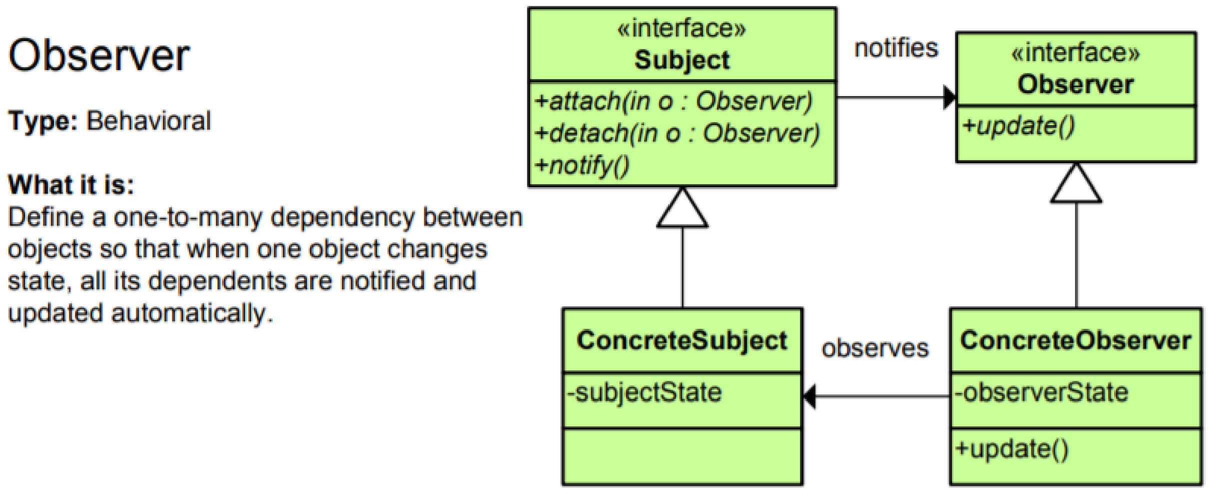
Design Patterns with SQLAlchemy

In this tutorial students would be exposed to how traditional design pattern theory can be translated for application in web frameworks and how they interplay with model design when using ORMS such as SQLAlchemy

Starter Repl

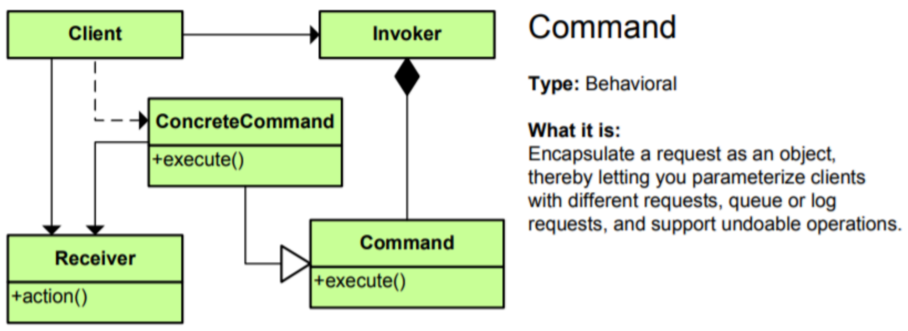
Observer



This pattern is very useful in message broadcasting applications where you want to maintain user inboxes. The inbox can be the individual subscriber's state which is updated when a message is broadcasted. This state will take the form of a bridge table linking the publisher and the subject.

Example https://replit.com/@Snickdx/DP-in-FlaskSQLalchemy-Completed#wsgi.py

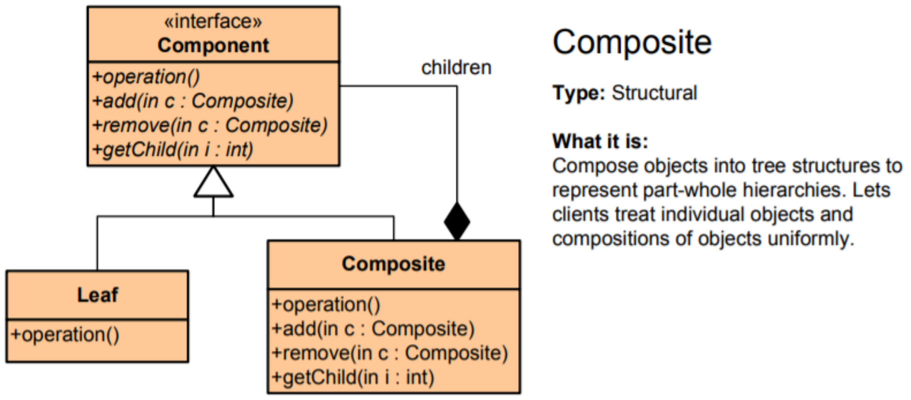
Command


Client
Invoker
ConcreteCommand
+execute()
Receiver
+action()
Command
+execute()
Command
Type: Behavioral
What it is:
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

The command pattern is useful when you wish to have an event store/transaction log which can give a historical account on the current state of an object. Eg for a bank each transaction may be stored as an event so the balance at some time T can always be calculated.

https://replit.com/@Snickdx/DP-in-FlaskSQLalchemy-Command#wsgi.py

Composite



Composite

Type: Structural

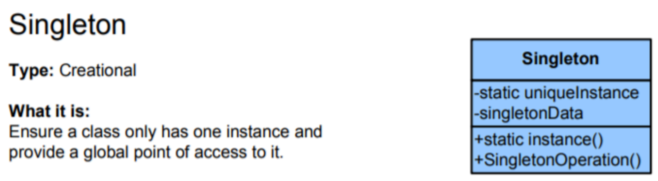
What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

In FlaskSQLAlchemy the composite pattern is already natively implemented via foreign keys and relationship fields. Relationship fields roll up all related objects in a property of another object (composes). An often secondary goal of the pattern is to allow for the traversal of recursive composite structures.

A recursive hierarchy can be achieved via self joins where a table has a relationship with itself. ie Employee managing other employees so each employee has a FK to their manager.

Example: https://github.com/Comp3613-SolitaireEngineers/COMP3613-UWI-Research-Assistant

Singleton



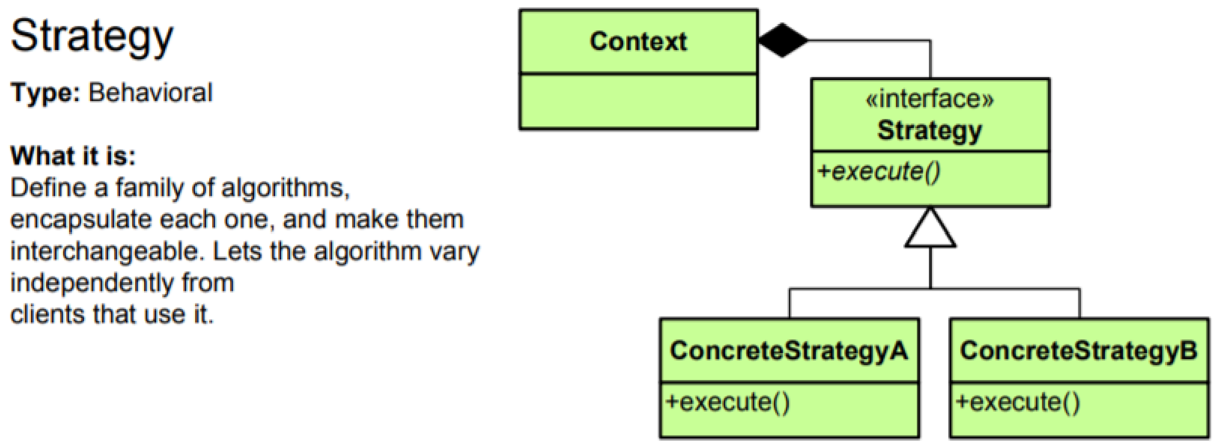
Singleton

Type: Creational

What it is:
Ensure a class only has one instance and provide a global point of access to it.

It is very common in web applications to have access of a single database object across the codebase. Flask apps are no different by instantiating SQLAlchemy once with db then importing across the app; this is a valid implementation of the singleton pattern even without creating a formal class.

Strategy


Strategy

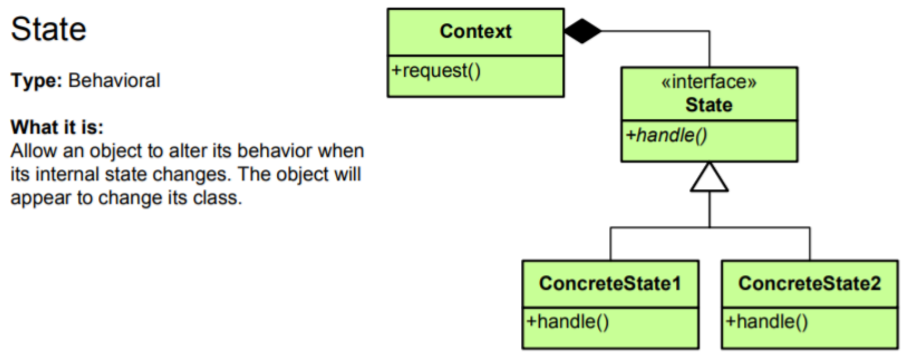
Type: Behavioral

What it is:
Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.

In the example below while strategy is implemented note how these classes do not store state and are therefore not sqlalchemy models. The benefit of strategy is allowing the extension and execution of algorithms without changing existing code or affecting how the client invokes it.

https://replit.com/@Snickdx/DP-in-FlaskSQLalchemy#models.py

State


State

Type: Behavioral

What it is:
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

The same can apply with the state. While strategy is about swapping out algorithms that get a solution differently, state allows the developer to represent how an object's behavior can vary depending on what state the object is in.