State pattern

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 11

Last Class

Chain of responsibility

- Break down the responsibilities and steps when handling a request
- Decouple classes that invoke operations from classes that carry out operations

Strategy

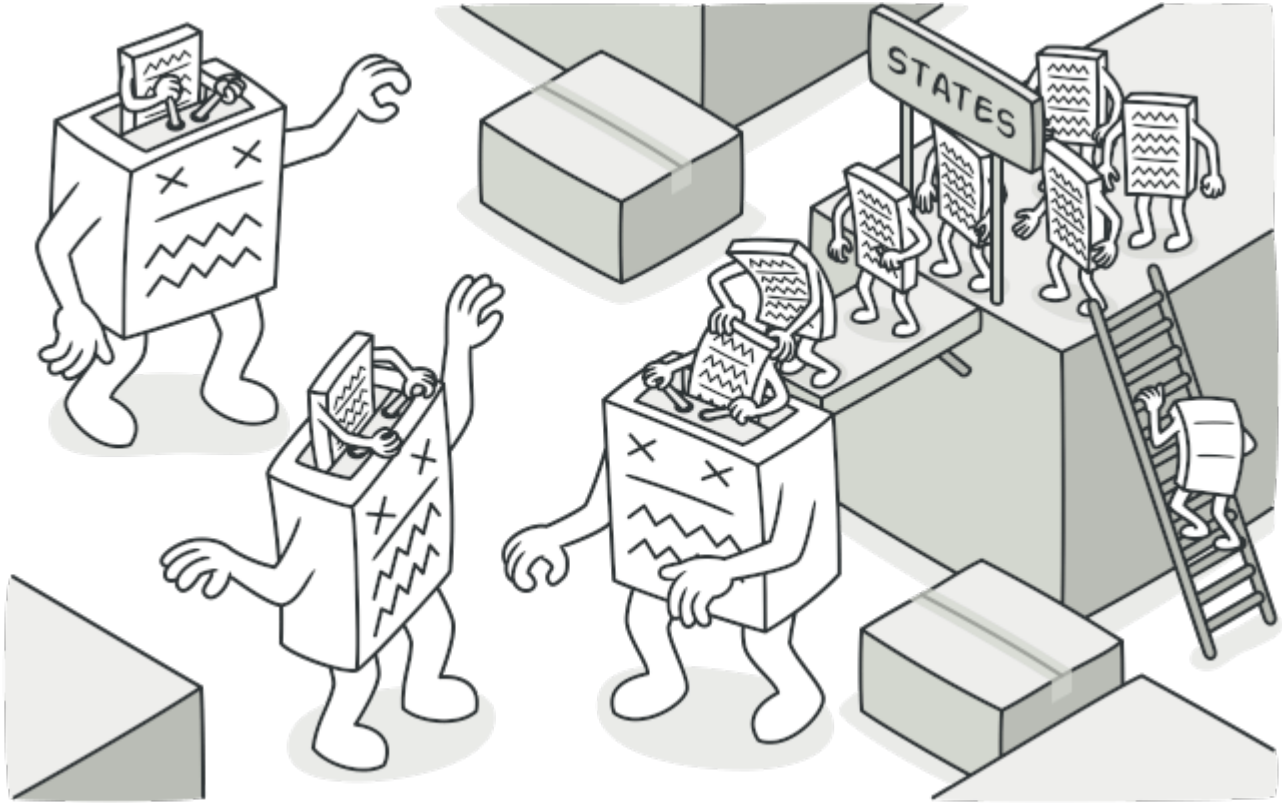
- Switch between different “Strategies” at runtime.
- These can be different implementations of the same process, or completely different algorithm.
- Embodies the Liskov Substitution Principle.
- Each strategy implements the same interface and is unaware of each other.

State

WHEN A CLASS IS AMBITIOUS AND WANTS TO BE MANY THINGS

State

- A pattern implemented when an object wants to alter its behaviour. It might behave like a different class.
- A quick recap: A state of an object is defined as the value and behaviours of the object at any given time during program execution.
- Depending on its internal state, an object may react differently to different scenarios.
- The state pattern enables a modular way to add states and control transitions. We decouple the state from the object.



A Finite State Machine

A program can be divided into a finite number of states across which its execution is tracked.

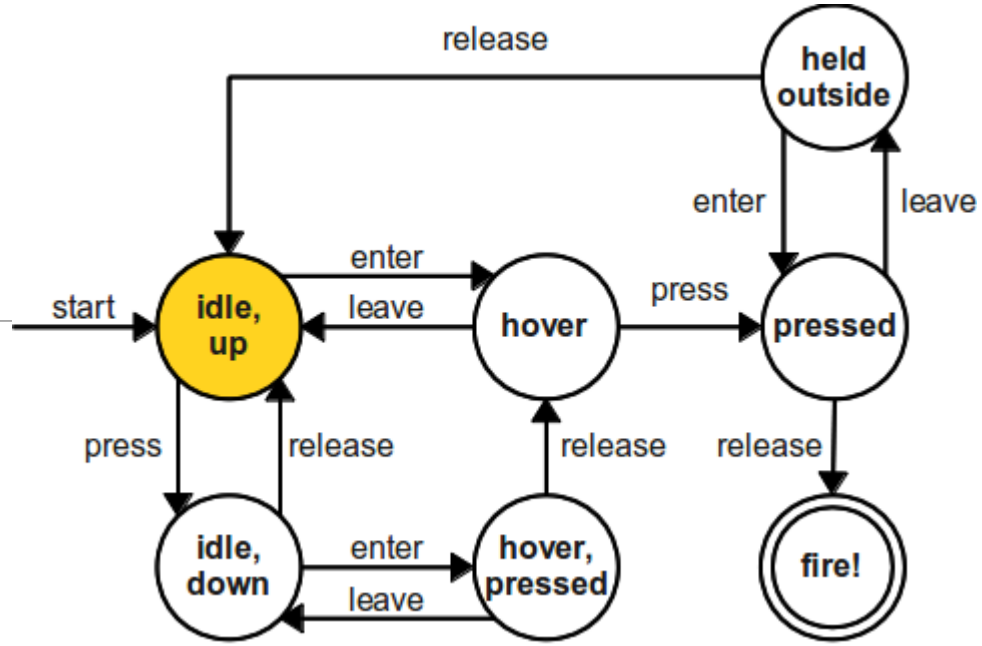
Within a state, the program behaves differently

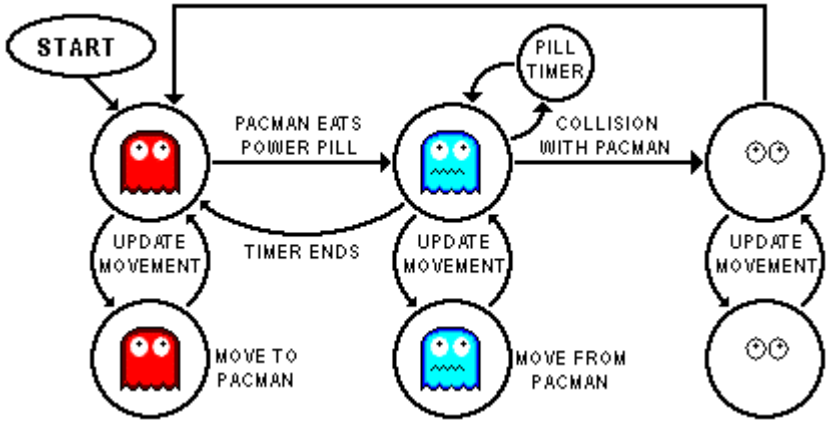
We can change the state of a program instantly.

Moving from one state to another state is bound by rules. These “Transitions” are also finite and predetermined.

The circles are our states

The arrows between them transitions that encapsulate rules for state switching





If you are in the mood for some *‘light’* reading: https://en.wikipedia.org/wiki/Finite-state_machine

Apply this concept to objects

An object can be in multiple states. It's behaviour is usually determined by its state:

- **Happy State:** It wags its tail, sticks out its tongue and is playful
- **Scared State:** Tail between its legs, togue inside it's mouth, whimpering
- **Aggressive State**: Tail is not wagging, growling, tail is raised high, bared teeth

Dog

- **Hover State:** might change color
- **Pressed:** Might change image and how its rendered
- **Idle:** Render default image
- **Disabled:** Button is not rendered

A UI Button:

State – Naïve Implementation

A ButtonEnum defines the different states a button can take.

A button stores its current state as an attribute

We use a series of if-else statements to change behaviour.

While this may be acceptable for simple scenarios, it isn't maintainable in larger systems.

```python
import enum

class ButtonEnum(enum.Enum):
    IDLE = 0,
    HOVER = 1,
    PRESSED = 2
    DISABLEd = 3


class Button:

    def __init__(self, idle_img, hover_img, pressed_img, callback):
        self.btn_state = ButtonEnum.IDLE
        self.btn_on_click_callback = callback
        self.idle_img = idle_img
        self.hover_img = hover_img
        self.pressed_img = pressed_img

    def render(self):
        if self.btn_state == ButtonEnum.IDLE:
            print(f"Displaying {self.idle_img}")
        elif self.btn_state == ButtonEnum.HOVER:
            print(f"Displaying {self.hover_img}")
        elif self.btn_state == ButtonEnum.PRESSED:
            print(f"Displaying {self.pressed_img}")
        else:
            print("Button disabled")


def on_button_press_callback():
    pass


def main():
    idle_img = "btn_save_idle_sprite.png"
    hover_img = "btn_save_hover_sprite.png"
    pressed = "btn_save_pressed_sprite.png"
    button = Button(idle_img, hover_img, pressed, on_button_press_callback)
    button.render()
    button.btn_state = ButtonEnum.PRESSED
    button.render()
```

State – Naïve Implementation

We want to be able to add and remove states without re-compiling the button class.

We want to be able to implement state-dependent behaviours easily as well.

In other words…

Making any changes to the states will effect the button class.

Button is coupled to it's state.

We want the button to be decoupled from its state.

```
import enum

class ButtonEnum(enum.Enum):
    IDLE = 0,
    HOVER = 1,
    PRESSED = 2
    DISABLEd = 3


class Button:

    def __init__(self, idle_img, hover_img, pressed_img, callback):
        self.btn_state = ButtonEnum.IDLE
        self.btn_on_click_callback = callback
        self.idle_img = idle_img
        self.hover_img = hover_img
        self.pressed_img = pressed_img

    def render(self):
        if self.btn_state == ButtonEnum.IDLE:
            print(f"Displaying {self.idle_img}")
        elif self.btn_state == ButtonEnum.HOVER:
            print(f"Displaying {self.hover_img}")
        elif self.btn_state == ButtonEnum.PRESSED:
            print(f"Displaying {self.pressed_img}")
        else:
            print("Button disabled")


def on_button_press_callback():
    pass


def main():
    idle_img = "btn_save_idle_sprite.png"
    hover_img = "btn_save_hover_sprite.png"
    pressed = "btn_save_pressed_sprite.png"
    button = Button(idle_img, hover_img, pressed, on_button_press_callback)
    button.render()
    button.btn_state = ButtonEnum.PRESSED
    button.render()
```

Another example – Document publishing


```python
class DocumentState(enum.Enum):
    DRAFT = 0,
    MODERATION = 1,
    PUBLISHED = 2

class Document:

    def __init__(self, state):
        self.state = DocumentState.DRAFT
        # ..
        # ..


    def publish(self):
        if self.state == DocumentState.DRAFT:
            self.state = DocumentState.MODERATION
            print("Document is now waiting for approval")
        elif self.state == DocumentState.PUBLISHED:
            if current_user.role == "admin" and self.valid:
                self.state = DocumentState.PUBLISHED
            elif current_user.role == "admin" and not self.valid:
                self.state = DocumentState.DRAFT
            else:
                print("Only a admin can approve")
        else:
            print("Document already published.")

    #...
    #...
```

document_state_pattern_example.py***

State Pattern

We separate the state out into its own hierarchy.

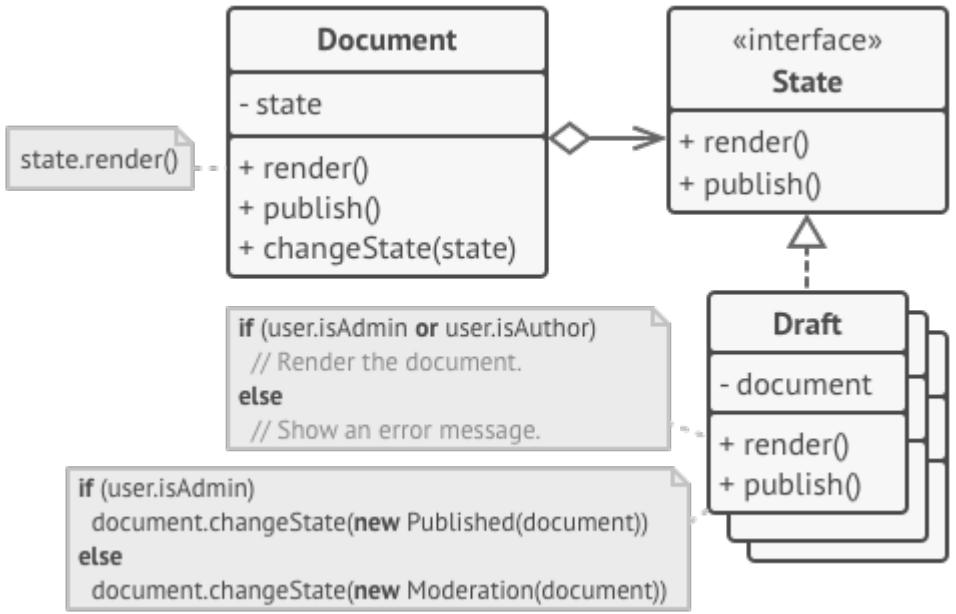
The document should be dependent on a State interface, not the concrete states (Draft, Moderation and Published).

Now we can add and remove states easily.

The State Interface consists of all the state-defined behaviours in the Document class. This allows the document class to communicate with it.

(In a way the Document wraps around its state)

Optionally, the state can keep a reference to the Document.



The generalized pattern

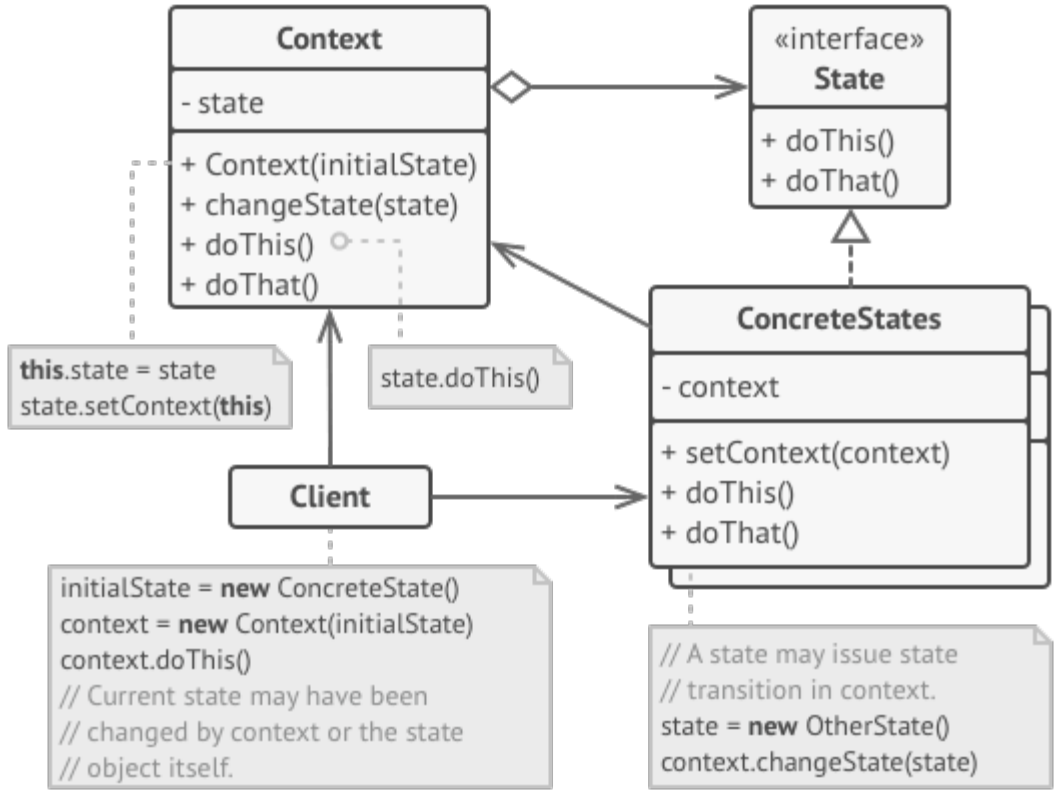
The object that has multiple states is called the **Context**

The context has a method to change its state.

The context delegates part or all of its behaviour to the State.

The concrete state may or may not have access to the context. If the state is responsible for changing the state of the context then this may be required.

The client instantiates the new state and assigns it to the context.



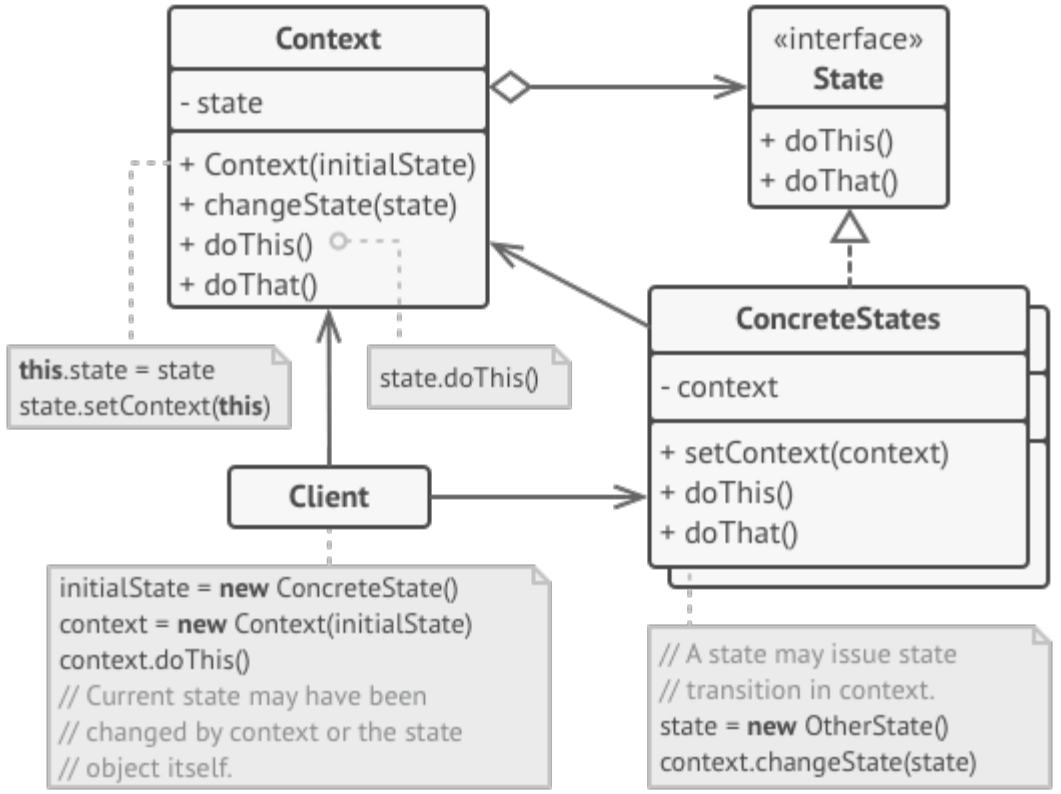
State example

We are working on a game where you want to have multiple game states controlled by a finite state machine (FSM)

Depending on user input, the FSM can change to one of the following states:

- Main Menu
- Gameplay
- Game Over

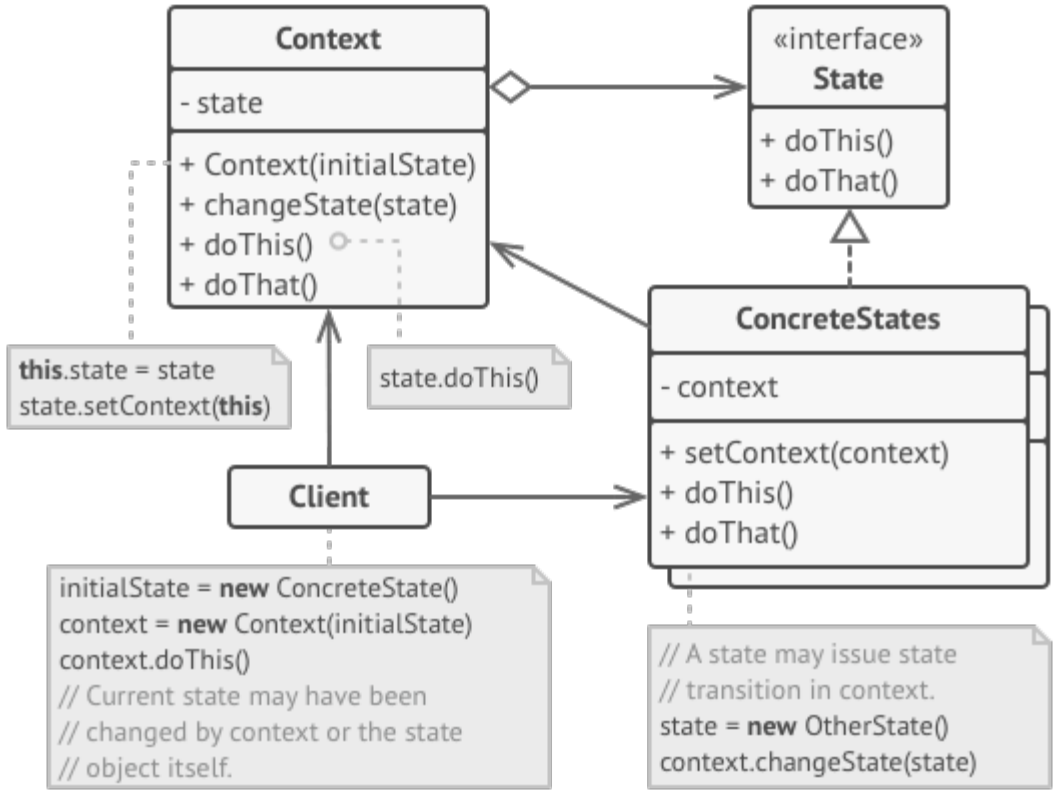
- The FSM's representation of the game changes based on its state. The game starts off in the Main Menu state



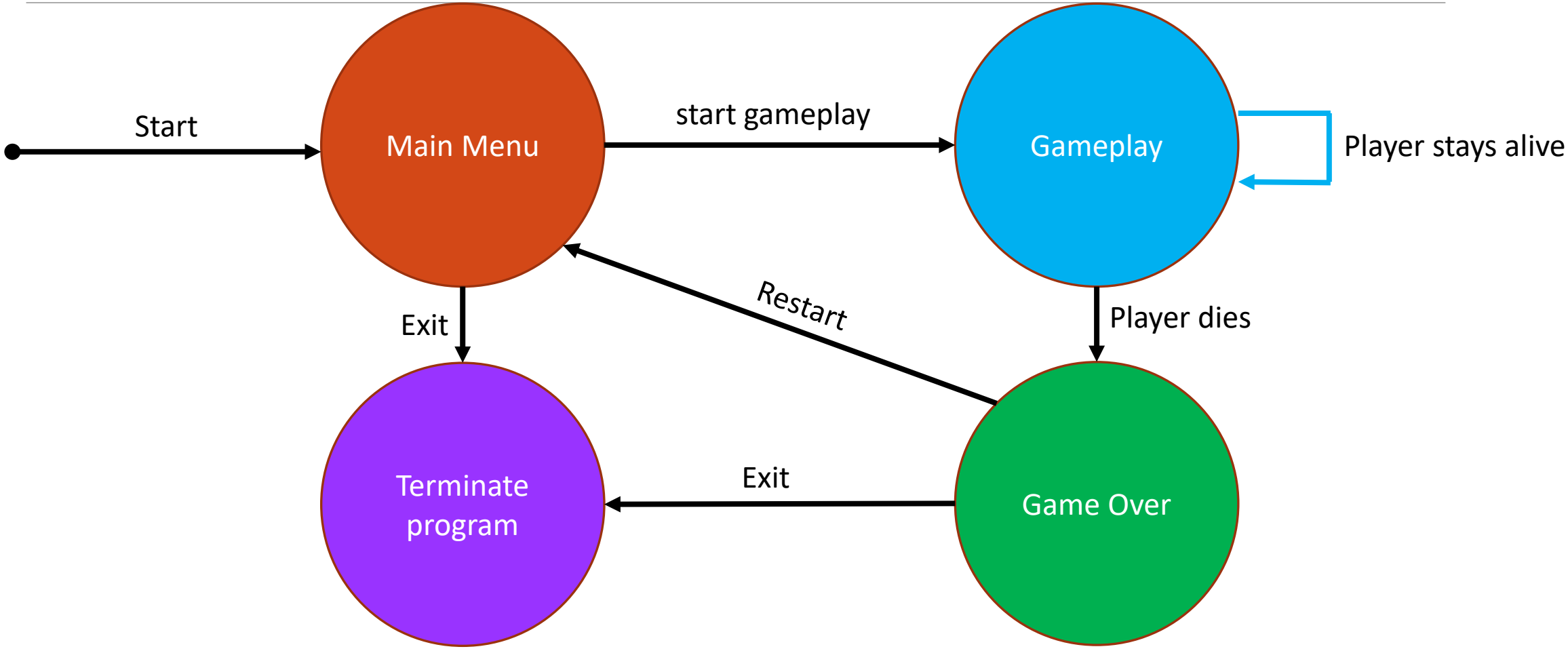
State example

- Different game events and transitions can occur depending on the state the game is in
- Main Menu state
  - Transition to gameplay state
  - Exit program
- Gameplay state
  - Keep playing the game
  - Character dies, going to the Game Over state
- Game Over state
  - Return to main menu state
  - Exit program

Draw a UML diagram depicting how you would use the state pattern to solve this.



State transition diagram



State example

Let's examine what happens in each state:

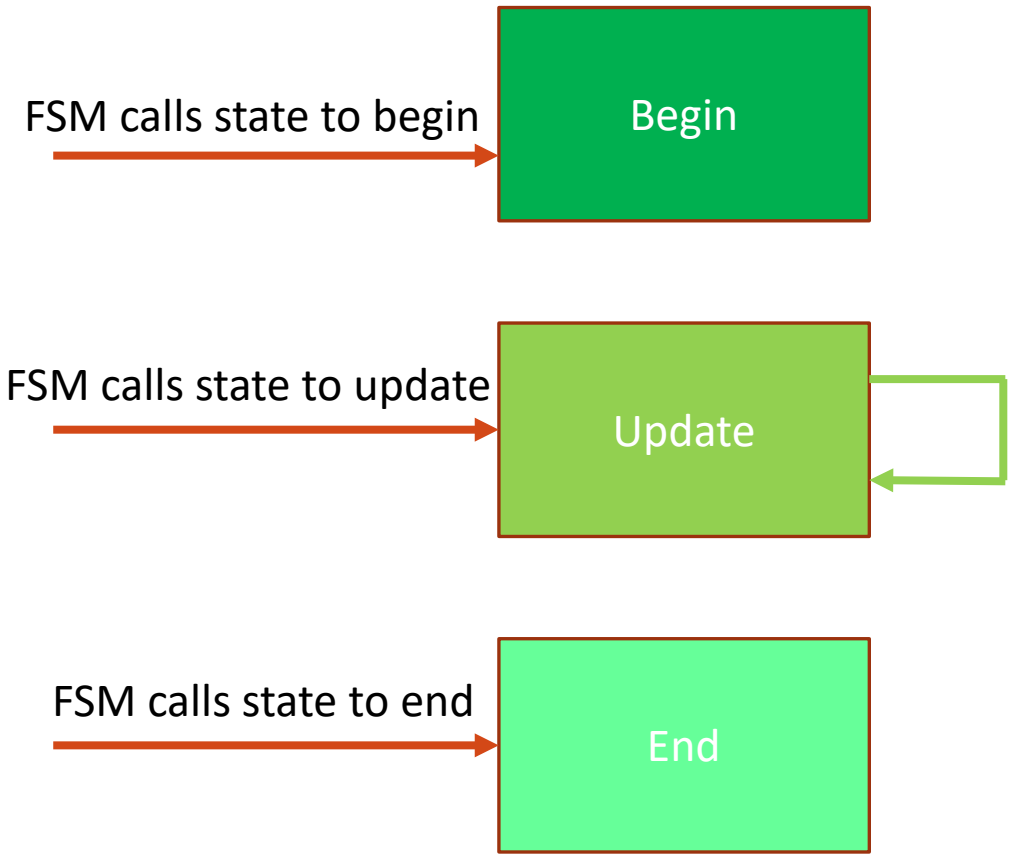
Generally contain **begin**, **update**, and **end** methods

Begin method – initialization executed once when entering the state

Update method – code executed every frame

End method – clean up executed done once when exiting the state

Each type of state (Main Menu, Gameplay, Game Over) has different code in each of the methods



State example

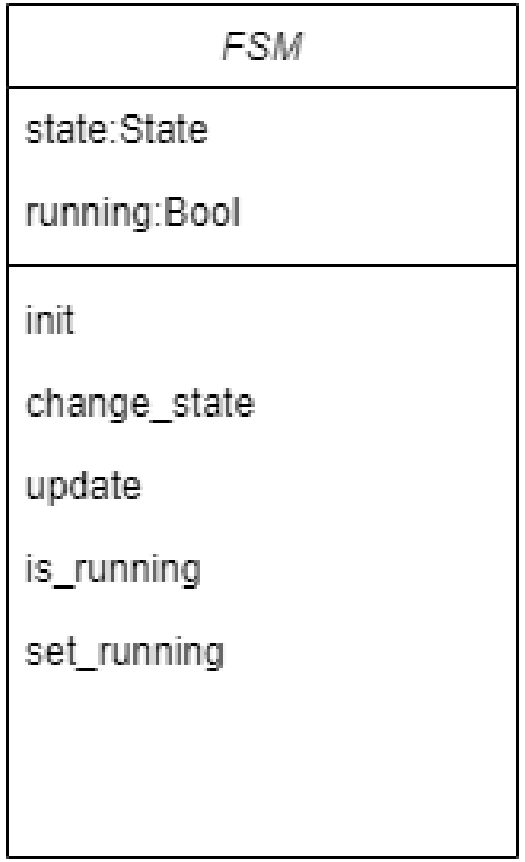
Let’s start with the FSM. This is the context that will keep changing states

Properties

- **state**
- running

Methods

- init
- change_state
- **update**
- is_running
- set_running



State example

Then the states

1 abstract base class

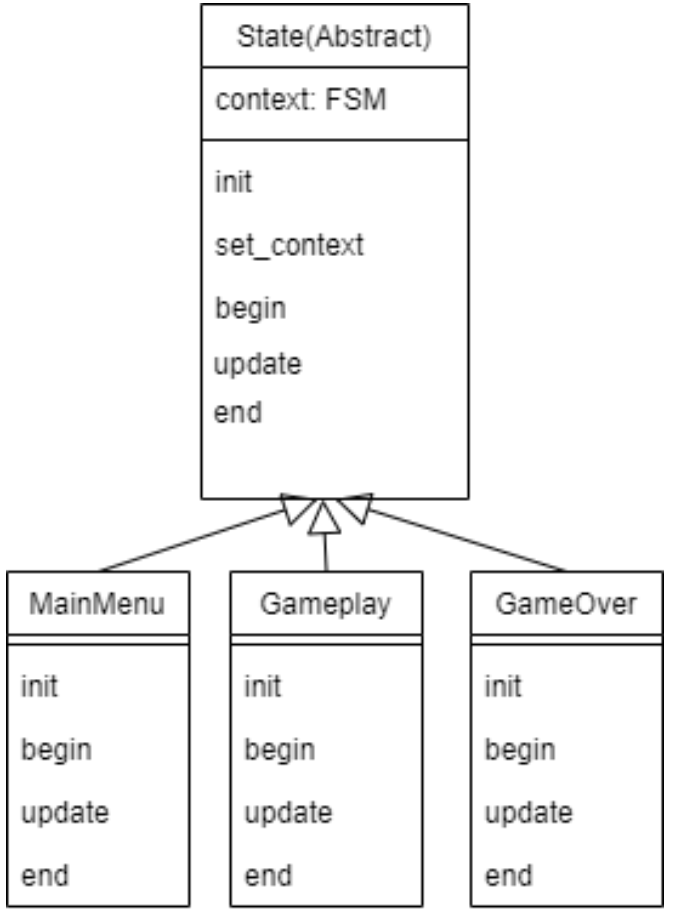
3 sub class states: MainMenu, Gameplay, GameOver

Properties

- **context – reference back to fsm context**

Methods

- init
- set_context – set reference back to context
- begin
- **update**
- end



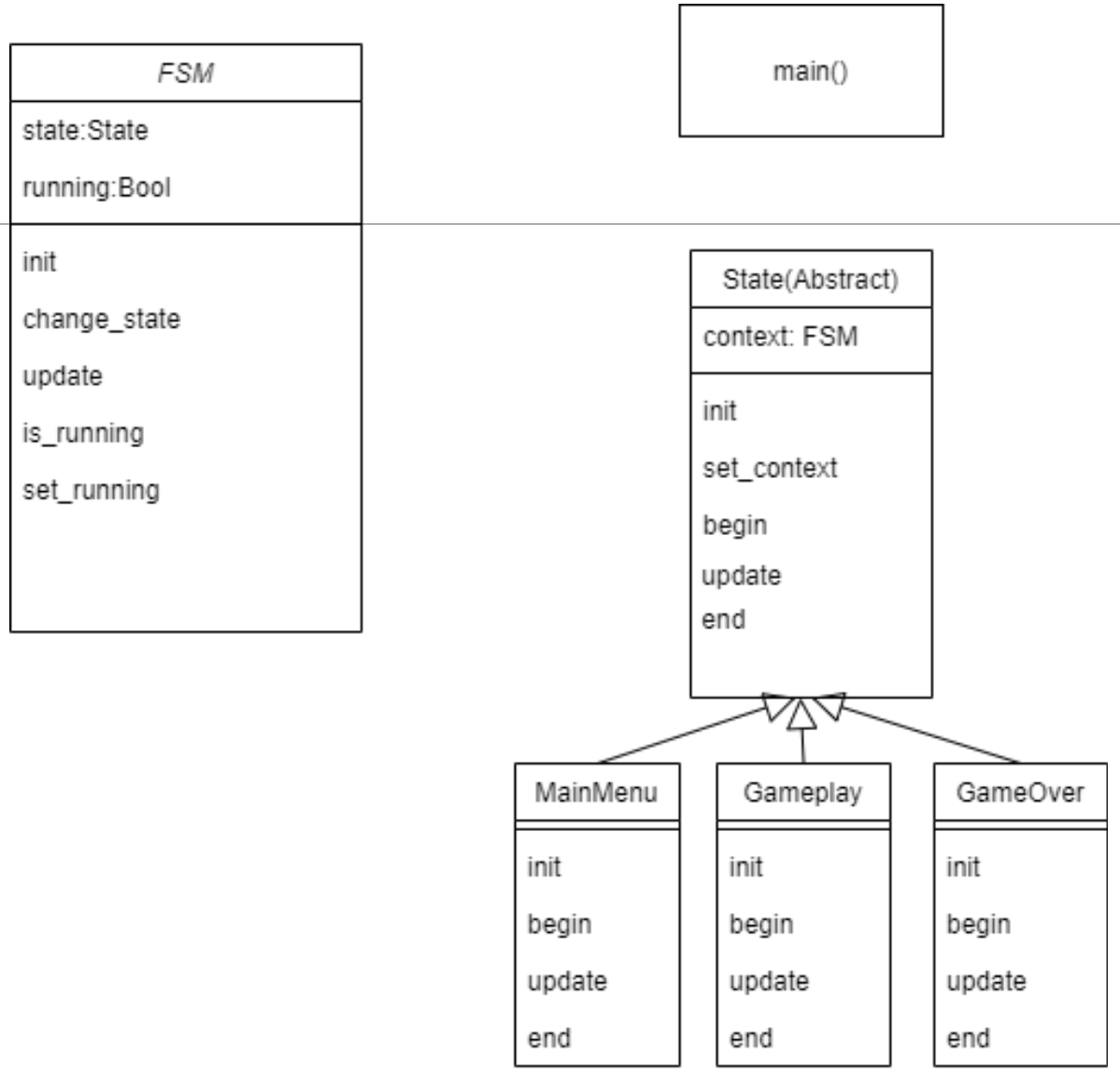
State example

Put it all together

main() is the client

Instantiate FSM and first state

Have a loop that will keep calling FSM update until FSM stops running



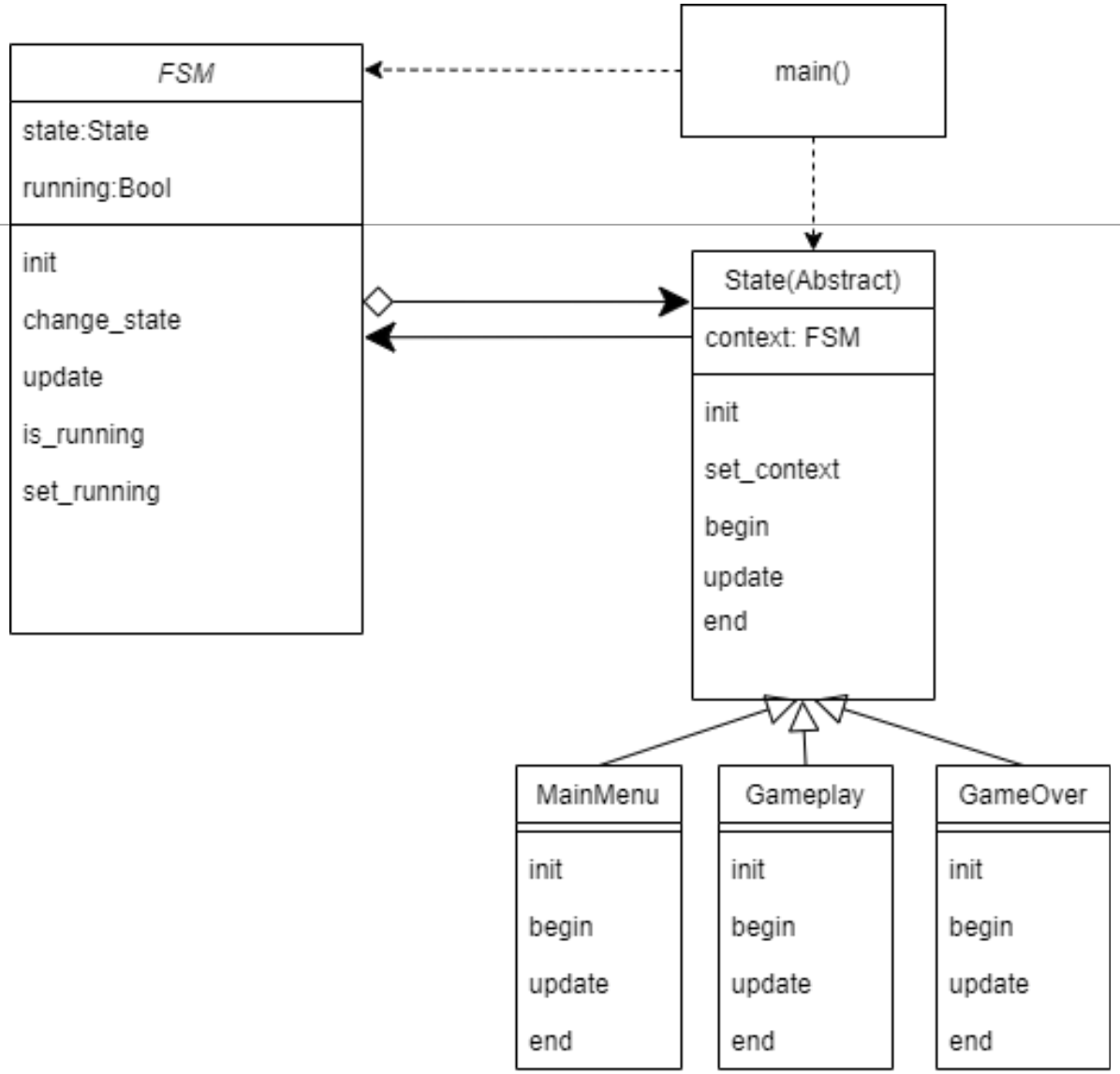
State example

Put it all together

main() is the client

Instantiate FSM and first state

Have a loop that will keep calling FSM update until FSM stops running



game_state_pattern.py

State: Why and When do we use it

- Use this pattern when an object's behaviour is heavily dependent on its states and the number of states can be large (or you may want to add more in the future)
- Avoid massive if statement blocks to handle object behaviours based on state conditions.
- You can create hierarchies of state classes if there are a lot of similar states that share code.
- Implements the Single Responsibility Principle. Each state is self-contained.
- Implements the Open/Closed Principle. We can add and remove states without modifying the context and multiple if-else statements.

State– Disadvantages

- Can become hard to maintain if the object rarely changes state or only has a few states. Sometimes a simpler solution involving conditional statements works better.
- State Transitions can be complex, especially if each state knows about its neighbouring states (that it transitions from/into). This can make the states highly coupled with each other.
- There are a lot more classes and code to maintain.

How is State different from Strategy?

They solve different problems

Strategy

- Choose an algorithm at run time
- Algorithms **don't know** about each other
- Context and Strategy **interfaces don't have to be the same**
- Strategy usually does not have a reference to context

State

- Change between states often based on some rules prescribed by a finite state machine.
- States **can know** about each other
- Context and state **interfaces match**
- Can have a reference to context

Summary

Strategy

- When you need to switch behaviours at runtime (sounds a lot like State)
- Another way of phrasing this is that it let's you swap algorithms at run-time.

State

- Allows a single object to have multiple behaviours.
- These may be completely different behaviours as if the object belonged to another class.
- The object is decoupled from its state.