Software Design

- Concepts
  - Abstraction & encapsulation (Feb 1)
  - Design patterns
    - Observer (Feb 3)
    - Strategy (Feb 5)
    - State
  - Dependency inversion
  - Dependency injection

Learning Objectives for Today

- Given a design/code and an appropriate feature request:
    - apply the Strategy design pattern to implement the feature
    - explain the benefits and limitations of the use of the Strategy design pattern for the design/code and feature request

Plan for Today

Observer pattern highlights (and exercise for the reader)

What to do when your algorithm needs to vary…
Strategy Pattern to the rescue!

A simple Strategy pattern example in Typescript

Applying Strategy for a new feature in Mario

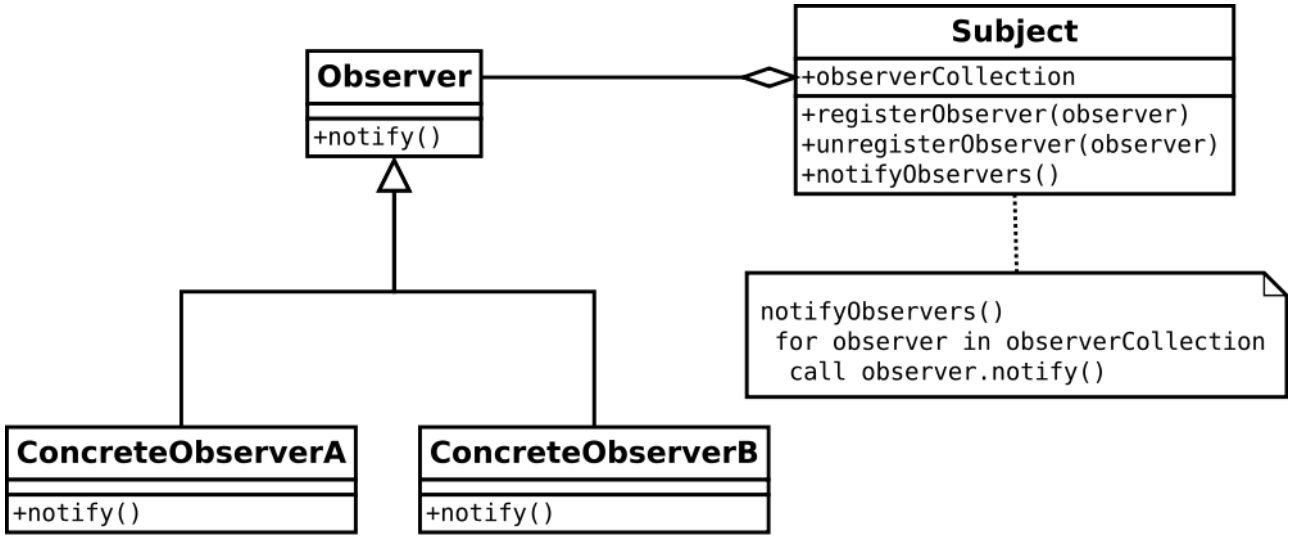
Strategy pattern highlights

Combining patterns

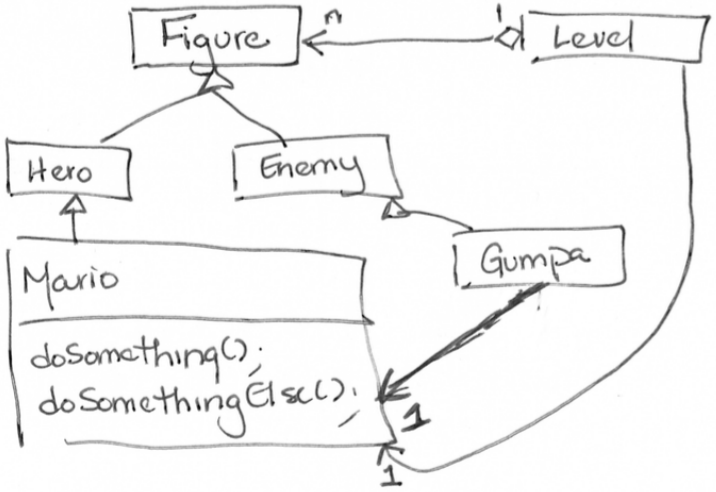
Observer Design Pattern

“The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.”

— Wikipedia entry

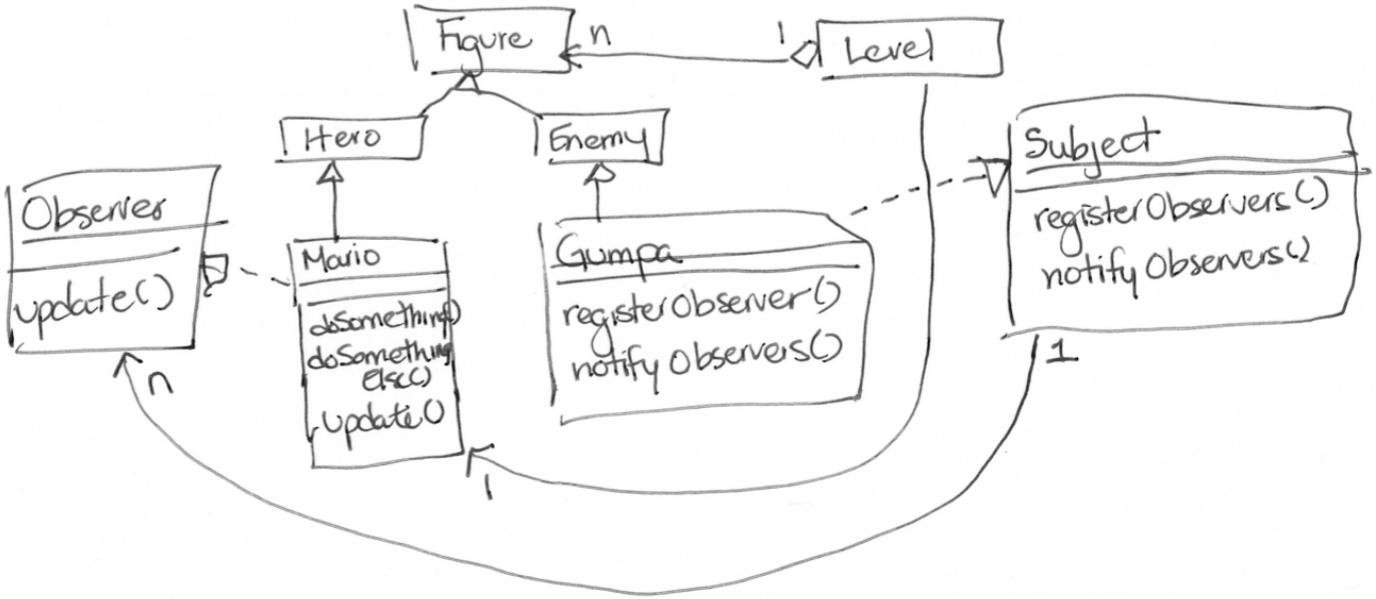


Design without Observer



Note the coupling between Gumpa and Mario. If we want to notify some other class other than Mario, we'd have to add additional coupling to Gumpa.

Design with Observer



Mario and Gump are unaware of each other. Can easily add another Observer to be notified on Gumpa changes.

Highlights of Observer Design Pattern

- Benefits:
  - Can dynamically change the objects notified when an event on the subject happens
  - Reduce coupling between “domain” classes
  - Can use different notification mechanisms, i.e., push or pull (we looked at push only)
  - Can more easily add new Classes (objects) to be notified when an event on the subject happens
- Limitations:
  - Can be awkward to pass a lot of data to an Observer
  - Be careful of spurious updates by the Subject

On-line Resources for Observer Design Pattern

- https://sourcemaking.com/design_patterns/observer
- https://en.wikipedia.org/wiki/Observer_pattern
- http://www.oodesign.com/observer-pattern.html
- You can find a lot of examples, etc. on the web

An Observer Pattern Exercise (for the reader)

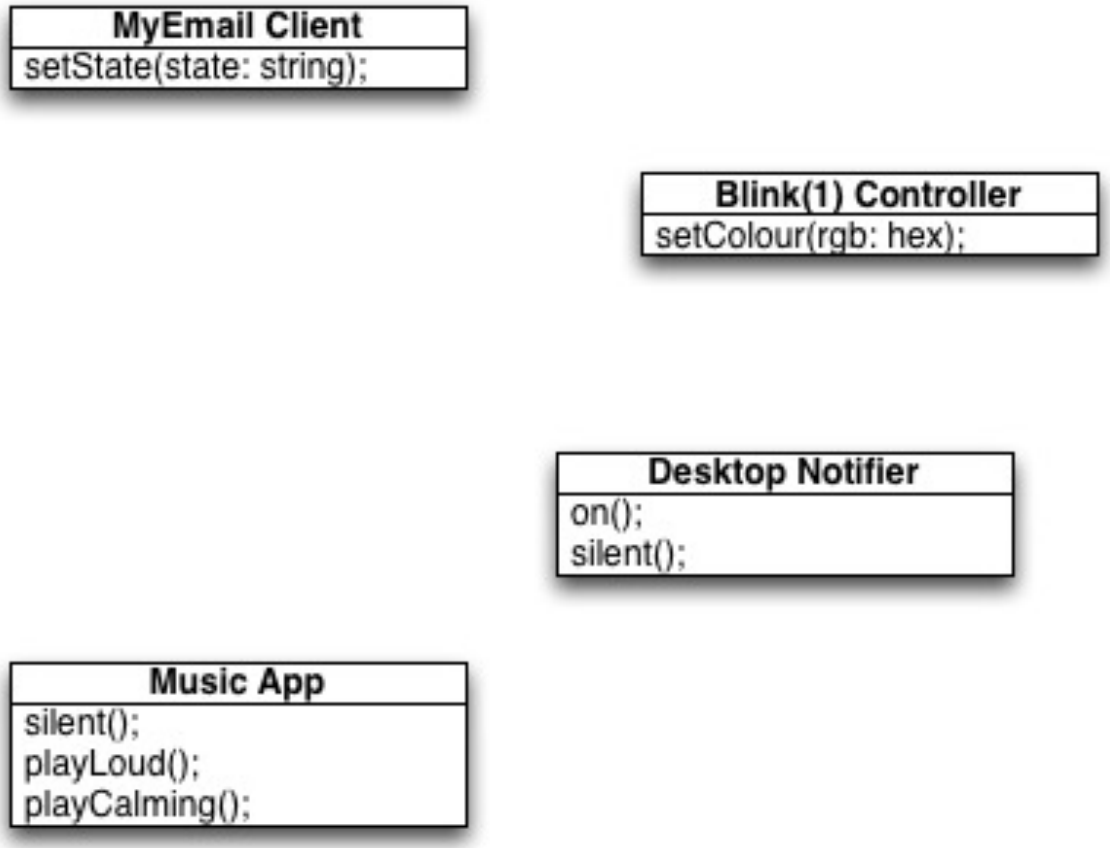
You are designing an email application for which you want to provide information to the user on their workflow.

The email application will recognize certain states: *overload* (more than 100 unread emails), *deluge* (in the last half hour, there has been an email every twenty seconds) and *all-is-well* (less than 5 unread emails).

When in *overload* state, a blink(1) will turn red. When in *deluge* state, desktop notifications will be silenced. When in *all-is-well* state your computer will play calming music.

You have been given the class diagram on the next slide. Apply the Observer pattern to make the desired behaviour happen.

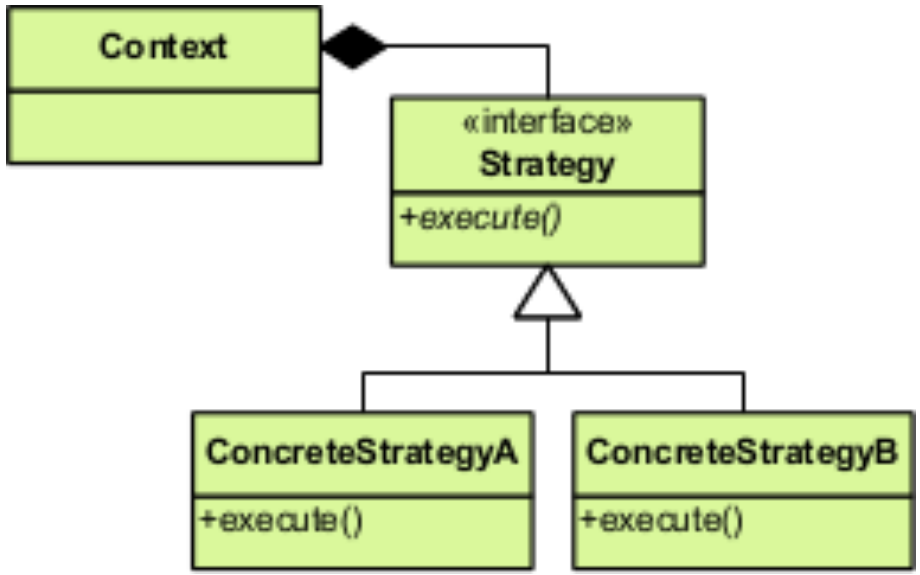
An Observer Pattern Exercise (for the reader)



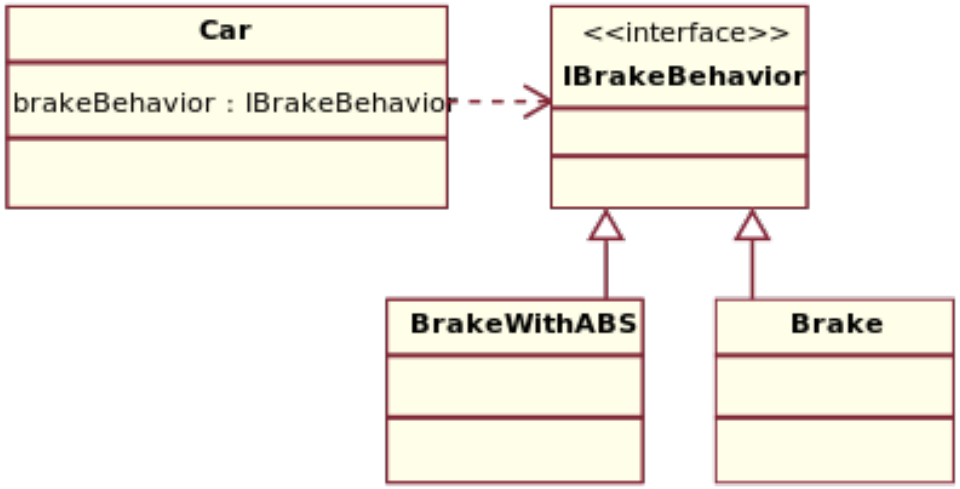
Strategy Design Pattern

“ a software design pattern that enables an algorithm's behavior to be selected at runtime. The strategy pattern: 1) defines a family of algorithms, 2) encapsulates each algorithm, and 3) makes the algorithms interchangeable within that family”

— Wikipedia Entry



An Example



A car may be sold with software that provides “regular” braking or with “anti-lock” braking

```
// An interface for the
// algorithm
interface IBrakeBehaviour {
   brake();
}

// Regular braking algorithm
class Brake implements
IBrakeBehaviour {
    brake() { // do regular; }
}

// ABS braking algorithm
class BrakeABS implements
IBrakeBehaviour {
    brake() { // ABS! };
}
```

```
class Car {

   // braking strategy
   IBrakeBehaviour b_strategy;

   Car() {
     setBrakeStrategy(
       new Brake());
   }

   // Allow dynamic strategy
   // change
   setBrakeStrategy(
     IBrakeBehaviour s) {
     b_strategy = s;
   }

   // actually brake!
   brake() {
     b_strategy.brake();
   }

}
```

delegation in action

But why don't we just use inheritance?

Strategy uses aggregation instead of inheritance.

This choice decouples the class (i.e., Car) from the behaviour (i.e., braking). We can change braking without breaking the class and we can dynamically change the behaviour of the braking for a car.

Consider the car. It probably has a lot of different algorithm choices, e.g., braking, heating, entertainment. If we tried to provide every combination by inheritance, we would have classes like "CarWithRegularBrakingSingleHeatingOnlyRadio", "CarWithABSBrakingSingleHeatingOnlyRadio", "CarWithRegularBrakingDualHeatingDeviceEntertainment", and so on and so on …

Strategy helps support the "open-closed principle". Classes are open for extension but closed for modification. (See https://en.wikipedia.org/wiki/Open/closed_principle for more details on this principle.)

Strategy Pattern: A Typescript Example

You have been asked to design an automated coach. This automated coach will watch your fitness chapter and let you know how it thinks you are doing.

Apply the Strategy Pattern to this problem so that either a “kind coach” or an “evil coach” can be set as a strategy in the Coach’s constructor.

The StrategyExample code linked from the “Software Design Unit” post on piazza has a solution.

The Strategy Pattern in Mario

You have been asked to add a new feature into Mario. Figures (including both Heros and Enemies) can either move in a normal way, or they can move at double speed.

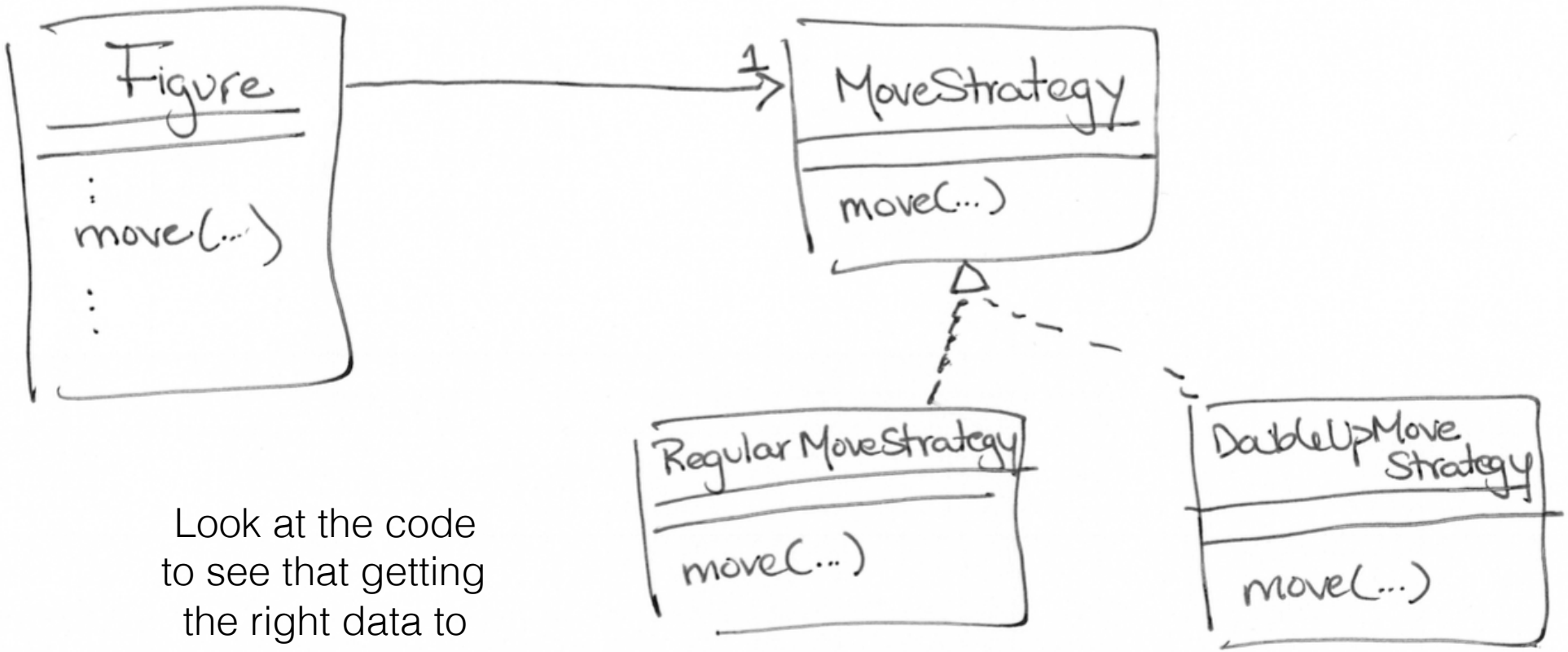
Apply the Strategy Pattern to enable the “move” algorithm to vary at run-time.

Applying a Design Pattern

- What are the critical roles in the Design Pattern?
- Two decisions are inter-related:
  - How are the roles going to map to the classes/ interfaces in the program (e.g., Mario5TS)?
  - What new classes or interfaces need to be added to the program (e.g., Mario5TS)
- How will the Design Pattern be initialized/triggered?

The Design using Strategy Pattern (working area for class)

The Design using Strategy Pattern



Look at the code to see that getting the right data to the algorithm is not always easy

Highlights of Strategy Design Pattern

- Benefits:
  - define families of related algorithms
  - alternative to subclassing
  - enable run-time choice of algorithms
- Limitations:
  - communication overhead between Strategy and Context roles

On-line Resources for Strategy Design Pattern

- https://en.wikipedia.org/wiki/Strategy_pattern
- https://sourcemaking.com/design_patterns/strategy
- https://dzone.com/articles/design-patterns-strategy
- You can find a lot of examples, etc. on the web

Using More than One Pattern (Exercise for the Reader)

Double speed the figures after the first gumpa dies.

Use both the Observer and the Strategy pattern.

Design Terminology

Some design terms you should be comfortable with:

- Coupling
- Cohesion
- Aggregation (Composition) and Association
- Inheritance
- Delegation
- Interface (what they are and why we use them)