Behavioural Patterns 1: Chain of Responsibility & Strategy & State

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 9

Last Class

Creational Patterns

- Patterns that help us instantiate an object or a group of objects in a manner that is decoupled.

Factory

- Separates creation code from the client code. Client depends on a factory and product interface not on concrete classes.

Abstract Factory

- Separates creation code from the client code. Client requires a family/group of objects that are related somehow.

Categorizing Design Patterns

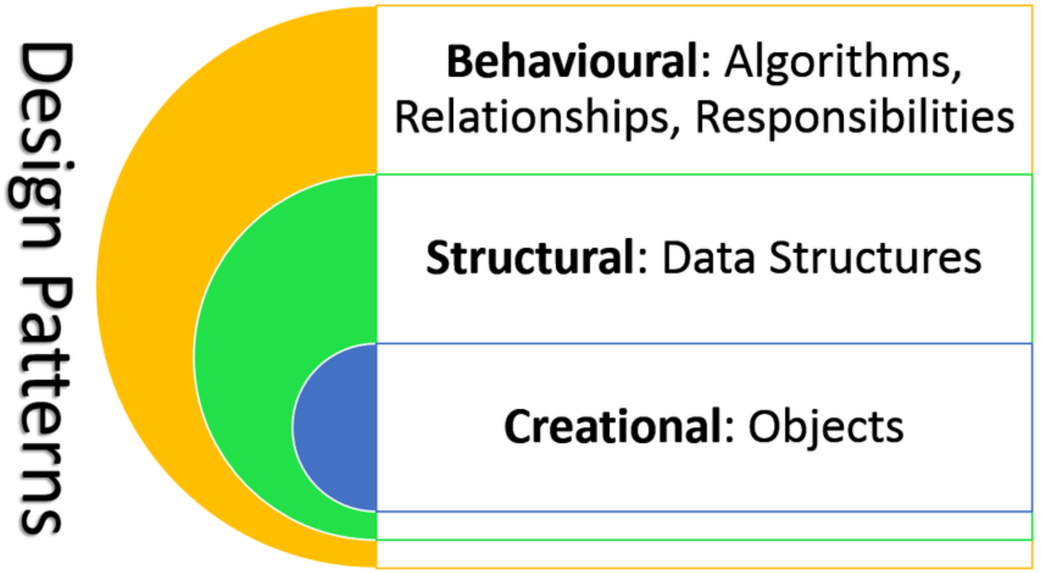
- **Behavioural (We are looking at these!)**

Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling? Algorithmic Patterns.

- **Structural** How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- **Creational**

All about class instantiation. Different strategies and techniques to instantiate an object, or group of objects


Design Patterns
Behavioural: Algorithms, Relationships, Responsibilities
Structural: Data Structures
Creational: Objects

Today we will be looking at 3 patterns…

Chain of Responsibility

- When a request needs to go through a series of steps.
- Decouples the sequencing of these steps and makes them re-usable.

Strategy

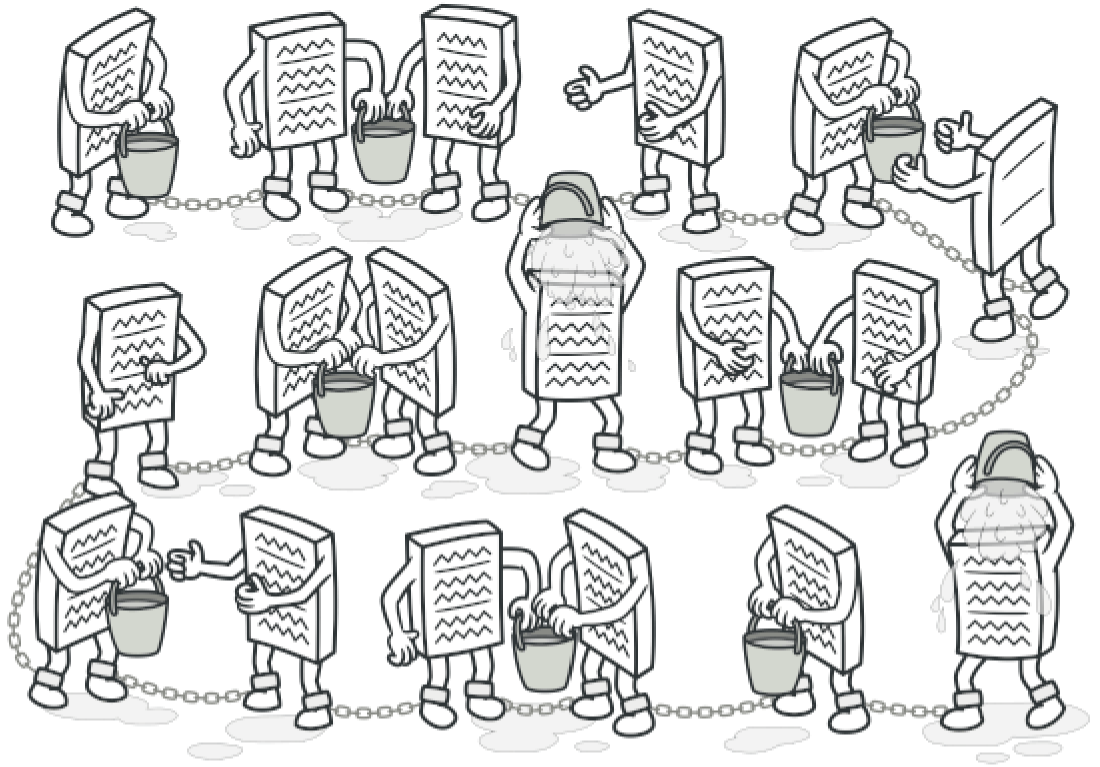
- When you need to switch behaviours at runtime (sounds a lot like State)
- Another way of phrasing this is that it let's you swap algorithms at run-time.

State

- When the same object can take on different behaviours depending on it's "state".
- Having different specialized behaviours within the same class,

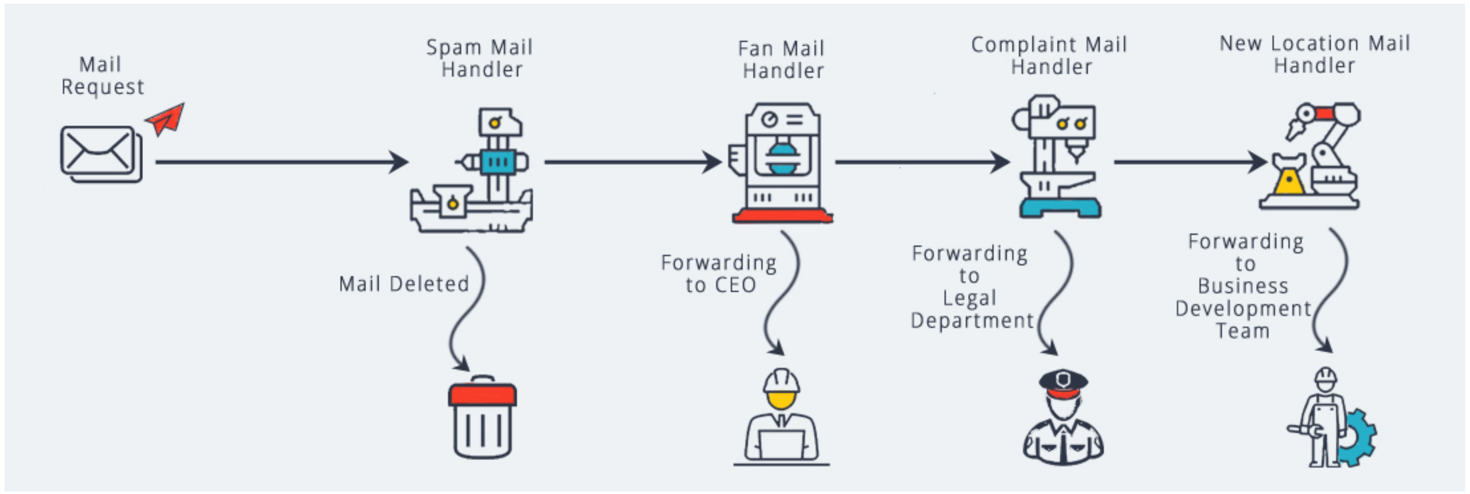
Chain of Responsibility

WHEN YOU WANT TO HAVE A LOT OF RESPONSIBILITIES BUT STILL BE FLEXIBLE



Chain of Responsibility

- When an object or a set of objects needs to undergo different "steps" of processing.
- These could be checks, validation, formatting, security, setup, etc.
- Set up a series of **Handlers**. Each handler is unique but **implements the same interface** and does something to a **Request**.
- Each Handler **has a reference to another handler** and it may, depending on it's code, pass on the request to another handler.



Chain of Responsibility - Scenario

Say there was an old fashioned school that did paper forms.

Students had to fill out a **Enrolment Form** every semester listing the courses they wanted to enrol in.

This form undergoes a series of validation checks, and get's processed by hand.

Student Enrolment Form

- Eventually (Finally!) the school went digital with their administrative tasks.
- Forms were processed by their administrative system.
- The code to process their enrolment form kind of looked like this.
- This is terrible.

```
def process_enrolment(student_application_form: EnrolmentApplicationForm) -
> (str, bool):
    validated = True

    student_record = database.get(student_application_form.student_id)
    if student_record.name != student_application_form.name:
        validated = False
        return "invalid record", validated
    if not student_application_form.fees_paid:
        validated = False
        return "fees not paid", validated

    for course in student_application_form.courses_for_enrolment:
        if not check_course_offering(course):
           validated = False
           return "course not offered", validated
        if student_record.age < course.min_age:
            validated = False
            return "age requirement not met", validated

    ..
```

Student Enrolment Form

- The system also processes other similar forms.
- Many of the forms share different validation checks.
- Sometimes the order in which they are done is changed depending on the form.
- Right now the code is duplicated and difficult to maintain. We have several big classes with redundant code.

```
def process_enrolment(student_application_form: EnrolmentApplicationForm) -
> (str, bool):
    validated = True

    student_record = database.get(student_application_form.student_id)
    if student_record.name != student_application_form.name:
        validated = False
        return "invalid record", validated
    if not student_application_form.fees_paid:
        validated = False
        return "fees not paid", validated

    for course in student_application_form.courses_for_enrolment:
        if not check_course_offering(course):
            validated = False
            return "course not offered", validated
        if student_record.age < course.min_age:
             validated = False
             return "age requirement not met", validated

    ..
```

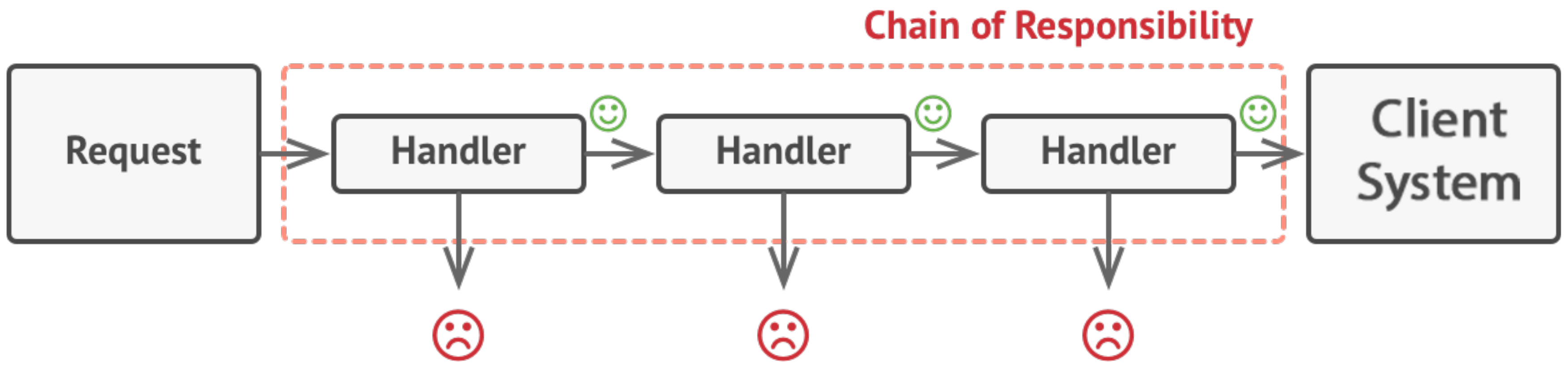
Chain of Responsibility

The Chain of Responsibility pattern separates these different processing steps into different classes. Each class is a **Handler**

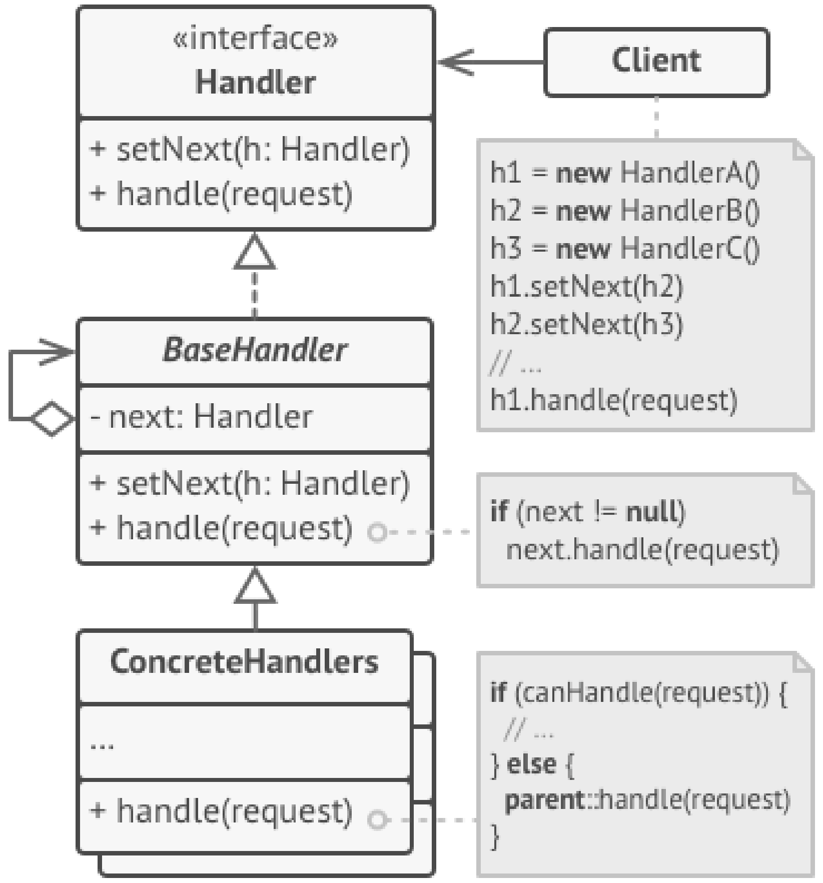
All these classes share the same interface, that is how one handler can pass on the request to another handler.

It kind of is like a linked list of responsibilities where each handler forms a node. We can arrange a different list for different scenarios.

The execution may stop midway and exit the chain if the Handler deems it necessary



Chain of Responsibility



```
h1 = new HandlerA()
h2 = new HandlerB()
h3 = new HandlerC()
h1.setNext(h2)
h2.setNext(h3)
// ...
h1.handle(request)
```

```
if (next != null)
  next.handle(request)
```

```
if (canHandle(request)) {
  // ...
} else {
  parent::handle(request)
}
```

Requirements:

- Each Handler implements the **same interface.**
- Base Handler is an **Optional** parent class that can hold some duplicate code (such as **setNext(h: Handler)**)
- Each handler implements a handle(request) method which is where they carry out their specific code.

Let’s implement our enrolment form!

Or at least a part of it.

Let’s draw out the UML diagrams

Enrolment Application Forms
- Student Account

Enrolment System

3 handlers to verify application forms:
- StudentValidationHandler
- CheckCourseOfferingHandler
- FeesHandler

Cheesy Teamwork Image for motivation

Let’s implement our enrolment form!

Or at least a part of it.

Let’s draw out the UML diagrams

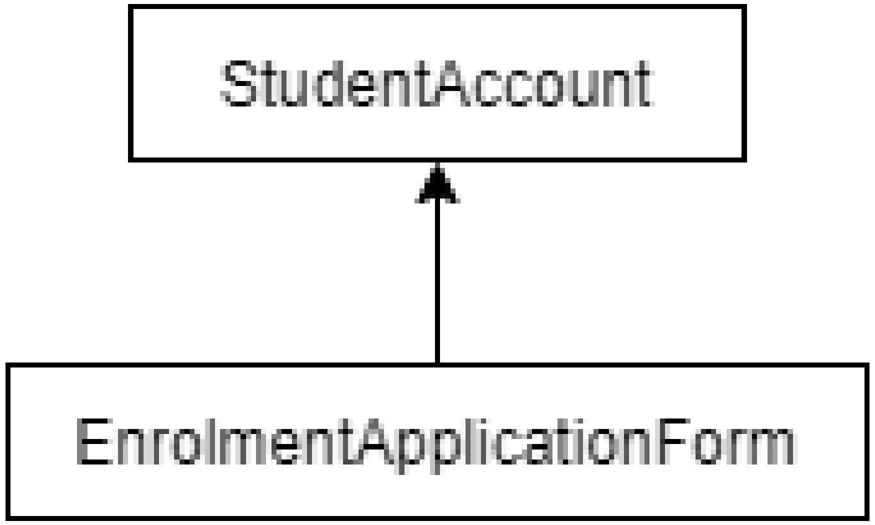
Enrolment Application Forms

- **Student Account**

Enrolment System

3 handlers to verify application forms:

- StudentValidationHandler
- CheckCourseOfferingHandler
- FeesHandler



Let’s implement our enrolment form!

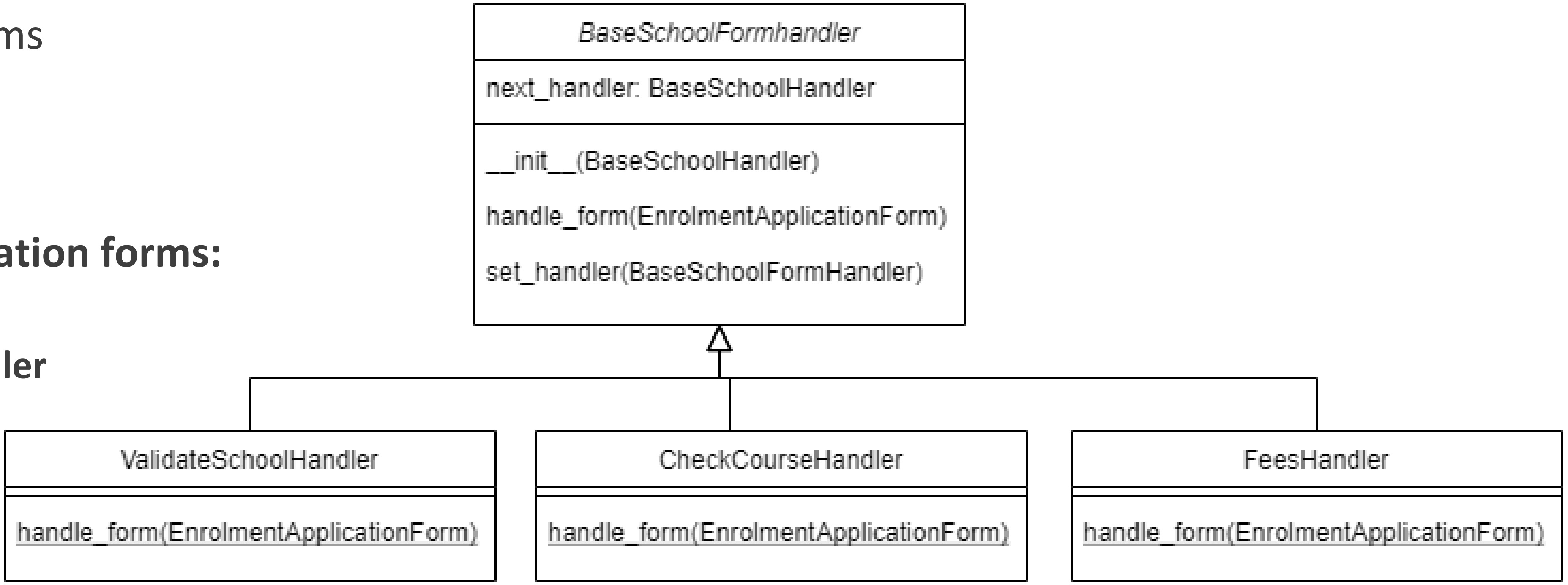
Or at least a part of it.

Let’s draw out the UML diagrams

Enrolment Application Forms
- Student Account

Enrolment System

3 handlers to verify application forms:
- **StudentValidationHandler**
- **CheckCourseOfferingHandler**
- **FeesHandler**



BaseSchoolFormhandler
next_handler: BaseSchoolHandler
__init__(BaseSchoolHandler)
handle_form(EnrolmentApplicationForm)
set_handler(BaseSchoolFormHandler)
ValidateSchoolHandler
handle_form(EnrolmentApplicationForm)
CheckCourseHandler
handle_form(EnrolmentApplicationForm)
FeesHandler
handle_form(EnrolmentApplicationForm)

Let’s implement our enrolment form!

Or at least a part of it.

Let’s draw out the UML diagrams

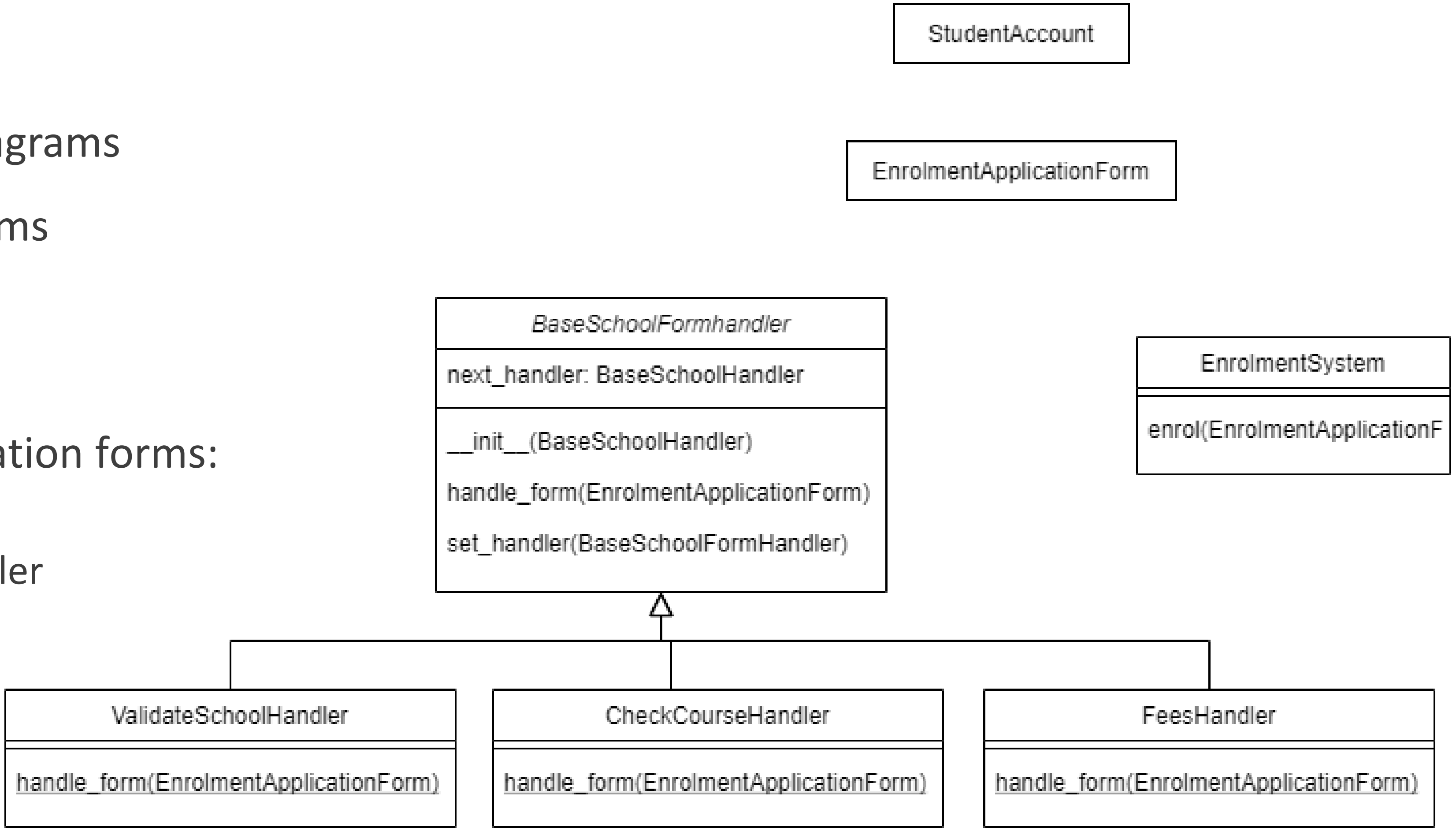
Enrolment Application Forms

- Student Account

Enrolment System

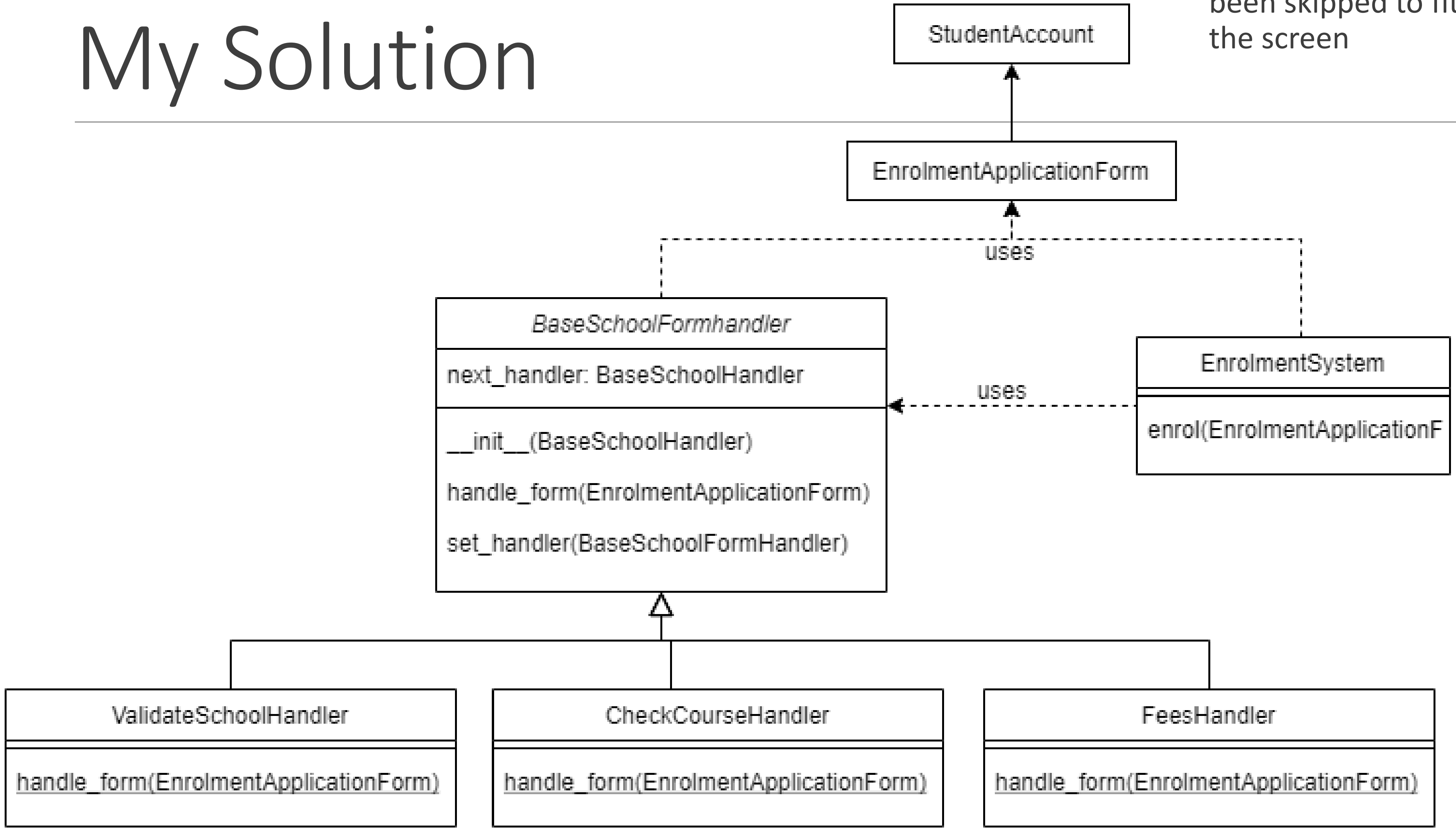
3 handlers to verify application forms:

- StudentValidationHandler
- CheckCourseOfferingHandler
- FeesHandler


StudentAccount
EnrolmentApplicationForm
BaseSchoolFormhandler
next_handler: BaseSchoolHandler
__init__(BaseSchoolHandler)
handle_form(EnrolmentApplicationForm)
set_handler(BaseSchoolFormHandler)
EnrolmentSystem
enrol(EnrolmentApplicationF
ValidateSchoolHandler
handle_form(EnrolmentApplicationForm)
CheckCourseHandler
handle_form(EnrolmentApplicationForm)
FeesHandler
handle_form(EnrolmentApplicationForm)

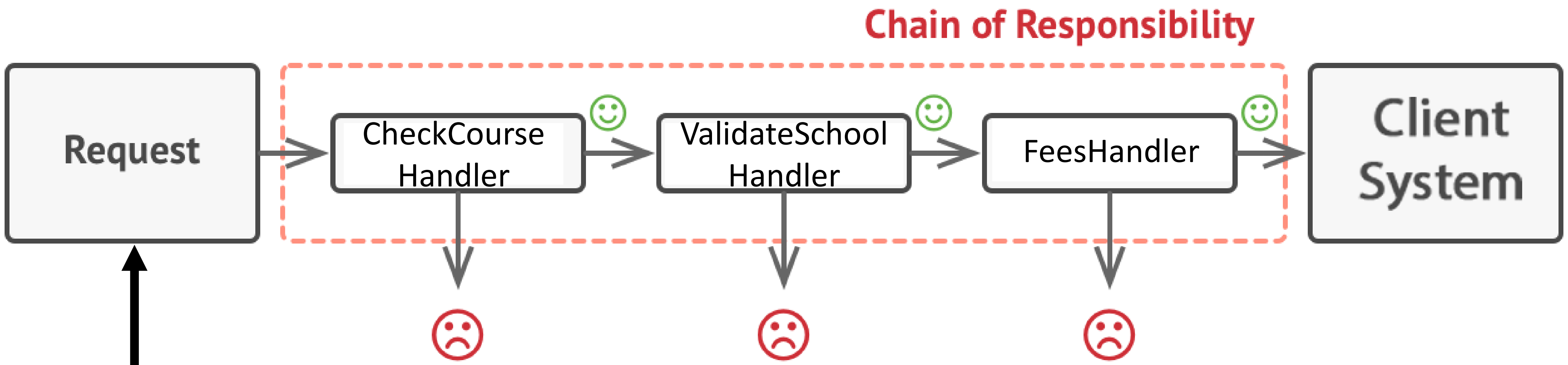
My Solution

NOTE: These would have attributes in them, but have been skipped to fit it all on the screen



StudentAccount
EnrolmentApplicationForm
uses
BaseSchoolFormhandler
next_handler: BaseSchoolHandler
__init__(BaseSchoolHandler)
handle_form(EnrolmentApplicationForm)
set_handler(BaseSchoolFormHandler)
uses
EnrolmentSystem
enrol(EnrolmentApplicationF
ValidateSchoolHandler
handle_form(EnrolmentApplicationForm)
CheckCourseHandler
handle_form(EnrolmentApplicationForm)
FeesHandler
handle_form(EnrolmentApplicationForm)

My Solution



Client System
1. Create student account and forms
2. Send form to enrolment system

chain_responsibility.py

Chain of Responsibility: Why and When do we use it

- If your program is expected to process different kinds of requests in various ways.
- When you need to do something in a particular order.
- If the sequence and ordering of request-processing is not known before hand and needs to be determined at run-time. We can control the order of request handling.
- Single Responsibility Principle. Each handler does one thing. We have decoupled **classes that invoke operations** (E.g. EnrolmentSystem) from **classes that perform operations** (the handlers).'
- Open/Closed Principle. We can introduce new handlers without modifying existing handlers or client code.

Chain of Responsibility – Disadvantages

- More classes to maintain.
- Some requests may end up unhandled. This may happen if we don’t set up the ordering of handlers properly.

Strategy

WHEN YOU WANT TO BE ABLE TO CHANGE YOUR PLANS

Liskov Substitution Principle

In programming, the Liskov substitution principle states that if ***S*** is a subtype of ***T***, then objects of type ***T*** may be replaced (or substituted) with objects of type ***S***.

Or

Objects in a program should be replaceable with instances of their base types without altering the correctness of that program.

Simply put, you should be able to refer to all the different UI components as its parent class, UIWidget

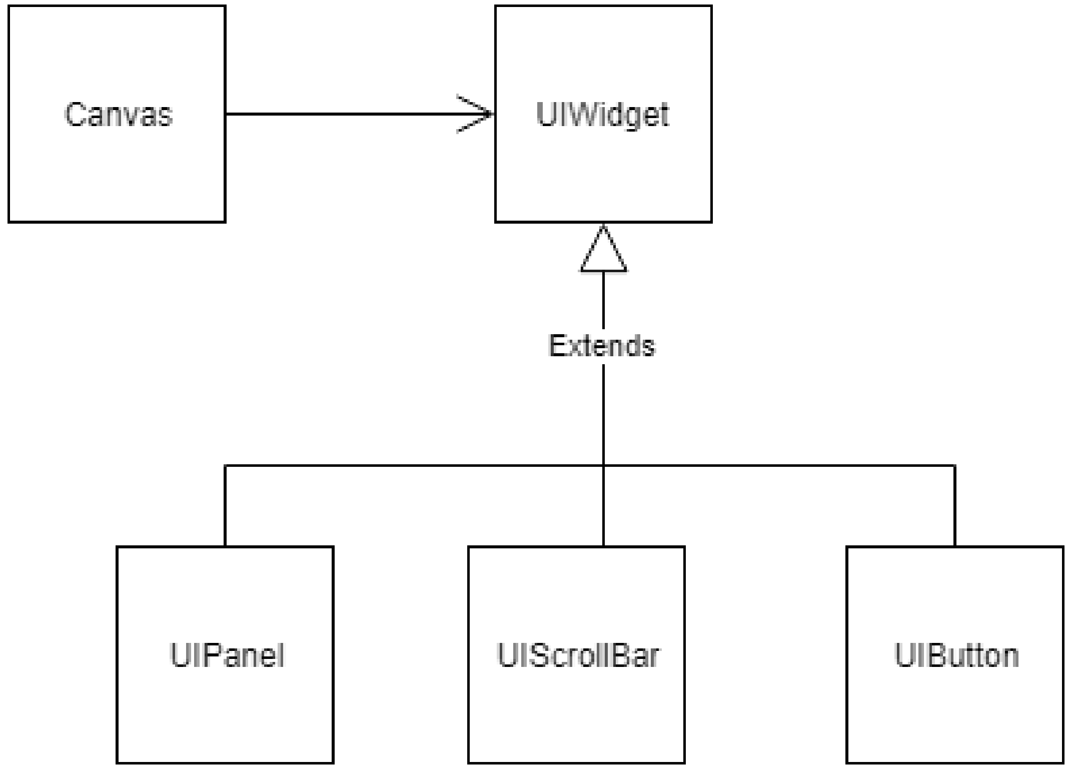
LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

In Python this is easy, since we don't worry about types!

```
Class Canvas:

    def __init__(self, widget_list):
        self.ui_widgets = widget_list

    def draw_screen(self):
        for widget in self.ui_widgets:
            widget.draw()
```

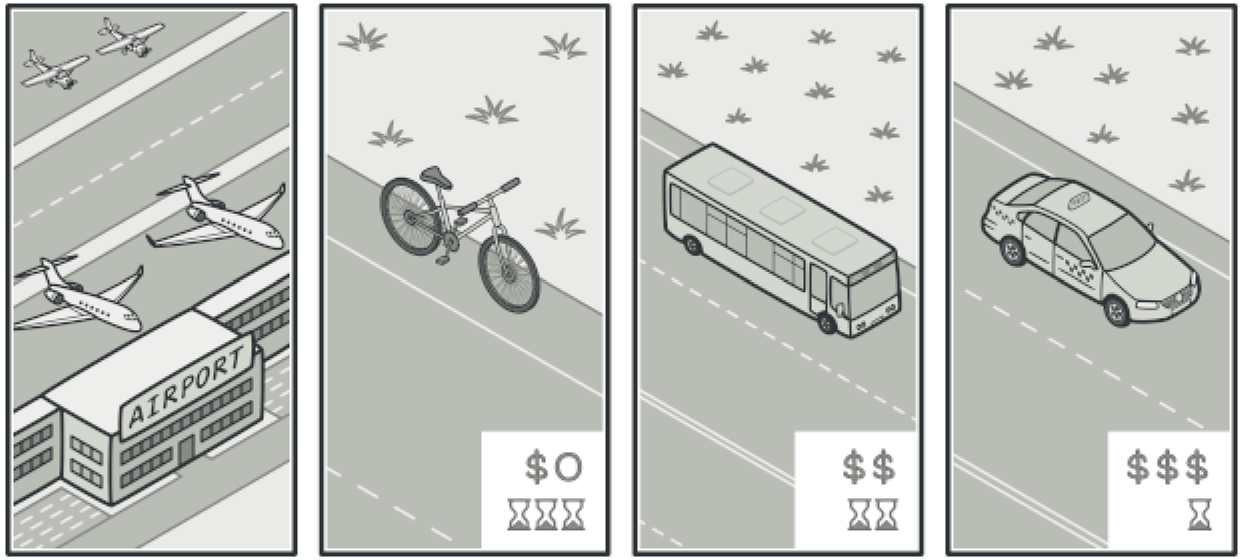
Wherever widget_list is used, we should be able to substitute its subclasses UIPanel, UIScrollBar, and UIButton

If we can not then the Liskov substitution principle fails. This means either the classes or the code calling the classes is implemented incorrectly



Strategy

- This is a really simple pattern that embodies Liskov's Substitution Principle
- In this pattern we create a hierarchy of behaviours that inherit from a base class
- We then use composition to be able to swap them out for each other at run time.
- You have done this already at some stage or form
- For example, you want to get to the airport and there are many strategies to do so.
- You can use a car, a bike or a bus. They all have similar behaviours with different implementations



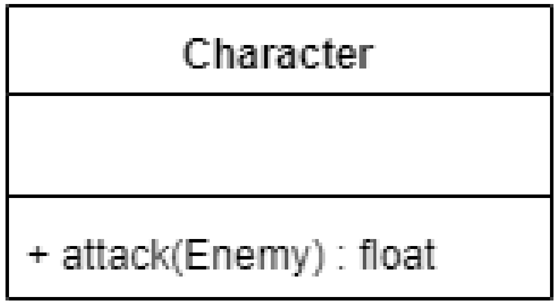
Strategy Scenario

Say you have built a game that lets you equip a character with weapons.

This weapon has a specific attack animation and damage calculation.

This is handled by the attack method of your character class

You make the best game in the world and it’s an instant success.



Strategy Scenario

Eager to continue pleasing your fans you decide to release an update with 2 new weapons!

You modified the Character class, and added code to the attack method to contain the attack animations and damage calculations for these new weapons.

Over time you added 5 more weapons.

Your Character class is huge. Any change to this causes errors.

Your team of developers keep having merge conflicts with this class. One developer is working on the character movement and you are working on character attack.

Single Responsibility Principle has been violated and so has open closed principle.

Strategy Scenario – Liskov Substitution Principle to the Rescue!

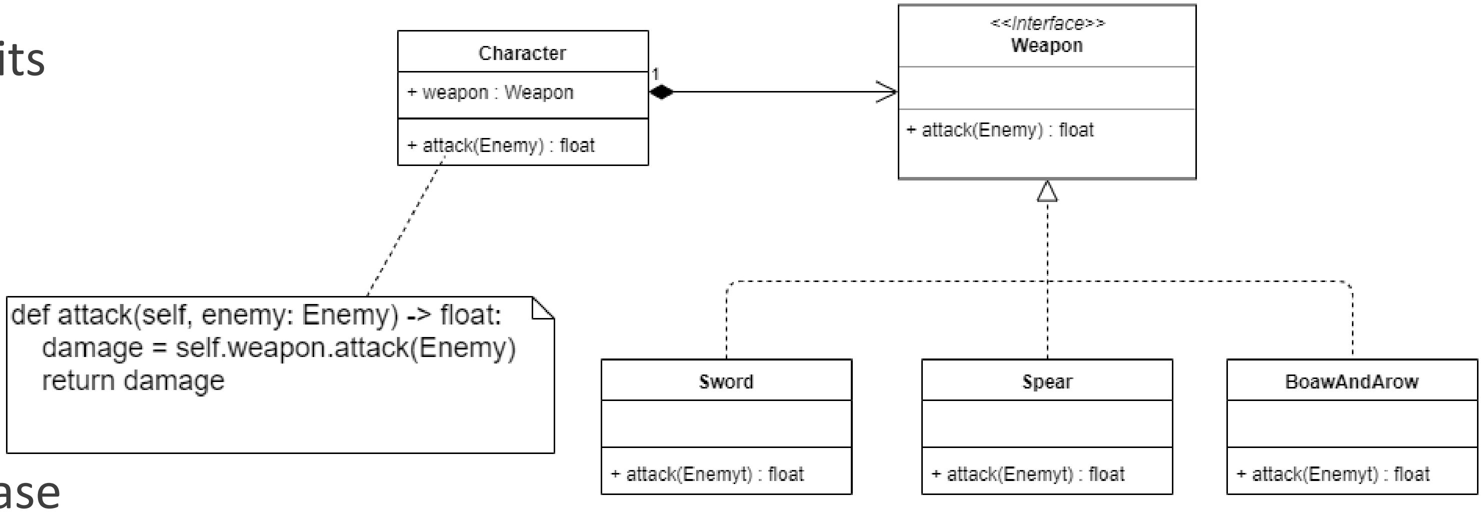
What if we used composition instead.

We can:

1) Separate the attack behaviour into its own class (Weapon)
Solves Single Responsibility Principle

2) Inherit from Weapon to implement different kinds of weapons.
Solves Open Closed Principle

3) Have the Character class store an attribute that holds a reference to a Base Weapon. We can replace this weapon (parent) with any of the different ones (child) during run time.
Liskov Substitution Principle

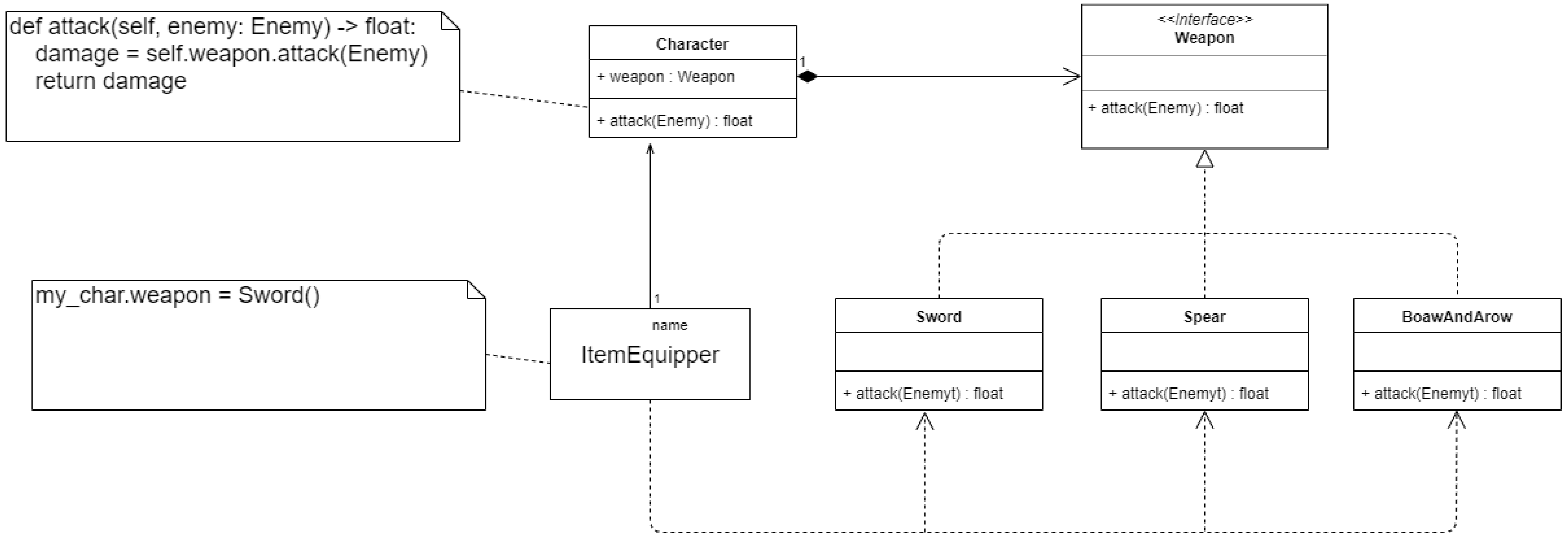


```
def attack(self, enemy: Enemy) -> float:
    damage = self.weapon.attack(Enemy)
    return damage
```

Strategy Scenario – How do we change the weapon?

A separate client class can change the weapon of the character during runtime.

Let's say this class was called ItemEquipper


```
def attack(self, enemy: Enemy) -> float:
    damage = self.weapon.attack(Enemy)
    return damage
```

```
my_char.weapon = Sword()
```

Strategy

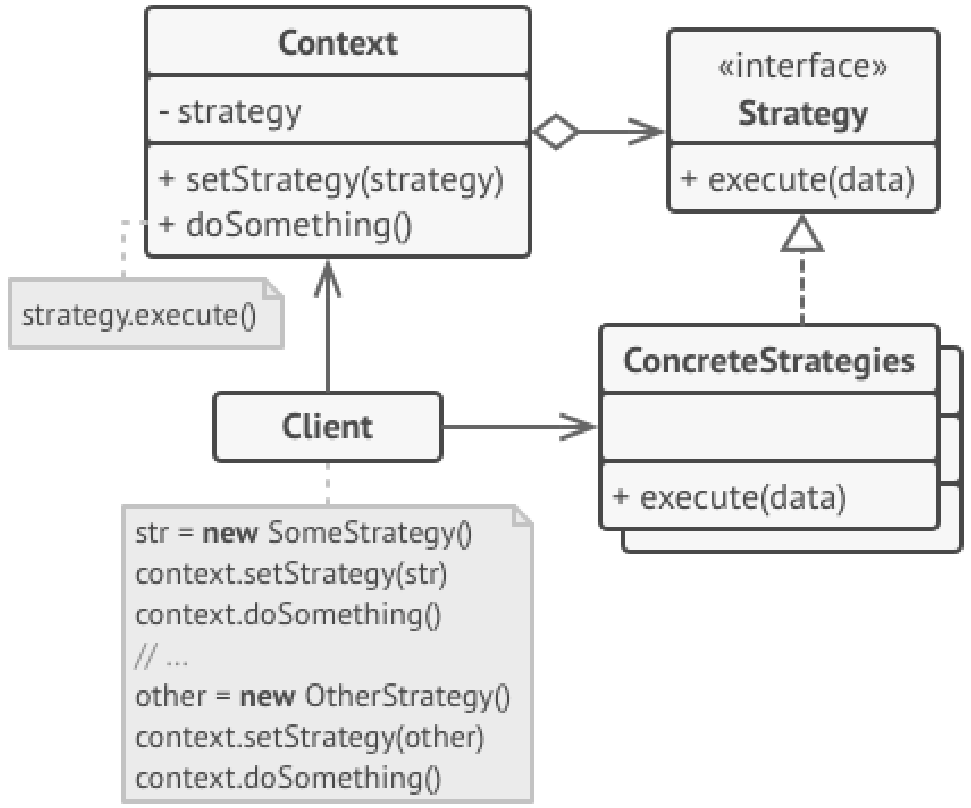
The original class that makes use of the behaviour/strategy is known as the **Context**.

In the previous example:

- the context was the character.
- the behavior/strategy was the weapon

This is a fairly simple pattern.

NOTE: The method(s) in the context and the method(s) in the Strategies don't need to have the same interface.



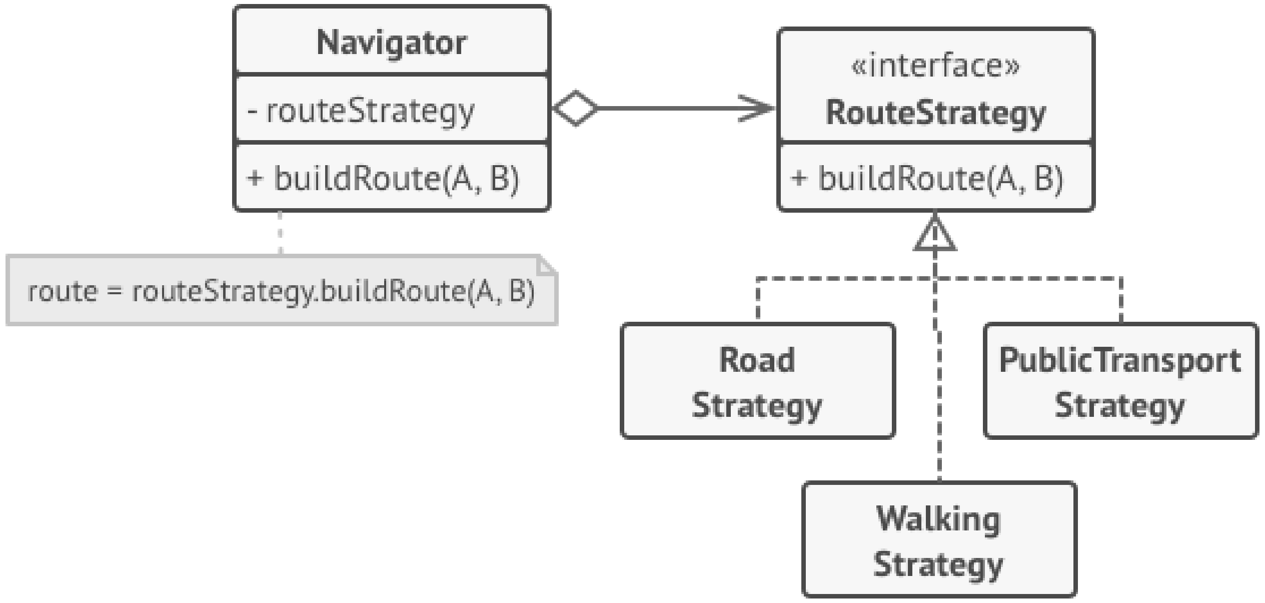
```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

Strategy – A non game example

But Jeff. We don’t all play games.

Give us other examples.

Consider an application that let you calculate the fastest route between two points. This works for cars. We release new versions where it also calculates routes for people who are walking (Depth First Search), and Public Transit (A* AI Path Algorithm) as well



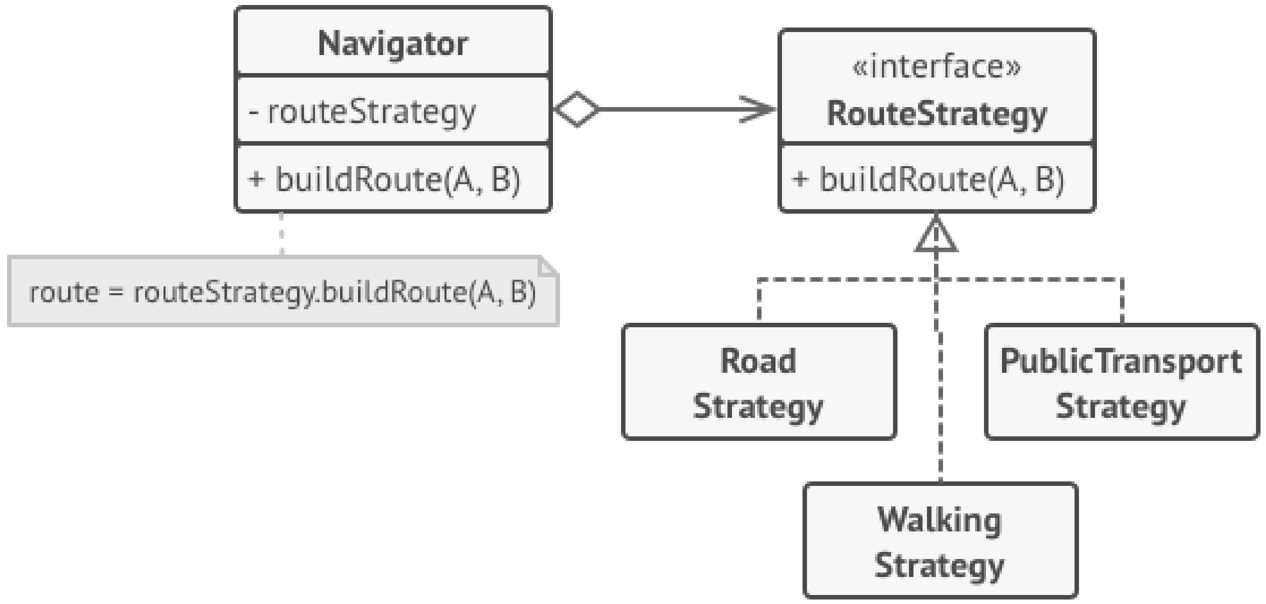
Strategy in Python

Remember Python doesn't have support for Interfaces.

Sure we could define a ABC as an interface, but if this is just for one method then it's kinda overkill.

So we can just define 3 classes and skip the base class:

- RoadRouter
- WalkRouter
- PublicTransportRouter



Strategy in Python

Now **IF, and ONLY IF** the classes contain one method and don’t make use of any instance variables. We could also replace these with simple Functions and not use Classes at all.

As long as the functions have the same signatures and return types we can pass them as an object to Navigator and still retain all the advantages of the Strategy Pattern.

```python
class Navigation:
    def __init__(self):
        self.route_function = None

    def __call__(self, a, b):
        return self.route_function(a, b)


def car_router(a, b):
    print("I find the fastest route when driving")

def walk_router(a, b):
    print("I find the fastest route when walking")

def public_transit_router(a, b):
    print("I find the fastest route when taking transit")

def main():
    point_a = (100,100)
    point_b = (200,200)
    nav = Navigation()
    nav.route_function = car_router
    nav(point_a, point_b)
    nav.route_function = walk_router
    nav(point_a, point_b)
```

Strategy Let’s Implement One

Say we have a ReportClass that can print reports.

A report consists of a heading, content and author.

We might want to format and print the report in different ways.

- We might want to format the content as bullet points and mention the author at the top
- We might want to use paragraphs and mention the author at the end.

strategy_sample_code.py

Strategy: When and Why do we use it

- When you want to use a different algorithm for different use cases. You can also swap them at run time
- Adheres to the open closed principle. We can add new strategies without modifying the context
- Replace inheritance with composition (similar to bridge pattern)
- Isolate the implementation of a strategy from the code that uses it.

Strategy– Disadvantages

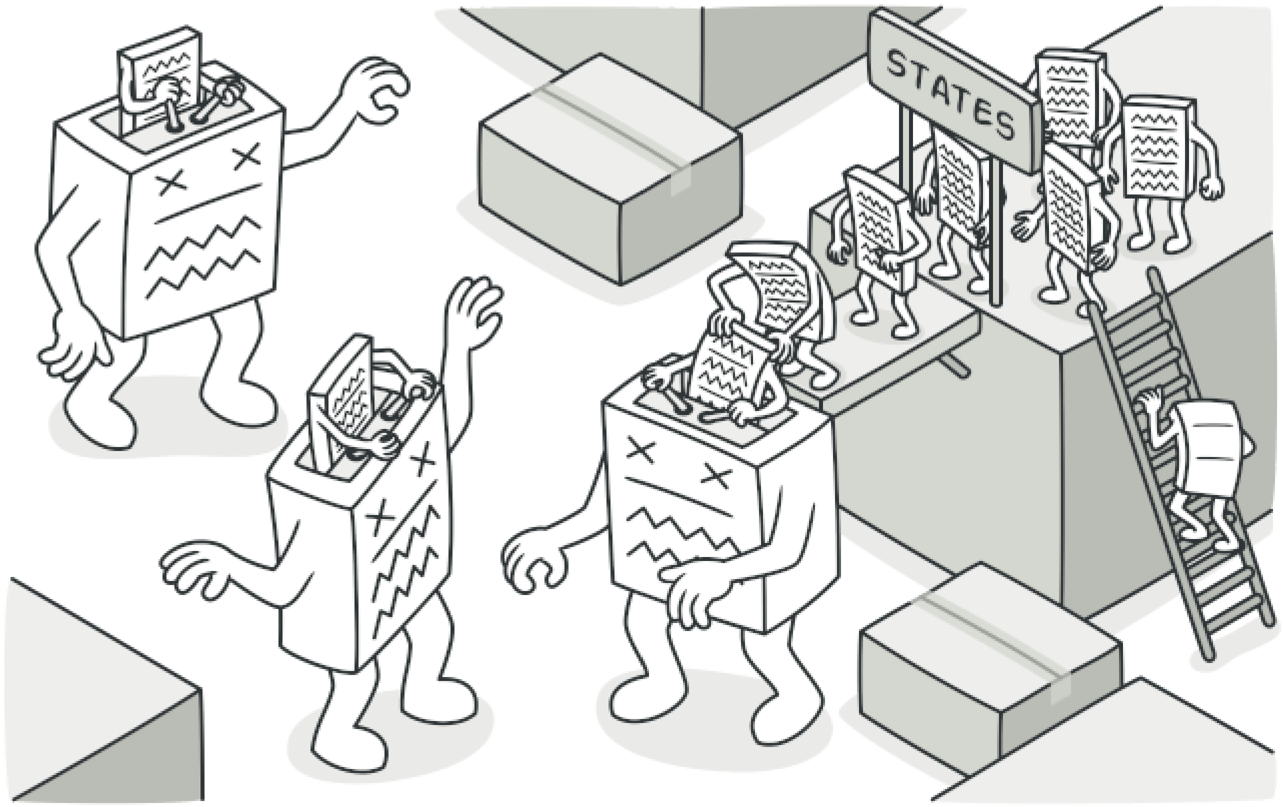
- It’s overkill if you only have a few algorithms that rarely switch.
- Client needs to be aware of the different strategies to be able to select between them.

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 it's 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 it's execution is tracked.

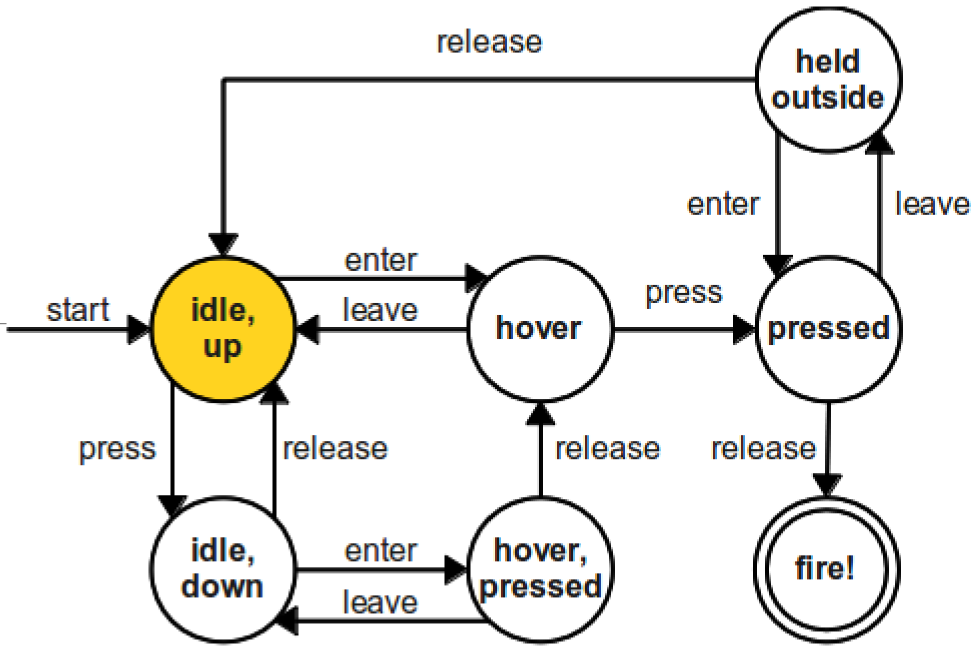
Within a state, the program behaves differently

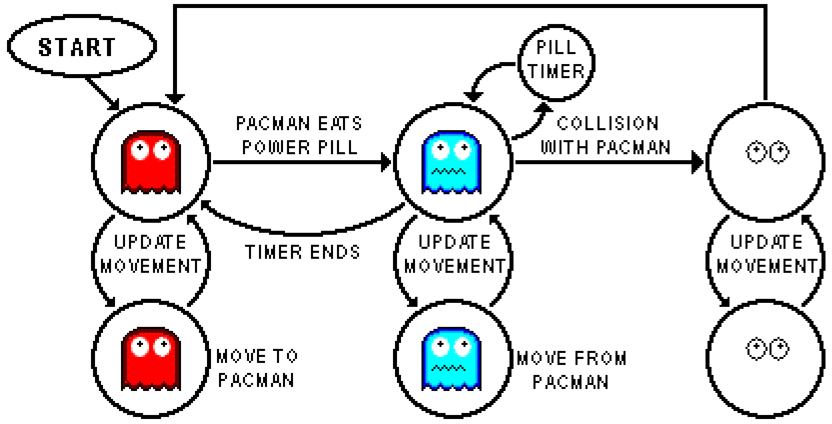
We can change the state of a program instantly.

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…

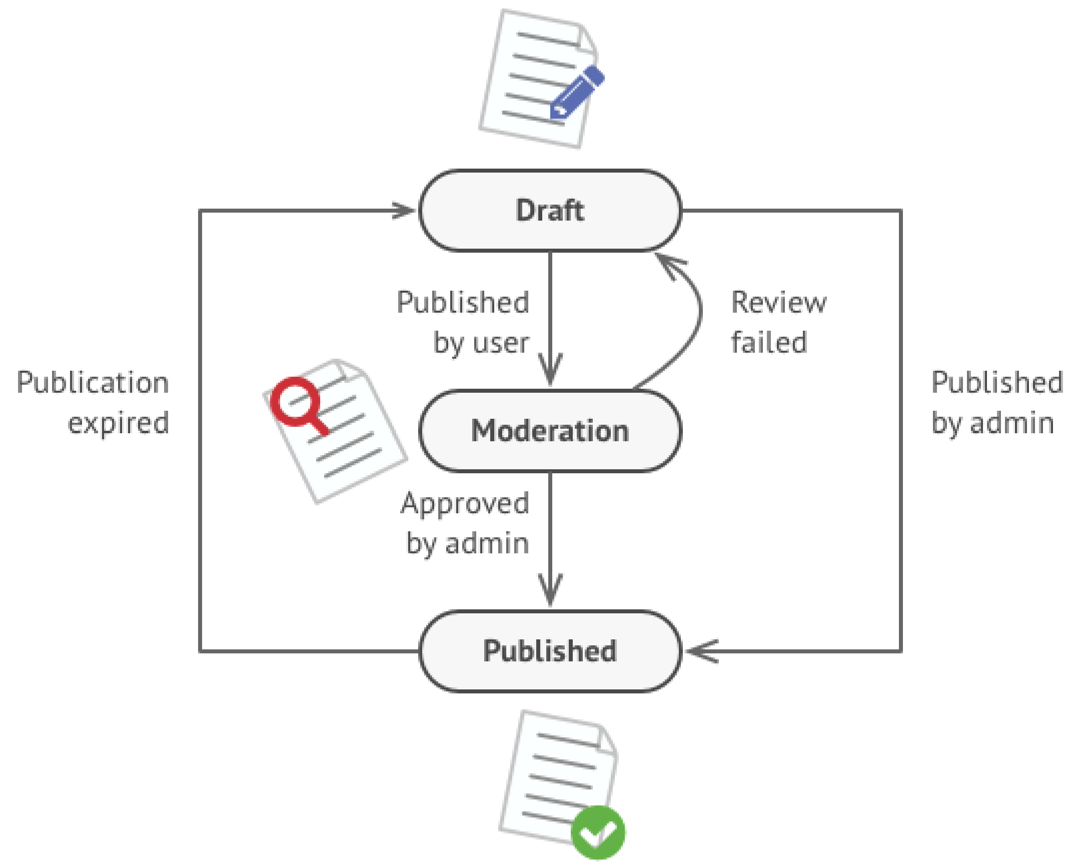
Making any changes to the states will effect the button class.

Button is coupled to it's state.

We want the button to be decoupled from its state.

```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()
```

Another example – Document publishing



document_state_pattern_example.py***

```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.")

    #...
    #...
```

State Pattern

We separate the state out into its own hierarchy.

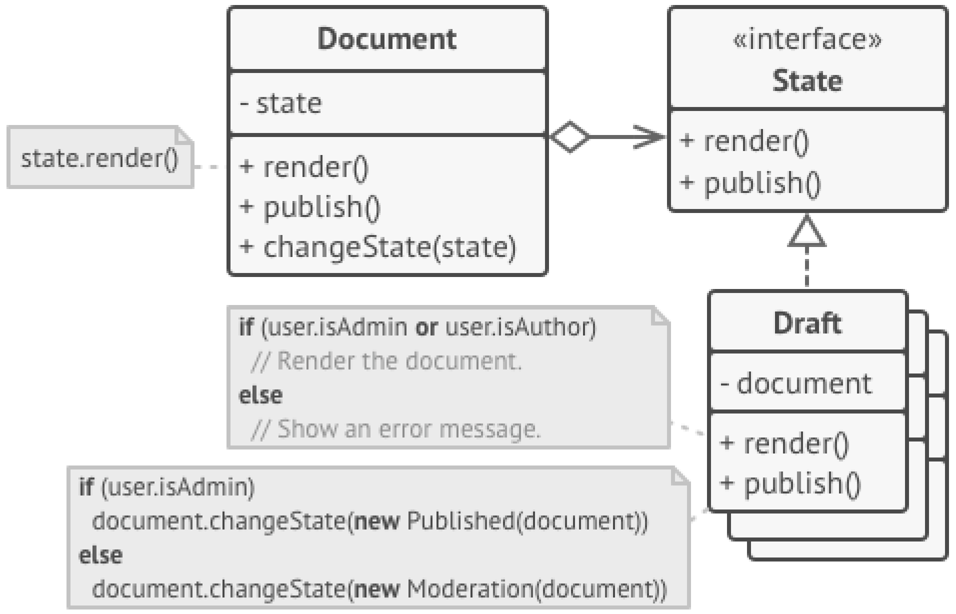
The document should be dependent on a State interface, not the concrete states (Draft, Moderation and Published).

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.



```
if (user.isAdmin or user.isAuthor)
  // Render the document.
else
  // Show an error message.
```

```
if (user.isAdmin)
  document.changeState(new Published(document))
else
  document.changeState(new Moderation(document))
```

The generalized pattern

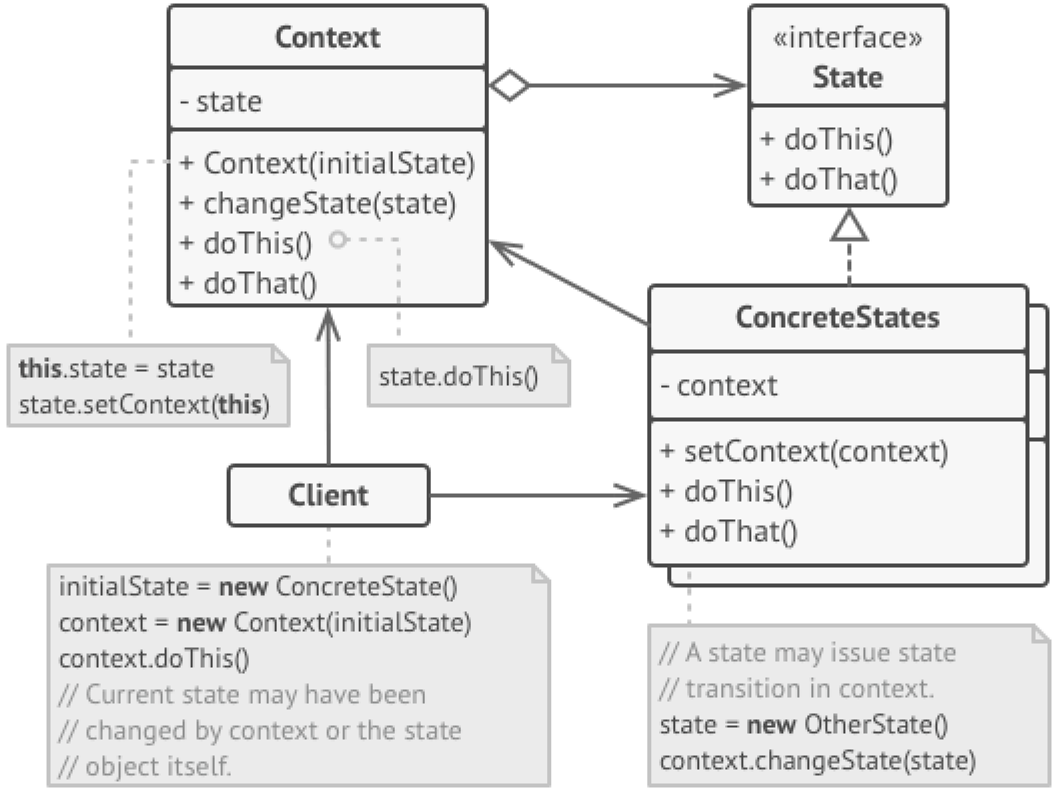
The object that has multiple states is called the **Context**

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.



```
this.state = state
state.setContext(this)
```

```
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
```

```
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)
```

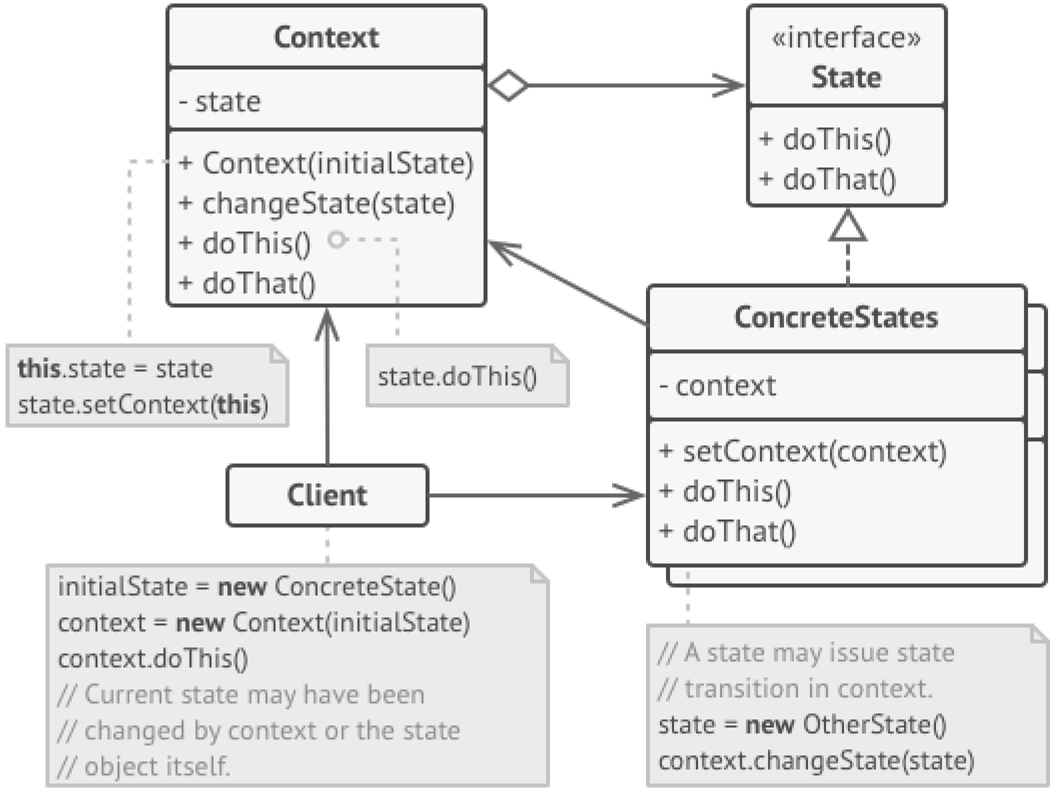
State example

We are working on a game where you control a fairy.

Depending on user input, the fairy can take one of the following forms:

- Fire
- Water
- Wind

- The fairy’s attack behaviour changes based on its state. The fairy starts off as a Fire Fairy.



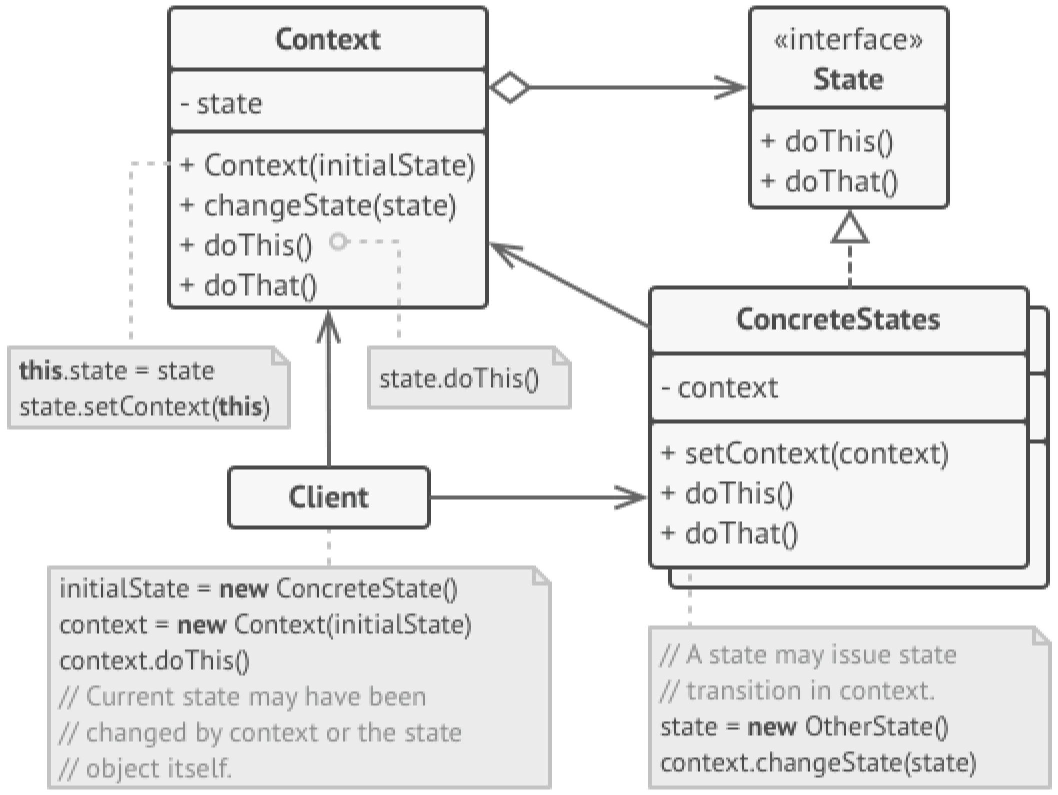
```
initialState = new ConcreteState()
context = new Context(initialState)
context.doThis()
// Current state may have been
// changed by context or the state
// object itself.
```

```
// A state may issue state
// transition in context.
state = new OtherState()
context.changeState(state)
```

State example

- Fire strong vs wind, Wind strong vs water, Water strong vs fire
- Strong attacks deal double damage
  - Fairy changes their state to avoid damage

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



State example

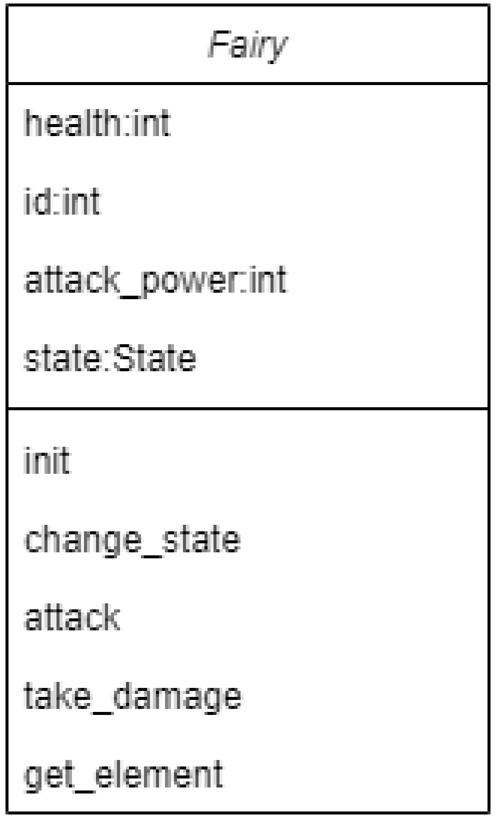
Let’s start with the fairy,

Properties

- health
- id
- attack power
- **state**

Methods

- attack
- take damage
- change state
- get element



State example

Then the states

1 abstract base class

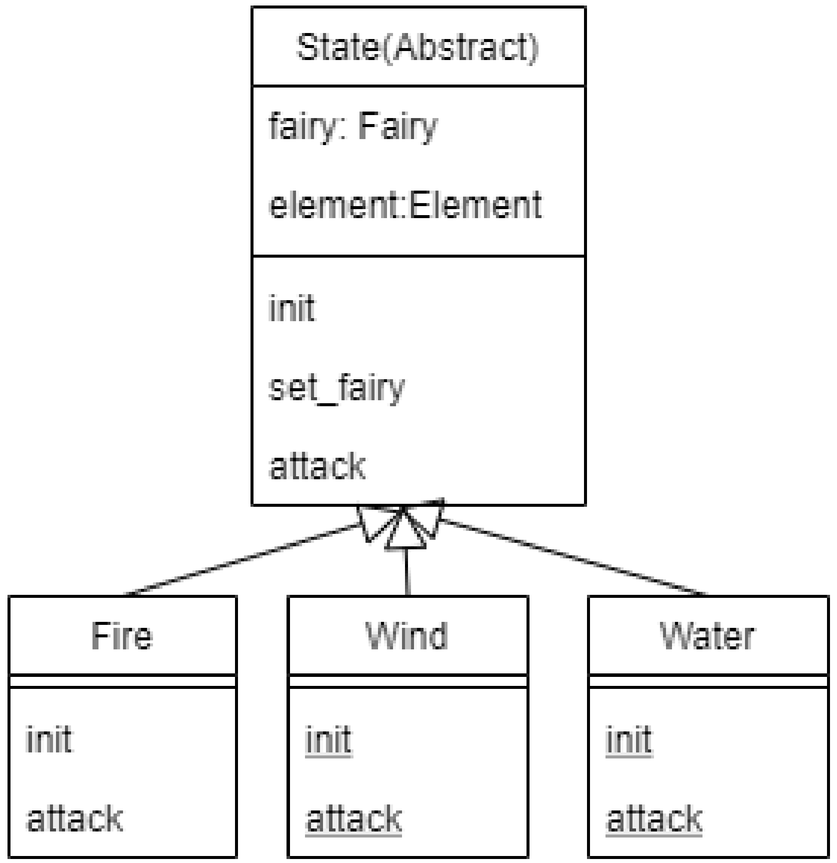
3 sub class states: Fire, Wind, Water

Properties

- **fairy – reference back to context**

Methods

- Init
- set_fairy – set reference back to context
- attack



State example

Then the states

1 abstract base class

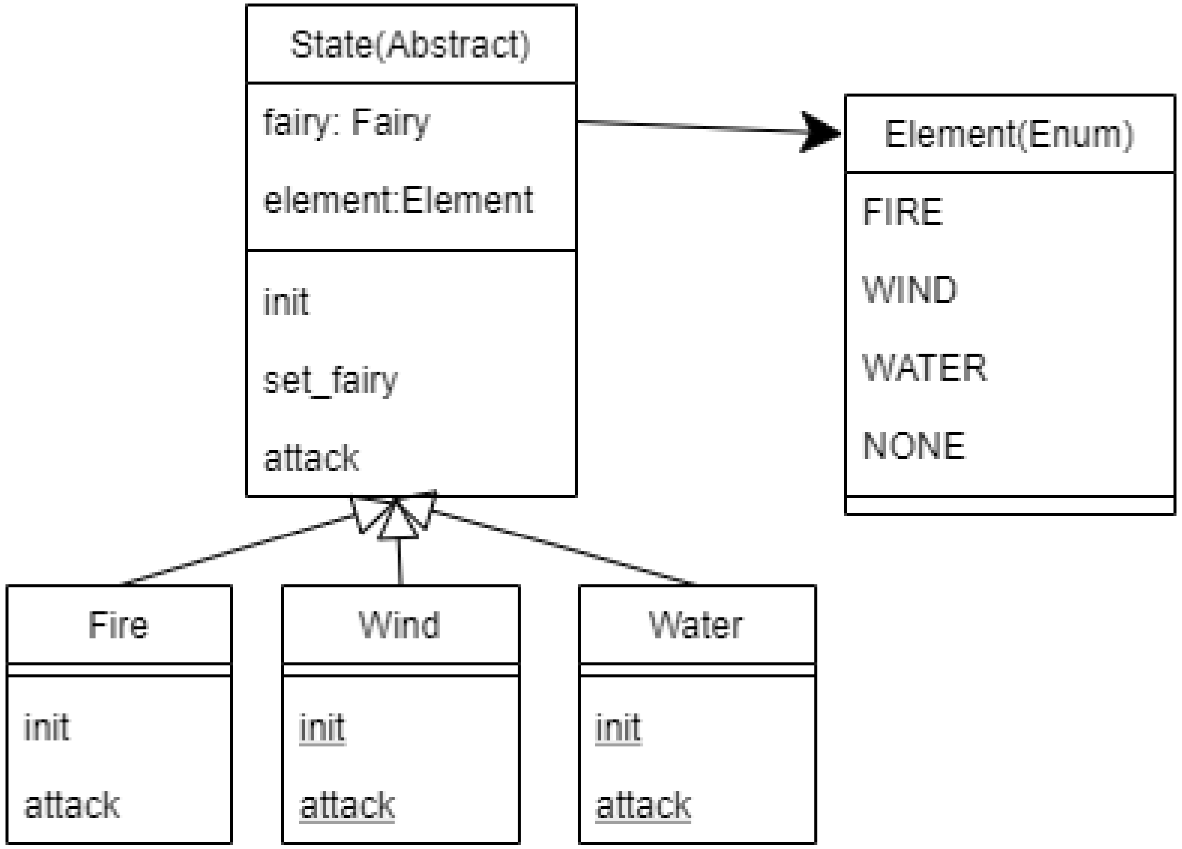
3 sub class states: Fire, Wind, Water

Properties

- **fairy – reference back to context**
- element – helpful enum indicating element. *Not required for pattern*

Methods

- Init
- set_fairy – set reference back to context
- attack



State example

Put it all together

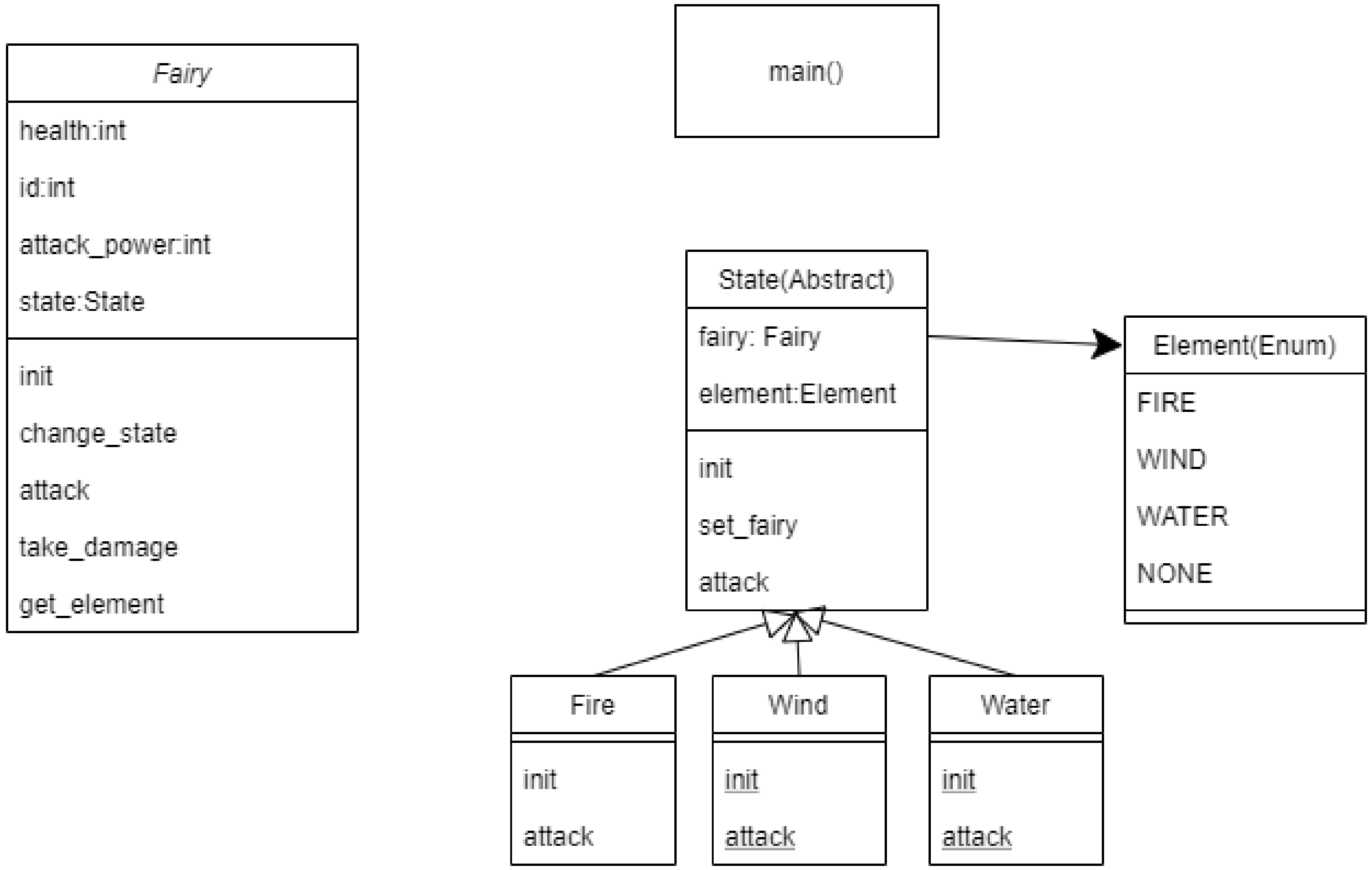
main() is the client

Instantiate Fairy and all the states

Have fairies attack

Change the state of a fairy to another state

Have fairies attack again



State example

Put it all together

main() is the client

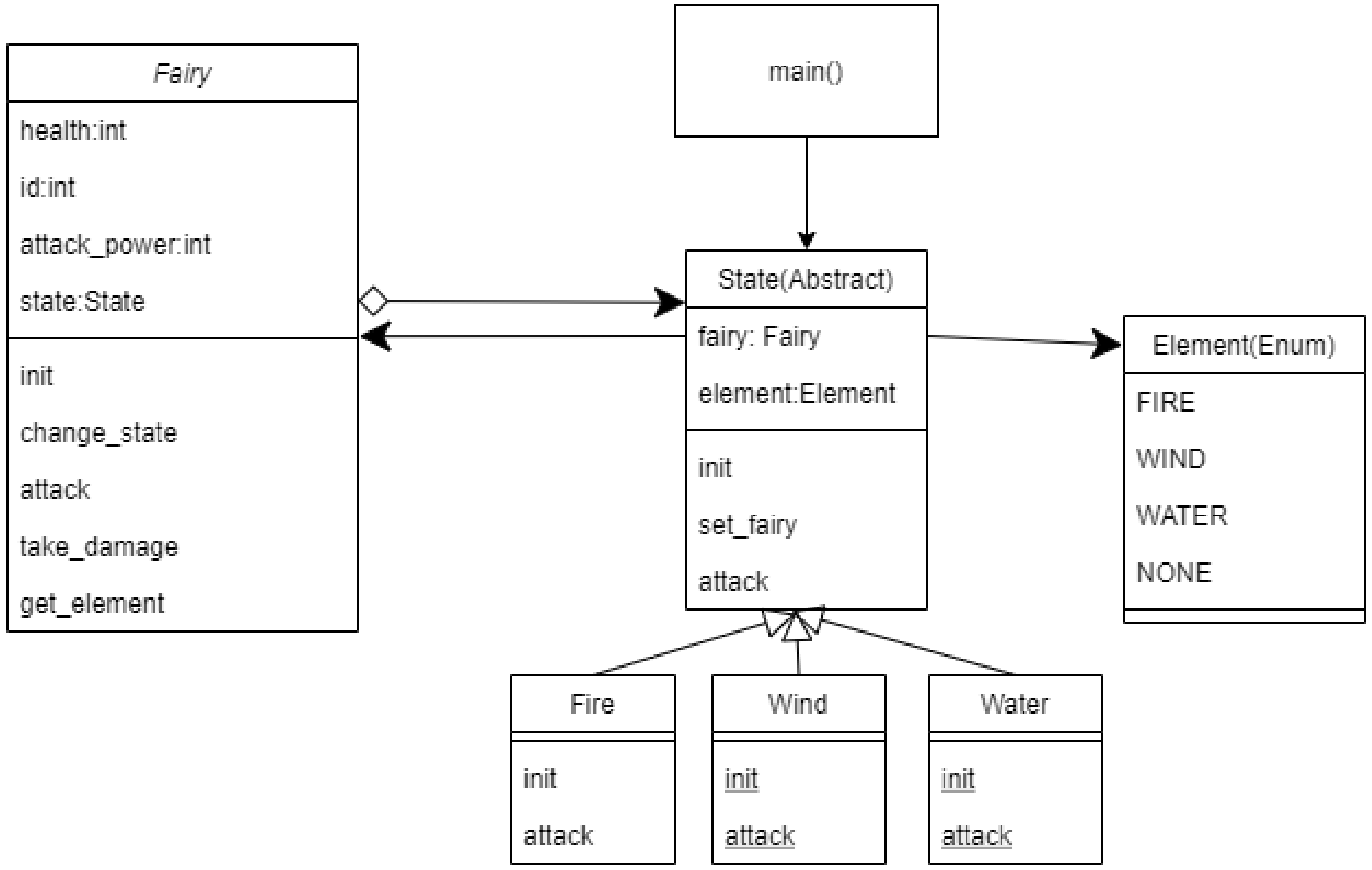
Instantiate Fairy and all the states

Have fairies attack

Change the state of a fairy to another state

Have fairies attack again

fairy_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

Behavioural Patterns

- These are patterns that change the way objects behave and interact.
- They are concerned with algorithms and assignment of responsibilities.

Chain of Responsibility

- Allows you to break down the responsibilities and steps when handling a request.
- Decouple classes that invoke operations from classes that carry out operations.

Summary

Strategy

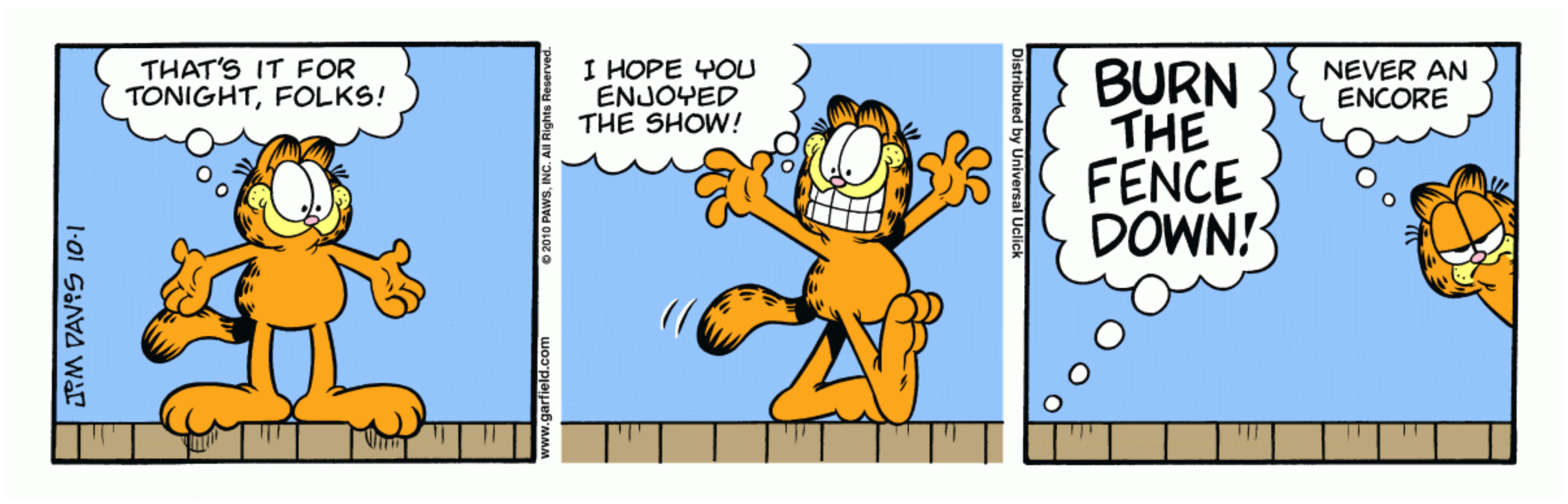
- When you need to switch behaviours at runtime (sounds a lot like State)
- Another way of phrasing this is that it let’s you swap algorithms at run-time.

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.

That’s it for today!

Assignment 2 is out



THAT'S IT FOR TONIGHT, FOLKS!
I HOPE YOU ENJOYED THE SHOW!
BURN THE FENCE DOWN!
NEVER AN ENCORE
JIM DAVIS 10·1
© 2010 PAWS, INC. All Rights Reserved.
www.garfield.com
Distributed by Universal Uclick