Chapitre. Game Programming Patterns

Master Imagine

Abdelkader Gouaïch, gouaich@lirmm.fr

2021

Introduction

Objectifs du cours:

- Introduction au concept d'architecture logicielle
- Présenter les patterns de programmation utiles à la programmation des jeux vidéo

Rappels sur les principes du génie logiciel

Une architecture logicielle

Une architecture logicielle:*organisation* du logiciel

Qu'est-ce qu'une bonne architecture ?

- Anticipe les changements
- S'accomode et intègre naturellement ces changements

Qu'entendons-nous par *changement* ?

- Ajout d'une *feature*
- Correction d'un *bug*

La dynamique du changement

Voici le cycle classique de travail pour un développeur :

1. Récupérer une tâche à réaliser
2. Apprentissage du code source
3. Programmation d'une solution
4. Intégration et validation
5. Aller à (1)

Propriétés importantes d'une bonne architecture

Une bonne architecture va:

- minimiser la phase d'apprentissage du code existant
- faciliter la phase de codage
- faciliter la phase d'intégration et de validation

Principes pour *une bonne* architecture

Principes:

- Decoupling
  - Séparation par l'analyse
- Simplicity
  - Parcimonie pour la réduction de la charge cognitive

Les design patterns

Design pattern: command

- Principe: encapsuler une requête dans un objet
- les requêtes objets seront:
  - paramétrées
  - mises dans une séquence

Exemple de Command

Une programmation directe:

```
function InputHandler()
{
    if( isPressed(BUTTON_A) )
    {
        run();
    }
    else if ( isPressed(BUTTON_B))
    {
        jump();
    }
}
```

Exemple de Command

Une programmation directe:

```
class Action{

    function execute(){

    }
}
class Run extends Action{

    function execute(){
        // do smthg
    }
}
class Jump extends Action{

    function execute(){
    // do smthg
    }
```

Exemple de Command

```
class InputHandler
{
    Action buttonA;
    Action buttonB;
    constructor(){
        buttonA = new Run()
        buttonB = new Jump()
    }

    function handleInput(){
        if( isPressed(BUTTON_A) )
        {
            buttonA.execute();
        }
        else if ( isPressed(BUTTON_B))
        {
            buttonB.execute();
        }
```

Variante de Command

```
class Action{

    function execute(GameActor actor){

    }
}
```

Variante pour la gestion de undo avec Command

```
    class Command{
        function execute(GameActor actor){
        }

        function undo(GameActor actor){
        }
    }
commandsQueue  = []
//trois pointeurs de commandes
let undoIndex ;
let currentIndex ;
let redoIndex ;
```

Design pattern: Observer

- Principe: créer une relation 1-many pour la notification des changements de la source vers les destinations

```
class Observer{
    function onNotify(entity, event){}
}

class Subject{
    observers = []
    function addObserver(observer){}
    function removeObserver(observer){}
    function notify(entity,event){
        observers.forEach( obs => obs.onNotify(entity,event) )
    }
}
```

Design pattern: State

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

```
function hundleInput(input){
    if(input == BUTTON_A)
    {
        yVelocity += VELOCITY
        setSprite(JMP_SPRITE)
    }
}
```

Un bug avec ce code !

Solution

```
function hundleInput(input){
    if(input == BUTTON_A)
    {
        if(! Jumping )
        {yVelocity += VELOCITY
        setSprite(JMP_SPRITE)
        Jumping = true
        }
    }
}
```

Comment réaliser cela de façon plus élégante ?

Finite State Machine

- Nombre d'états fini
- Un seul état dans un temps donné
- Des événements sont reçus par le FSM
- Les transitions d'états sont déclenchées par les événements

```
const STATES = {
    STATE_STANDING  = 0,
    STATE_JUMPING   = 1,
    STATE_DUCKING   = 2,
    STATE_DIVING    = 3,
    }
```

```
function handleInput(input){

    switch(state){

        case STATES.STATE_STANDING:
            if(input == BUTTON_B){
                yVelocity += VELOCITY
                setSprite(JMP_SPRITE)
                state = STATES.STATE_JUMPING
            }
            break;
        ...
    }
}
```

Le pattern STATE

```
class ActorState {

    function handleInput(actor,input){}
    function update(actor){}

}
class DuckingState extends ActorState {
...
    function handleInput(actor,input){
        if(input == BUTTON_A) {
            ...
        }
    }
    function update(actor){...}

}
```

```
class MyActor extends Actor {
    ActorState state;

    function handleInput(input){
        state.handleInput(this,input)
     }
    function update(){
        state.update(this)
    }

}
```

Extension de la FSM : Push Down Automata

On ajoute une pile (stack) pour ranger l'état courant avant de passer à un nouvel état.

Deux opérations nouvelles:

- push : sauvegarde du State courant dans la pile et remplacement avec le nouveau state
- pop: retirer l'état courant et le remplacer par la sauvegarde