COSC 310: DESIGN PATTERNS (3)

Dr. Gema Rodriguez-Perez

University of British Columbia

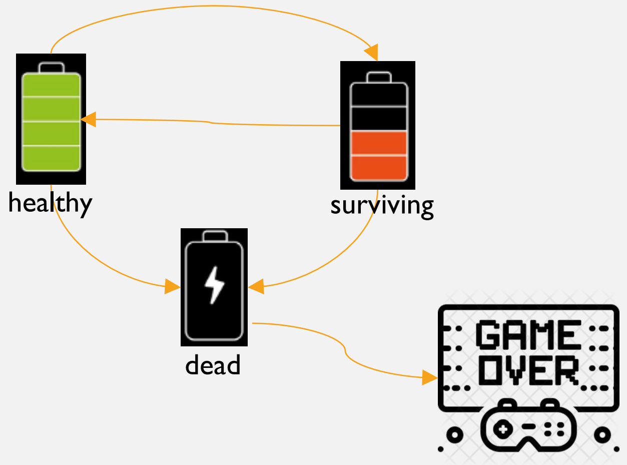
gema.rodriguezperez@ubc.ca

STATE

Behavioral design pattern which deals with the runtime object behaviour based on the current state.

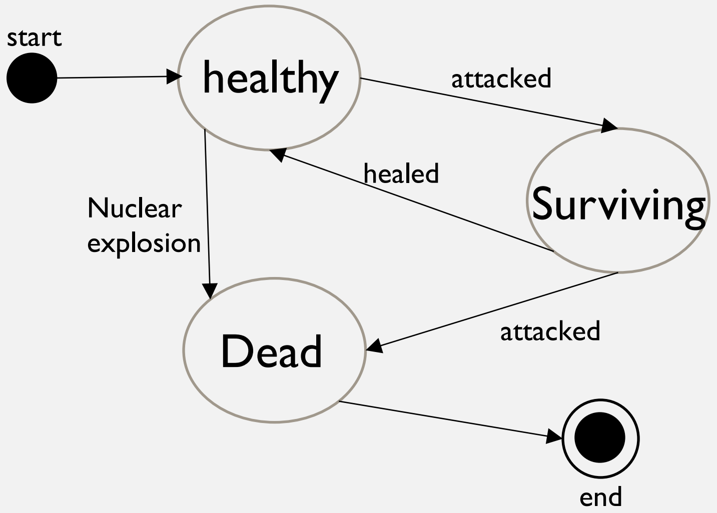
SCENARIO

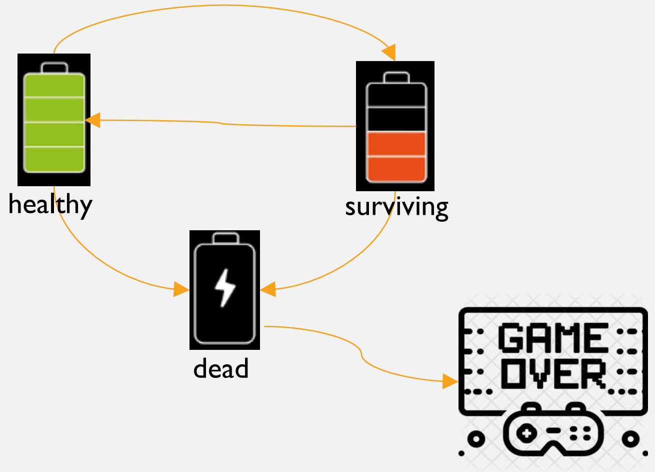
You are implementing a game where a game character can be in different states: healthy, surviving, and dead.



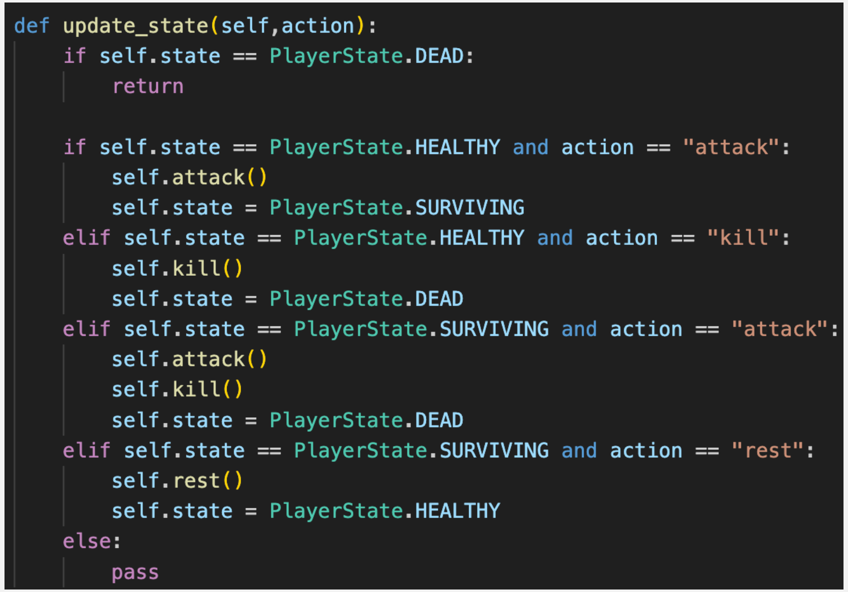
STATE MACHINE

Often used for problems that are natural to define using state machines





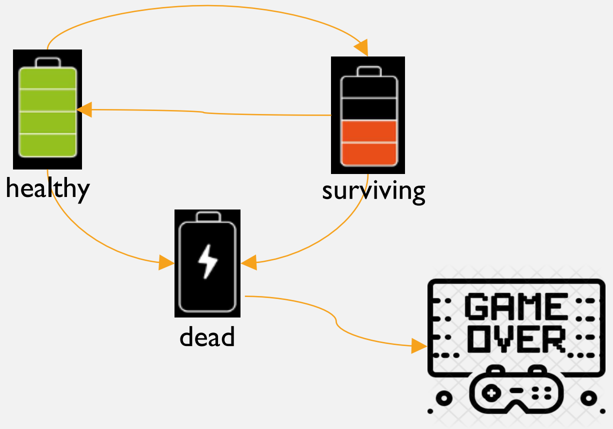
INITIAL IMPLEMENTATION



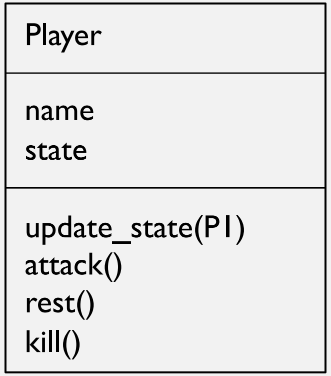
```python
def update_state(self,action):
    if self.state == PlayerState.DEAD:
        return

    if self.state == PlayerState.HEALTHY and action == "attack":
        self.attack()
        self.state = PlayerState.SURVIVING
    elif self.state == PlayerState.HEALTHY and action == "kill":
        self.kill()
        self.state = PlayerState.DEAD
    elif self.state == PlayerState.SURVIVING and action == "attack":
        self.attack()
        self.kill()
        self.state = PlayerState.DEAD
    elif self.state == PlayerState.SURVIVING and action == "rest":
        self.rest()
        self.state = PlayerState.HEALTHY
    else:
        pass
```

- Player manages its transitions
- Methods must **switch** depending on the state



INITIAL IMPLEMENTATION



- Hard to predict all possible states
- State often scattered across fields and parameters
- Code becomes very difficult to maintain as more states are added (need to change conditionals in many methods)
- Hard to add new states
- Large class has many responsibilities
- Relying on concrete implementation and not abstraction

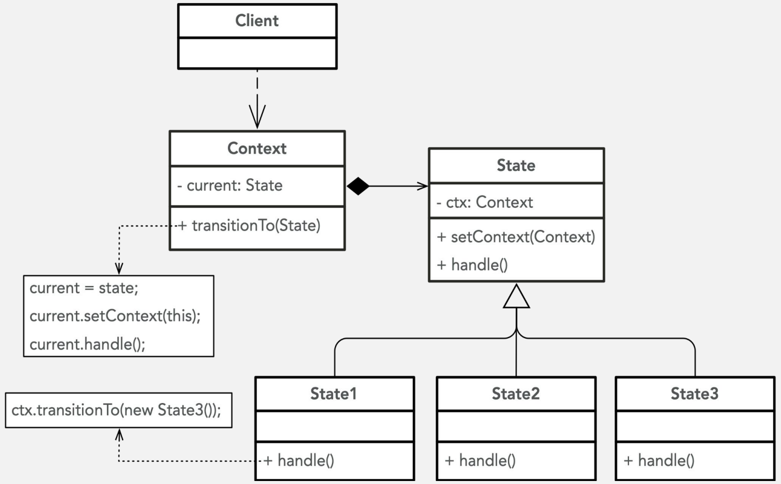
STATE SUGGESTED IMPLEMENTATION

Intent: let an object alter its behavior when its internal state changes. Let it appear as if the object changed its class.

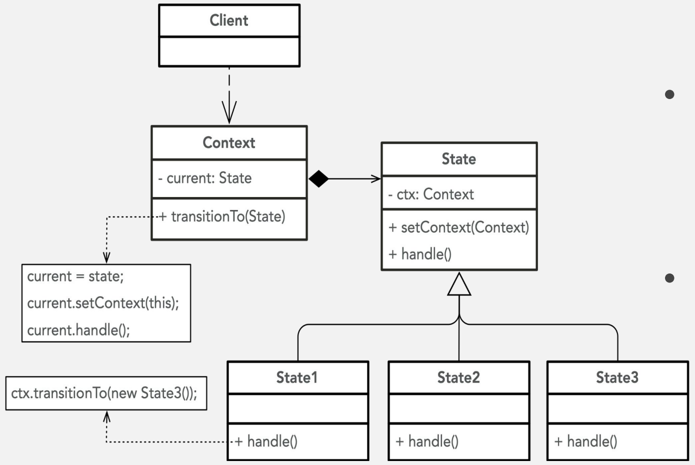
```
StateHealthy created by
 -   StateSurviving

StateSurviving created by
 -   StateHealthy

StateDead created by
 -   StateHealthy
 -   StateSurviving
```



STATE SUGGESTED IMPLEMENTATION



- You create new classes for all possible states of an object and extract all state-specific behaviors into these classes.
- The original object (*context*) stores a reference to one of the state objects that represents its current state, and delegates all the state-related work to that object.
- To transition the context into another state, replace the active state object with another object that represents that new state. All state classes should follow the same interface and the context itself should work with these objects through that interface.

STATE SUGGESTED IMPLEMENTATION

- **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter for passing it a new state object.
- The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will never be called.
- **Concrete States** provide their own implementations for the state-specific methods.
- Both **context and concrete states** can set the next state of the context and perform the actual state transition

EXERCISE

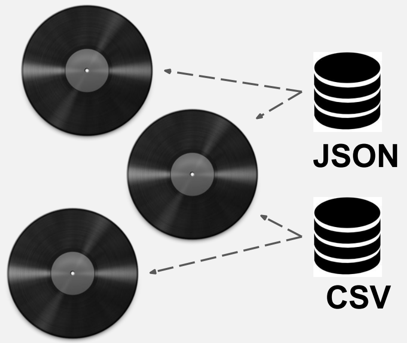
Modify the "**COSC_310_game.py**" file available on Canvas (Modules>Design Patterns Implementations) to implement the **state design pattern.** Define a player entity capable of transitioning between the states: **healthy, surviving, and dead**. The Java version of COSC_310_game.py is also available (Player.java)

ADAPTER

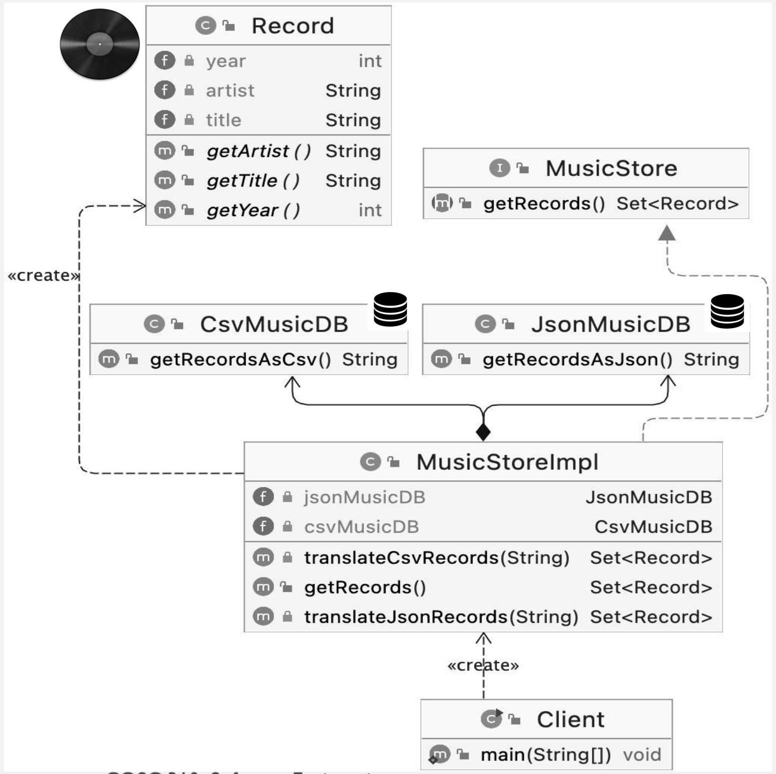
STRUCTURAL PATTER that allows objects with incompatible interfaces to collaborate

SCENARIO

You are implementing a web application for a music store. You already have an interface for returning music record objects and you want to integrate it with data sources that return records as JSON and CSV.



INITIAL IMPLEMENTATION



Problems:

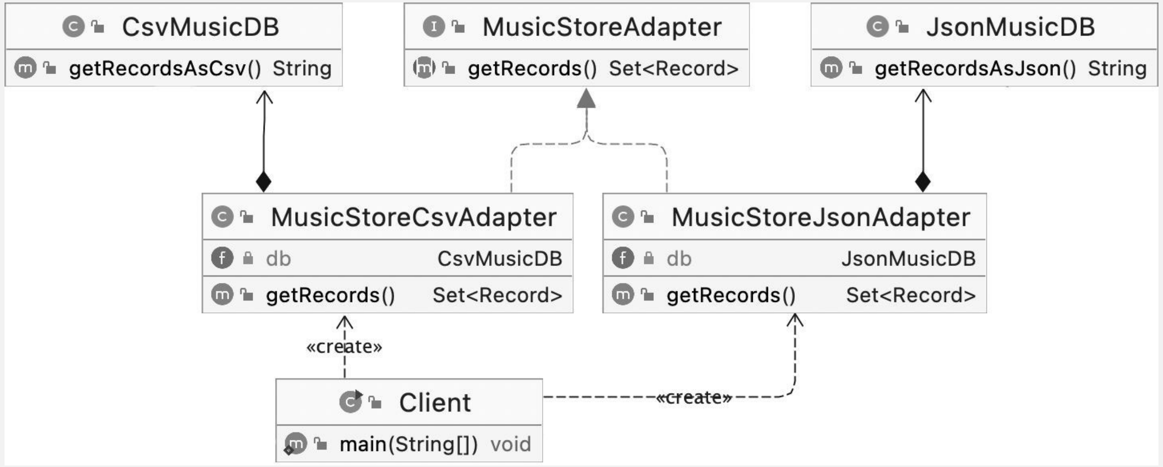
- Multiple **conversions** implemented in one class
- Solution not open for other formats/conversions
- Each additional algorithm makes the code in this class more complex (**divergent changes** and **duplicate code** smells)
- Significant "bloating" over time in a large class

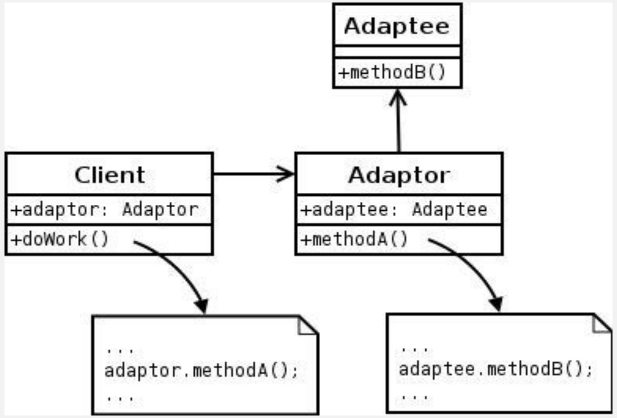
ADAPTER PATTERN APPLICABILITY

- Use the Adapter class when you want to use some existing class, but its interface isn’t compatible with the rest of your code.
- Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can’t be added to the superclass.

ADAPTER SUGGESTED IMPLEMENTATION

Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Wrap an existing class with a new interface.





ADAPTER SUGGESTED IMPLEMENTATION

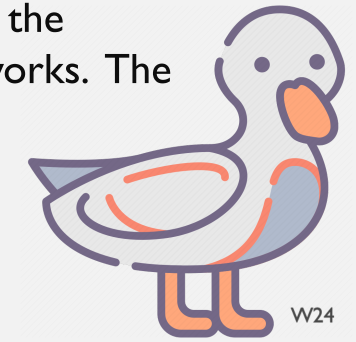
- Make sure that you have at least two classes with incompatible interfaces
- Declare the client interface and describe how clients communicate with the service
- Create the adapter class and make it follow the client interface.
- Add a field to the adapter class to store a reference to the service object.
- One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object.
- Clients should use the adapter via the client interface.

EXERCISE

A Turkey amongst DUCKS: If it walks like a duck and quacks like a duck, then it must be a duck!

Or …. It **might** be a **turkey wrapped with a duck adapter**!

Checkout the "Adapter" available on Canvas and implement the TurkeyAdapter class to make the DuckSimulator program works. The Adapter folder contains the Python and Java versions



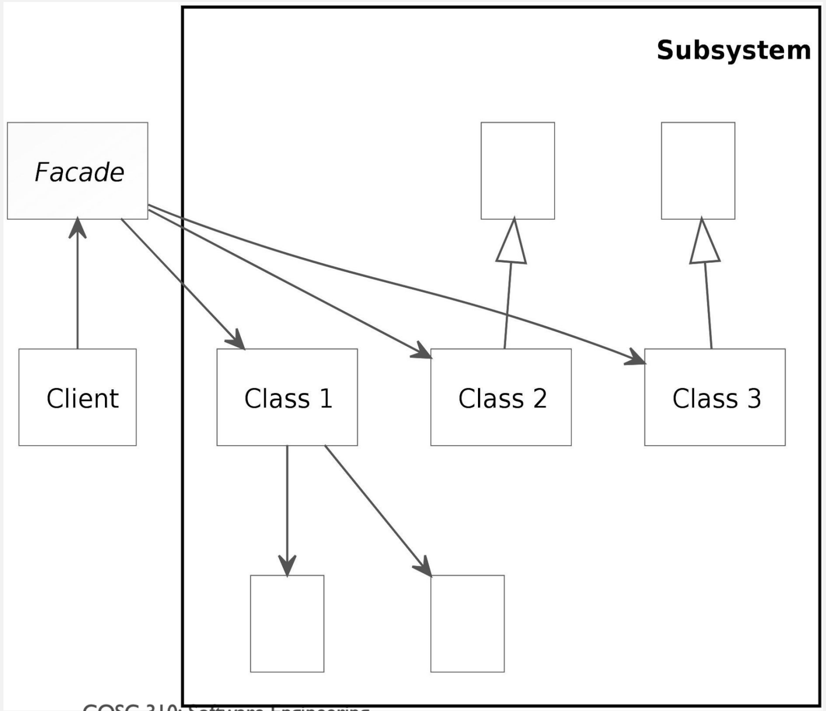
FACADE

STRUCTURAL PATTERN that provides a simplified interface to a library, a framework, or any other complex set of classes.

SCENARIO

You are implementing a query engine that lets clients add a dataset from an archive, remove a dataset, and perform complex queries on the dataset. Your code base has grown significantly and you want to provide a simplified interface to clients.

FACADE PATTERN



- Useful when you need a simple interface to a complex subsystem
- You want to avoid tight coupling of your implementation to the complex subsystem

FACADE SUGGESTED IMPLEMENTATION

- Check whether it’s possible to provide a simpler interface than what an existing subsystem already provides.
- Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem.
- The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.
- To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade.