

Module Series Game.

Game programming Patterns

Abdelkader Gouaich

gouaich@tirmont.fr

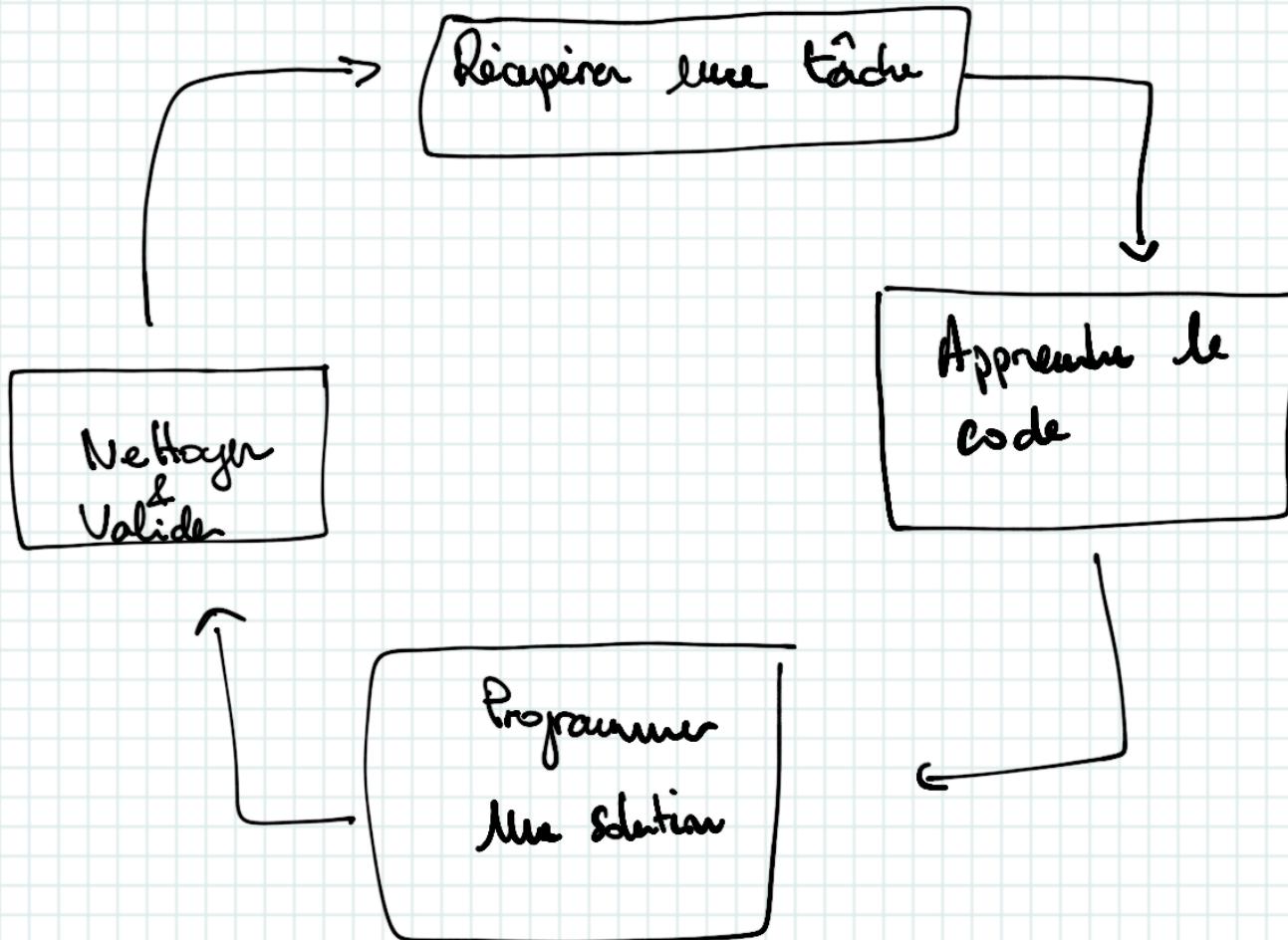
Objectifs du cours:

- o présenter les patterns de programmation les plus utilisés dans la programmation de jeux
- o Donner des exemples / cas
- o Savoir organiser son code
- o Savoir définir une bonne architecture logiciel

② Prérequis: Software architecture.

- . Organisation du logiciel
- . C'est quoi une bonne architecture?
 - Architecture → Anticipe les changements!
 - S'adapte avec les changements.
- . Changements?
 - Ajout d'une feature
 - Correction d'un bug

. Comment effectuer son changement?



Cycle de travail
d'un développeur.

Propriétés importantes d'une bonne architecture :

- Minimiser la phase d'apprentissage ✓
- Faciliter la phase de codage ✓
- Simplifier la phase de validation ✓

Comment obtenir une bonne architecture ?

Decoupling

{ Discours de la méthode
Principe de "separation" of concerns.

Simplicity

{ Cohérence
minimiser la charge cognitive.

Chapitre 2

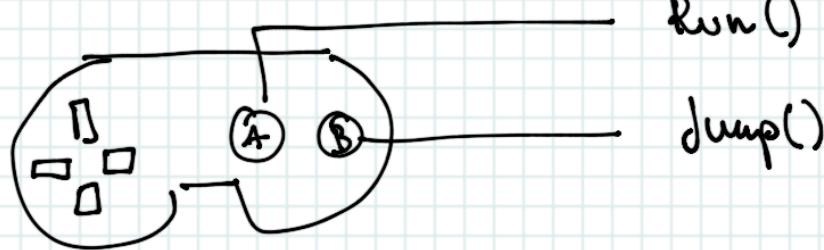
Les design patterns

Design Pattern : Command

- Incorporer une requête dans un objet
- Ces requêtes objets peuvent être:
 - paramètres
 - mise en file d'attente
 - faire des undo

Command est une réification des appels de méthodes

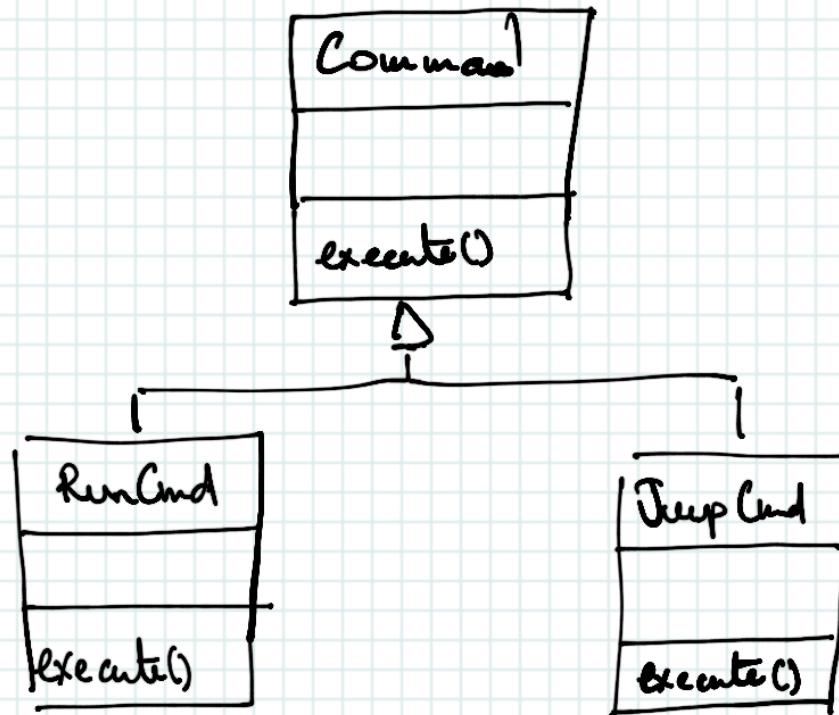
Example:



Primer programme simple :

```
void Input Handler  
{  
    if (isPressed (Button_A)) run ();  
    else if (isPressed (Button_B)) jump();  
}
```

Avec le pattern Command :



Class Input Handler

```
{  
    Command buttonA,  
    Command buttonB;  
  
    InputHandler()  
    {  
        buttonA = new RunCommand();  
        buttonB = new JumpCommand();  
    }  
  
    handleInput()  
    {  
        if (isPressed(BUTTON_A))  
        {  
            buttonA.execute();  
        }  
        else if (isPressed(BUTTON_B))  
        {  
            buttonB.execute();  
        }  
    }  
}
```

Variante :

→ Prendre en compte l'acteur du jeu sur lequel exécuter la commande.

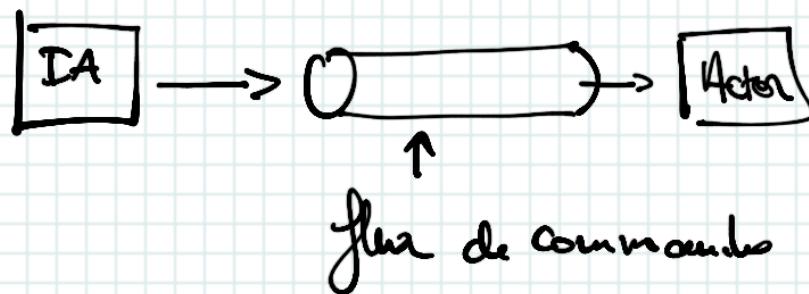
class Command

{

void execute (Game Actor actor);

}

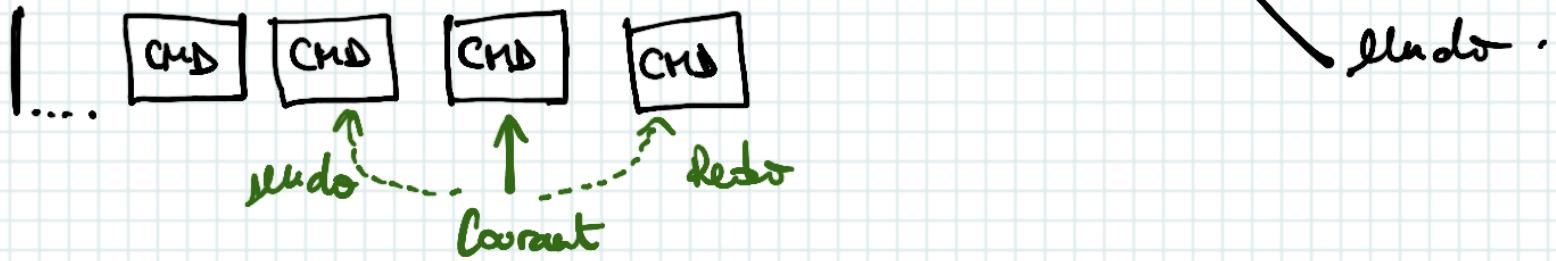
Exemple d'utilisation pour une IA:



Gestion des Undo / Redo :

```
class Command {  
    . execute();  
    . undo();  
}
```

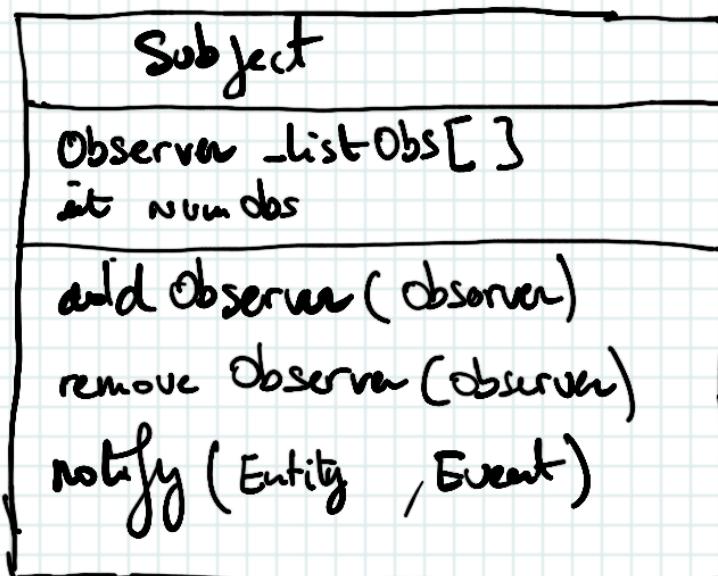
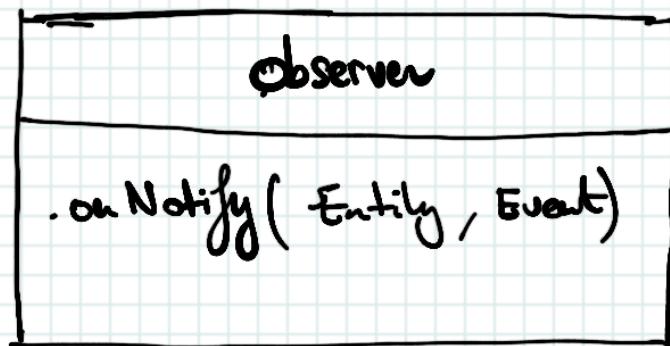
- Définir une file de commandes : + 3 pointers



Design Pattern : Observer.

. Description : Permet de créer une relation 1-Many pour la notification des changements de la source vers les destinataires.

. Classes de base :

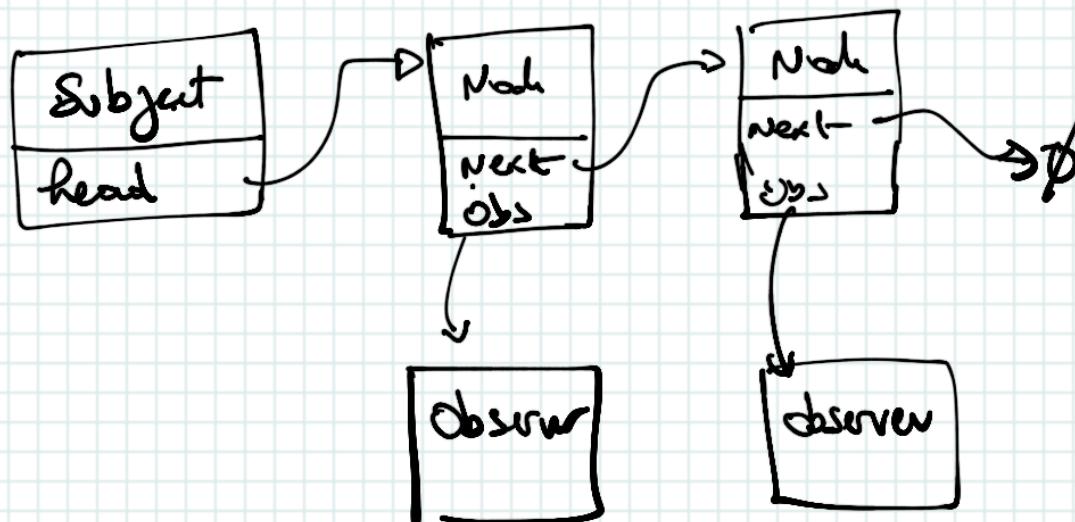


Exemple de code de la fonction notify:

```
...
void notify(Entity entity, Event event)
{
    for (i in 0..numObs)
    {
        -list Obs[i].onNotify(entity, event)
    }
}
```

Variants :

Replace la tableau par une liste



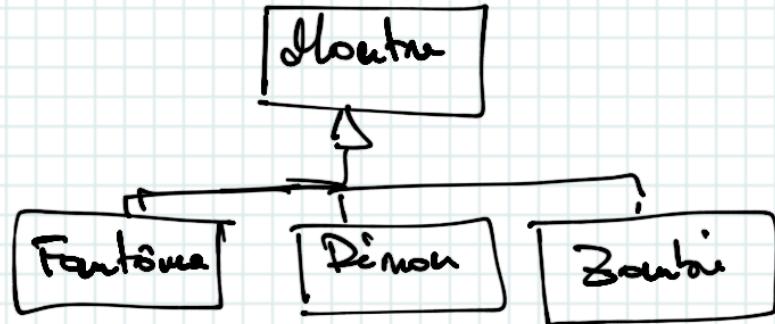
Le design pattern : Prototype

Principe :

- . Spécifier les objets à créer en utilisant une instance
- . Copier cette instance pour créer d'autres objets.

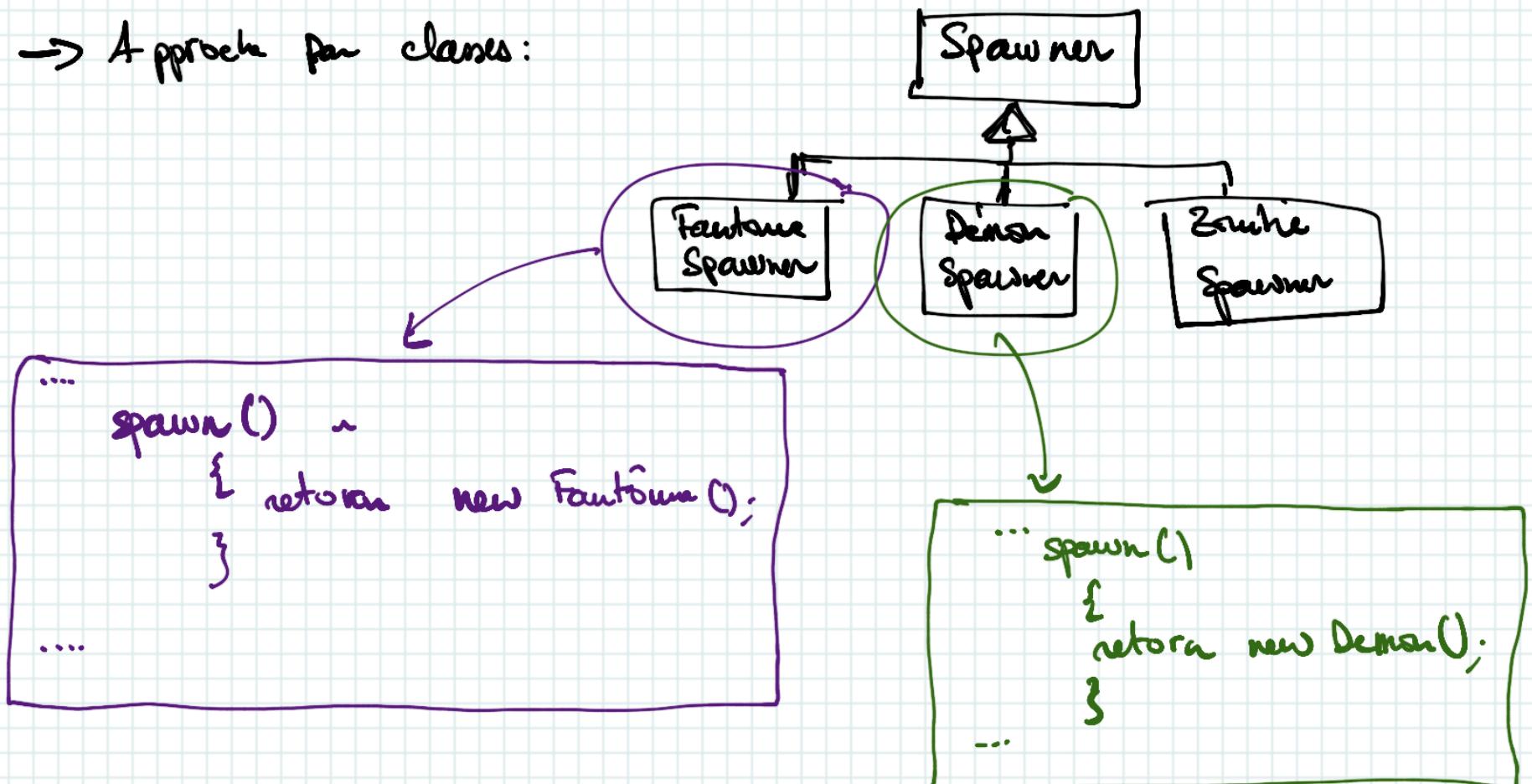
Exemple:

Mon jeu propose les entités suivantes :



je cherche à "spawner" ces entités :

→ Approche par classes :



→ Approach per Prototype:

```
class Monster {  
    ...  
    Monster clone();  
}
```

```
class Faetome {  
    health, Speed;  
    ...  
    Monster clone(){  
        return new Faetome(health, speed);  
    }  
    ...
```

Maintenant le code des Spawner devient plus facile!

Nous avons besoin d'une seule classe:

```
class Spawner
{
    Monstre -prototype;
    Spawner (Monstre proto)
    {
        -prototype = proto;
    }
    gloire spawn () {
        return -prototype.clone();
    }
}
```

Exemple:

Spawner fantomeSpawner = new Spawner (new Fantome (10, 10));

Spawner demonSpawner = new Spawner (new Demon (...));

Design Pattern: State

Principe : Permettre à un objet de modifier son comportement en fonction de son état interne.

Exemple :

```
--  
void handleInput ( Input* i )  
{ if ( input == BUTTON_A )  
{ yVelocity = V  
setSprite ( Jump_Sprite );  
}  
}
```

--

Correction:

```
void handleInput (Input in)
{
    if (input == BUTTON_A) {
        yVelocity = V
        setSprite (Jump_Sprite);
    }
}
```

bug!

```
void handleInput (Input in)
{
    if (input == BUTTON_A)
    {
        if (!Jumping) {
            yVelocity = V
            setSprite (Jump_Sprite);
            Jumping = true;
        }
    }
}
```

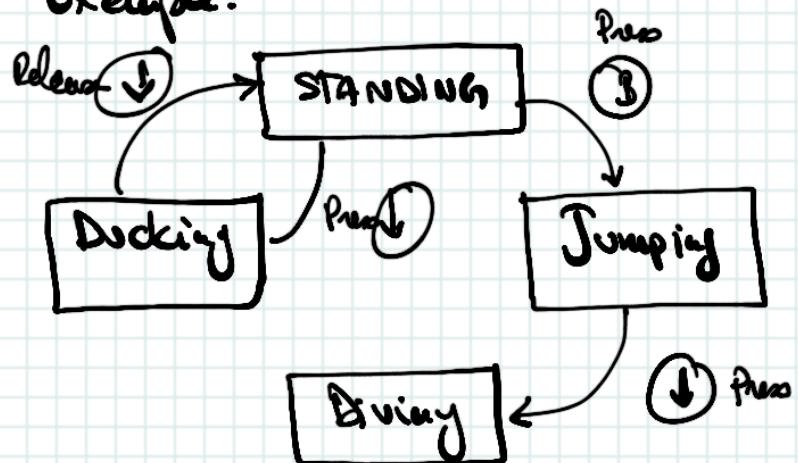
Comment réaliser cela de façon plus élégante?

→ Réponse: Structure à état.

Machine à états finis :

- Nombre d'états finis
- . On ne peut être qu'en dans un état à la fois
- . une séquence d'événements sont envoyés à la machine
- . Pour chaque état il y a un ensemble de transitions vers les autres états

Exemple:



Exemple d'implémentation de la FSM:

- des états représentés par une énumération

```
enum State
```

```
{ STATE_STANDING;  
STATE_JUMPING;  
STATE_DUCKING;  
STATE_DIVING;  
}
```

- . les transitions par un switch:

```
void handleInput ( Input in )
```

```
{  
    switch ( -state )  
    {  
        case STATE_STANDING:  
            if ( in == PRESS_B )  
            {  
                -state = STATE_JUMPING  
                yVelocity = V  
                set Sfmita ( JUMP_ Inv );  
            }  
            else if ( in == PRESS_DOWN )  
            {  
                ...  
            }  
    }  
}
```

le pattern State :

→ Définition d'une interface de State
interface ActorState

```
{ void handleInput(Actor ac, Input in);  
void update(Actor ac);  
}
```

→ Une implementation de l'interface pour changer état
class DuckingState implements ActorState

```
{  
    ...  
    handleInput(Actor ac, Input in) {  
        if (in == RELEASE_DOWN)  
        {  
            ...  
        }  
    } ..  
    update(Actor ac) { ... };  
}
```

Côté de l'acteur : utiliser la délégation !

```
class MonActeur extends Actor
{
    ActorState state;

    handle Input( Input in )
    {
        state. handle Input( this , in );
    }

    update( ) { state. update( this ); }

}
```

Variante de l'implémentation :

- utile si l'état conserve des variables propres à chaque acteur.
- il sera donc important d'instancier les Etats à chaque transition.

class MonActeur {

 handle Input (Input ii)
 {

 ActorState s = _state->handle Input (...);

 if (s != null)

 {

 _state = free();

 _state = s;

 }

 }

 '; class StandingState {

 handle Input (...)

 {

 if (...) {

 return new DuckyState();

 }

 return null;

 }



ne pas
changer d'état.

Extension de la FSM :

Push Down Automate.

A mériter...

- une FSM ne conserve pas la mémoire des états. ✓
- Nous savons où nous sommes mais pas d'où nous venons.
- Solution: utiliser une pile pour stocker l'état précédent.

Avec une pile, nous avons donc deux opérations supplémentaires:

- **push**: mettre l'état courant en haut de la pile
- **pop**: retirer le top de la pile; l'état en dessous devient l'état courant.

