Game Programming Patterns

The Component

Game Programming Patterns

Component Design Pattern in Games

- Allow a **single entity** to **span multiple domains** without coupling the domains to each other
    - e.g., graphic rendering, physics, sound, etc..
- Keep the the **domains isolated**
    - The **code** for **each one** is kept in its own **component class**
- **The entity is reduced to a simple container of components**

Objective

The idea is to keep the coupling between classes as “loose” as possible in the system, spawning components that represent an Object in a tree structure.

x2

The Component Pattern

Example: The Spaghetti

```
if (collidingWithFloor() && (getRenderState()
!= INVISIBLE)) {

  playSound(HIT_FLOOR);
}
```

=> **To** make a **change** in code like this **we need to know about physics, graphics, and sound**, just to make sure we don't break anything.

Note

When it starts to get bad enough, developers will start **putting hacks** in **other parts** of the **code**. **Compromising** even more the **code maintainability**.

x3

Don’t be that cat!

Flying Spaghetti-Code Monster

x4

Sample Code - Entity Mario

From a monolithic class to a decoupled solution

x5

A Monolithic Class - Mario Implementation

```cpp
class Mario {
public:
  Mario()
  : velocity_(0),
    x_(0), y_(0)
  {}

  void update(World& world, Graphics& graphics);

private:
  static const int WALK_ACCELERATION = 1;

  int velocity_;
  int x_, y_;

  Area area_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

```cpp
void Mario::update(World& world, Graphics& graphics) {
  // Acquire user input to determine the acceleration
  switch (Controller::getJoystickDirection()) {
    case DIR_LEFT:
    velocity_ -= WALK_ACCELERATION;
    break;

    case DIR_RIGHT:
    velocity_ += WALK_ACCELERATION;
    break;
  }

  // Calculates the new position with the physics engine
  x_ += velocity_;
  world.resolveCollision(volume_, x_, y_, velocity_);

  // Finally draw the appropriate sprites
  Sprite* sprite = &spriteStand_;
  if (velocity_ < 0) {
    sprite = &spriteWalkLeft_;
  }
  else if (velocity_ > 0){
    sprite = &spriteWalkRight_;
  }

  graphics.drawQueue(*sprite, x_, y_);
}
```

x6

A Monolithic Class - Mario Implementation

- This is a **simple** and trivial **implementation**
- There's **no gravity** and **animations**
  - or any of the dozens of other details that makes a character fun to play
- However, we start to see that a **single function** will **require attention** from **several developers**
  - **Plus:** it's already starting to get a bit messy
  - **Imagine this scaled up to a thousand lines**
    - gravity, animations, etc...

```cpp
void Mario::update(World& world, Graphics& graphics) {
  // Acquire user input to determine the acceleration
  switch (Controller::getJoystickDirection()) {
    case DIR_LEFT:
    velocity_ -= WALK_ACCELERATION;
    break;

    case DIR_RIGHT:
    velocity_ += WALK_ACCELERATION;
    break;
  }

  // Calculates the new position with the physics engine
  x_ += velocity_;
  world.resolveCollision(volume_, x_, y_, velocity_);

  // Finally draw the appropriate sprites
  Sprite* sprite = &spriteStand_;
  if (velocity_ < 0) {
    sprite = &spriteWalkLeft_;
  }
  else if (velocity_ > 0){
    sprite = &spriteWalkRight_;
  }

  graphics.drawQueue(*sprite, x_, y_);
}
```

x7

Splitting Out a Domain

Decouple First Domain

- **First domain** is the **input**
  - Mario starts by reading user input and adjust his velocity based on it. Let's decouple it.

```
class InputComponent {
public:
  void update(Mario& mario) {

    switch (Controller::getJoystickDirection()) {
      case DIR_LEFT:
        mario.velocity -= WALK_ACCELERATION;
        break;

      case DIR_RIGHT:
        mario.velocity += WALK_ACCELERATION;
        break;
    }
  }

private:
  static const int WALK_ACCELERATION = 1;
};
```

```
class Mario {
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics) {
    input_.update(*this);

    // Modify position by velocity.
    x += velocity;
    world.resolveCollision(area_, x, y, velocity);

    // Draw the appropriate sprite.
    Sprite* sprite = &spriteStand_;
    if (velocity < 0) {
      sprite = &spriteWalkLeft_;
    }
    else if (velocity > 0) {
      sprite = &spriteWalkRight_;
    }

    graphics.drawQueue(*sprite, x, y);
  }

private:
  InputComponent input_;

  Area area_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

x8

Splitting Out a Domain

First Component Class

- Mario now owns an **InputComponent** object
- We've only started, but **already gotten rid** of some **coupling** — the **main** Mario **class no longer** has any **reference to Controller**
    - This will come in handy later

```
class Mario {
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics) {
     input_.update(*this);

    // Modify position by velocity.
    x += velocity;
    world.resolveCollision(area_, x, y, velocity);

    // Draw the appropriate sprite.
    Sprite* sprite = &spriteStand_;
    if (velocity < 0) {
      sprite = &spriteWalkLeft_;
    }
    else if (velocity > 0) {
      sprite = &spriteWalkRight_;
    }

    graphics.drawQueue(*sprite, x, y);
  }

private:
   InputComponent input_;

  Area area_;

  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

x9

Splitting Out the Rest

We repeat the cut-and-paste job

```
class PhysicsComponent {
public:
  void update(Mario& mario, World& world) {

    mario.x += mario.velocity;
    world.resolveCollision(area_,
     mario.x,
     mario.y,
     mario.velocity);
  }

private:
  Area area_;
};
```

```
class GraphicsComponent {
public:
  void update(Mario& mario, Graphics& graphics) {

  Sprite* sprite = &spriteStand_;
    if (mario.velocity < 0) {
      sprite = &spriteWalkLeft_;
    }
    else if (mario.velocity > 0){
      sprite = &spriteWalkRight_;
    }

    graphics.drawQueue(*sprite, mario.x, mario.y);
  }

private:
  Sprite spriteStand_;
  Sprite spriteWalkLeft_;
  Sprite spriteWalkRight_;
};
```

x10

Final Result - Mario Implementation

Mario with a Component Pattern

- Mario class now **holds** the set of **components** that **defines it**, and **holds** the **state** that is **shared** among **domains**
- **More importantly**, it gives an **easy way** for the **components** to **communicate without being coupled**

```
class Mario {
public:
  int velocity;
  int x, y;

  void update(World& world, Graphics& graphics) {
    input_.update(*this);
    physics_.update(*this, world);
    graphics_.update(*this, graphics);
  }

private:
  InputComponent input_;
  PhysicsComponent physics_;
  GraphicsComponent graphics_;
};
```

2 - Abstracting the behavior

Adding an AI controlled player

Adding an AI Controlled Player

Abstracting the behavior

- Previously we've **pushed our behavior** out to separate **component** classes, but we **haven't abstracted the behavior out**
- Let's take the ***InputComponent*** and **hide** it **behind an interface**
    - Turn ***InputComponent*** into an **abstract class**

```cpp
class InputComponent {
public:
  virtual ~InputComponent() {}
  virtual void update(Mario& mario) = 0;
};
```

Adding an AI Controlled Player

Implementing the Interface

- Then, we **take** our **existing user input handling** code and **push it down** into a **class** that implements that interface.

```
class PlayerInputComponent : public InputComponent
{
public:
  virtual void update(Mario& mario) {

    switch (Controller::getJoystickDirection()) {
      case DIR_LEFT:
        mario.velocity -= WALK_ACCELERATION;
        break;

      case DIR_RIGHT:
        mario.velocity += WALK_ACCELERATION;
        break;
    }
  }

private:
  static const int WALK_ACCELERATION = 1;
};
```

Adding an AI Controlled Player

Abstracting the behavior

- Then we change Mario to **hold a pointer** to the ***InputComponent***
- Now when we **instantiate** Mario class we can **pass** an ***InputComponent***:

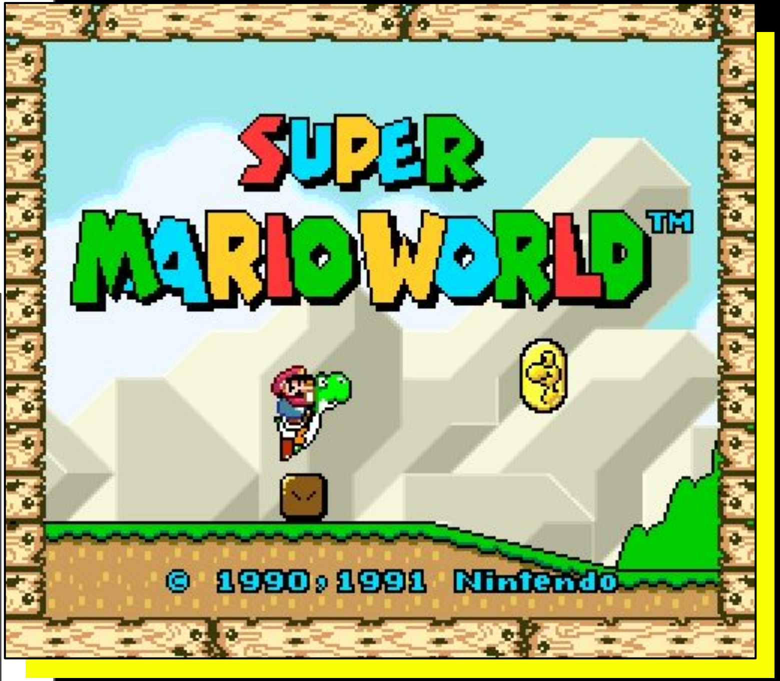
```
Mario* mario = new Mario(new PlayerInputComponent());
```

```
class Mario {
public:
  int velocity;
  int x, y;

  Mario(InputComponent* input)
  : input_(input)
  {}

  void update(World& world, Graphics& graphics) {
    input_->update(*this);
    physics_.update(*this, world);
    graphics_.update(*this, graphics);
  }

private:
  InputComponent* input_;
  PhysicsComponent physics_;
  GraphicsComponent graphics_;
};
```

Creating a demo scene

- **By hiding** the **input component** behind an **interface**, we can **easily create** a **demo** scene
- In **demo mode** we can **start** the **controller** like this:

```cpp
class DemoInputComponent : public InputComponent {
public:
  virtual void update(Mario& mario) {
    // AI to automatically control Mario...
  }
};

// Construct mario with the new input
Mario* mario = new Mario(new DemoInputComponent());
```



From: Super Mario World, SNES
Starting screen

Finally: No Mario at all?

Abstracting Mario into a generic (base) component

Creating a Generic Game Object

Abstracting Mario Class

- Our Mario class is now a **component "bag"**
- It looks like a **good candidate** for a **base game object**

```
class Mario {
public:
  int velocity;
  int x, y;

  Mario(InputComponent* input)
  : input_(input)
  {}

  void update(World& world, Graphics& graphics) {
    input_->update(*this);
    physics_.update(*this, world);
    graphics_.update(*this, graphics);
  }

private:
  InputComponent* input_;
  PhysicsComponent physics_;
  GraphicsComponent graphics_;
};
```

Creating a Generic Game Object

Implement Interfaces

- We **take** the remaining **concrete components**
  - i.e. Physics and Graphics
- And **hide** them **behind interfaces**
  - Like we did with the Input

```
class PhysicsComponent
{
public:
  virtual ~PhysicsComponent() {}
  virtual void update(GameObject& obj, World&
world) = 0;
};

class GraphicsComponent {
public:
  virtual ~GraphicsComponent() {}
  virtual void update(GameObject& obj, Graphics&
graphics) = 0;
};
```

Creating a Generic Game Object

The GameObject

- Finally, we **rewrite** our Mario class into a **generic GameObject** using those **Interfaces**
- Then, **create** our **concrete Components** with these **Interfaces**

```cpp
class MarioPhysicsComponent : public PhysicsComponent {
public:
  virtual void update(GameObject& obj, World& world) {
    // Physics code...
  }
};

class MarioGraphicsComponent : public GraphicsComponent {
public:
  virtual void update(GameObject& obj, Graphics& graphics)
  {
    // Graphics code...
  }
};
```

```cpp
class GameObject {
public:
  int velocity;
  int x, y;

  GameObject(InputComponent* input,
             PhysicsComponent* physics,
             GraphicsComponent* graphics)
  : input_(input),
    physics_(physics),
    graphics_(graphics)
  {}

  void update(World& world, Graphics& graphics) {
    input_->update(*this);
    physics_->update(*this, world);
    graphics_->update(*this, graphics);
  }

private:
  InputComponent* input_;
  PhysicsComponent* physics_;
  GraphicsComponent* graphics_;
};
```

Creating a Generic Game Object

Concluding the Example

- Now we can easily create a **method** that provides a **GameObject** with the **Mario behavior**
- **Plus:** By defining **other methods** we can **create** all kinds of **objects (or entities)** to **fit our game needs**

```
GameObject* createMario() {
  return new GameObject(new PlayerInputComponent(),
                        new MarioPhysicsComponent(),
                        new MarioGraphicsComponent());
}
```

Component Pattern

When to Use It

- Components are **most common** within the **core class** that defines **entities** in a game
- However the pattern is useful when:
    - A class **touches multiple domains** that **should** be **decoupled**
    - A class is getting **massive** in **complexity** and **hard to work**
    - You need to **change things on-the-fly** and **Inheritance** will **not work**

Keep in Mind

Each conceptual object becomes a cluster of objects, requiring instancing, initialization, and communication.

For a large codebase this complexity may be worth it for the decoupling, maintainability and code reuse.

But take care and ensure you aren't over-engineering before applying the pattern.

Game Programming Patterns

The Component