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

2024 - 2025
3dev3a

3dev3a – Développement 3

Exercice : Météo - 2

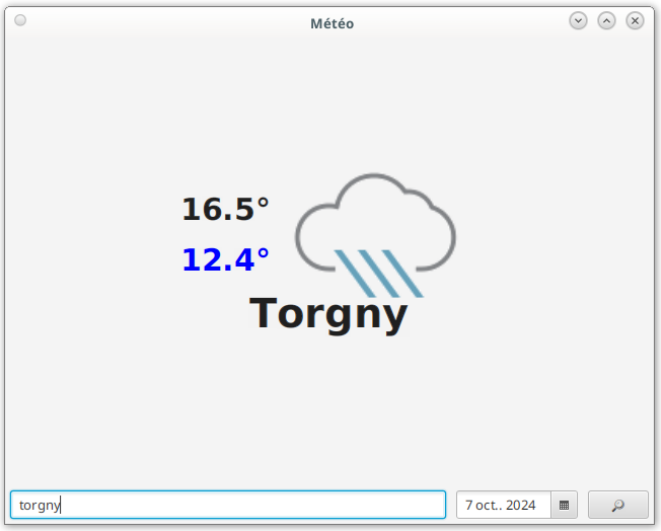
Structure MVC et composants

Cet exercice couvre deux objectifs :

1. la refactorisation (*refactoring*[a]) du code pour qu'il utilise le patron d'architecture (*architectural pattern*) MVC ; modèle, vue, contrôleur (*model, view, controller*) tel que vu en BLOC1[b] ;
2. le découpage des interfaces graphiques en composants graphiques indépendants.

a. *réusinage* en français du Québec
b. Vous aurez peut-être envie de relire le TD15 du cours 1DEV2A

Table des matières



Version du document
15 octobre 2024

1 Exercice : Météo 2

Préalable

Avant de commencer, taguez votre solution précédente de l'exercice. La commande suivante devrait suffire et vous permettra de revenir à cette version si nécessaire :

```
$ git tag -a meteo-v1.0 -m "Version 1.0 - Un truc qui
marche"
```
terminal

▷ Pour pousser le *tag* sur le dépôt,

```
$ git push origin tag meteo-v1.0
```
terminal

1.1 MVC - Model View Controller

La première étape consiste à découper son code en le répartissant dans 3 packages : `g12345.meteo.model`, `g12345.meteo.view` et `g12345.meteo.controller`[1] (cfr. figure 3 page 5).

Le modèle

Votre modèle contiendra :

▷ une exception propre au modèle.
Cette exception encapsule toutes les exceptions qui peuvent subvenir dans le modèle (principalement lors des appels à l'API) afin que le modèle ne lance qu'une seule exception : `WeatherException`.
Vous pouvez choisir si cette exception hérite de `Exception` ou de `RuntimeException`... et en « assumer les conséquences » ;
(Un peu de lecture sur le sujet ?)
https://reflectoring.io/do-not-use-checked-exceptions/

▷ un objet de transfert de données entre les différentes parties de l'application. Nous l'appelons `WeatherObject` et nous conseillons un `record` pour l'implémenter ;

▷ une classe qui *va effectivement faire le boulot* d'interrogation des API et de recherches des éléments demandés ;
Sa responsabilité est la requête aux API. C'est tout.

Remarque UML

Dans le diagramme de classes, la méthode `fetch` de la classe `WeatherApi` est **statique** et a une visibilité **paquetage** (*package*) ce qui se traduit par

▷ `~` pour la visibilité `package` ;

▷ `soulignement` pour tout ce qui est statique.

1. 12345 est à remplacer par votre matricule.

- ▷ une classe façade qui sera l'interface unique avec les autres *packages* (dans ce cas uniquement le contrôleur) et qui s'occupera de la persistence des données.
  Sa responsabilité est la cohérence du modèle et la persistance des données.

Persistence

Un des rôles du modèle est de gérer la persistence des données.
Dans ce cas simple il est inutile de faire des requêtes aux API à chaque demande de données au modèle si la question reste la même (*aka* il s'agit de la même date et du même endroit).
Le modèle conservera donc l'adresse, la date… et la météo associée si elle a déjà été demandée.

Le contrôleur

Son rôle est ici très simple : lorsque la vue demande une météo, il

- ▷ fait la demande correcte au modèle,
- ▷ met la vue à jour.

Dans le cas où le modèle est en erreur, le contrôleur informera l'utilisateur ou l'utilisatrice *via* la vue.

La vue

La vue va être découpée en plusieurs classes (cfr. ci-dessous et la figure 3 page 5).

1.2 Composants graphiques

Aux composants graphiques habituels (*textfield*, *button*…) nous allons ajouter deux composants dont les seules responsabilités sont d'afficher des données et de fournir les éventuelles informations qu'ils contiennent.

Un composant graphique hérite d'un layout (`Parent`, `HBox`…) et place ses composants au sein de ce layout.

La classe `WeatherView` (cfr. figure 1 page suivante) affiche ce qu'on lui demande d'afficher. C'est-à-dire, une image, la température minimale, la température maximale et un texte représentant la localité [2].

La classe `InputView` (cfr. figure 2 page suivante) quant-à elle fournira les accesseurs (*getters*) nécessaires pour que la vue principale puisse accéder aux informations transmises par l'utilisateur ou l'utilisatrice.

Aucune de ces deux classes n'a accès au contrôleur.

À ces deux composants nous ajoutons une vue principale qui connaitra le contrôleur et qui contiendra les différentes actions (ici, une seule action). C'est la responsabilité de cette classe de solliciter le contrôleur lorsque l'on clique sur le bouton. C'est elle qui recevra l'ordre de mise à jour de la part du contrôleur.

2. Ceci pour l'exemple d'interface donné. Si votre interface affiche d'autres informations, c'est très bien aussi.

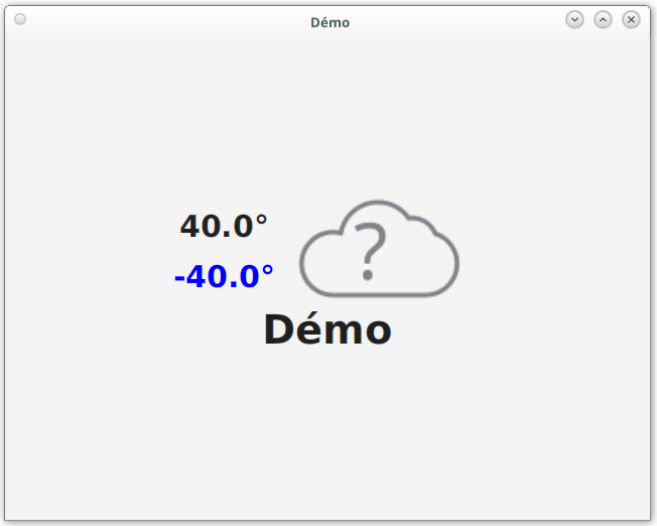


FIGURE 1 – Exemple du composant `WeatherView` placé dans une fenêtre

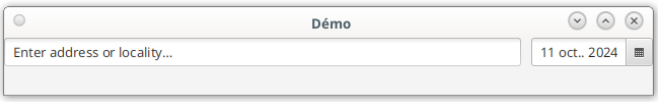


FIGURE 2 – Exemple du composant `InputView` placé dans une fenêtre

Vous pouvez maintenant taguer votre version : *v2*.

```
$ git tag -a meteo-v2.0 -m "Structure MVC et composants"
```
terminal

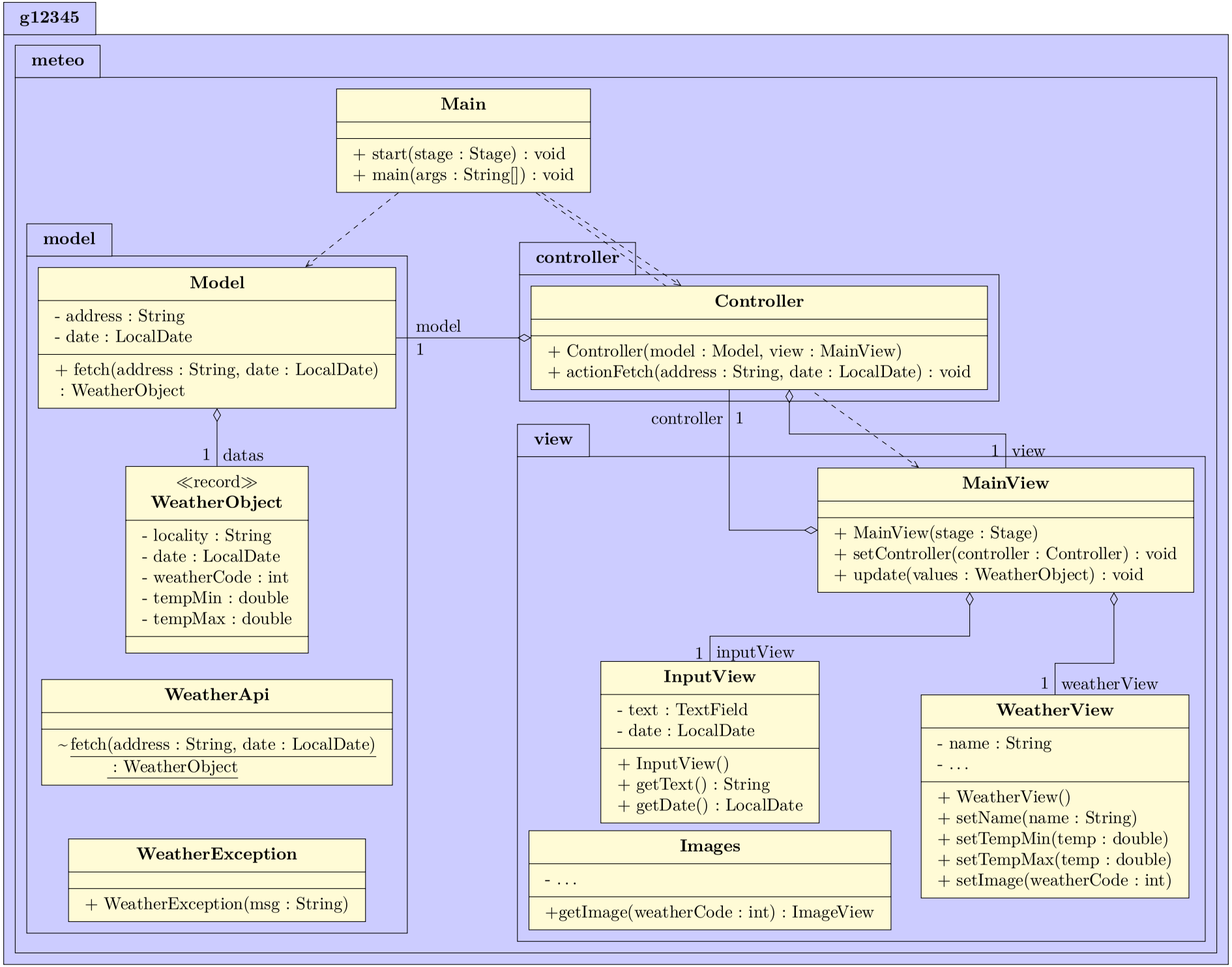


FIGURE 3 – Diagramme de classes sans les classes de l'API telles que `Application`, `HBox`, `TextField`...