



Conception de l'application 2048

martine.gautier@univ-lorraine.fr

- 1 Captures d'écran
- 2 Arborescence des composants JavaFX
- 3 Identification des classes utiles
- 4 Construction du diagramme de classes
 - Mode procédural
 - Patron de conception Observer entre Modèle et Vues
- 5 Ecriture des classes, de façon incrémentale

Jeu

NouveauCTRL+N

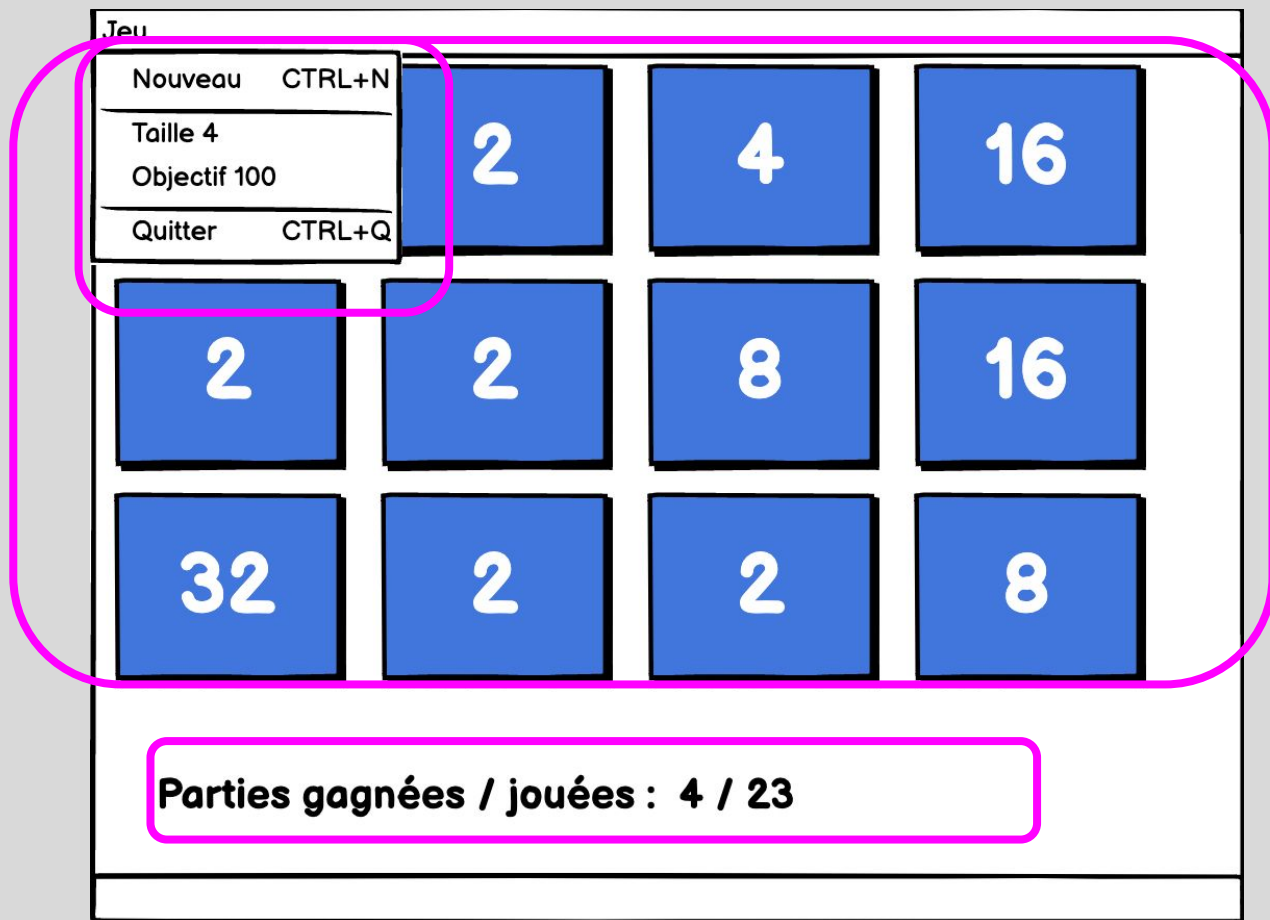
Taille 4

Objectif 100

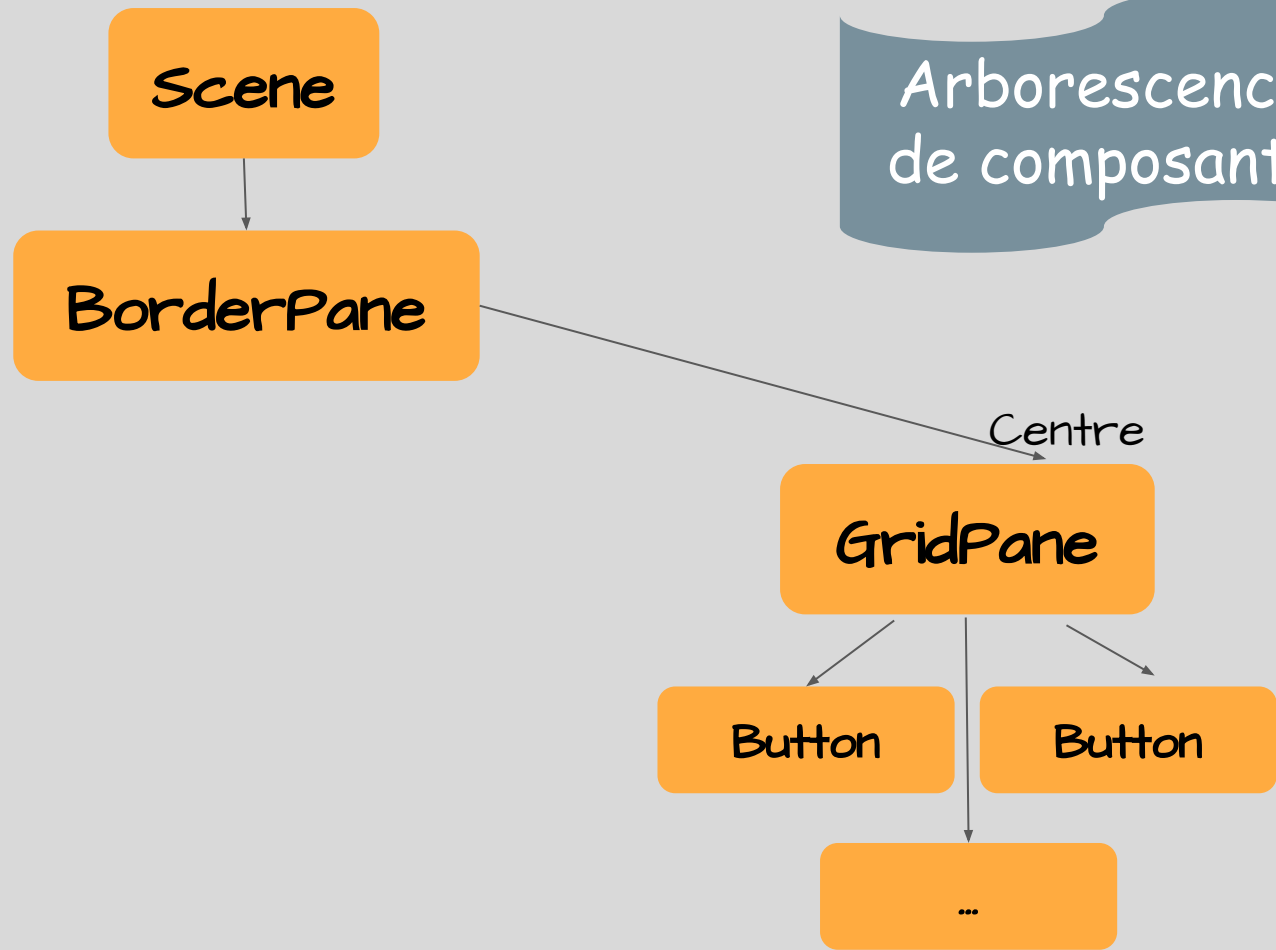
QuitterCTRL+Q

| | | | |
|----|---|----|----|
| 2 | 4 | 16 | |
| 2 | 2 | 8 | 16 |
| 32 | 2 | 2 | 8 |

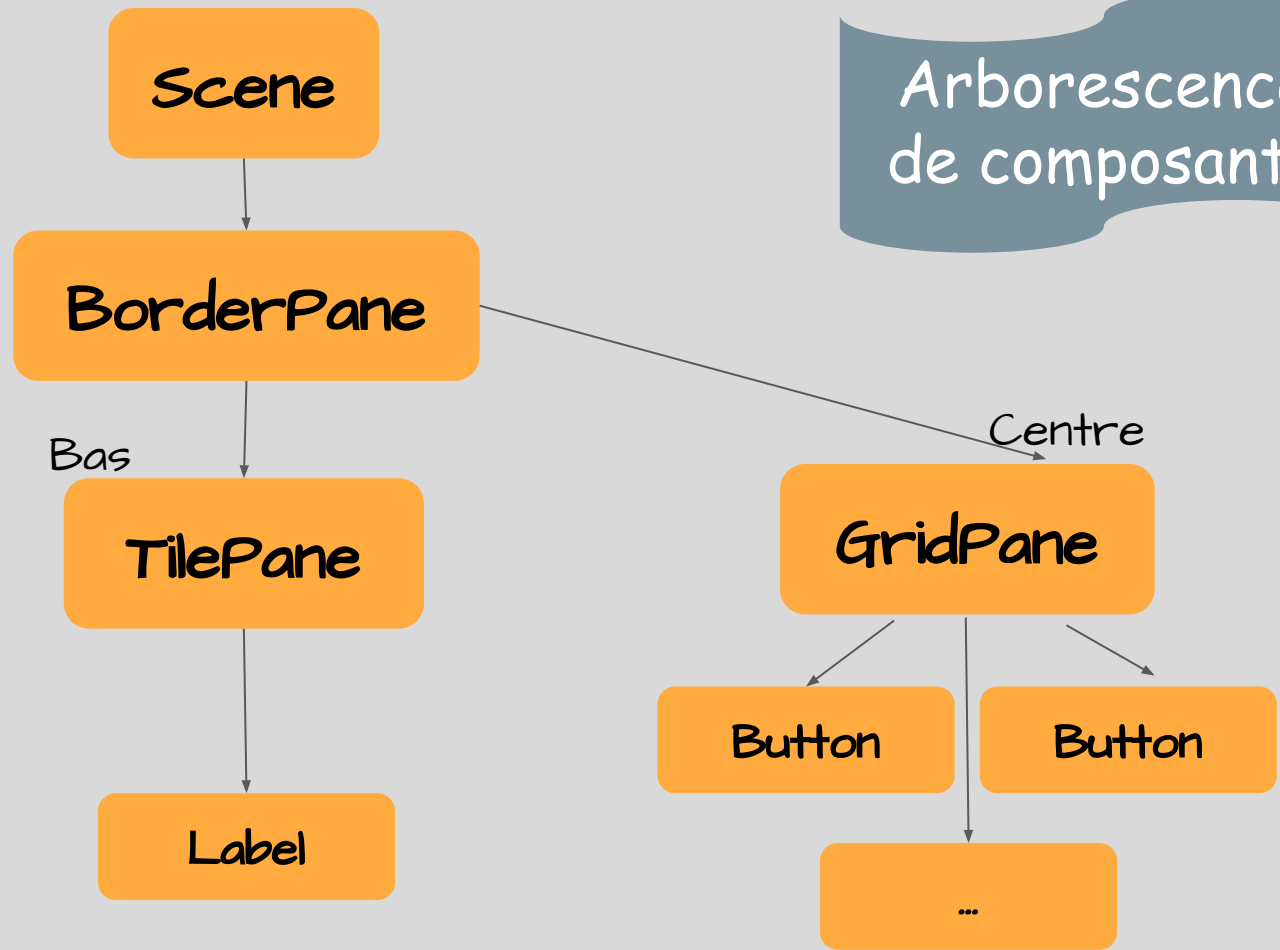
Parties gagnées / jouées : 4 / 23



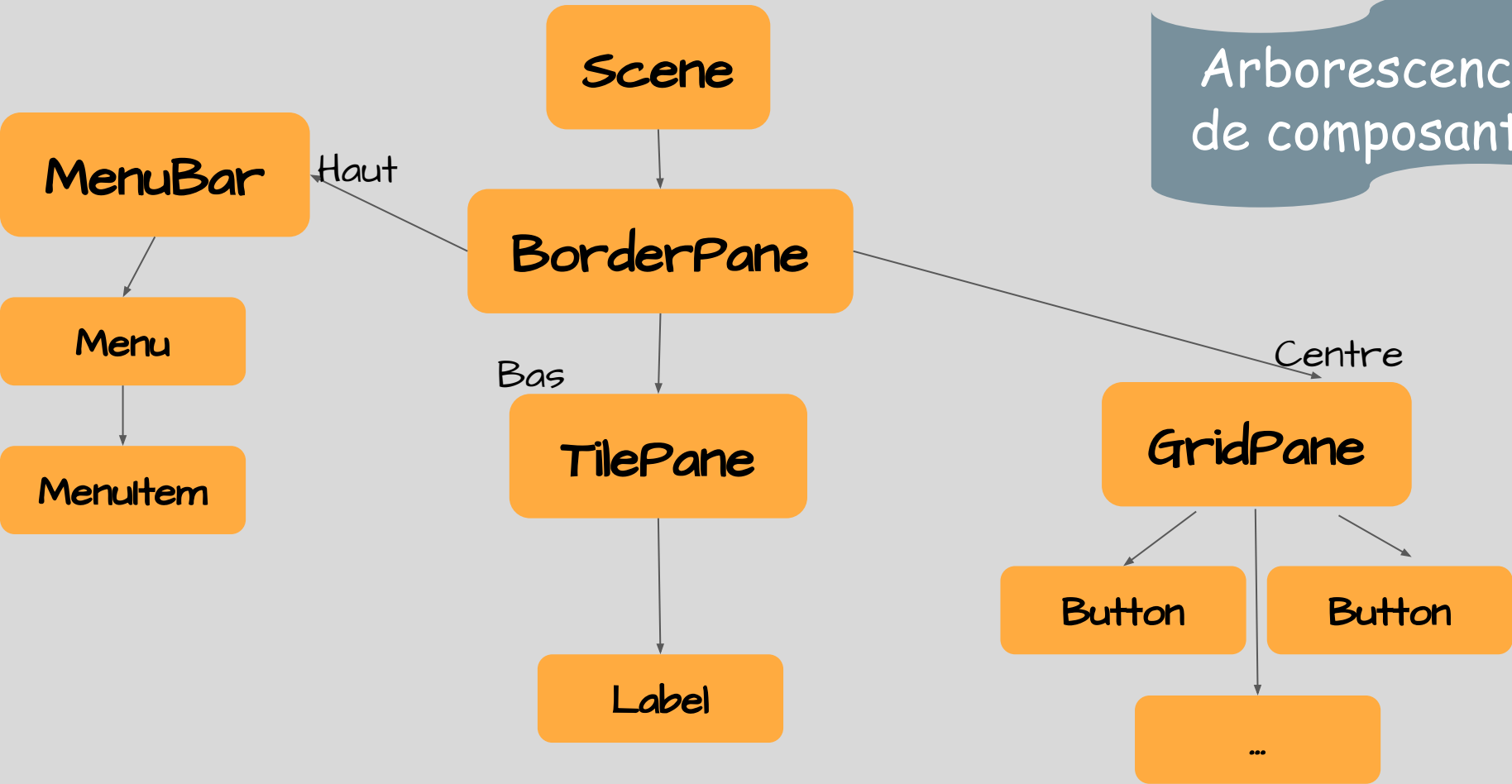
Arborescence de composants



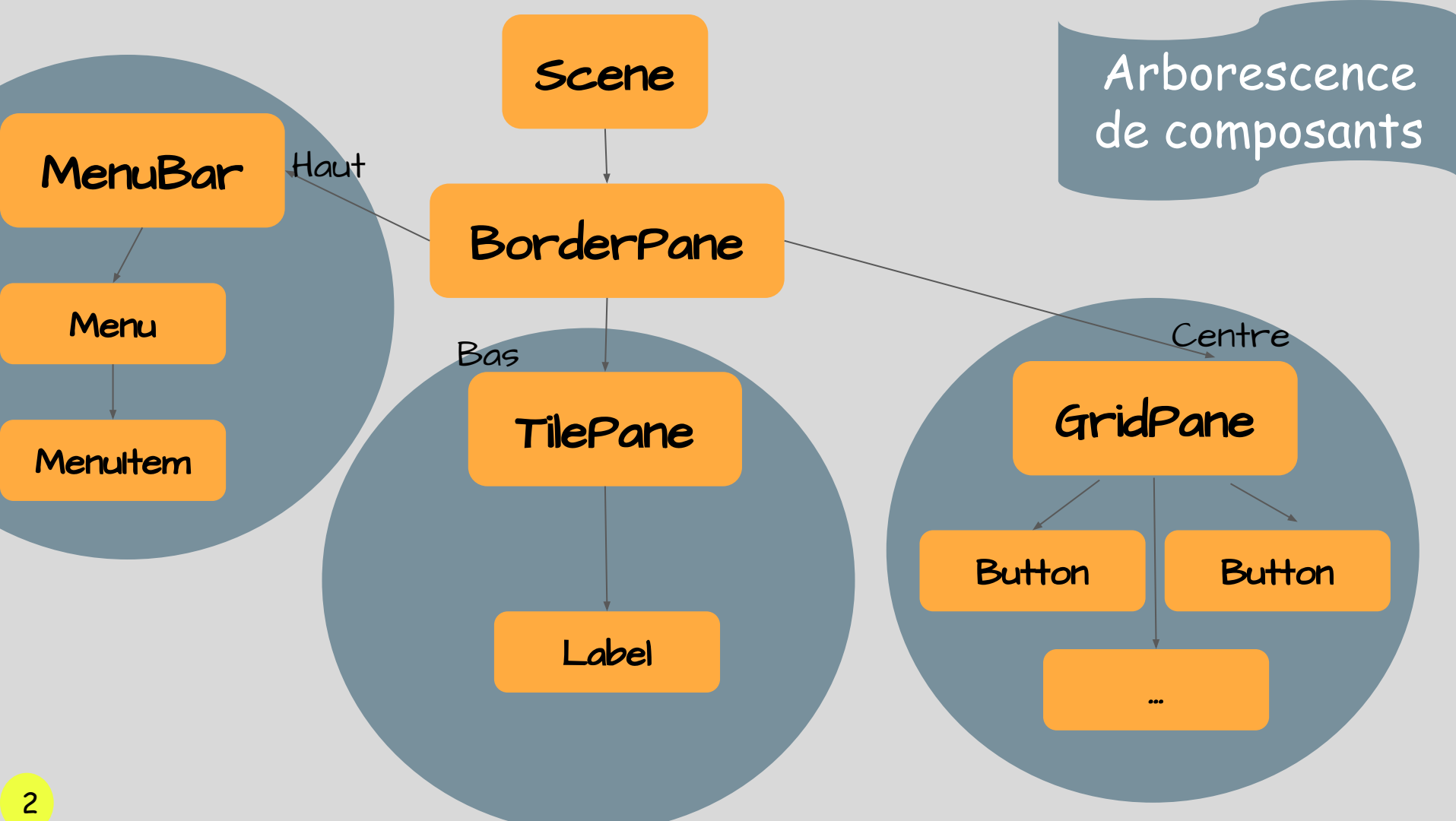
Arborescence de composants



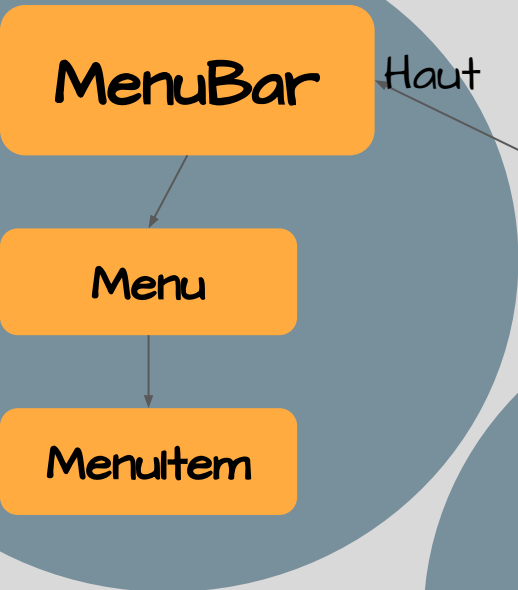
Arborescence de composants



Arborescence de composants



VueMenu



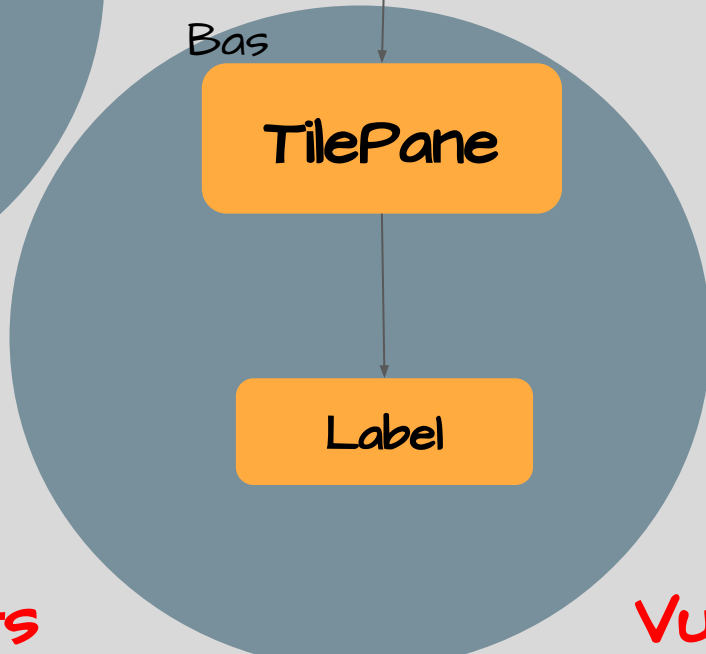
Scene

BorderPane

Bas

TilePane

Label



Arborescence
de composants

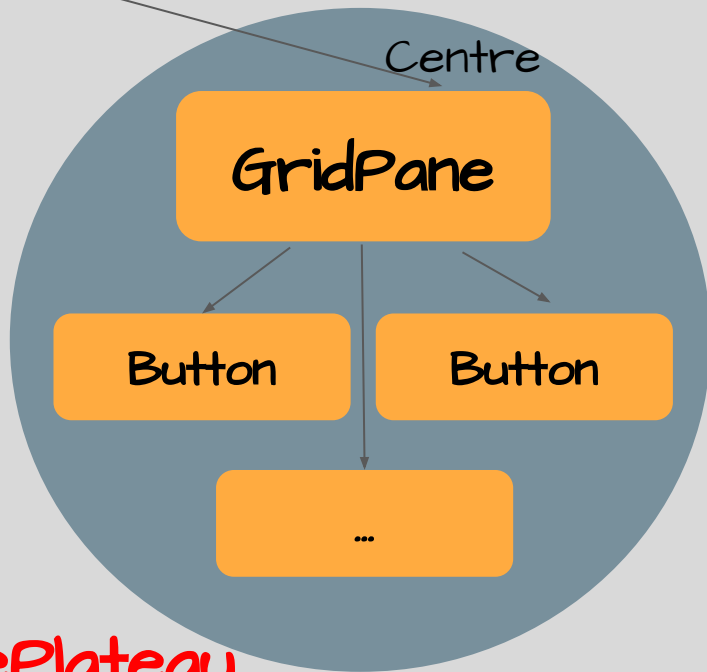
Centre

GridPane

Button

Button

...

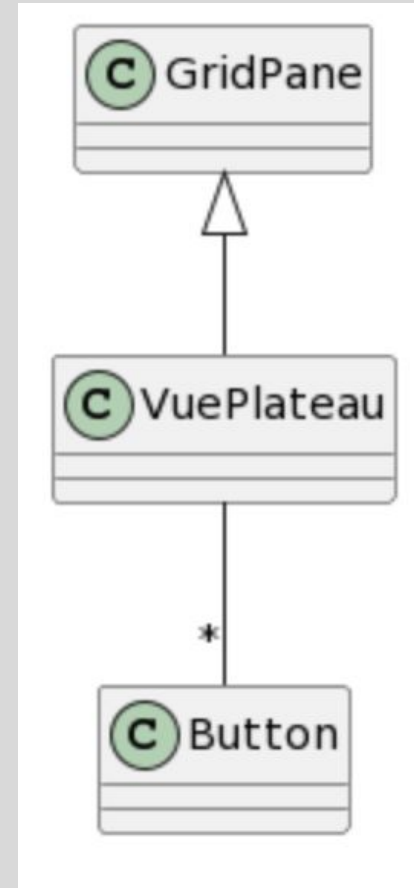
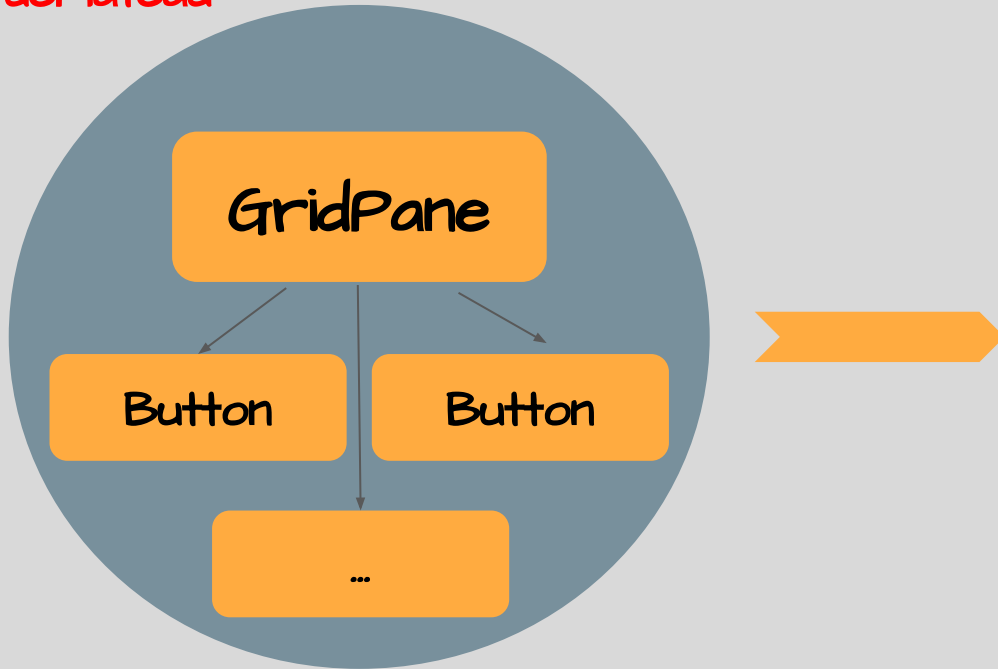


4 classes, en plus de la classe Main

- VueMenu, VueStats, et VuePlateau sont des composants graphiques ; ce sont différentes vues du modèle.
- Jeu gère le modèle (les valeurs des cases, le nombre de parties, etc.)

De l'arborescence au diagramme de classes UML

VuePlateau



De la même
façon pour
chaque panneau

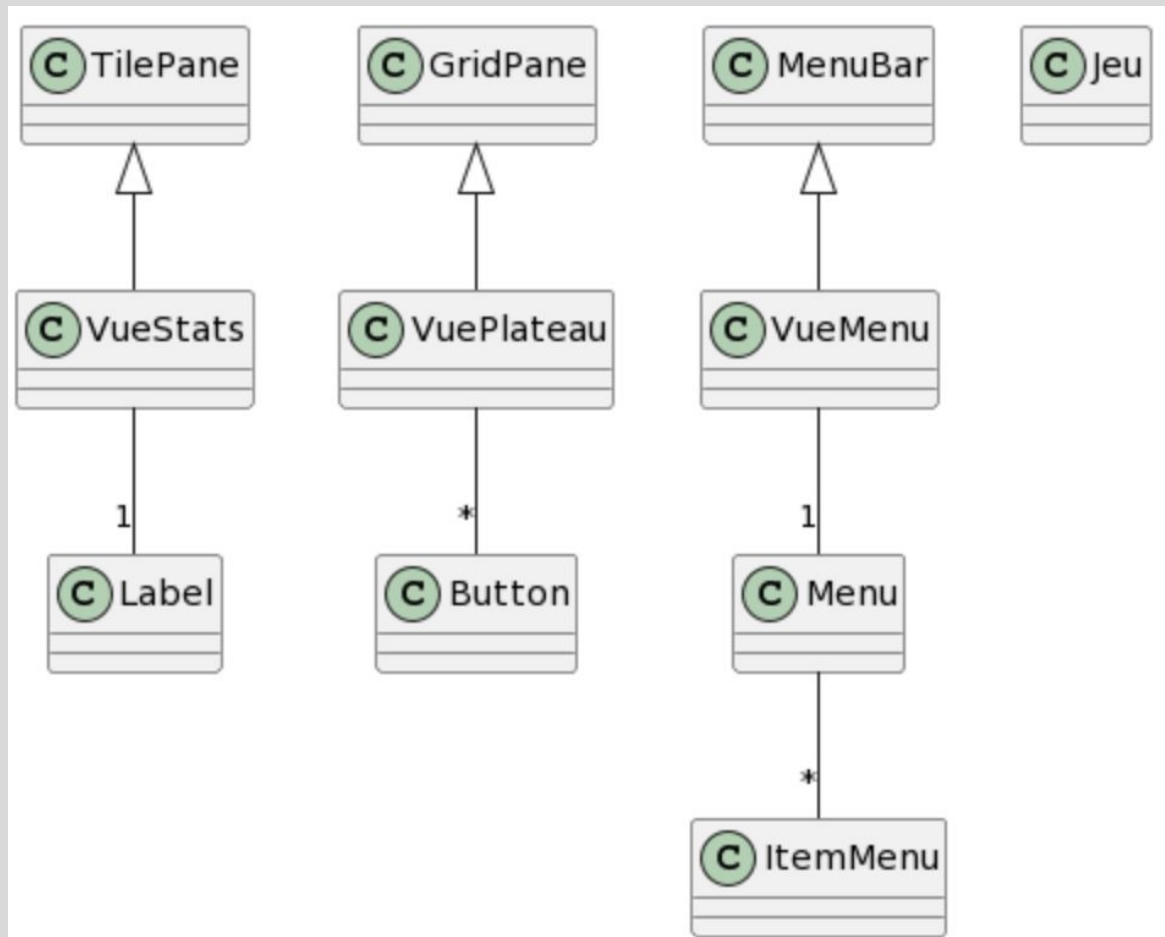


Diagramme incomplet

On ajoute le modèle de données

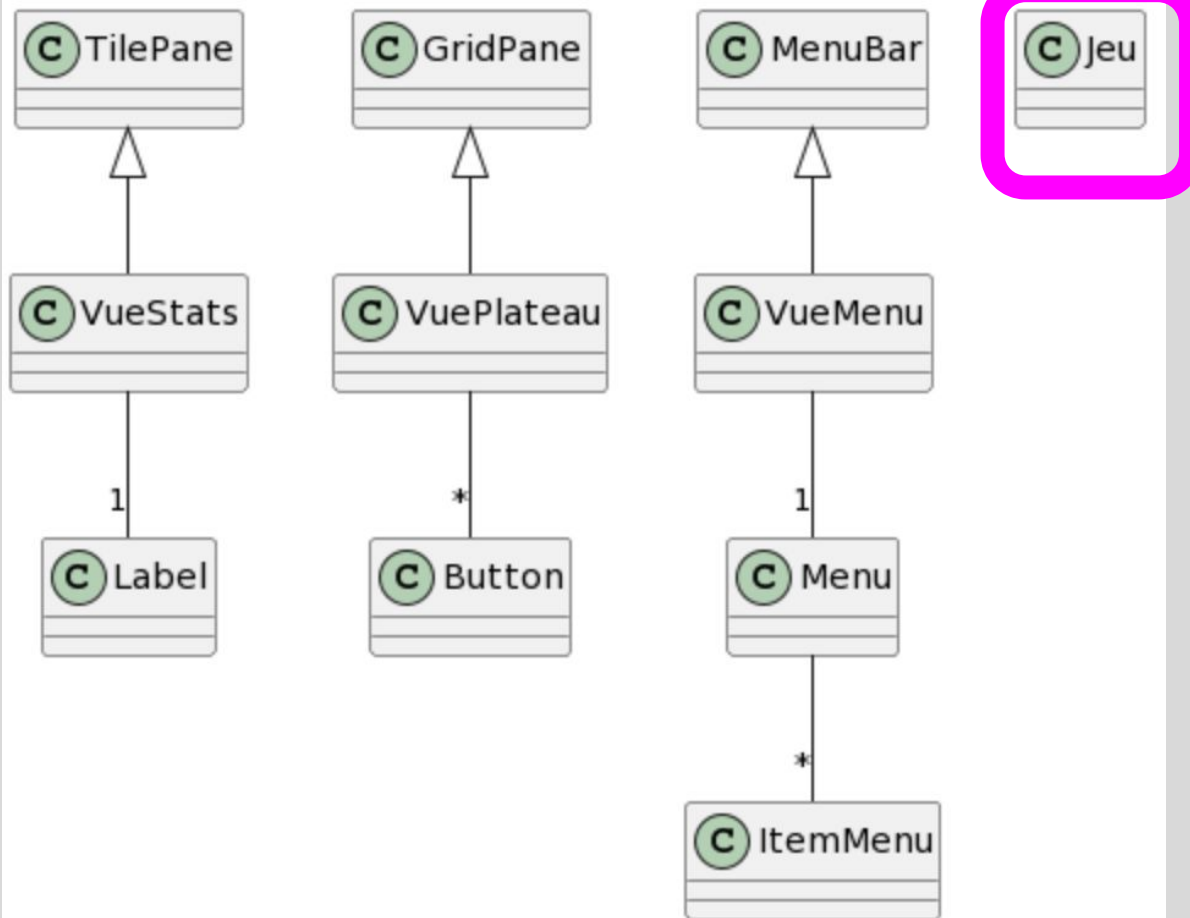


Diagramme incomplet

Un peu de code pour comprendre (incomplet)

```
public class VueMenu extends MenuBar {  
  
    public VueMenu (. . . ) {  
        super() ;  
        Menu menu = new Menu("Jeu");  
        MenuItem nouveau = new MenuItem("Nouveau");  
        nouveau.setAccelerator(KeyCombination.keyCombination("Ctrl+N");  
        MenuItem quitter = new MenuItem("Quitter");  
        quitter.setAccelerator(KeyCombination.keyCombination("Ctrl+Q");  
        ...  
        menu.getItems().addAll(nouveau, quitter);  
        this.getMenus().add(menu)) ;  
    }  
}
```

La classe principale (encore incomplète)

```
public class Main extends Application {  
  
    @override  
    public void start (Stage primaryStage) {  
  
        BorderPane root = new BorderPane() ;  
        root.setTop(new VueMenu(..)) ;  
        root.setCenter(new VuePlateau(..)) ;  
        root.setBottom(new VueStats(..)) ;  
        primaryStage.setScene(new Scene(root, 1000, 700));  
        primaryStage.show();  
    }  
    public static void main(String[] args) {  
        launch(args);  
    }  
}
```

Une classe par zone clairement identifiée sur l'IG

- De façon naturelle, les composants regroupés dans un panneau ont un rapport entre eux.
 - = différentes vues (partielles) du modèle
- Chaque classe ainsi créée décrit un nouveau composant graphique, avec des responsabilités limitées
 - Rafraîchir les informations si besoin
 - Gérer la réactivité de ses composants

Gérer la réactivité des composants

- Créer un écouteur à l'item de menu *Quitter*

```
public class VueMenu extends MenuBar {  
    public VueMenu(...) {  
        ...  
        MenuItem quitter = new MenuItem("Quitter ") ;  
        quitter.setOnAction(event -> Platform.exit()) ;  
        ...  
    }  
}
```

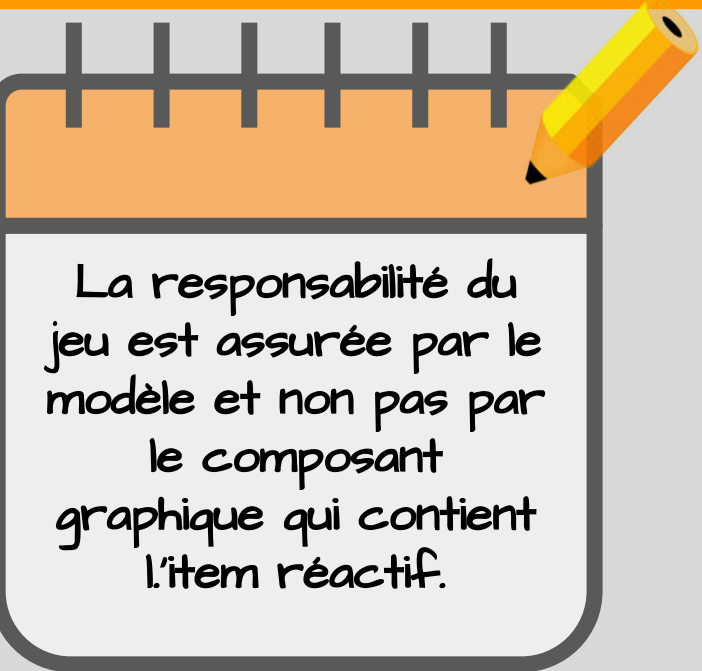
Gérer la réactivité des composants

- Créer un écouteur à l'item de menu *Nouveau*

```
public class VueMenu extends MenuBar {  
    public VueMenu(...) {  
        ...  
        MenuItem nouveau = new MenuItem("Nouveau ");  
        nouveau.setOnAction( ..... );  
        ...  
    }  
}
```




Gérer la réactivité des composants

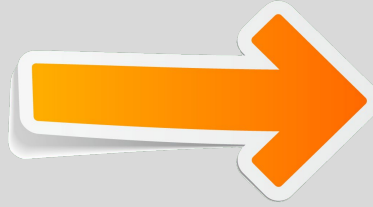
A stylized illustration of a notepad with a yellow cover and a yellow pencil resting on it. The notepad has a white page with a grey border and a grey spiral binding on the left. The text is written in a black, handwritten-style font.

La responsabilité du
jeu est assurée par le
modèle et non pas par
le composant
graphique qui contient
l'item réactif.

Gérer la réactivité des composants

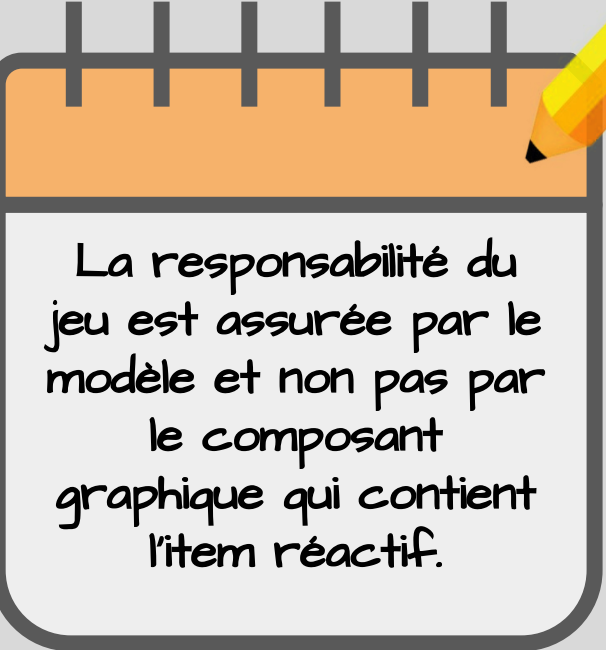



La responsabilité du jeu est assurée par le modèle et non pas par le composant graphique qui contient l'item réactif.

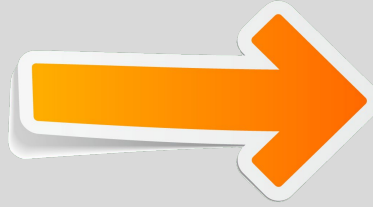


L'écouteur prévient le modèle qu'il réinitialiser les valeurs des cases.

Gérer la réactivité des composants



La responsabilité du jeu est assurée par le modèle et non pas par le composant graphique qui contient l'item réactif.



L'écouteur prévient le modèle qu'il réinitialiser les valeurs des cases.



L'écouteur doit connaître le modèle.

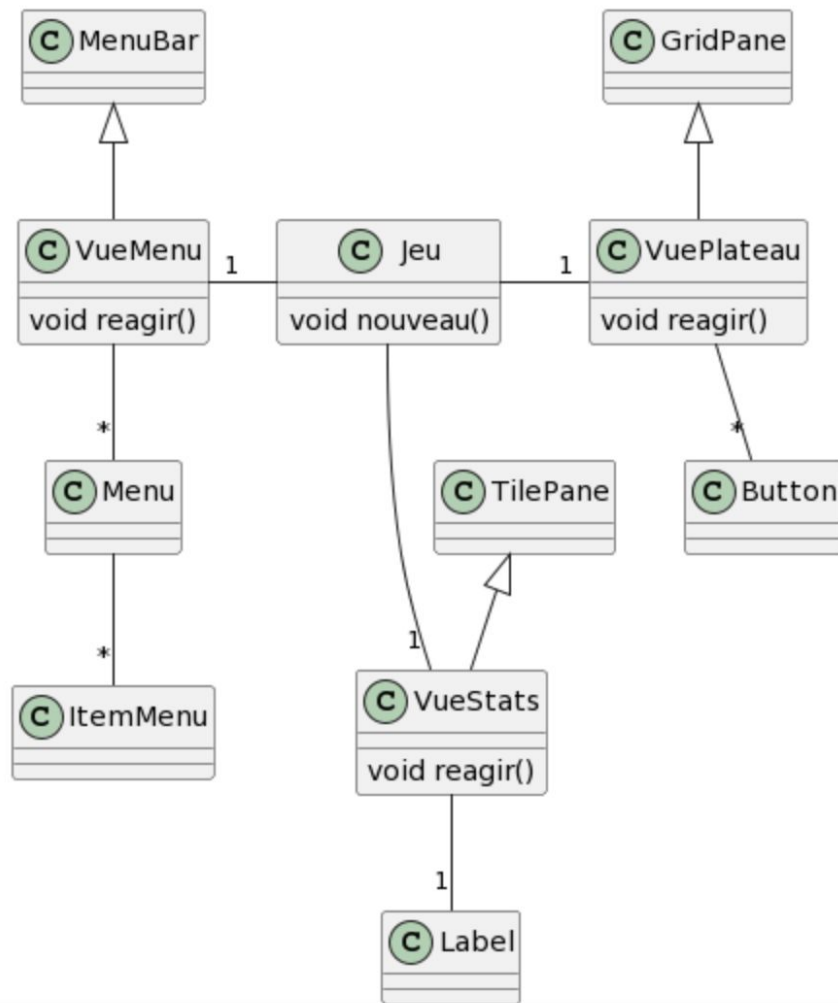
Gérer la réactivité des composants

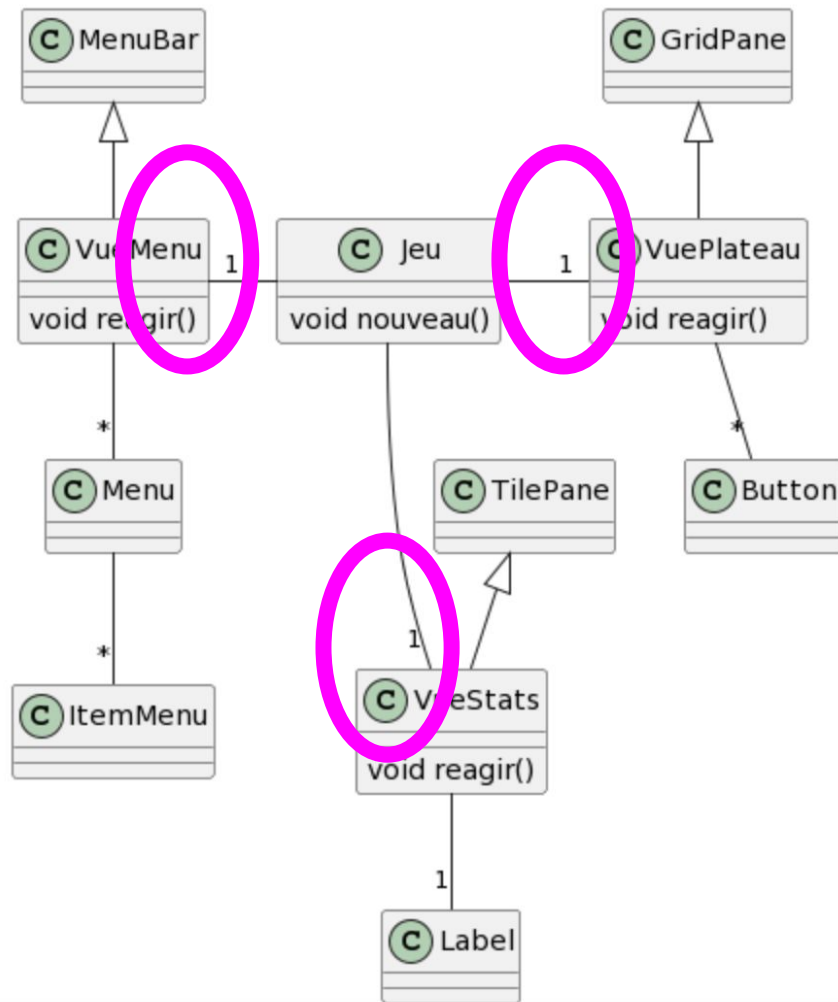
- Créer un écouteur à l'item de menu *Nouveau*

```
public class VueMenu extends MenuBar {  
    public VueMenu(Jeu jeu) {  
        ...  
        MenuItem nouveau = new MenuItem("Nouveau ") ;  
        nouveau.setOnAction( event -> jeu.nouveau() ) ;  
        ...  
    }  
}
```

Rafraichir les composants, si besoin

- La fonction nouveau de la classe **Jeu** met à jour les données.
 - = réinitialise les valeurs des cases
 - = comptabilise le nombre de parties
- Comment rafraîchir les composants ?
 - Solution : le modèle connaît tous les composants graphiques et leur demande de se rafraîchir.
 - = chaque composant définit la fonction reagir()







```
public class Jeu    {  
    private VuePlateau vp ;  
    private VueStats vs ;  
    private VueMenu vm ;  
    public Jeu(VuePlateau vp, VueStats vs, VueMenu pvm)  
    {  
        ...  
        this.vp = vp ; this.vs = vs ; this.vm = vm ;  
    }  
    public void nouveau() {  
        ... // Réinitialiser le plateau  
        this.vp.reagir() ;  
        this.vs.reagir() ;  
        this.vm.reagir() ;  
    }  
}
```

```
public class Jeu {  
    ...  
    public Jeu( ... ) {  
        ...  
    }  
    public void nouveau() {  
        ... // Réinitialiser le jeu  
        prévenirVues();  
    }  
    public void prévenirVues() {  
        this.vp.reagir() ;  
        this.vs.reagir() ;  
        this.vm.reagir() ;  
    }  
}
```

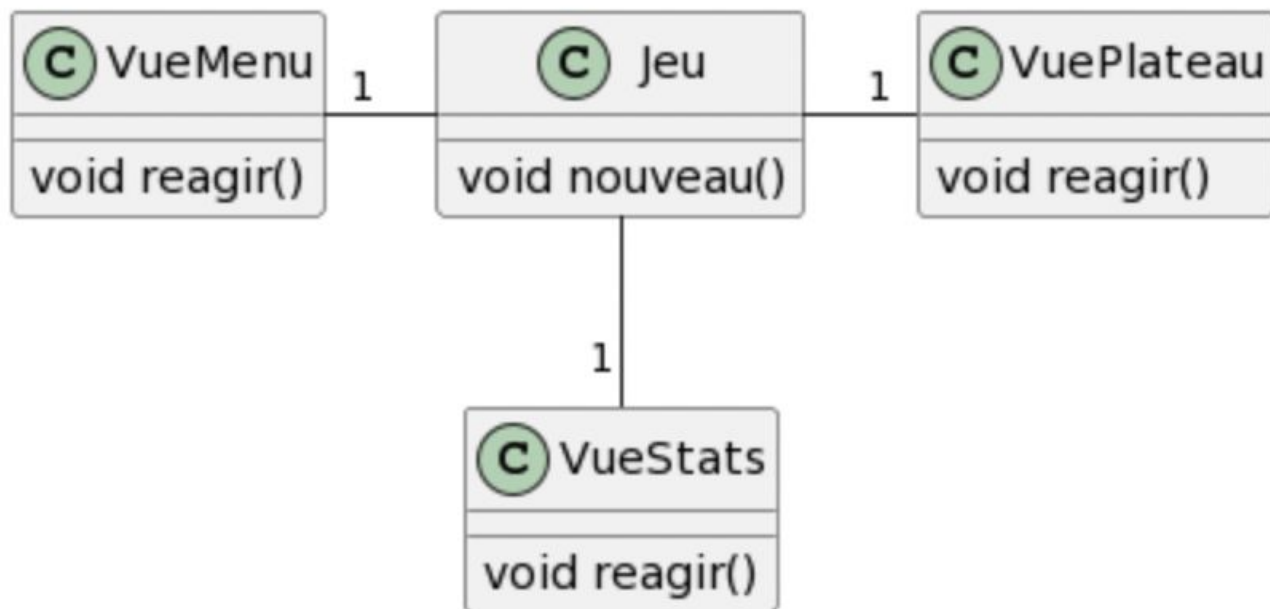
En factorisant le code

```
public class VueStats extends TilePane {
    private Label stats ;
    private Jeu jeu ;
    public VueStats(Jeu jeu) {
        ...
        this.jeu = jeu ;
    }
    public void reagir() {
        String st = "Parties gagnées/jouées : ";
        st += jeu.getNbGagnees() + "/" + jeu.getNbJouees();
        this.stats.setText(st);
    }
}
```

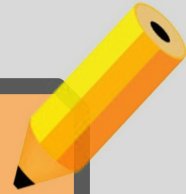
Rafraîchir les composants

- Que penser de cette solution ?
 - Chaque composant se rafraîchit tout seul en puisant les infos utiles dans le modèle. 
 - Solution évolutive ? Modification/Ajout de composant ? 
- D'où vient le problème ?
 - Le modèle est dépendant des noms et du nombre de composants.

→ mauvaise évolutivité

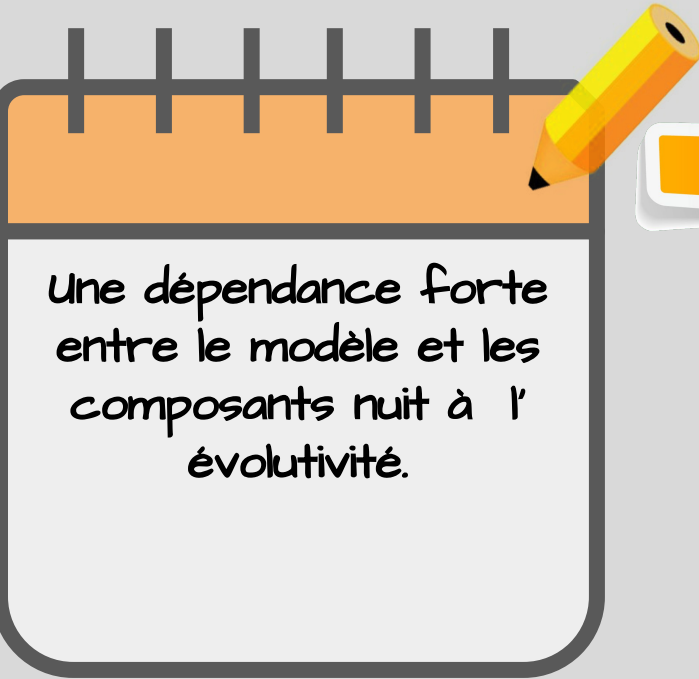


Gérer la réactivité des composants



Une dépendance forte
entre le modèle et les
composants nuit à l'
évolutivité.

Gérer la réactivité des composants

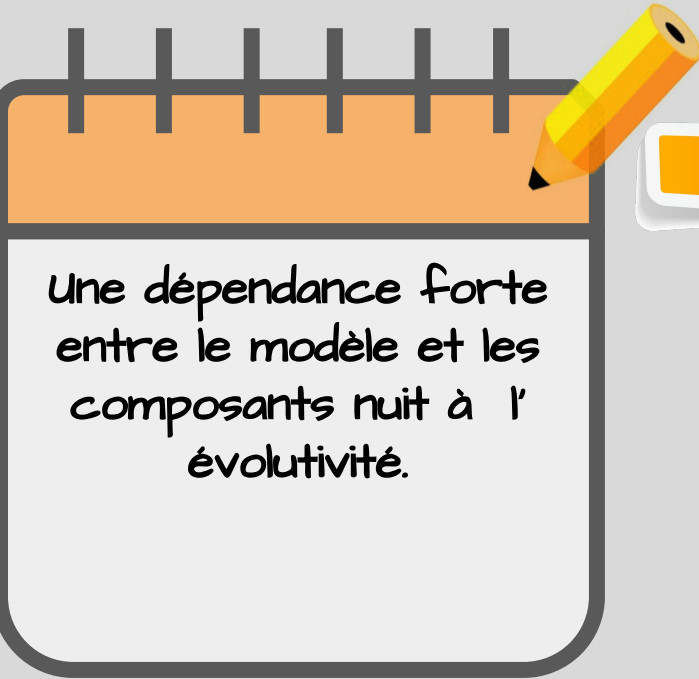


Une dépendance forte
entre le modèle et les
composants nuit à l'
évolutivité.

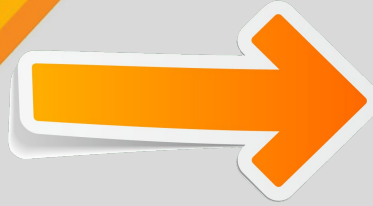


Réduction du couplage.

Gérer la réactivité des composants



Une dépendance forte
entre le modèle et les
composants nuit à l'
évolutivité.



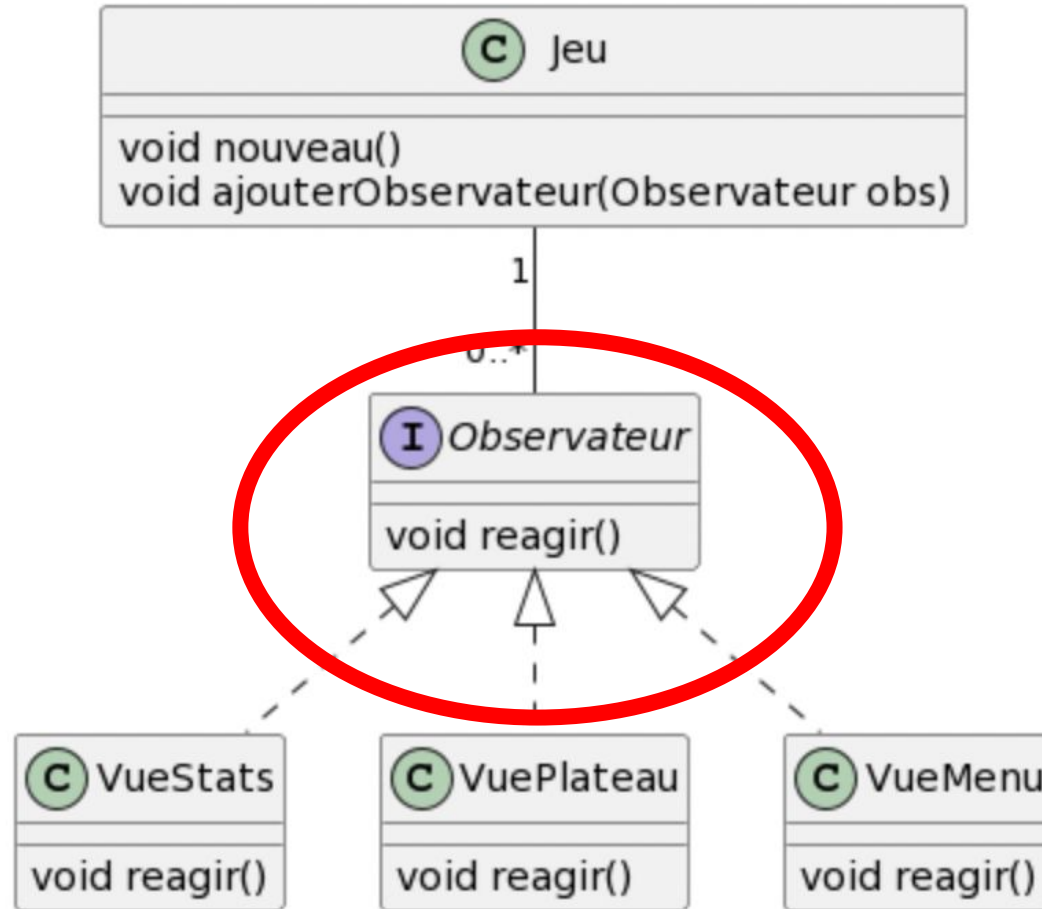
Réduction du couplage.



Introduire une
interface, dont le
modèle dépend.

Modèle indépendant du nom et du nombre de composants

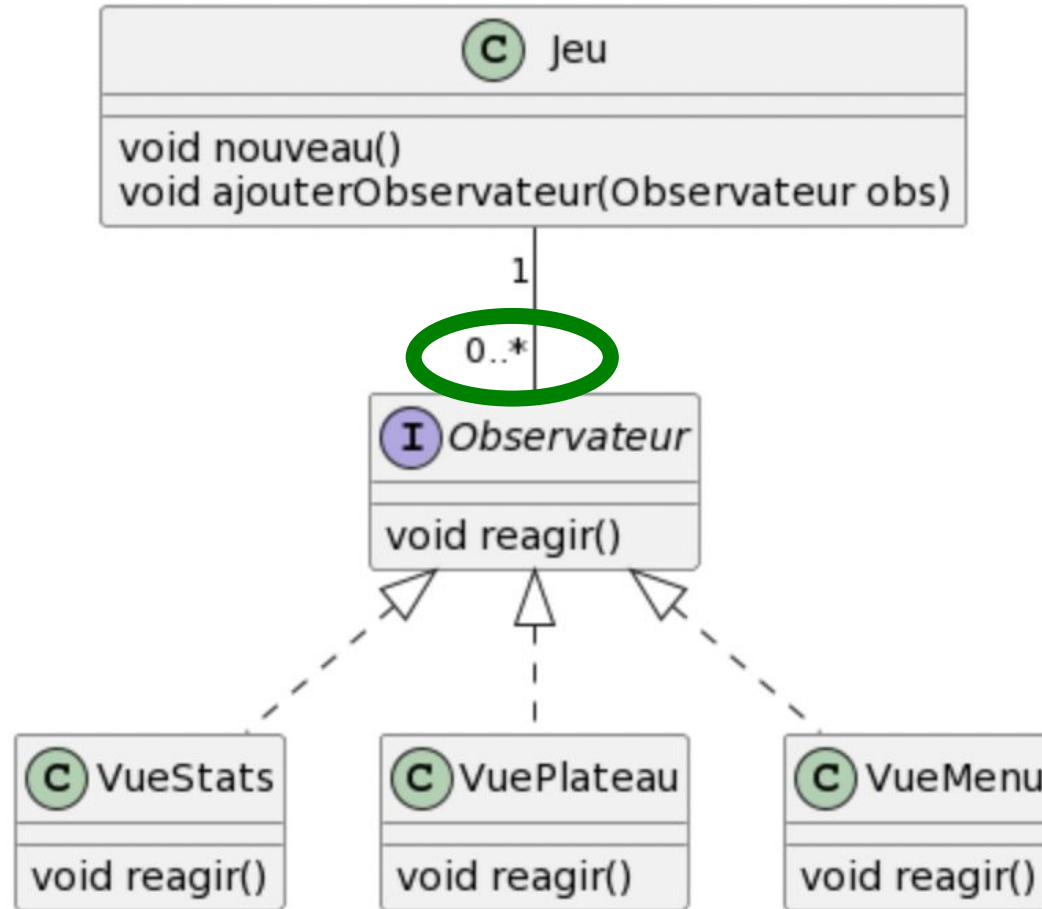
Les vues
implémentent
une interface
commune.



Modèle indépendant du nom et du nombre de composants

Les vues implémentent
une interface commune.

Le modèle
connaît un
nombre
quelconque
d'observateurs

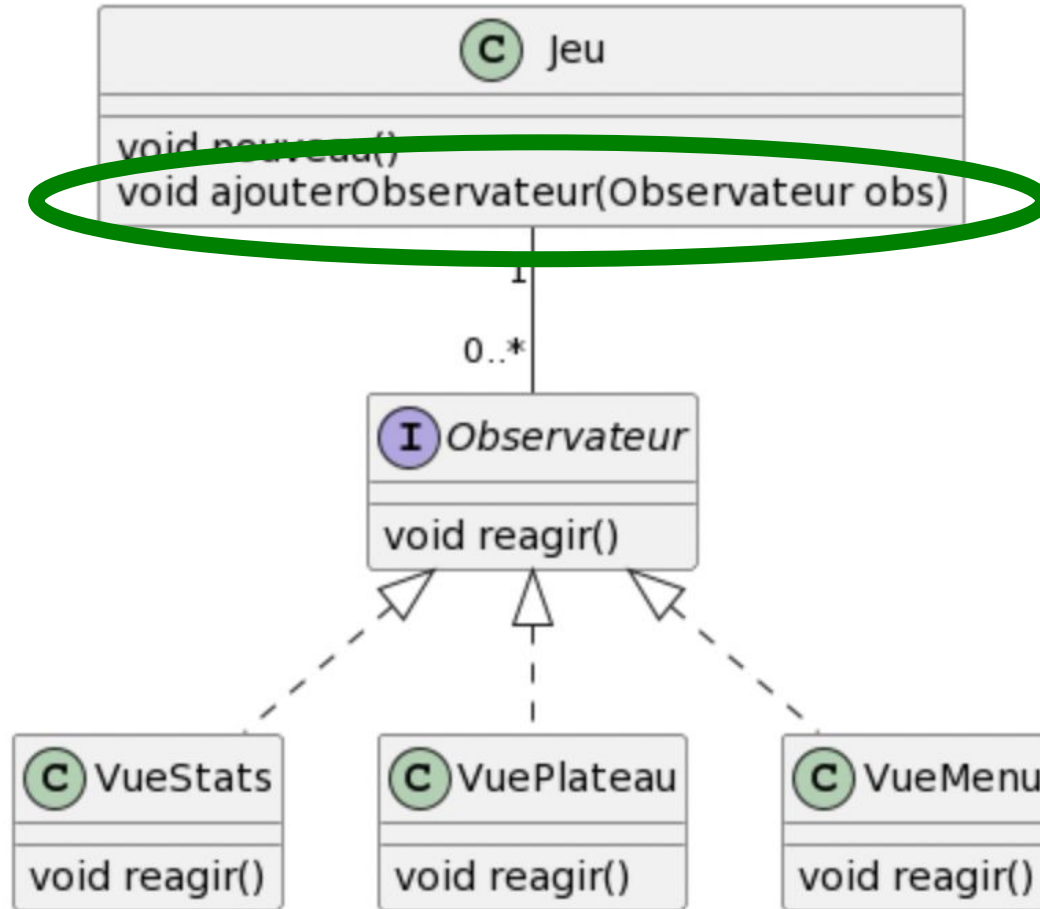


Modèle indépendant du nom et du nombre de composants

Les vues implémentent
une interface commune.

Le modèle connaît un
nombre quelconque
d'observateurs.

Pour ajouter un
observateur au
modèle

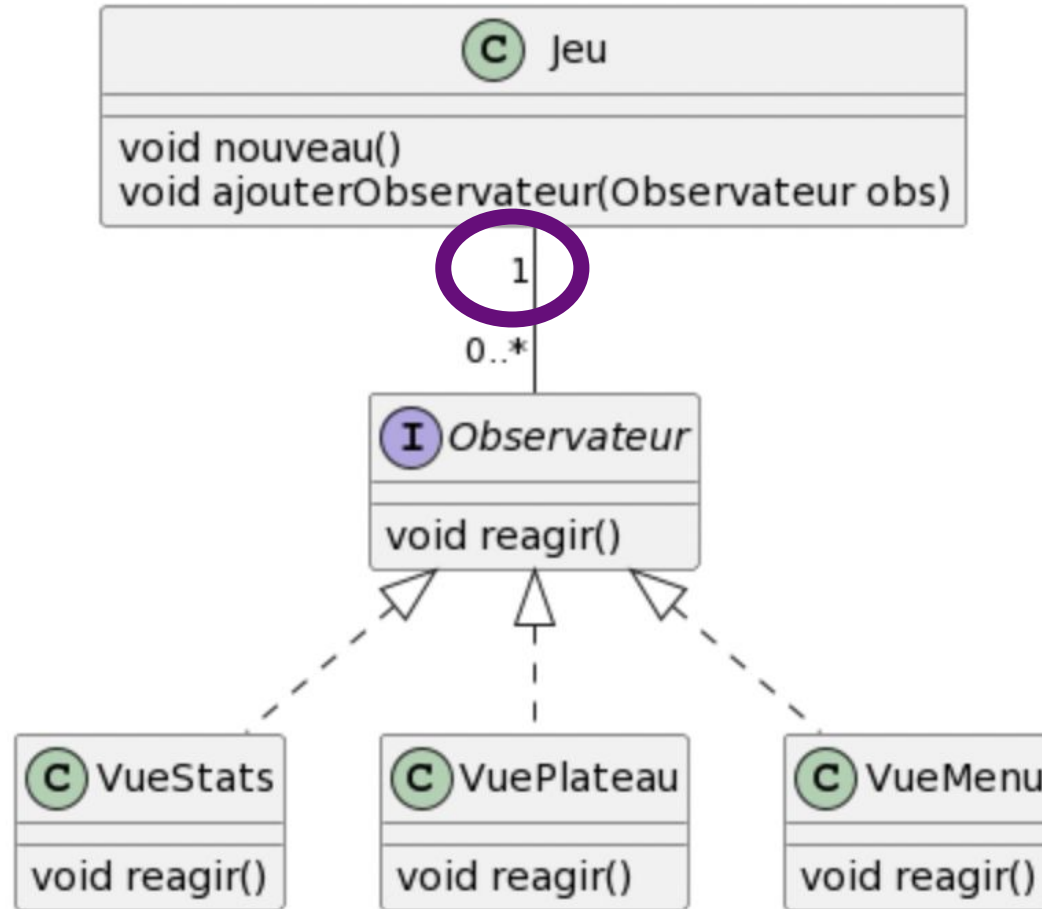


Modèle indépendant du nom et du nombre de composants

Les vues implémentent
une interface commune.

Le modèle connaît un
nombre quelconque
d'observateurs.

Chaque
observateur
connaît le
modèle.



```
public class Jeu    {  
    private ArrayList<Observateur> obs = new ArrayList<>(10);  
    public Jeu() { ... }  
    public void ajouterObservateur(Observateur o) {  
        this.obs.add(o) ;  
    }  
    public void nouveau() {  
        ...      // Réinitialiser le plateau  
        this.notifierObservateurs() ;  
    }  
    public void notifierObservateurs() {  
        for (Observateur o : this.obs) o.reagir() ;  
    }  
}
```

```
public class VueStats extends Pane
                                implements Observateur {

    private Label stats ;
    private Jeu jeu ;
    public VueStats(Jeu jeu) {
        ...
        this.jeu = jeu ;
        this.jeu.ajouterObservateur(this) ;
    }
    public void reagir() {
        ...
    }
}
```



Chaque vue est responsable de son inscription auprès du modèle.

Application du design pattern Observer

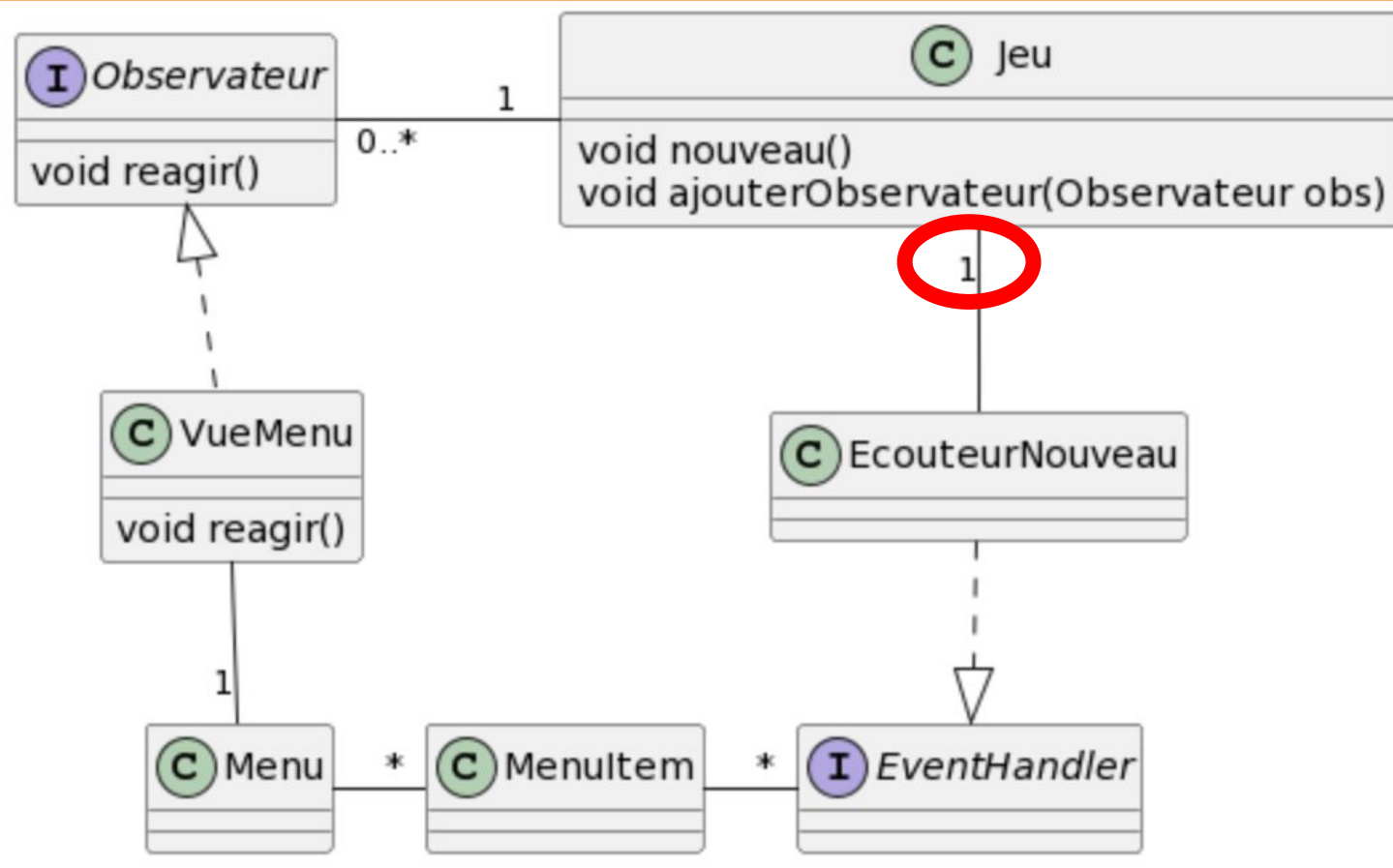
- Le modèle connaît les vues.
 - pas nominativement, pour ne pas créer de dépendance superflue
- Le modèle, lorsqu'il est modifié, prévient les vues de sorte que celles-ci se rafraichissent.
 - chaque fonction de transformation du modèle se termine par `notifierObservateurs()`
- Chaque vue s'inscrit auprès du modèle ; sa fonction `reagir()` rafraîchit le composant en puisant les informations utiles dans le modèle.

Création du modèle et des vues

```
public class Main extends Application {  
  
    @override  
    public void start (Stage primaryStage) {  
  
        Jeu jeu = new Jeu(5, 20) ;  
        BorderPane root = new BorderPane() ;  
        root.setTop(new VueMenu(jeu)) ;  
        root.setCenter(new VuePlateau(jeu)) ;  
        root.setBottom(new VueStats(jeu)) ;  
        primaryStage.setScene(new Scene(root, 450, 400));  
        primaryStage.show();  
    }  
    ....  
}
```

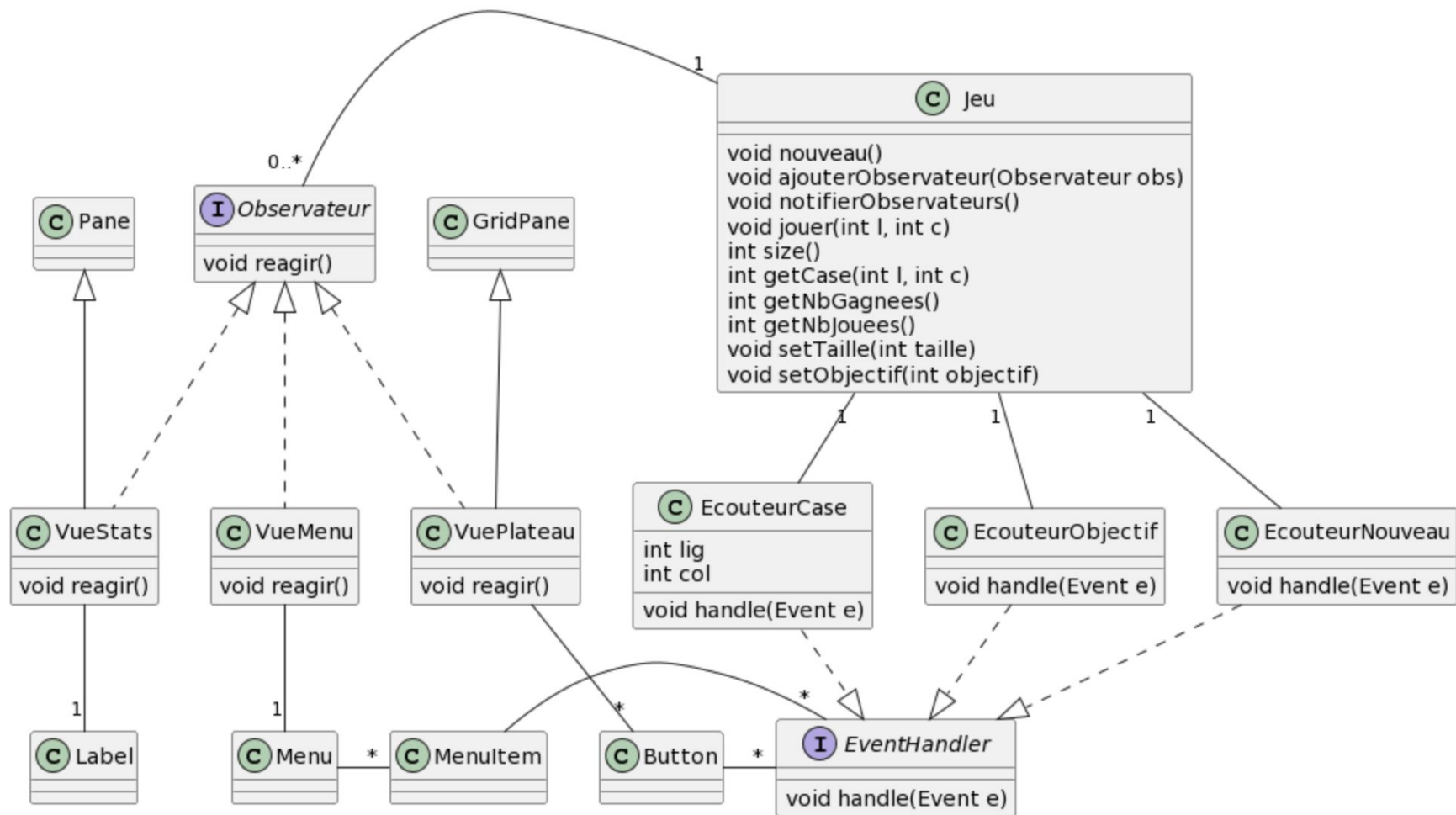
Intégration d'un écouteur

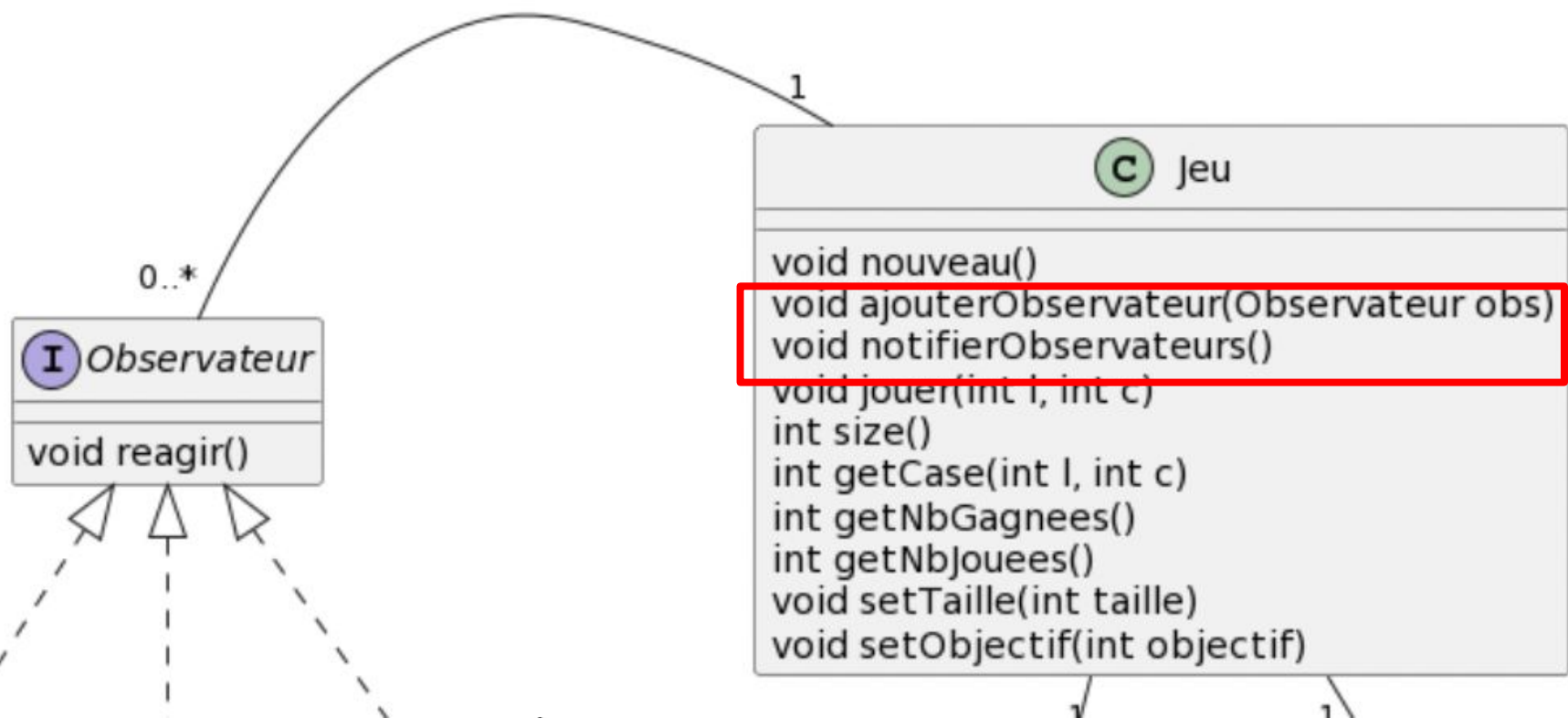
L'écouteur
de l'item
connaît le
modèle pour
lui demander
la
réinitialisation
du jeu.

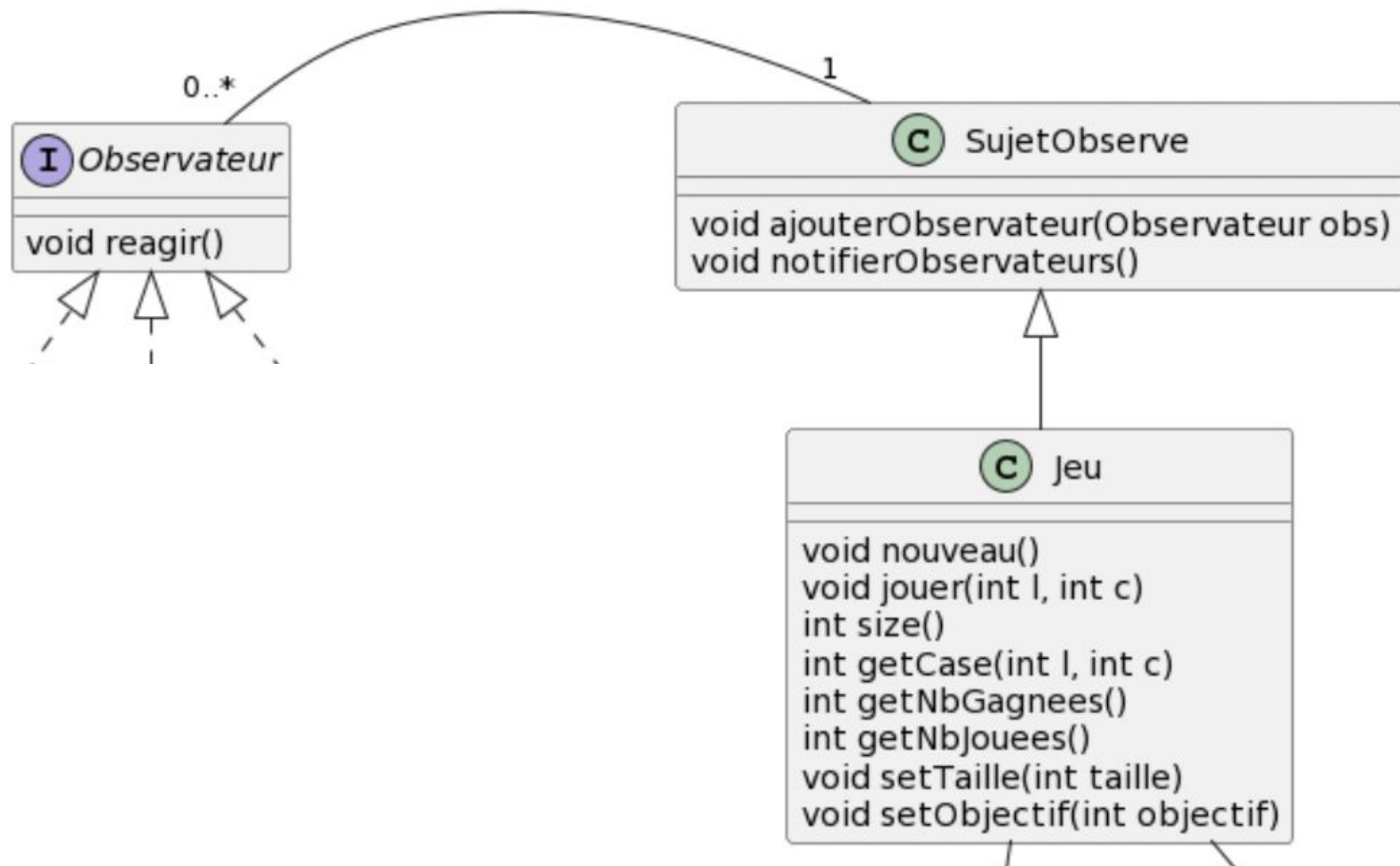


Imaginer les fonctions du modèle

- ❑ Seule une classe façade est en lien avec les vues.
- ❑ Fonctions de transformation
 - utiles aux écouteurs : nouveau, jouer, ...
- ❑ Fonctions d'observation
 - utiles aux vues : getCase, getNbGagnees, ...







De la conception au développement

- Diagramme de classes, incluant toutes les fonctions nécessaires à la construction de l'interface graphique
- Développer les classes de façon incrémentale/itérative
- **Mauvaise méthode**
 - Écrire complètement toutes les classes
 - Lancer l'exécution en priant très fort
- **Bonne méthode**
 - Se concentrer sur une fonctionnalité
 - Ecrire le minimum nécessaire pour la tester



MVW en résumé

- ❑ Le modèle gère les données du jeu et une collection de vues (+ fonctions de gestion de ces collections).
- ❑ Chaque vue est un composant graphique, en général un panneau contenant des composants élémentaires. Elle connaît le modèle, fourni lors de la construction.
- ❑ Chaque vue s'inscrit elle-même auprès du modèle, pour être prévenue en cas de mise à jour.
- ❑ Chaque vue implante l'interface *Observateur*, donc propose la fonction *reagir()*.
- ❑ A la fin de chaque fonction de transformation du modèle, toutes les vues inscrites auprès du modèle sont prévenues ; elles sont rafraîchies par l'exécution de leur fonction *reagir()*.
- ❑ La fonction *start* crée le modèle et place chaque vue dans un panneau.