>>+activerPeriodiquement(cmd : Command)
        <<const>>+activerApresDelay(cmd : Command)
        +desactiver() : void
    }
    Controller --> TocTemps : ctl
    Controller --> TocMesure : ctl
    Controller --> Toc : mot
    Controller --> Moteur : mot
    Controller --> Horloge : ihm
    Controller --> EteindreLed : led
    Controller --> Led : ledTemps
    Controller --> Led : ledMesure
    Controller --> Horloge : timer
    Controller --> Horloge : h
    TocTemps --> Command : tocTemps
    TocMesure --> Command : tocMesure
    Toc --> Moteur : -mot
    Moteur --> Command : Command
    Moteur ..> Command : Invoker
    Moteur ..> Command : Receiver
    EteindreLed --> Led : -led
    Led --> Horloge : -h
    
```

The diagram illustrates the Command pattern implementation for a car engine control system. It features several classes: **TocTemps**, **Controller**, **TocMesure**, **Toc**, **Moteur**, **Command**, **EteindreLed**, **Led**, and **Horloge**. The **Controller** class acts as the central orchestrator, managing interactions between other components. It holds references to **TocTemps**, **TocMesure**, **Toc**, **Moteur**, and **Horloge**. The **Command** interface defines the **execute()** method. Concrete command classes like **TocTemps**, **TocMesure**, and **EteindreLed** implement this interface by delegating requests to the **Controller**. The **Moteur** class maintains state (e.g., **enMarche**, **mesure**, **t**, **tempo**) and implements methods like **start**, **stop**, **inc**, **dec**, **setTempo**, and **setMesure**. The **Led** class has state variables **LEDTEMPS** and **LEDMESURE** and interacts with the **Horloge** for timing. The **Horloge** class provides periodic and delayed activation services.

FIGURE 2 – diagramme de séquence de la V1 : démarrage du métronome

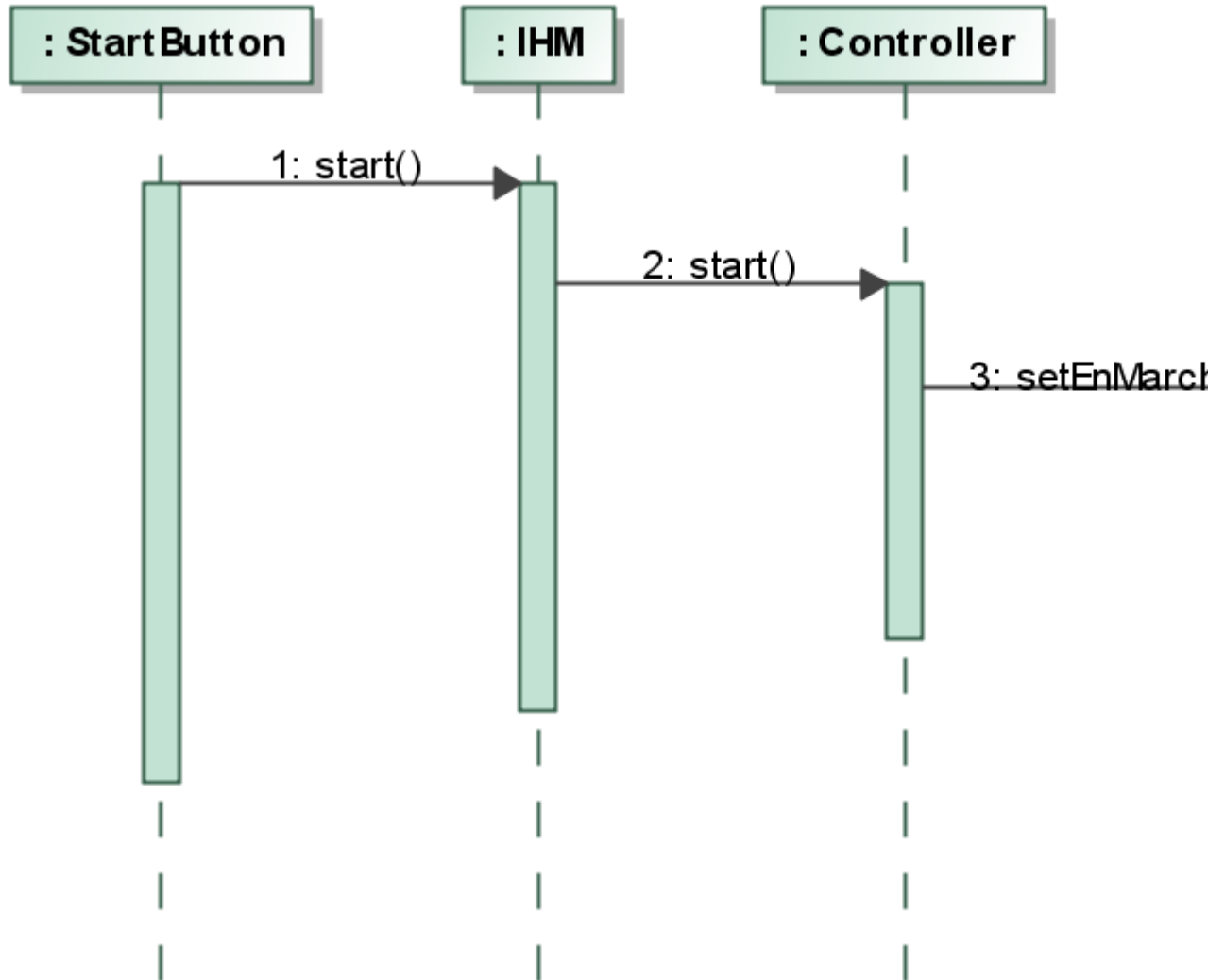


FIGURE 3 – diagramme de classes de la V2

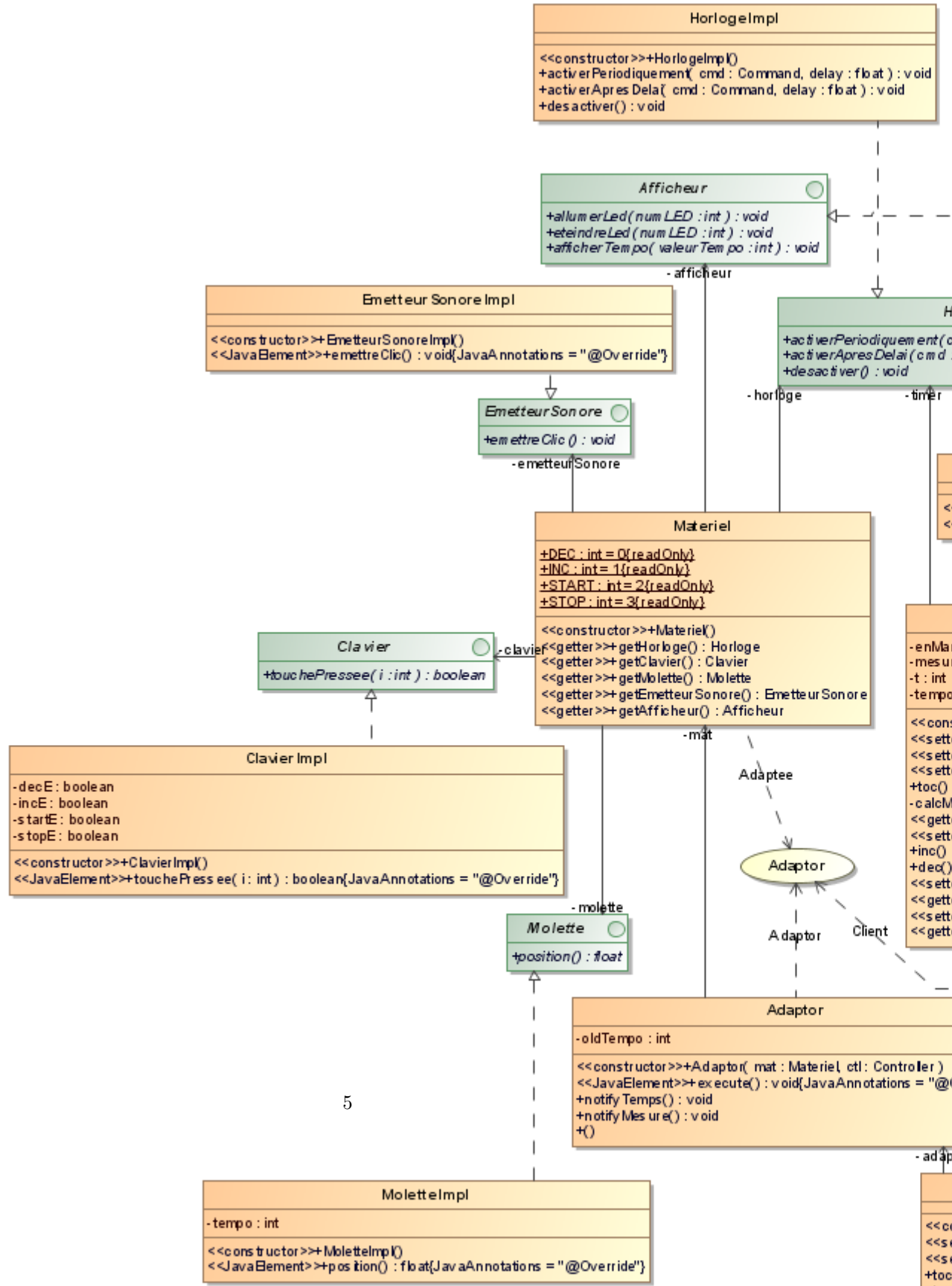


FIGURE 4 – diagramme de séquence de la V2 : diminution de la mesure

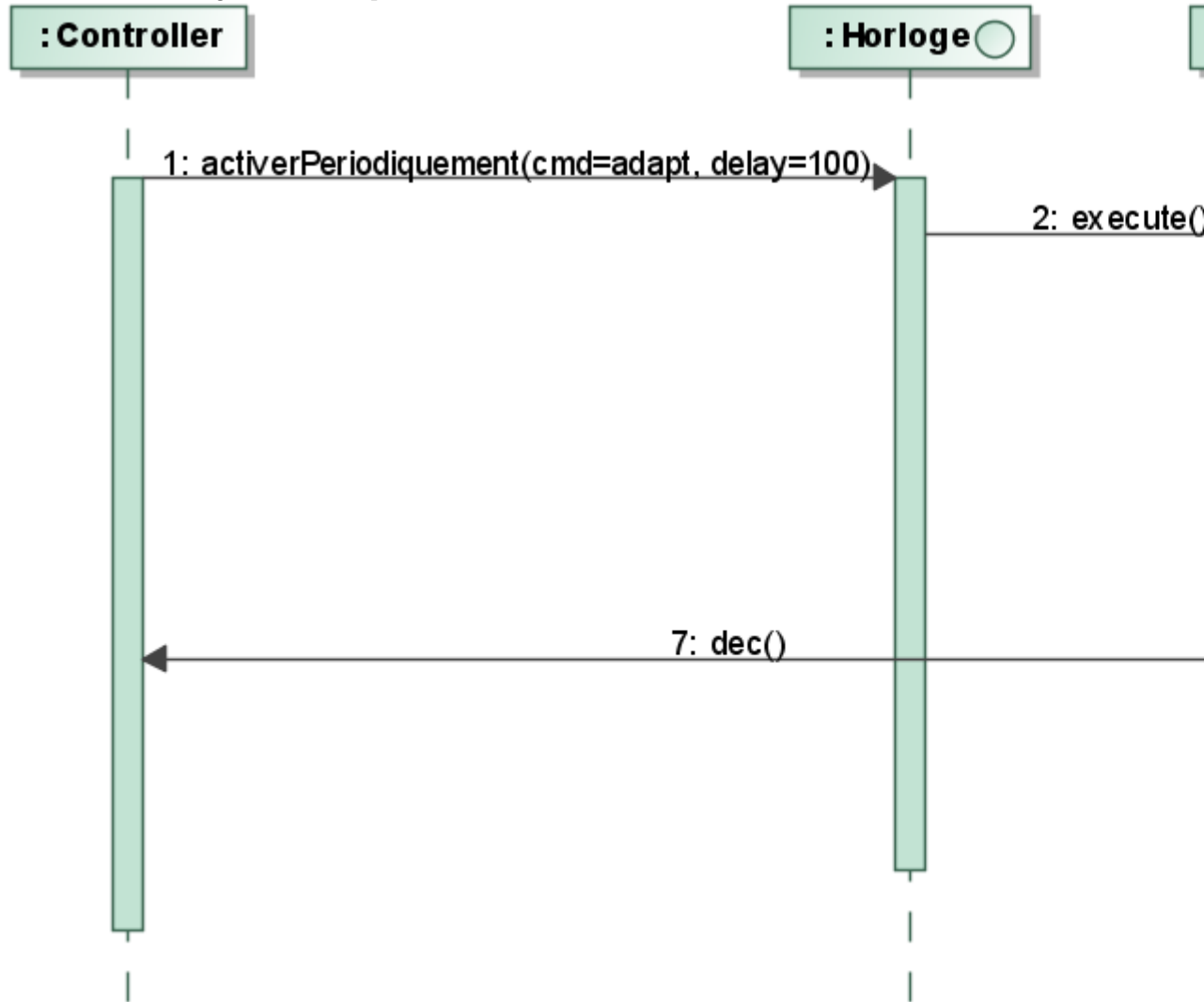


FIGURE 5 – diagramme de séquence de la V2 : "toc temps"

