Software Engineering - Architecture and design patterns for Graphical User Interfaces

Victorien Elvinger

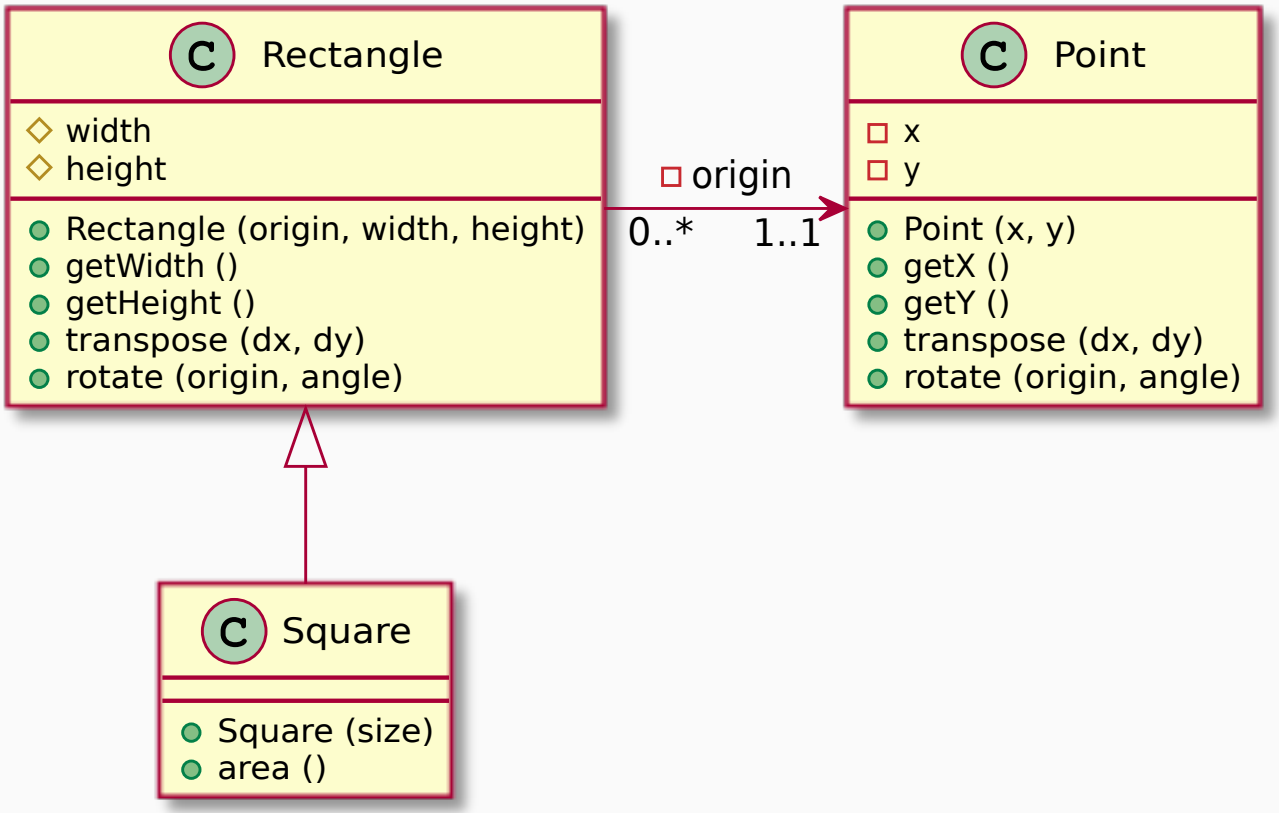
November 2019


Notions de couplage

Modules

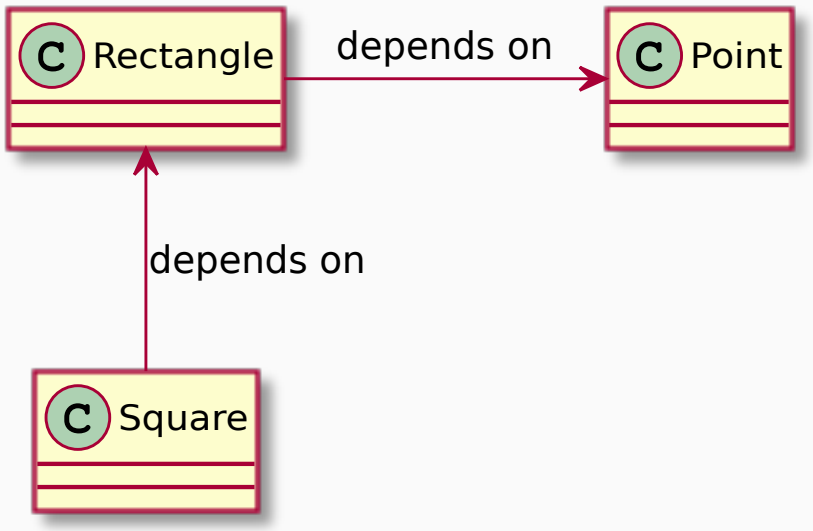
- Un programme est découpé en modules
  - Fonctions
  - Classes et Interfaces
  - Paquets

Dépendances entre modules



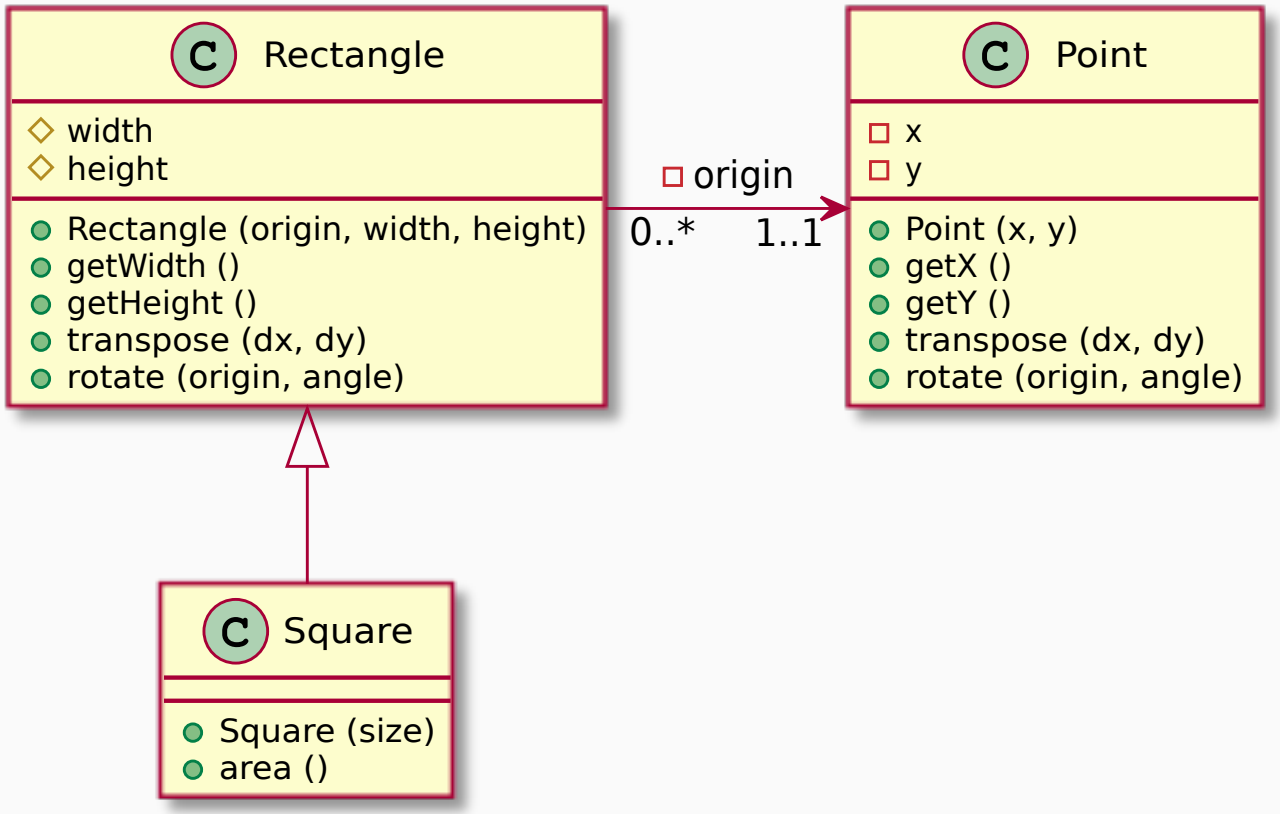
- Un module peut dépendre sur un autre
  - Une fonction qui appelle une autre fonction
  - une classe qui hérite d'une autre classe
  - Une classe en relation cliente avec une autre classe

Dépendances entre modules



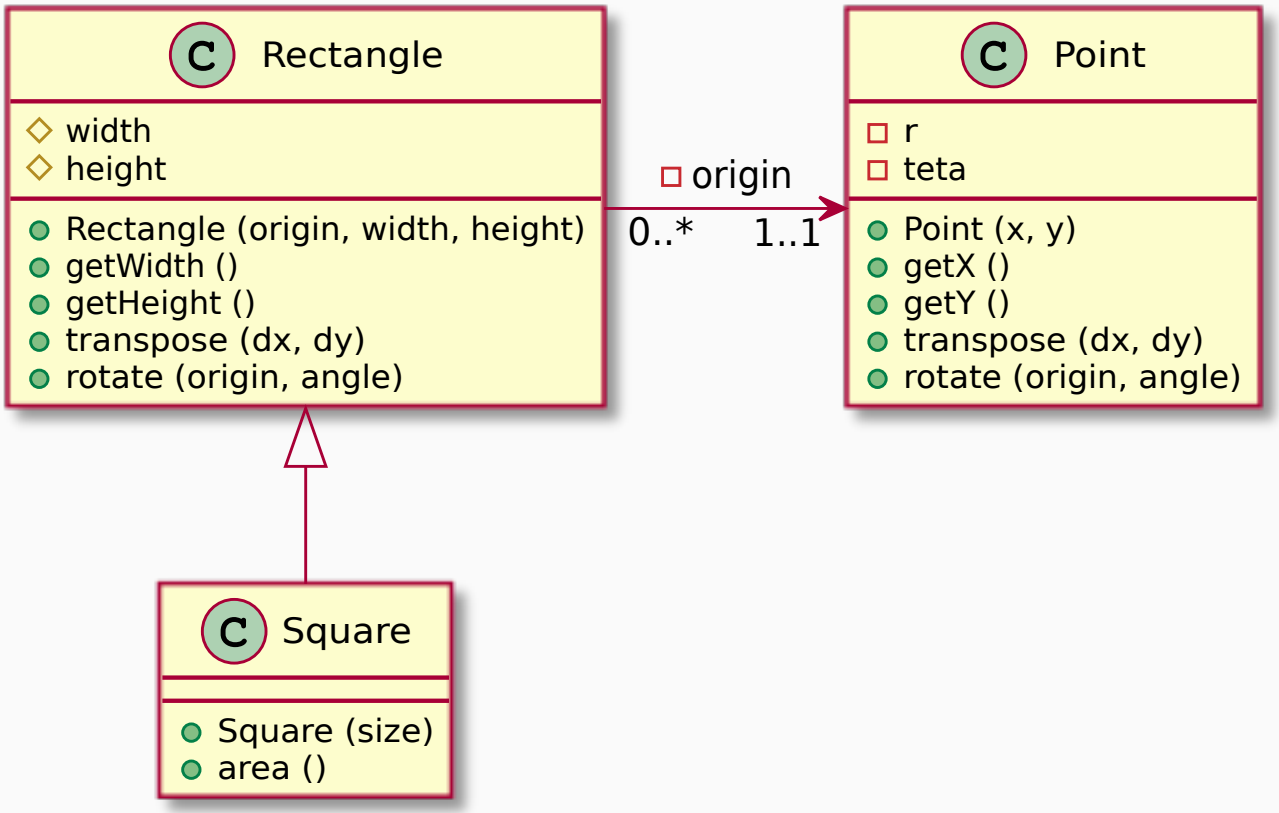
- Un module peut dépendre sur un autre
  - Une fonction qui appelle une autre fonction
  - une classe qui hérite d'une autre classe
  - Une classe en relation cliente avec une autre classe

Couplage



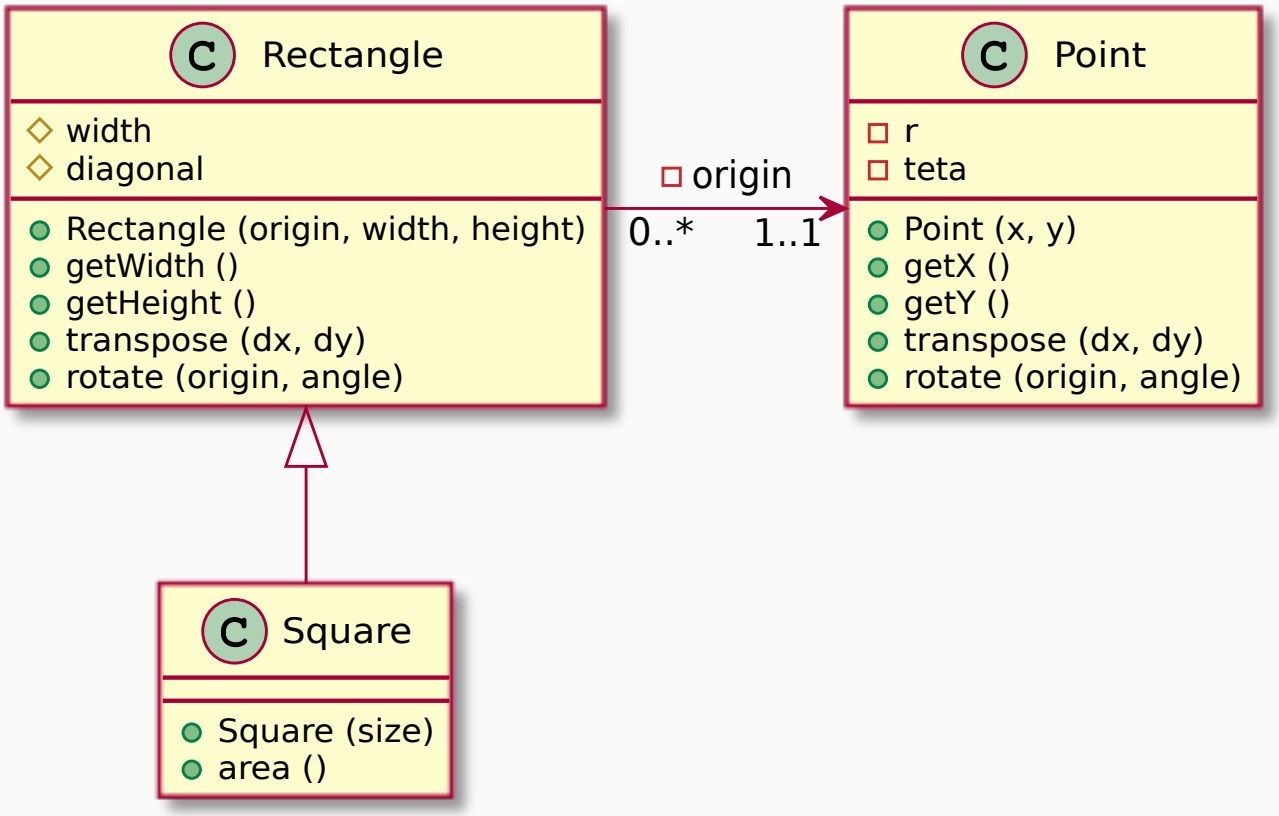
- Degré de dépendance entre deux modules
- Difficulté de modifier indépendemment 2 modules fortement couplés

Couplage



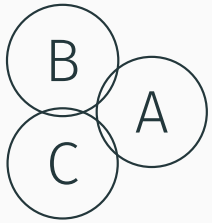
- Degré de dépendance entre deux modules
- Difficulté de modifier indépendemment 2 modules fortement couplés

Couplage

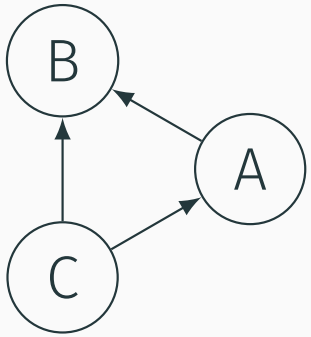


- Degré de dépendance entre deux modules
- Difficulté de modifier indépendemment 2 modules fortement couplés

Découplage et couplage faible



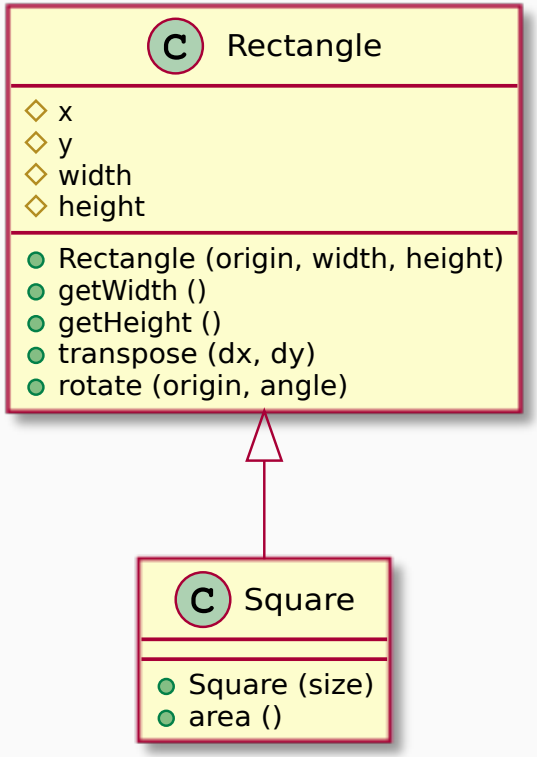
Couplage fort



Couplage plus faible

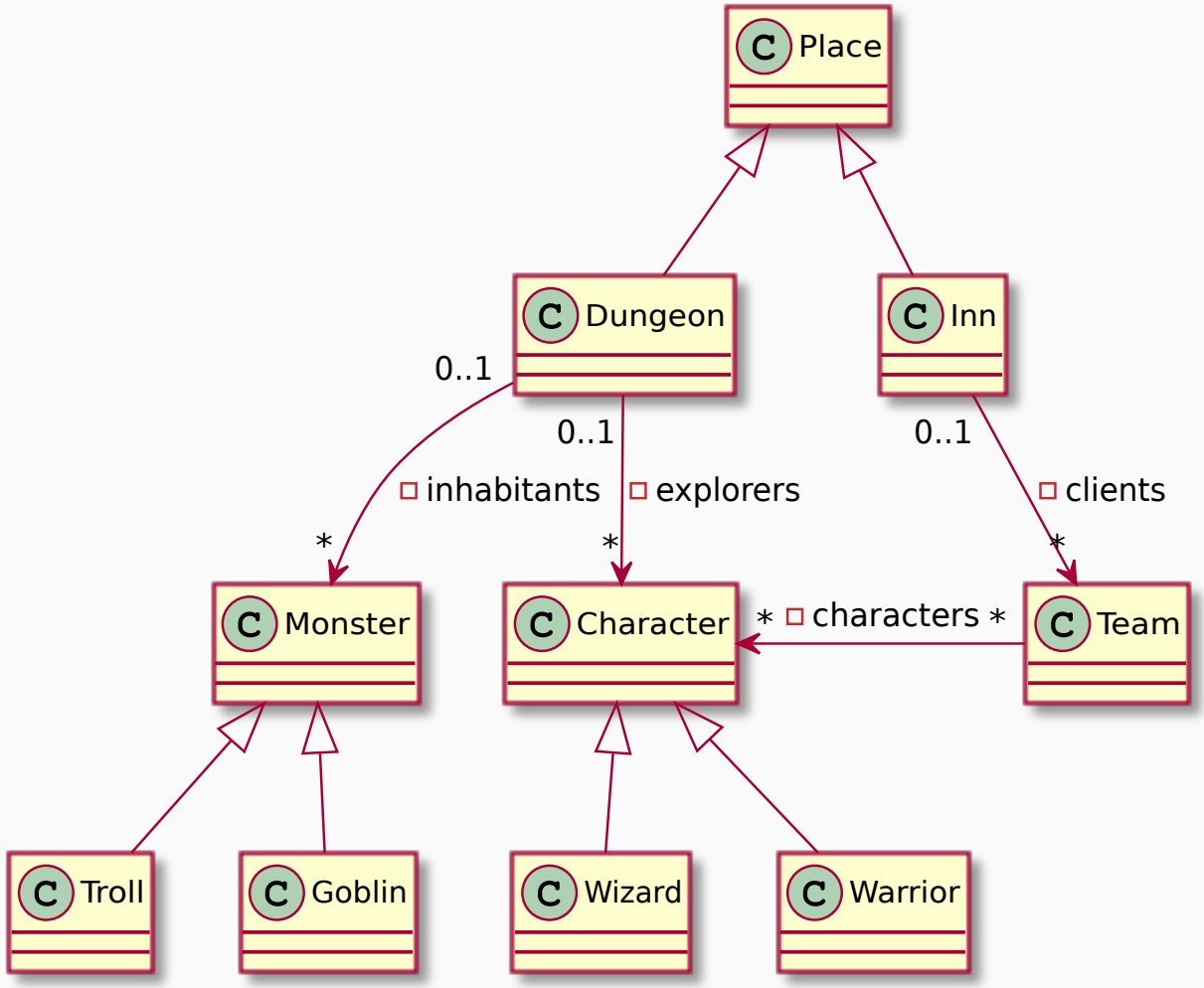
- Faciliter la modification et l'ajout de fonctionnalités
- Faciliter la réutilisation de modules
- Faciliter le travail en équipe
- Faciliter l'écriture de tests unitaires
- Souvent synonyme d'une bonne conception

Découplage et couplage faible



- Encapsulation
- Préférer la composition à l'héritage
- Séparation de préoccupations
    - Principe de responsabilité unique

Exemple

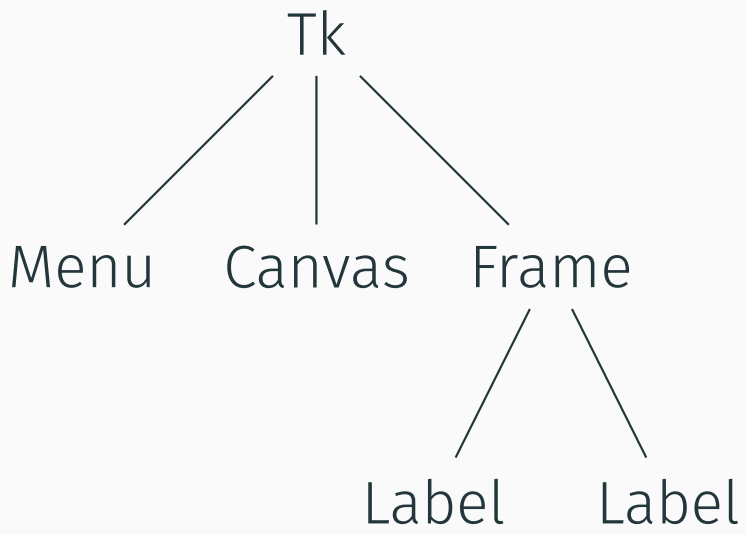

Place
Dungeon
Inn
0..1
0..1
0..1
inhabitants
explorers
clients
*
*
*
Monster
Character
* characters *
Team
Troll
Goblin
Wizard
Warrior

Design Patterns for Graphical User Interfaces

Arbre de vues

- Une interface est structurée comme un arbre de *vues*
- Une *vue* est un élément qui occupe une région de l'écran

Arbre de vues



- Une interface est structurée comme un arbre de *vues*
- Une *vue* est un élément qui occupe une région de l'écran

Arbre de vues - Construction procédurale

```
from tkinter import *
root = Tk()
menu = Menu(root)
canvas = Canvas(root)
bar = Frame(root)
name_info = Label(bar, text="filename:")
size_info = Label(bar, text="size:")
```

- Construction étape par étape de l'arbre de vues
- Utilisation d'un langage de programmation polyvalent

Arbre de vues - Construction déclarative

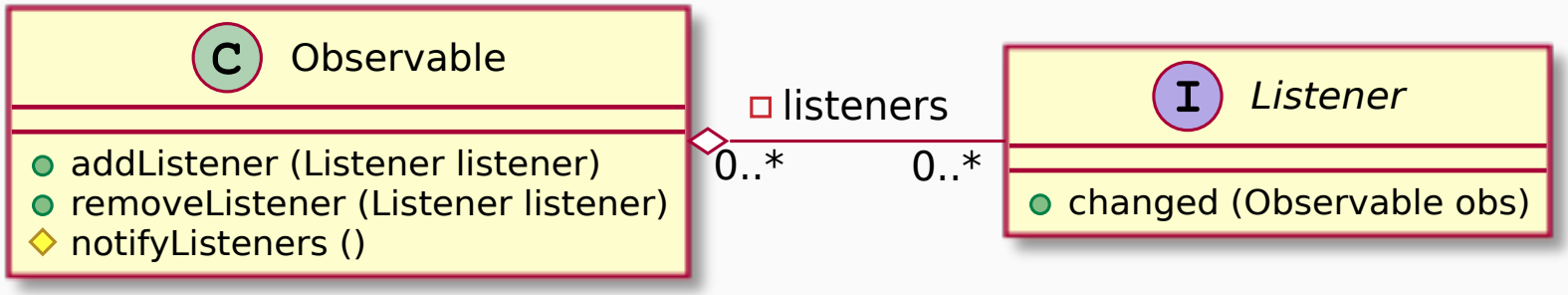
```
<Tk>
    <Menu> ... </Menu>
    <Canvas/>
    <Frame>
        <Label side="left" text="filename:"/>
        <Label side="left" text="size:"/>
    </Frame>
</Tk>
```

- Représentation directe de l'arbre de vues
- Utilisation d'un langage adapté à la représentation d'arbres

Avantages de la construction déclarative

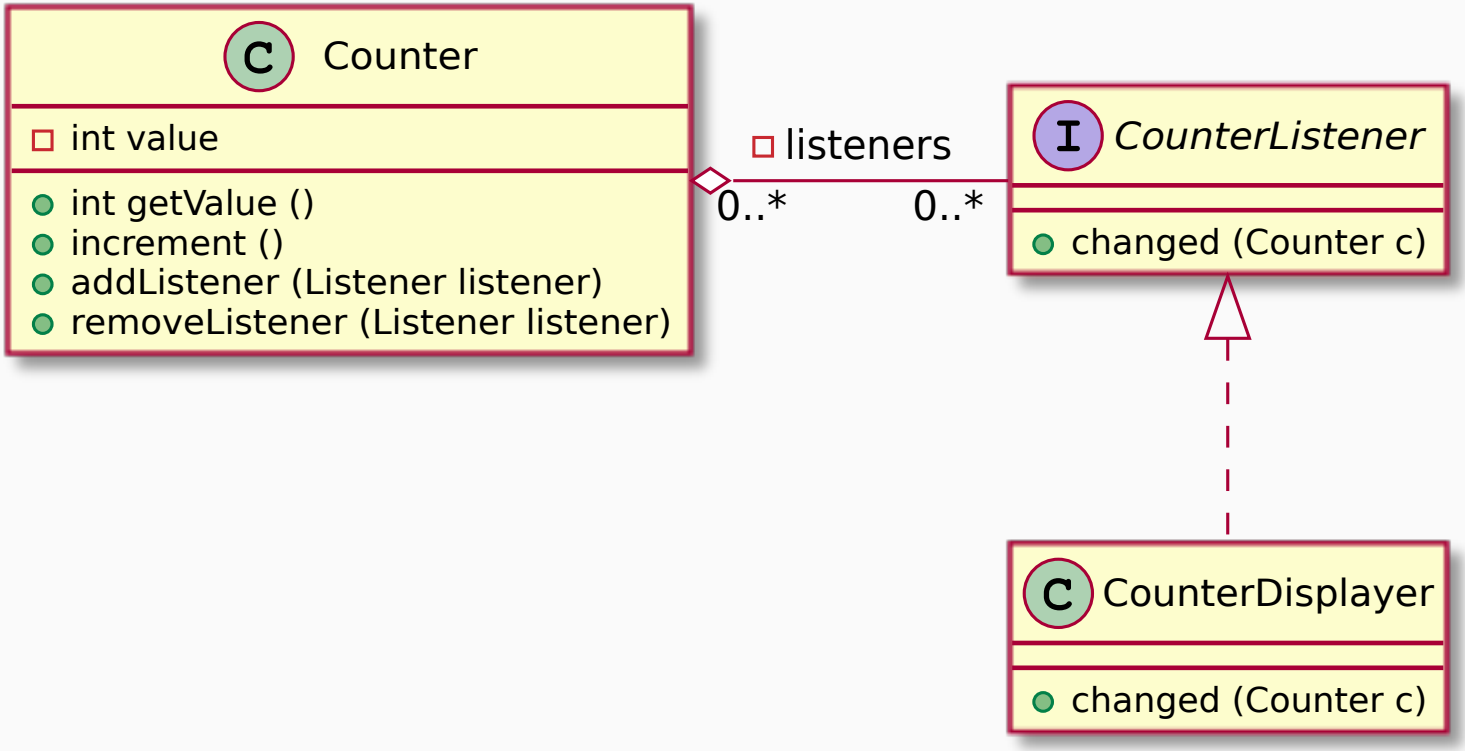
- Souvent plus concise
- Plus simple pour les non-programmeurs
- Développement plus simple d'outils
  - Validation de code
  - Génération d'interface

Listener design pattern



- Aussi nommé *Observer design pattern*
- Permet à un objet d'avertir ces observateurs que son état a changé en conservant un couplage faible

Counter example



Counter example

```
public class Counter {
    public void increment () {
        this.value++;
        for (x : this.listeners) { x.changed(this); }
    }
}
public class CounterDisplayer implements CounterListener {
    public void changed(final Counter c) {
        System.out.println(c.getValue());
    }
}
```

Counter example

```
public Main {
    public static void main(final String[] args) {
        final Counter c = new Counter();
        final CounterListener v = new CounterDisplayer();
        c.addListener(v)

        c.increment(); // 1
        c.increment(); // 2
    }
}
```

Counter example

```
public Main {
    public static void main(final String[] args) {
        final Counter c = new Counter();
        final CounterListener v = (c) -> {
            System.out.println(c.getValue());
        }};
        c.addListener(v)

        c.increment(); // 1
        c.increment(); // 2
    }
}
```

UI conclusion

- Les Entrées / Sorties sont séparées
  - La sortie est représentée par un arbre de vues
  - Les entrées sont traitées par des observateurs attachés à des *vues*
- Où se trouve la logique applicative ?
  - Les données
  - Les opérations pour modifier ses données

Architecture MVC

Architecture logicielle

- Manière de découper une application en limitant les couplages

Architecture MVC

Rails

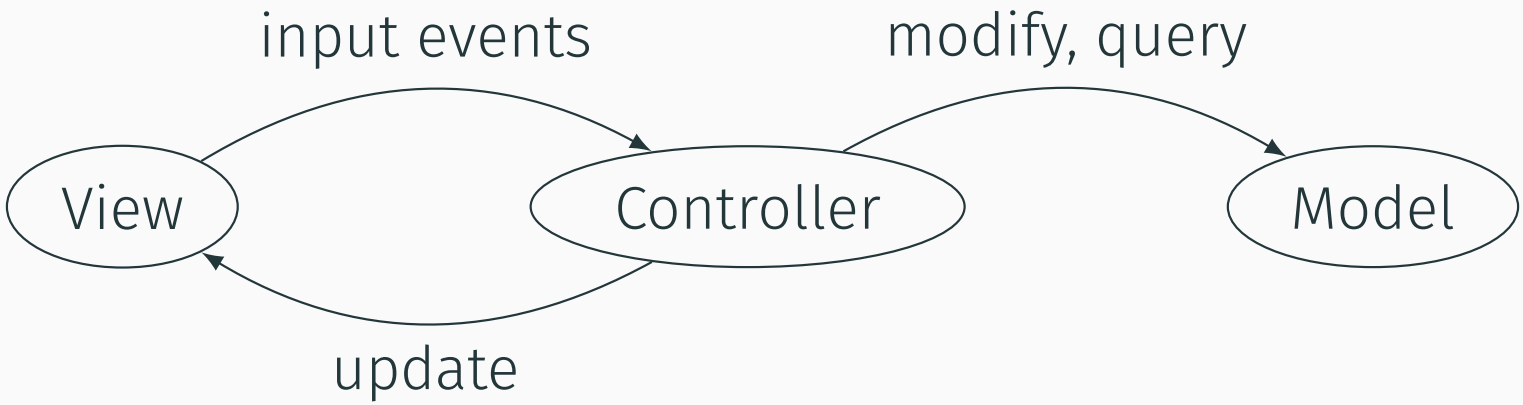
Django

Angular

Struts

- Apparition dans les années 80
- Séparer la vue de la représentation et la manipulation des données de l'application
- Devenue très populaire notamment pour la conception d'applications webs
    - Nombreuses variantes de l'architecture

Architecture Modèle-Vue-Contrôleur (MVC)

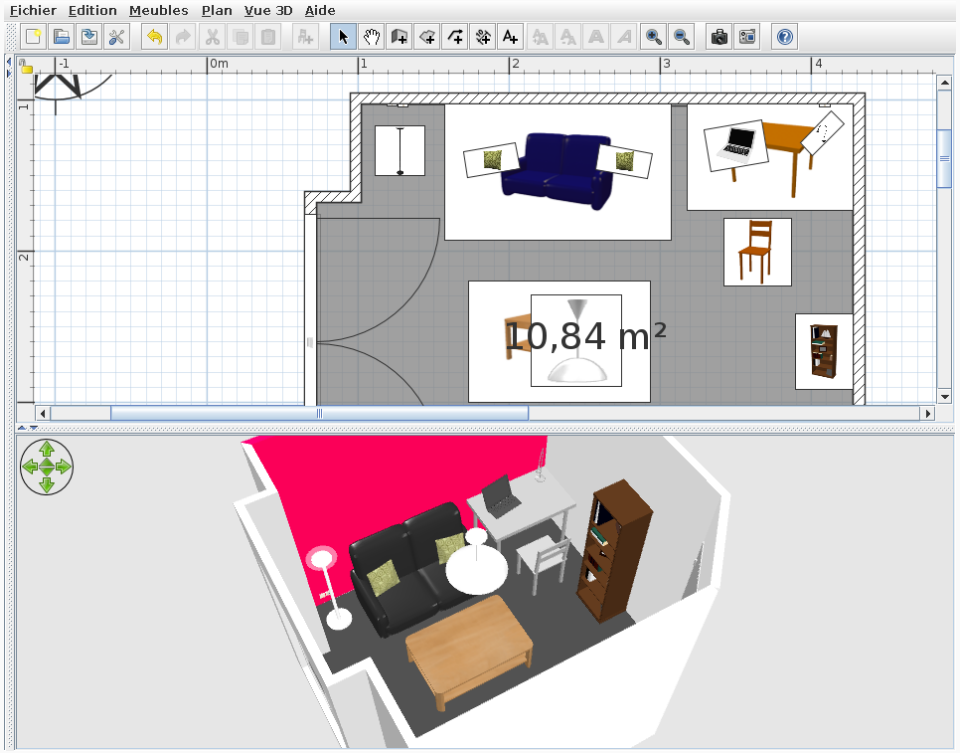


Modèle : Représentation des données de l'application, opérations pour modifier et interroger les données

Vue : Interface graphique (Sortie)

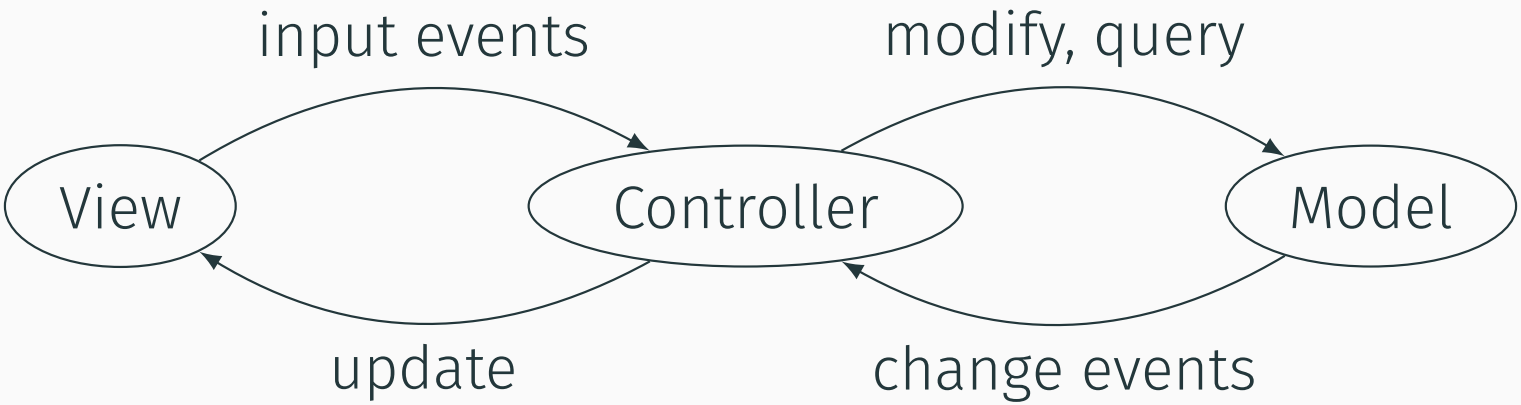
Contrôleur : Traitement des entrées des utilisateur·ice·s, modification des données et de la vue

Avantages de MVC



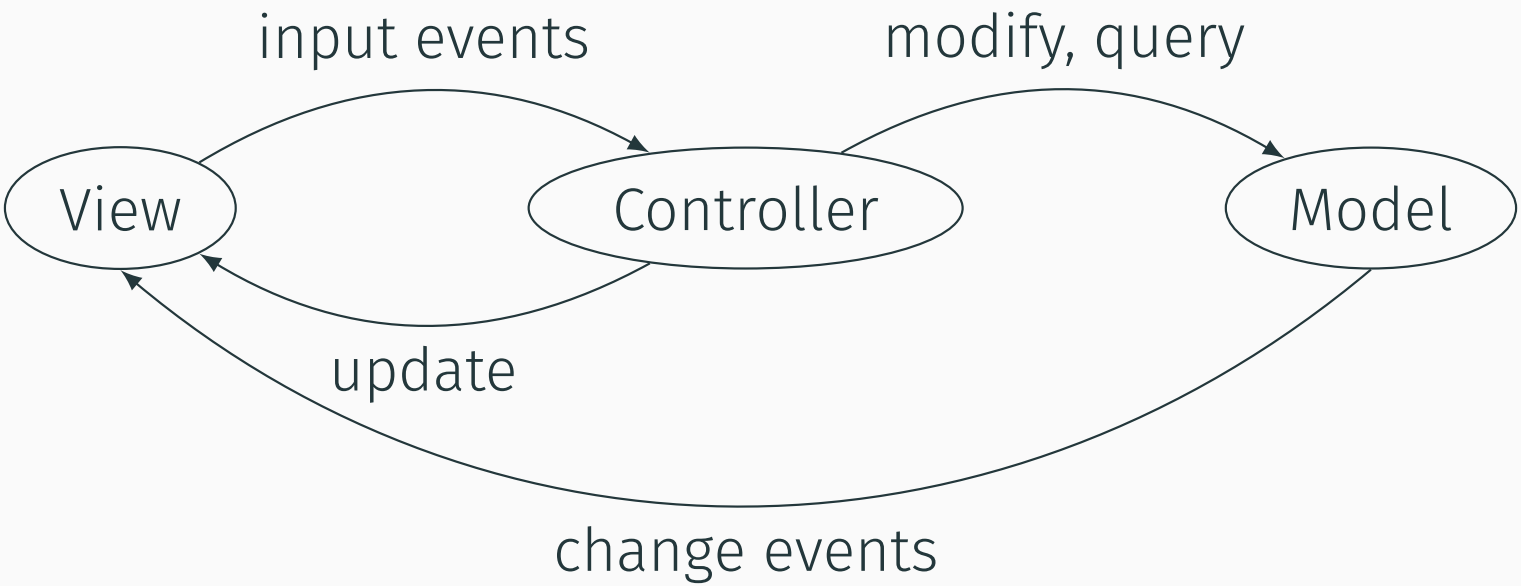
- Découplage du modèle de la vue
  - Développement en parallèle de la vue et du modèle
  - Vues multiples pour un même modèle
  - Réutilisation de la vue pour un autre modèle

Architecture MVC



- Le contrôleur doit savoir qu'estc e qui est modifié dans le modèle
  - Le contrôleur est couplé assez fortement au modèle
- Le modèle est le mieux placé pour savoir ce qui est modifié

Architecture MVC



- Le modèle pourrait être modifié par différents contrôleurs

JavaFX