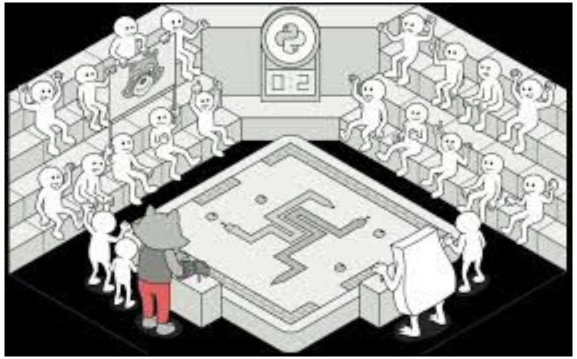
COMS W4156 Advanced Software Engineering (ASE)

November 17, 2022

Agenda: Design Patterns

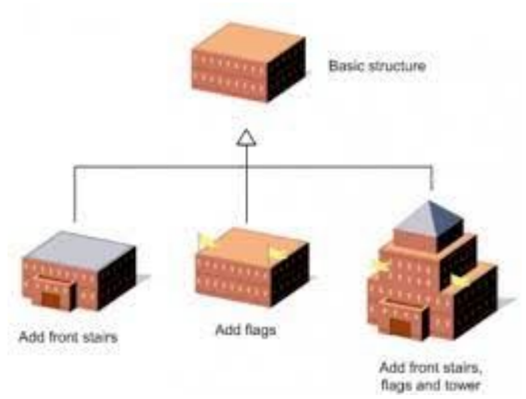
1. Overview
2. Observer
3. Mediator
4. Facade
5. Adapter
6. Strategy



Design Patterns

Design patterns are general reusable solutions to commonly occurring problems, typically defined by a description or template rather than a library or framework with reusable code

Finer granularity than architectural styles like client/server, instead operating at the class and method level



Gang of Four book

Initial set of 23 design patterns presented in GoF book, heavily influenced by Christopher Alexander's architectural patterns for buildings and cities

The book gives examples in C++ and Smalltalk,
but equally relevant to Java and other OO languages

See "15 years later" interview with GoF authors here

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Agenda: Design Patterns

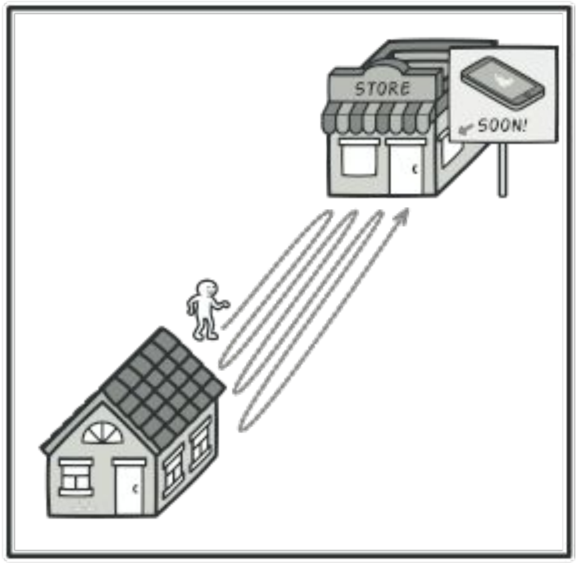
1. Overview
2. Observer
3. Mediator
4. Facade
5. Adapter
6. Strategy

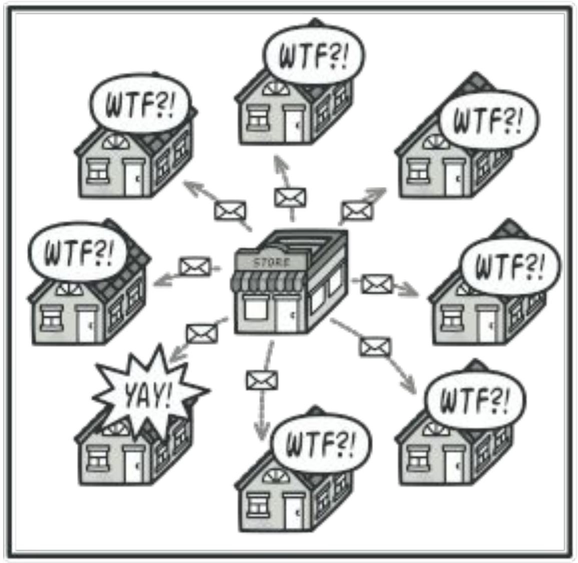
Observer Pattern

Behavioral design pattern that lets you define a subscription mechanism to notify multiple objects (called subscribers) about any events that happen to the object they're observing (called publisher or subject)

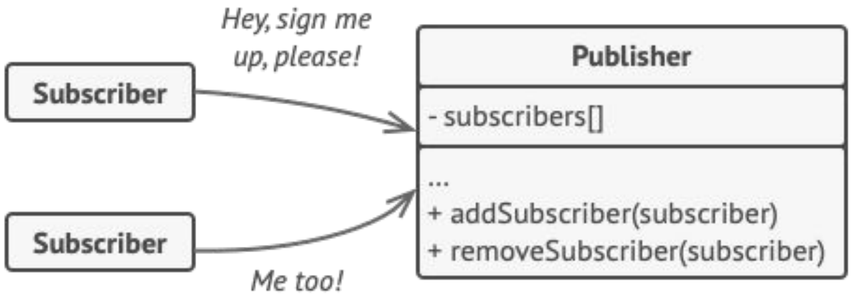
Behavioral patterns realize common communication paradigms and the assignment of responsibilities between objects

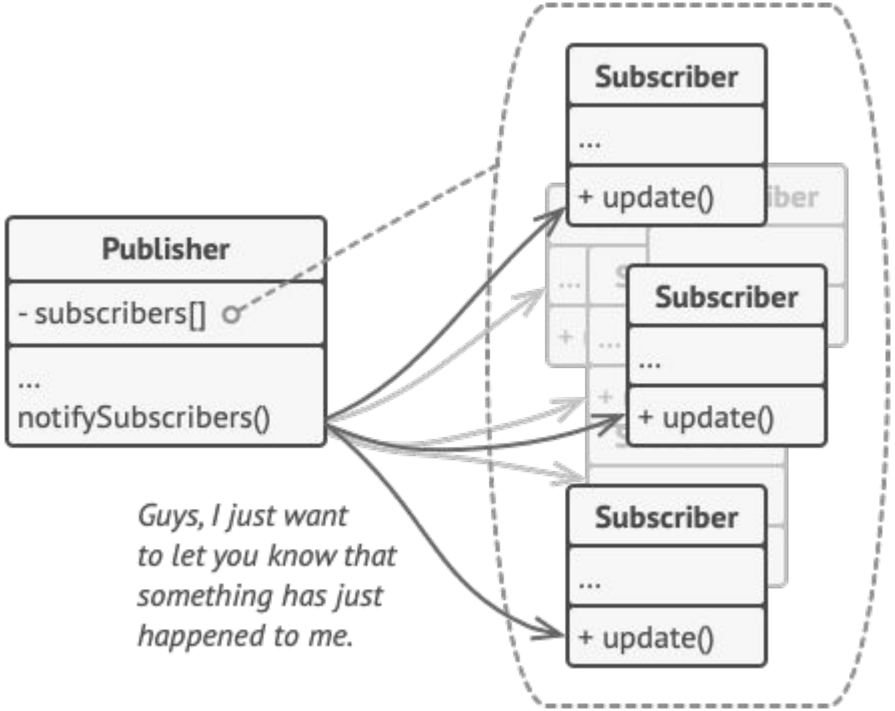
Problem that Observer Addresses



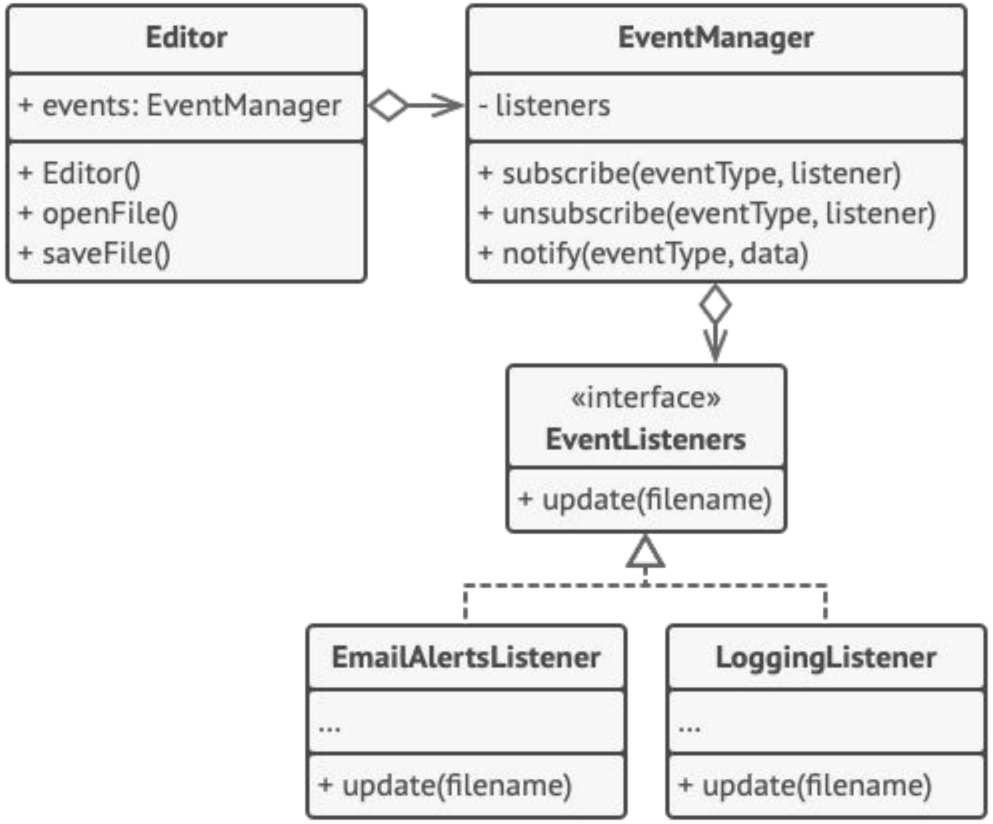


Observer Solution





Example Observer



Note: Diagrams on this page and previous page (and upcoming pages) are UML class diagrams. This course does not cover UML but it's very easy to learn the frequently used kinds of models on your own (explanation of UML arrows)

When to Use Observer

Use when changes to the state of one object may require changing other objects, and the actual set of objects is unknown beforehand or changes dynamically - The Observer pattern lets any object that implements the subscriber interface subscribe for event notifications in publisher objects, e.g., to hook up custom code (callbacks) via custom subscriber classes

Use when some objects in your program must observe others, but only for a limited time or in specific cases - the subscription list is dynamic, so subscribers can join or leave the list whenever they need to

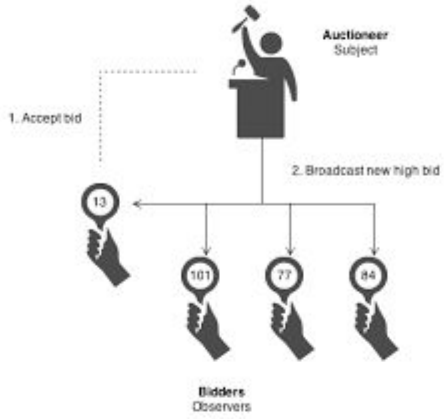
Refactoring to use Observer

1. Break your business logic into two parts: core functionality will act as the publisher, the rest will turn into a set of subscriber classes
2. Declare subscriber interface with, at least, an update method that takes “context” as parameter(s) - perhaps a reference to the publisher itself so subscribers can ask it more specifically for what they want when they are notified
3. Declare publisher interface with a pair of methods for adding and removing a subscriber object
4. Decide where to put the actual subscription list and subscription methods, maybe an abstract class extended by concrete publishers or a separate object referenced by concrete publishers (composition)
5. Create concrete publisher classes that notify all their subscribers whenever something important happens
6. Implement update notification methods in concrete subscriber classes and code to register and deregister with publishers as needed

Observer Pattern Code Examples

[C++]

[Java]



Agenda: Design Patterns

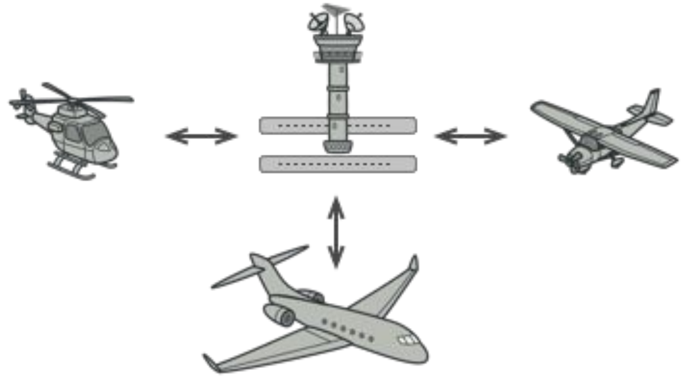
1. Overview
2. Observer
3. Mediator
4. Facade
5. Adapter
6. Strategy

Mediator Pattern vs. Observer Pattern

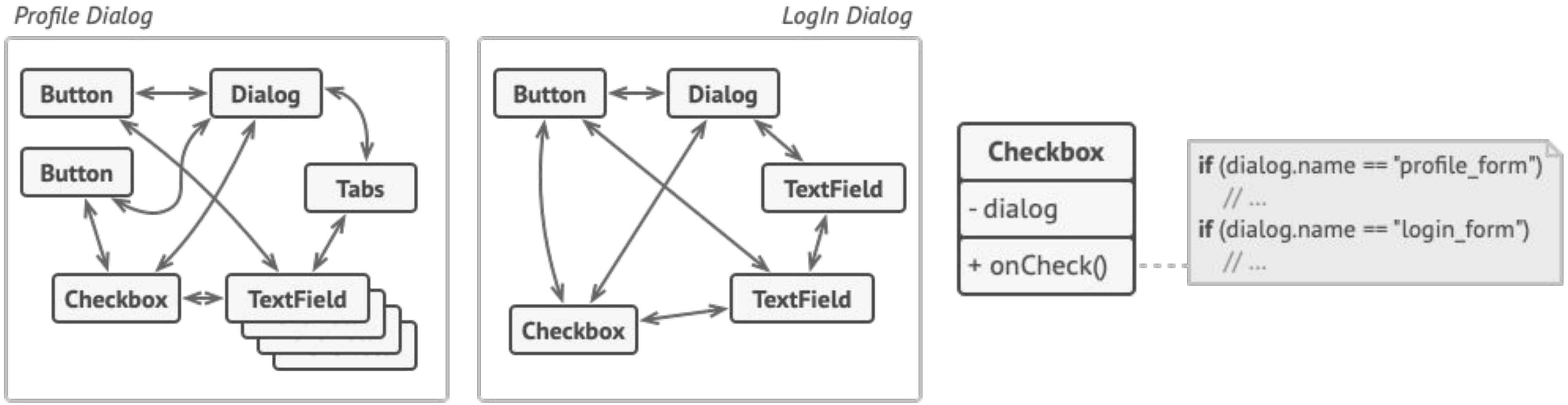
Behavioral design pattern that lets you eliminate mutual (two-way) dependencies among objects, restricts direct communications between the objects and forces them to instead depend only on a single mediator object

In contrast, observer dependencies are dynamic and one-way

Could use observer to implement mediator

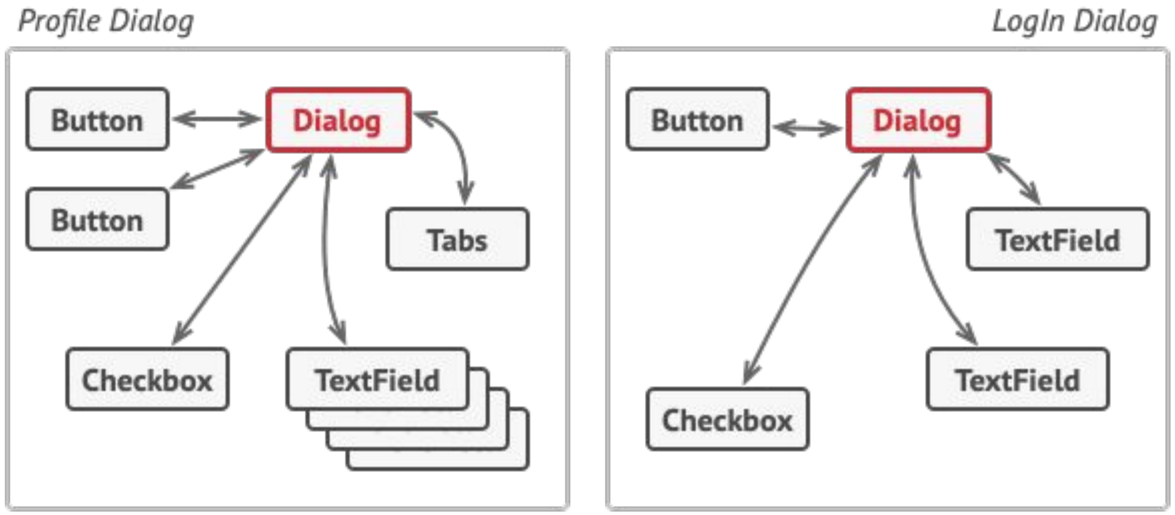


Problem that Mediator Addresses

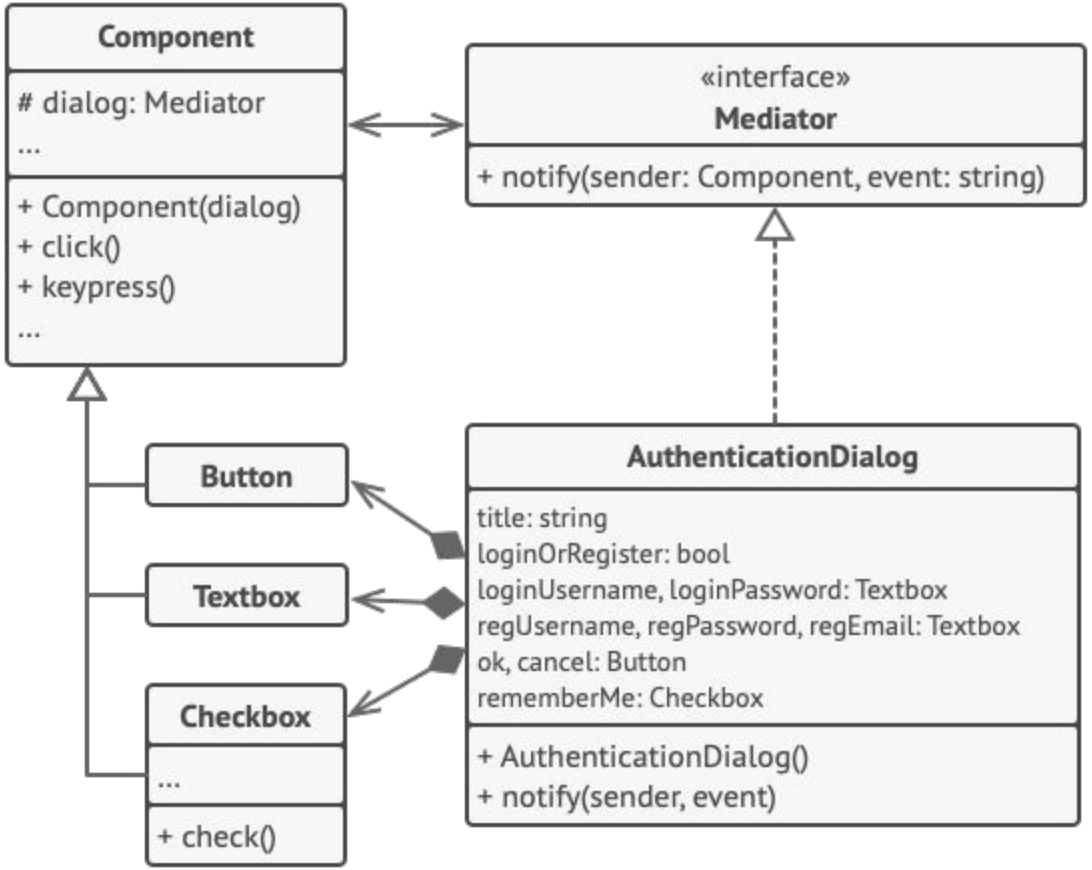


```
if (dialog.name == "profile_form")
    // ...
if (dialog.name == "login_form")
    // ...
```

Mediator Solution



Mediator Example



When to Use Mediator

Use when some classes are tightly coupled to several other classes, so they are hard to change - the Mediator pattern lets you extract all the relationships between classes into a separate class, isolating any changes to a specific component

Use when you can't reuse a component in a different program because it's too dependent on your other components - After you apply Mediator, individual components become unaware of the other components so can reuse in a different app by providing a new mediator class

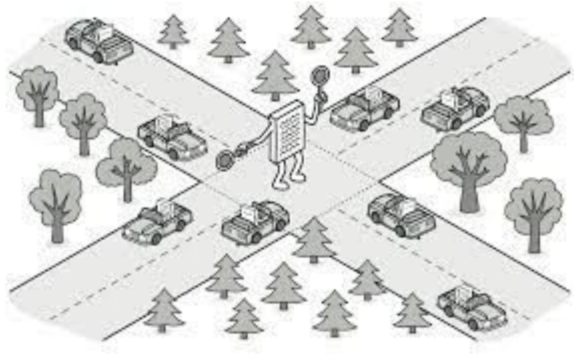
Refactoring to Use Mediator

1. Identify a group of tightly coupled classes
2. Declare a mediator interface with the desired communication protocol between mediators and those classes, in many cases a single method for receiving notifications is sufficient
3. Implement a concrete mediator class, which should store references to all components it manages
4. Components should also store a reference to the mediator object, e.g., pass the mediator object as an argument to the component’s constructor
5. Change the components’ code so that they call the mediator’s notification method instead of methods on other components
6. Extract the code that involves calling other components into the mediator class, and execute this code whenever the mediator receives notifications from that component

Mediator Pattern Code Examples

C++

Java



Agenda: Design Patterns

1. Overview
2. Observer
3. Mediator
4. Facade
5. Adapter
6. Strategy

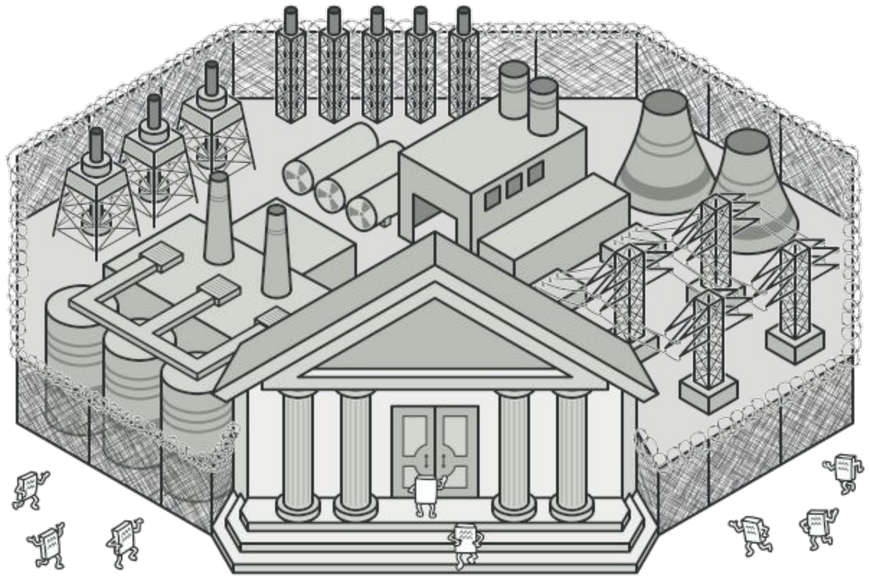
Facade Pattern is Like a Mediator for a Subsystem

Facade is a structural design pattern that provides a simplified interface to a library, framework, or other complex set of classes

Structural patterns explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient

Facade and Mediator have similar jobs: they try to organize collaboration between lots of tightly coupled classes

Facade defines a simplified interface to a subsystem of objects, but objects *within* the subsystem can still communicate directly



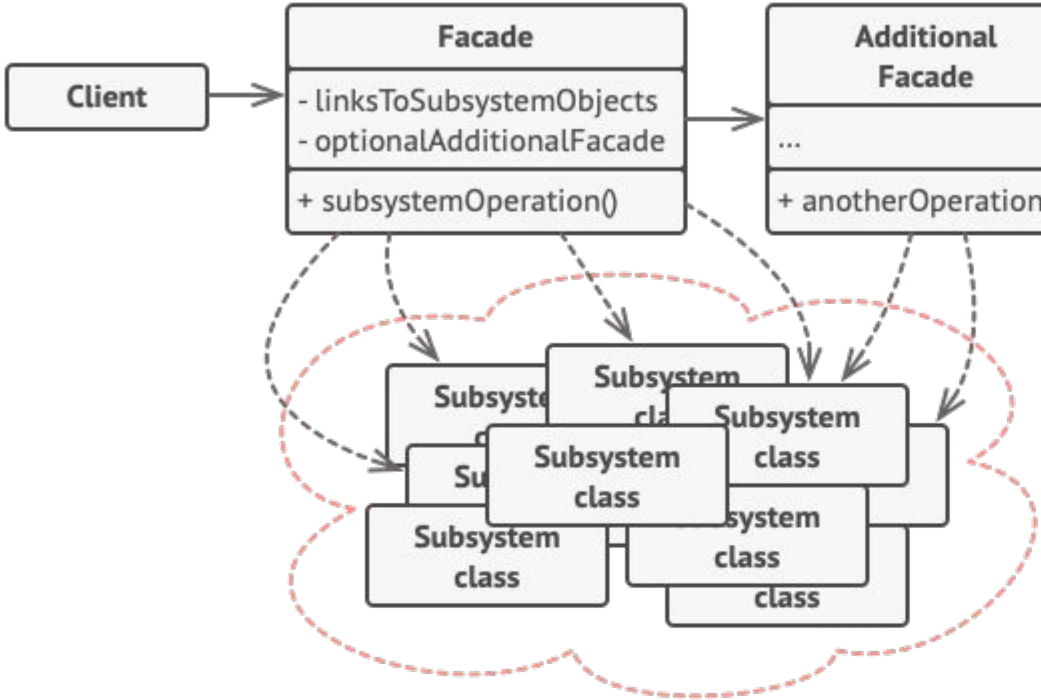
Problem that Facade Addresses

Imagine your code needs to work with a *subset* of some fancy third-party library or framework, you’re not a power-user who needs full control of “everything”

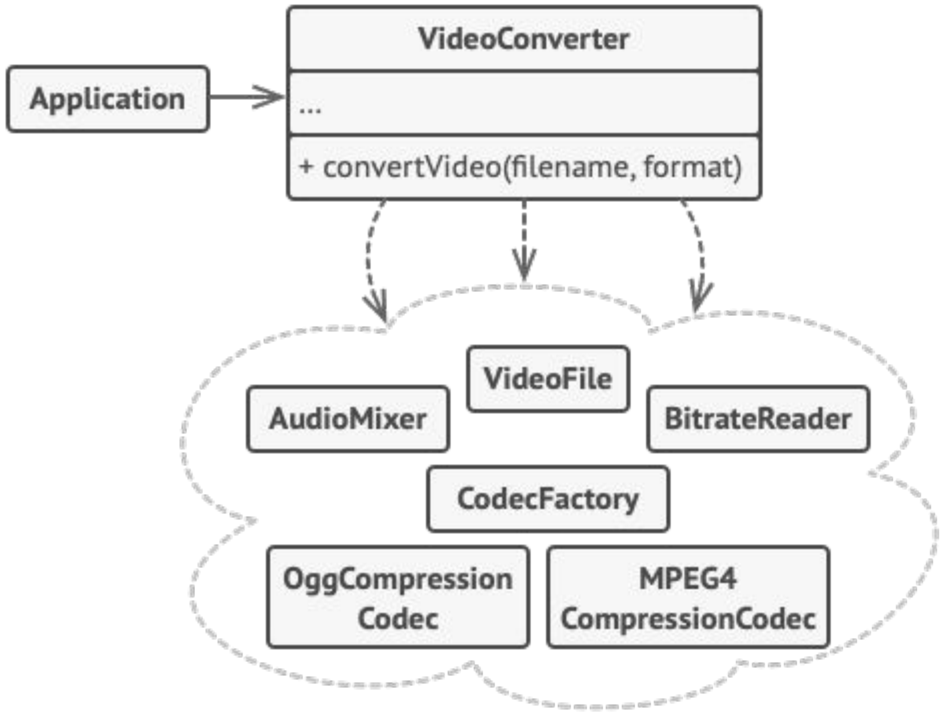
You’d still need to initialize all the third-party objects you use (and possibly others you don’t), keep track of dependencies among those objects, execute method calls on different objects in the correct order, supply data in proper format that might be different for different objects, and so on

Your business logic would become tightly coupled to the implementation details of 3rd-party classes

Facade Solution



Facade Example



When to Use Facade

Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem

Notice the Facade pattern is presented as the developer adding a facade to *someone else's* library or framework, but alternatively the subsystem could come with multiple facades to choose from

Similarly to applying Mediator, your components become unaware of the individual components within the subsystem so can replace with a different subsystem implementing the same interface

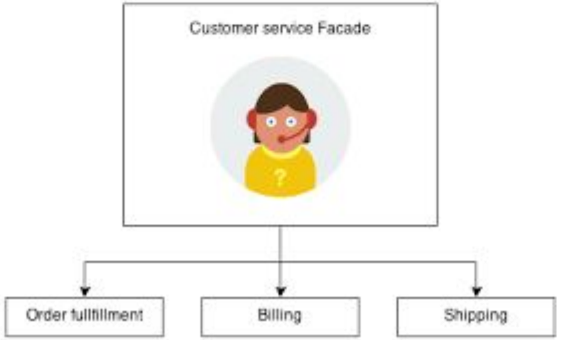
Refactoring to use Facade

1. Check whether it’s possible to provide a simpler interface than what an existing subsystem already provides - you’re on the right track if this interface makes the client code independent from many of the subsystem’s classes.
2. 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, and is responsible for initializing the subsystem and managing its life cycle
3. To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade, protecting the client from any changes in the subsystem code
4. If the facade becomes too big, consider extracting part of its behavior to a new, refined facade class

Facade Pattern Code Examples

C++

Java



Agenda: Design Patterns

1. Overview
2. Observer
3. Mediator
4. Facade
5. Adapter
6. Strategy

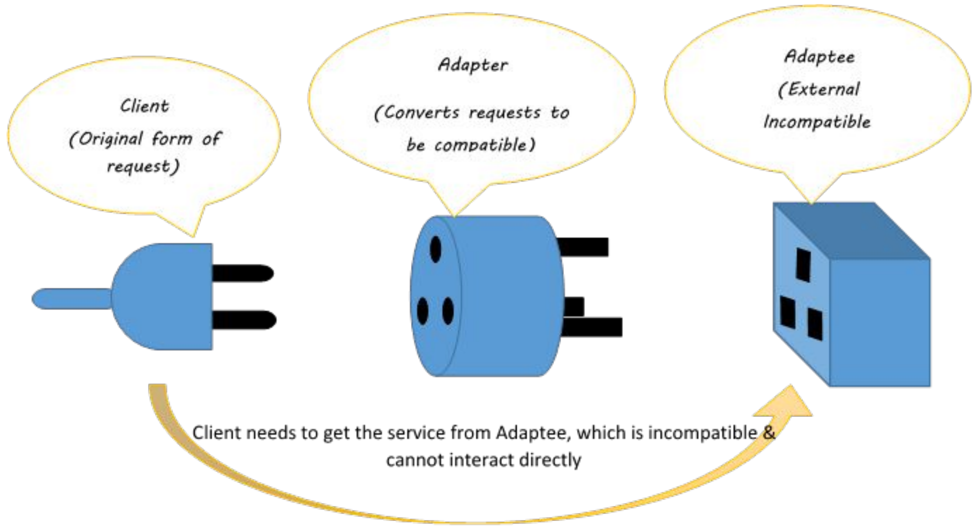
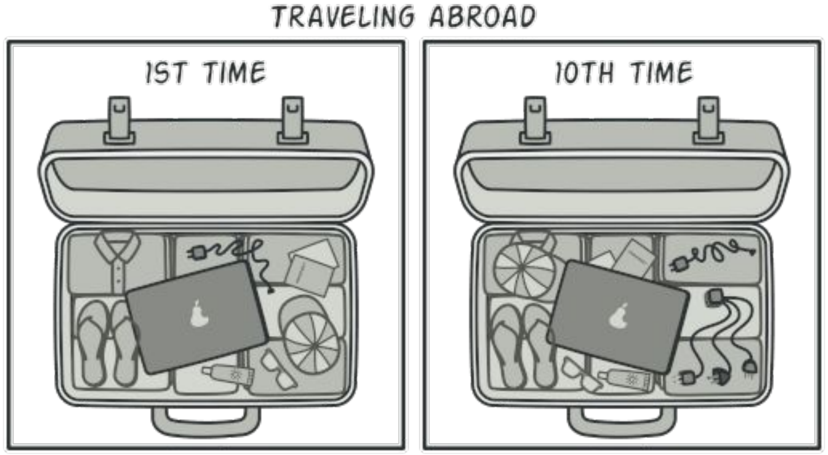

Client
(Original form of request)
Adapter
(Converts requests to be compatible)
Adaptee
(External Incompatible
Client needs to get the service from Adaptee, which is incompatible & cannot interact directly

Figure 1-Adapter Pattern Concept

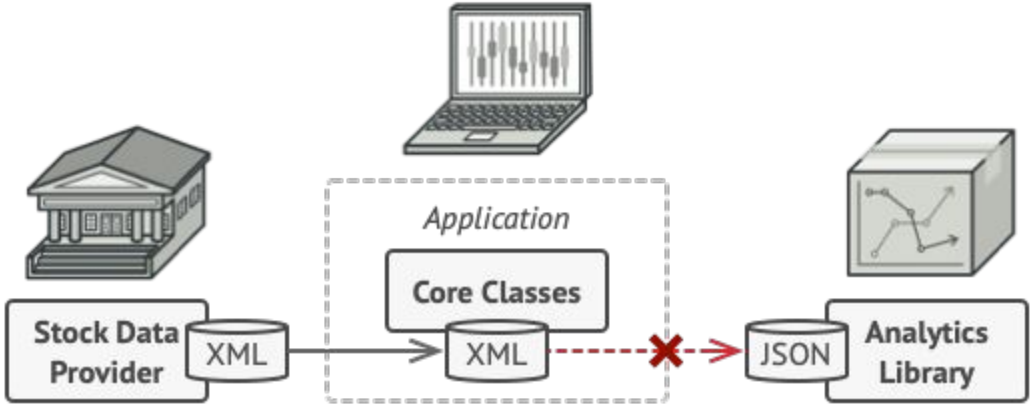
Adapter Pattern Converts Interface instead of Simplifying

Structural design pattern that allows objects with incompatible interfaces to collaborate

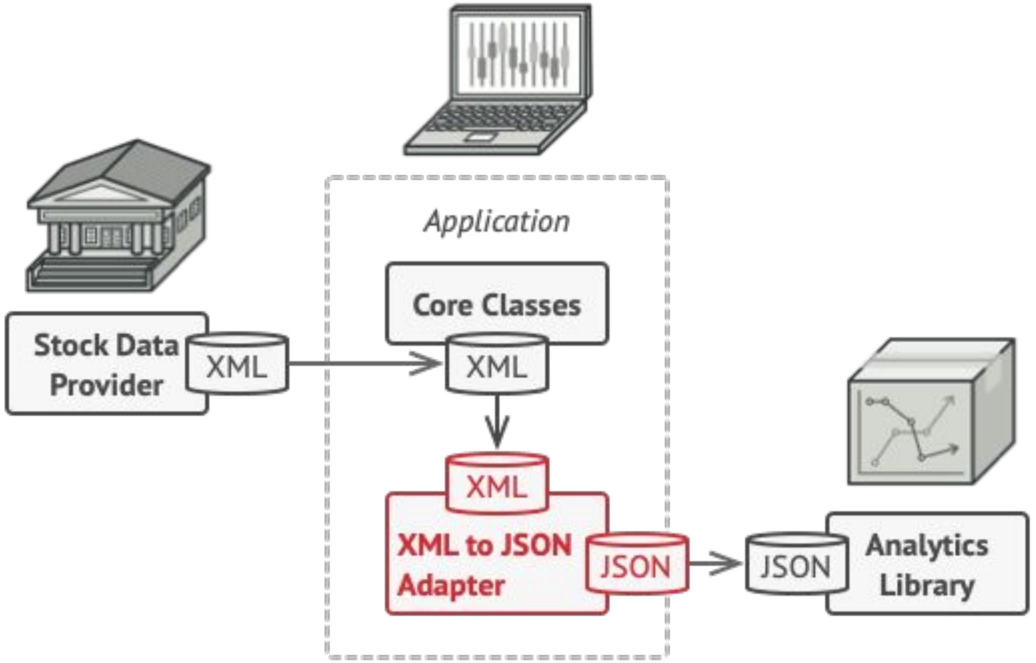
Adapter (often called wrapper) usually wraps one class of objects or literally one singleton object, not a whole subsystem like Facade



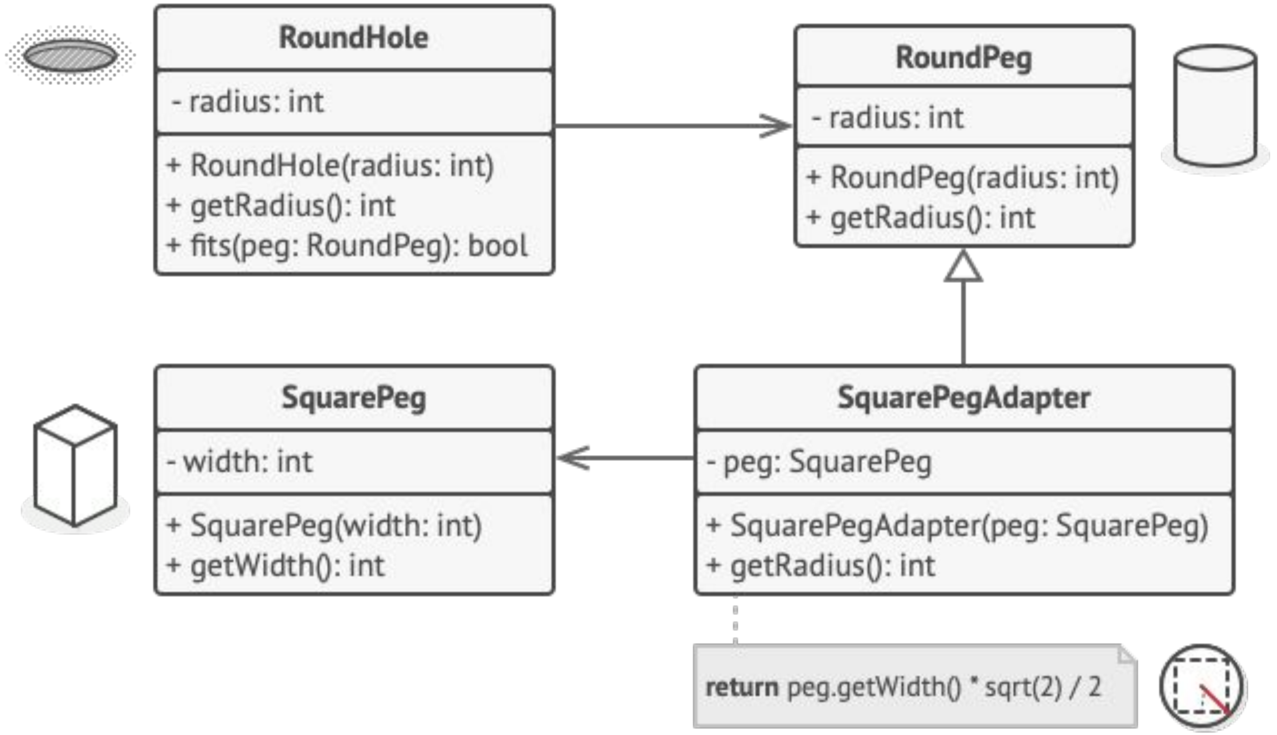
Problem that Adapter Addresses



Adapter Solution



Adapter Example



When to Use Adapter

Use when you want to use some existing class, but its interface isn't compatible with the rest of your code - the Adapter pattern creates a middle-layer class to translate between your code and a legacy class, 3rd-party class or any other class that cannot be changed

Another application is when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass - rather than duplicate the missing functionality in the child classes, wrap inside an Adapter that implements that functionality once

Refactoring to Use Adapter

1. Declare the client interface for how clients should communicate with the service
2. Create an adapter class that follows the client interface, initially with empty methods
3. Add a field to the adapter class to store a reference to the service object and initialize this field via the adapter's constructor (alternatively could pass the relevant service object to the adapter as an argument to each method)
4. Implement all methods of the client interface in the adapter class, handling only the interface or data format conversion and delegating most of the real work to the service object
5. Clients then use the adapter via the client interface

Adapter Pattern Code Examples

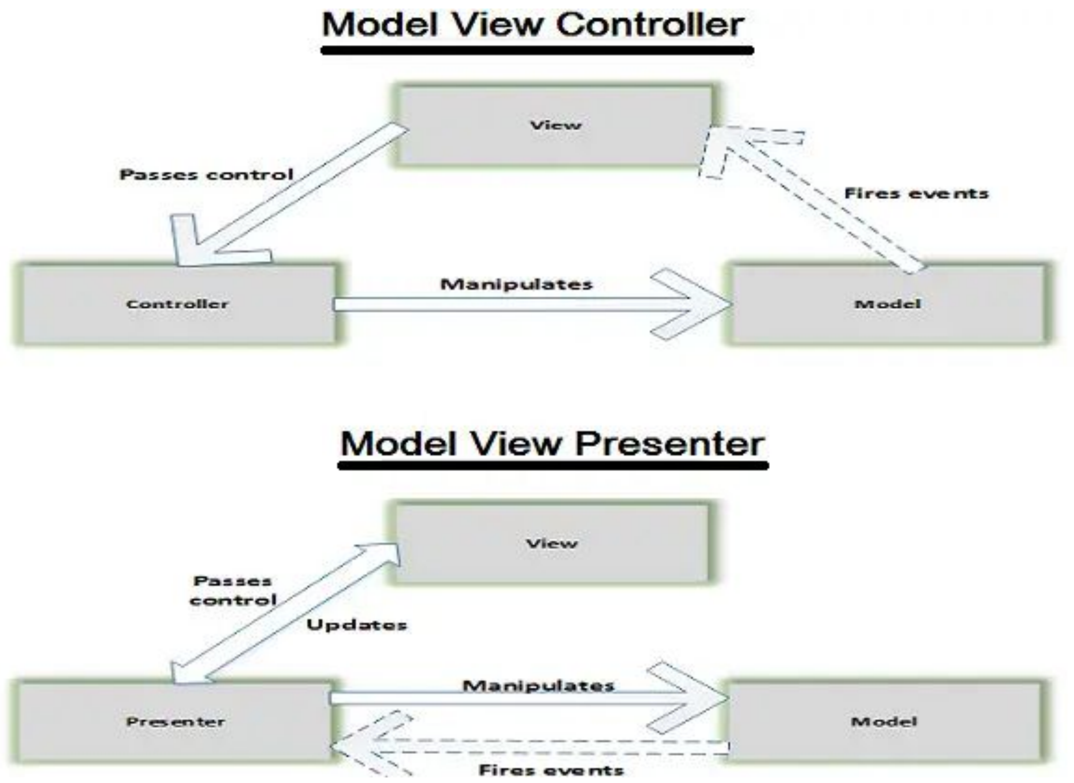
C++

Java

Agenda: Design Patterns

1. Overview
2. Observer
3. Mediator
4. Facade
5. Adapter
6. Strategy

Reprise: Is MVC truly an architecture or is it a design pattern?



MVC - uses Observer Pattern to passively update view

MVP - uses Observer Pattern to passively update presenter, which in turn actively updates the view

(There are other variants)

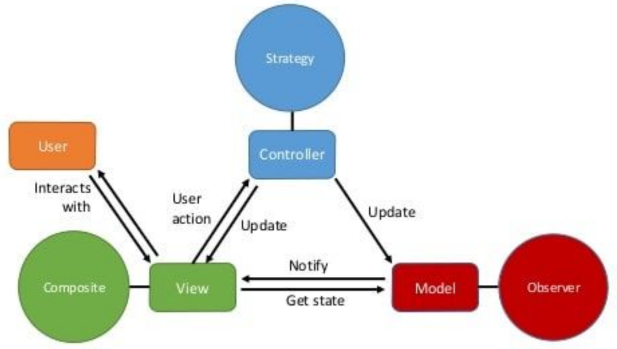
MVC/P is an Architecture <u>and</u> a *Compound* Design Pattern

Observer Pattern + Strategy Pattern

Observer Pattern (ultimately) updates the view to display the model, possibly multiple different views for the same model, e.g., desktop vs. mobile browser

Strategy Pattern supports different ways ("strategies") to leverage the model, e.g., multiple apps using the same service

Traditional MVC & associated design patterns

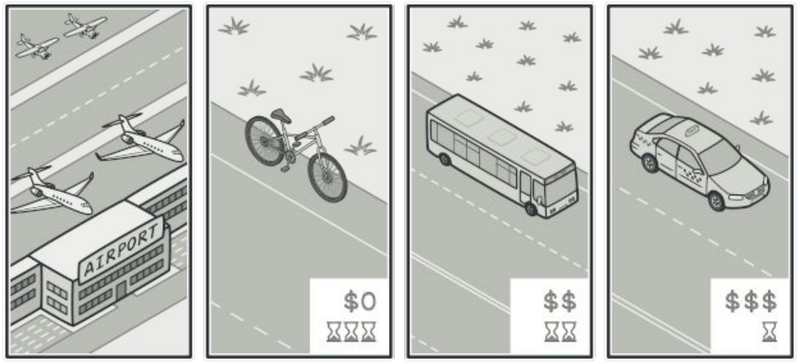


1. R. Buencamino: Learning the Model-View-Controller Design Patterns in iOS. Lynda.com, 2015

What’s the Strategy Pattern?

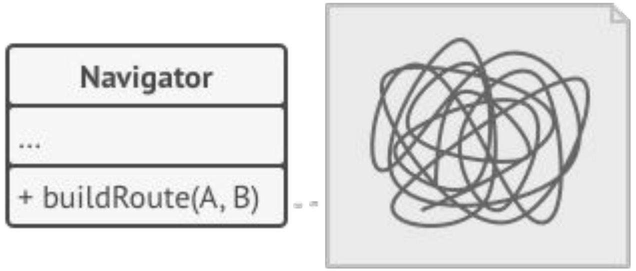
Behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable

May allow swapping algorithms at runtime - e.g., in MVC/P an administrator might switch on the fly been privileged and regular user interfaces with different controller/presenter not just different view (e.g., mice fake student)

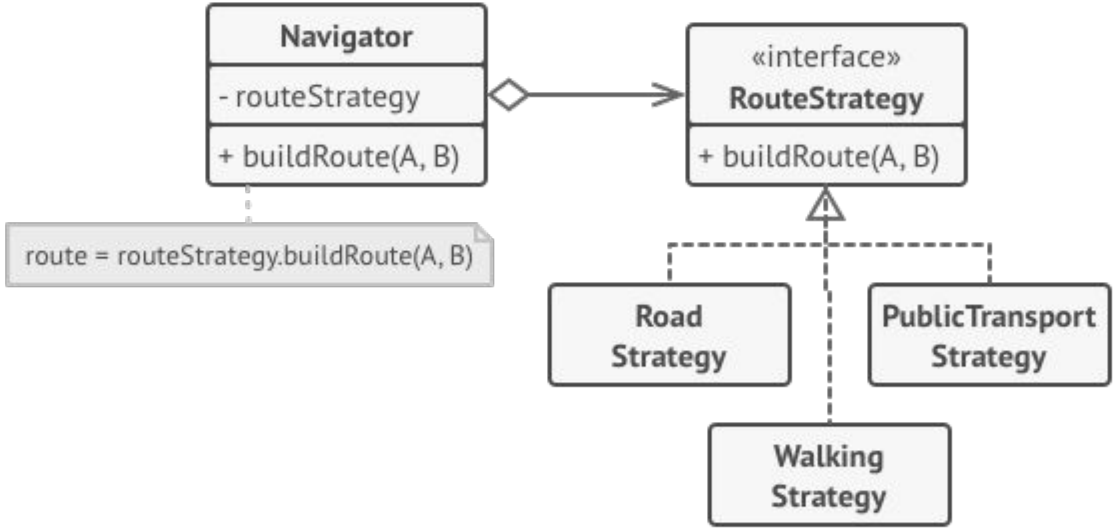


Problem that Strategy Addresses

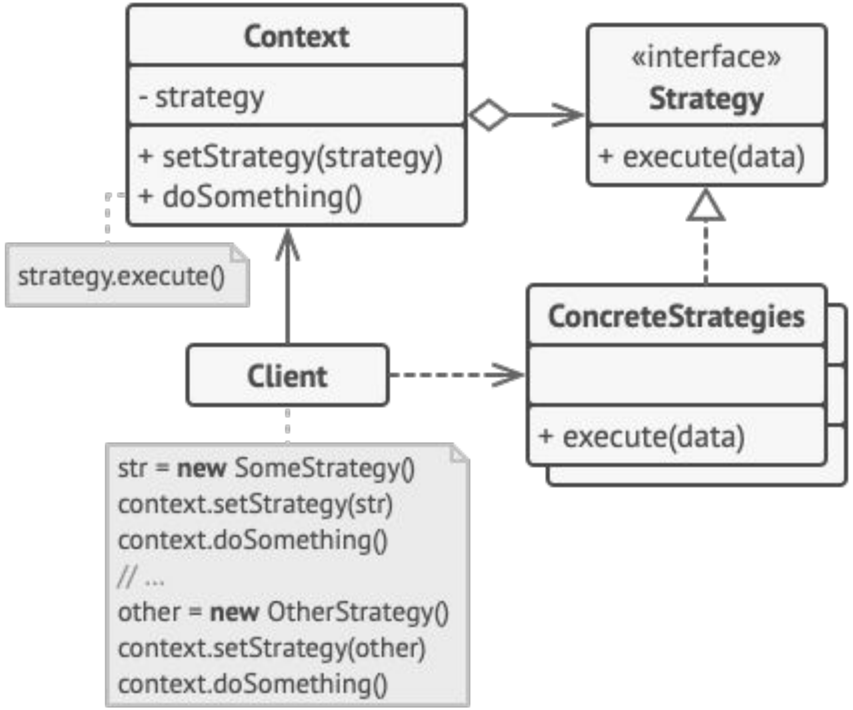
Google maps automatic route planning



Strategy Solution



Strategy Example

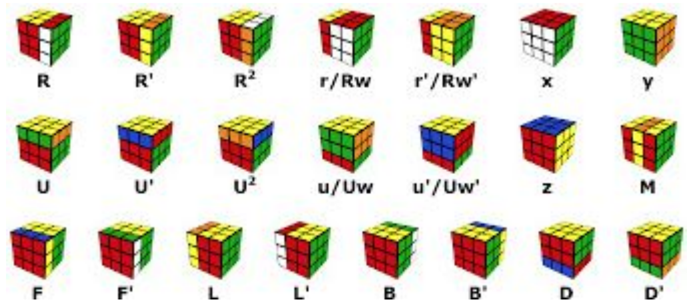


```
str = new SomeStrategy()
context.setStrategy(str)
context.doSomething()
// ...
other = new OtherStrategy()
context.setStrategy(other)
context.doSomething()
```

When to Use Strategy

Use when you want to use different variants of an algorithm within an object and be able to switch from one algorithm to another - the Strategy pattern lets you indirectly alter the object's behavior at runtime by associating it with different sub-objects that perform specific sub-tasks in different ways

Use when you have a lot of similar classes that only differ in the way they execute some behavior - extract the varying behavior into a separate class hierarchy and combine the original classes into one, thereby reducing duplicate code



Use the Strategy pattern when your class has a massive conditional operator that switches between different variants of the same algorithm

Refactoring to Use Strategy

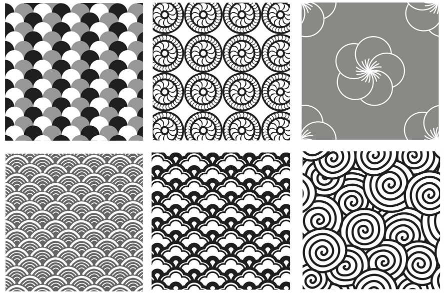
1. In the context class, identify an algorithm that’s prone to frequent changes or has many different variants
2. Declare the strategy interface common to all variants of the algorithm.
3. Extract all algorithms into their own classes that implement the strategy interface
4. In the context class, add a field for storing a reference to a strategy object and provide a setter for replacing values of that field
5. The context class also needs to supply an interface for the strategy object to access the context’s data
6. Clients of the context must associate it with a suitable strategy that matches the way they expect the context to perform its primary job

Strategy Pattern Code Examples

C++

Java

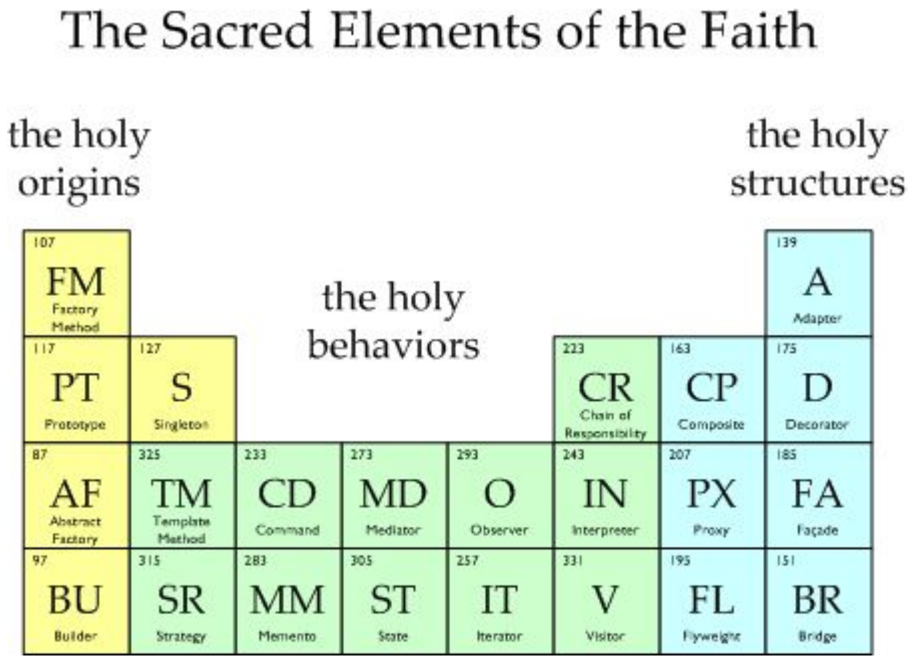
Behavioral vs. Structural vs. Creational



Singleton design pattern is creational, Observer, Mediator and Strategy are behavioral, Facade and Adapter are structural

- Creational patterns provide class instantiation and object creation mechanisms that increase flexibility and reuse of existing code
- Structural patterns explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient
- Behavioral patterns realize common communication paradigms and the assignment of responsibilities between objects

Many More Design Patterns



Upcoming Assignments

Second Iteration due November 28 (note this is the Monday after Thanksgiving)

Second Iteration demo due December 5

Next Week

Tuesday - Class Cancelled

Thursday - Thanksgiving

The Rest of the Semester (not necessarily in this order)

Test Ordering, Test Suite Acceleration

Test Oracles, Metamorphic Testing

Test Generation, Fuzzing, Symbolic Execution

Testing the Test Suite, Mutation Testing

Ask Me Anything