Design Patterns

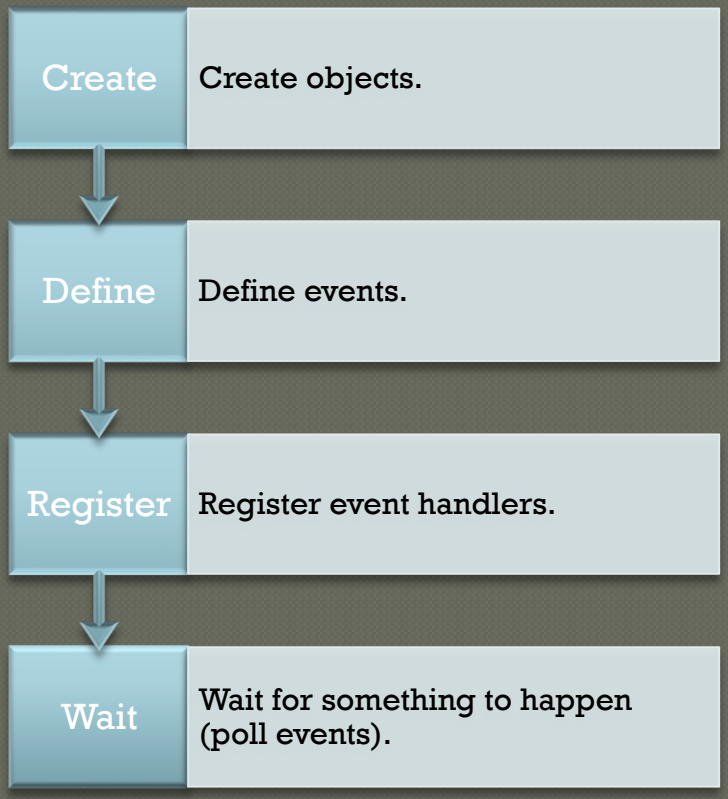
- The Strategy Pattern
- The Factory Method
- Generics
- The Abstract Factory Pattern
- The State Pattern
- **The Observer Pattern**
- The Adapter Pattern
- The Composite Pattern
- The Iterator Pattern
- The Builder Pattern
- Fallen Patterns
  - The Singleton Pattern
  - The Visitor Pattern

Procedural Programming

- Program starts in "main"
  (or at the beginning of the script)
- Create objects, call methods
- Methods call other methods
- Eventually, all methods return, main exits

Event-Driven Programming

- Also known as publish/subscribe



What are events?

AKA signals, triggers

Events are objects which represent an important occurrence

Objects are *polled*. Polling means checking to see if an event has happened.

Events can be *handled.* A handler is a method/function that runs when an event is triggered.

Why Bother?

- Often times the procedural logic of a program is not the focus
- We don't care about drawing the buttons, updating the buttons, and moving the buttons
  - We care about what happens when the user clicks a button
- Event-driven programming models systems of effects:
  - Simulations: what happens when two objects collide?
  - Social media: what happens when someone uses a certain hashtag?
  - GUI: what happens when the scrollbar is clicked

Why Bother?

- Event-driven programming allows us to decouple the detection of an event from its response
- *Reminder*: coupling is how much modules depend on each other
  - Lower is better
- Simplifies the *detector*, no longer needs to know how its events are handled
- Simplifies the *handler*, no longer needs to detect the event

What Does it Look Like?

- C# has events built-in
- Simple example:
  - https://www.codeproject.com/Articles/11541/The-Simplest-C-Events-Example-Imaginable

Metronome: The event raiser (subject)

```
class Metronome {
  public delegate void Handler();
  public event Handler OnTick;

  public void Start() {
    while( true ) {
      System.Threading.Thread.Sleep(3000);
      OnTick();
    }
  }

}
```

Metronome: The event listener

```
class Listener {

  public Listener( string msg, Metronome m ) {
    m.OnTick += HeardATick;
    message = msg;
  }

  private string message;

  private void HeardATick() {
    System.Console.WriteLine( message );
  }
}
```

Main: Setting up the system

```
public static void Main() {

  var m = new Metronome();

  var listener1 = new Listener(
    "Listener 1 heard a tick!", m );

  var listener2 = new Listener(
    "Listener 2 heard a tick!", m );

  m.Start();
}
```

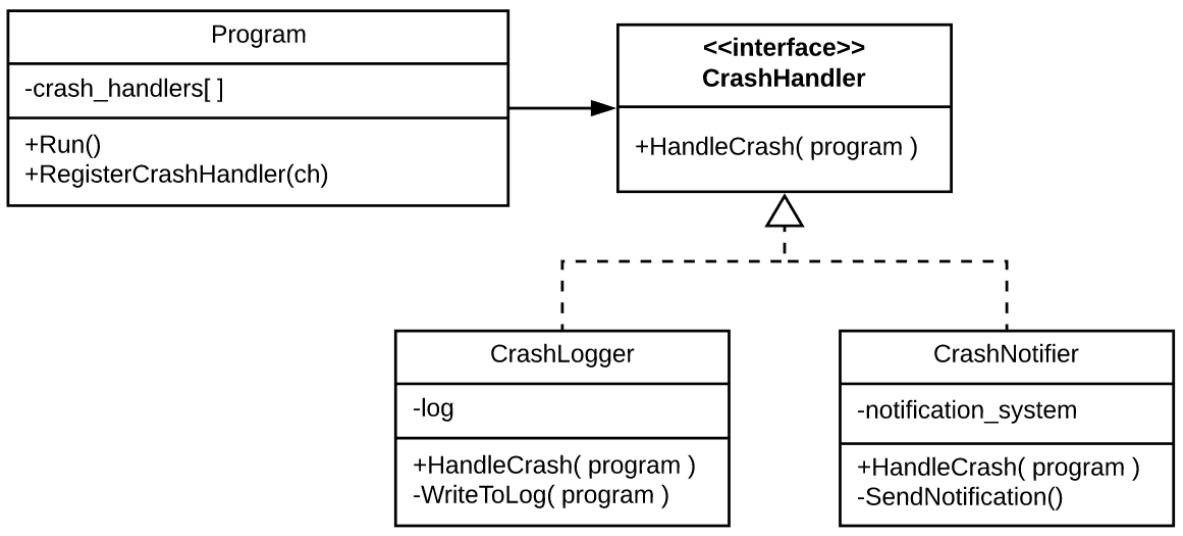
The Observer Pattern

- A specific kind of event handler
  - Basically the observer is what happens when you implement your own event system
- Similar to an event, but not necessarily built-in
- An observer subscribes to a subject
- The subject publishes an event
- When the subject publishes, the observer handles

Example: Crash Reporter

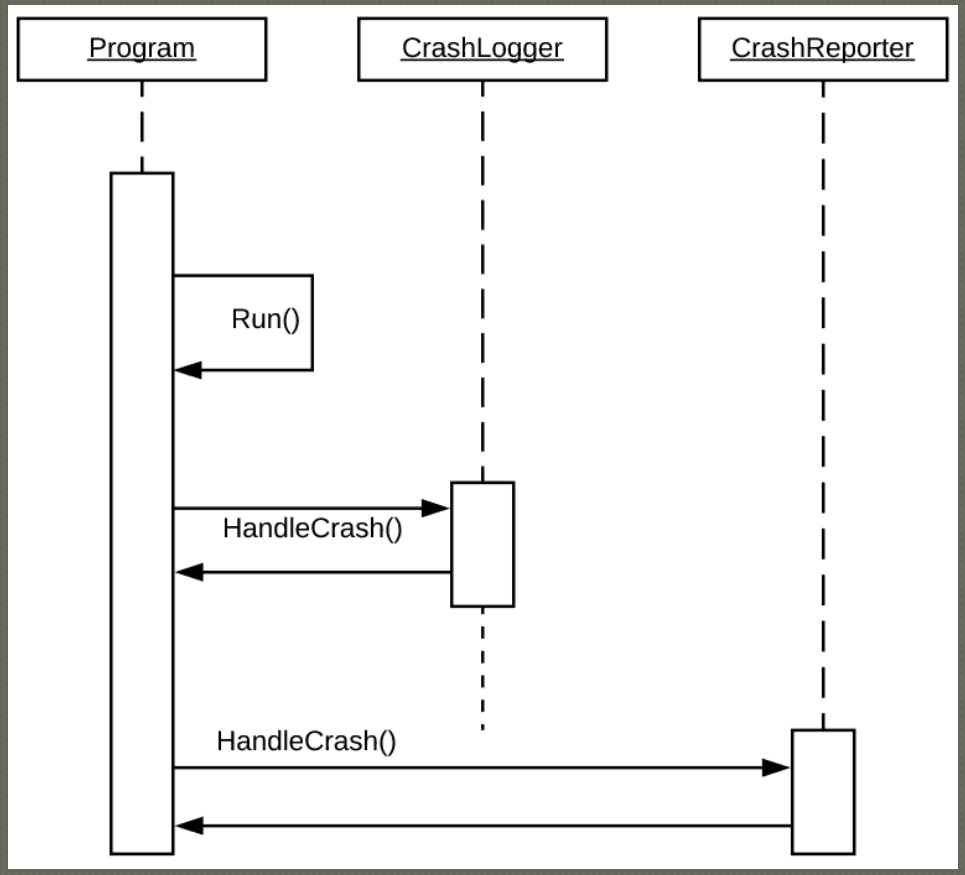
- Every time a program crashes, we want to log the crash report, and notify the user
- Subject: the program
- Listeners:
  - The logger
  - The notifier

Class Design



- Run starts the program running
- If a program raises a signal that causes it to crash (i.e., SIGSEGV) it notifies each of its listeners
- One listener will write the crashing program's information to a log
- The other will notify the user that the program has crashed
  - "An unexpected error has occurred, send error report?"

Sequence Diagram

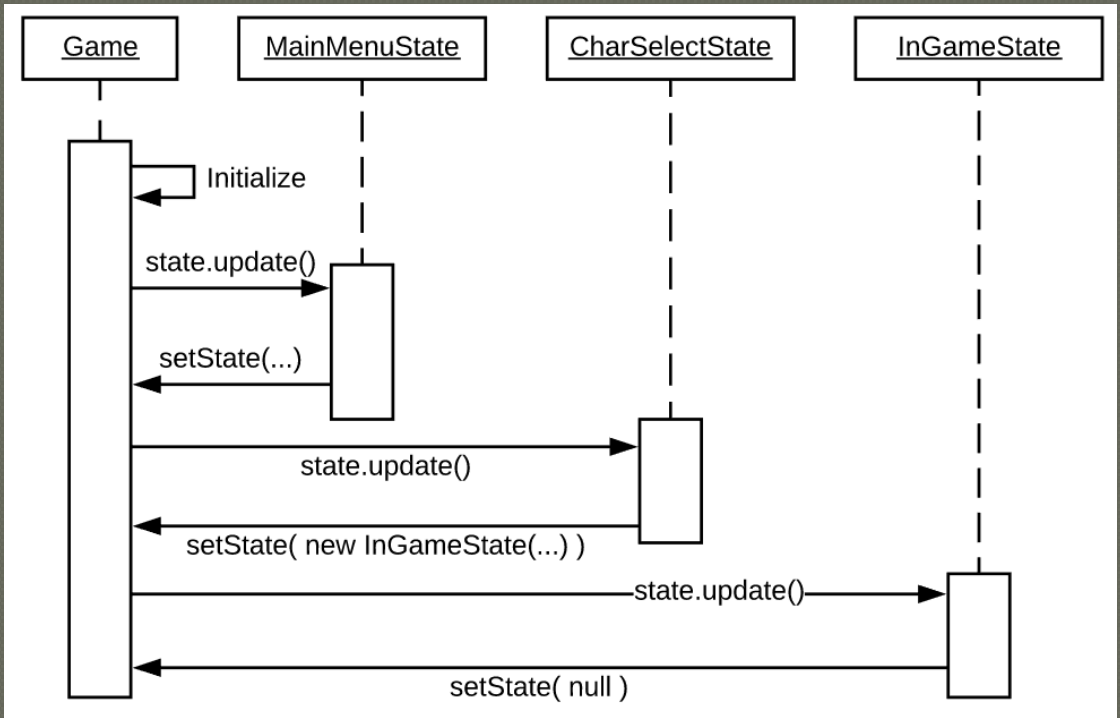


Show the temporal behavior of software

Each arrow represents a method call or return

Objects are active over *activations* (vertical white bars)

Sequence Diagram for State Pattern

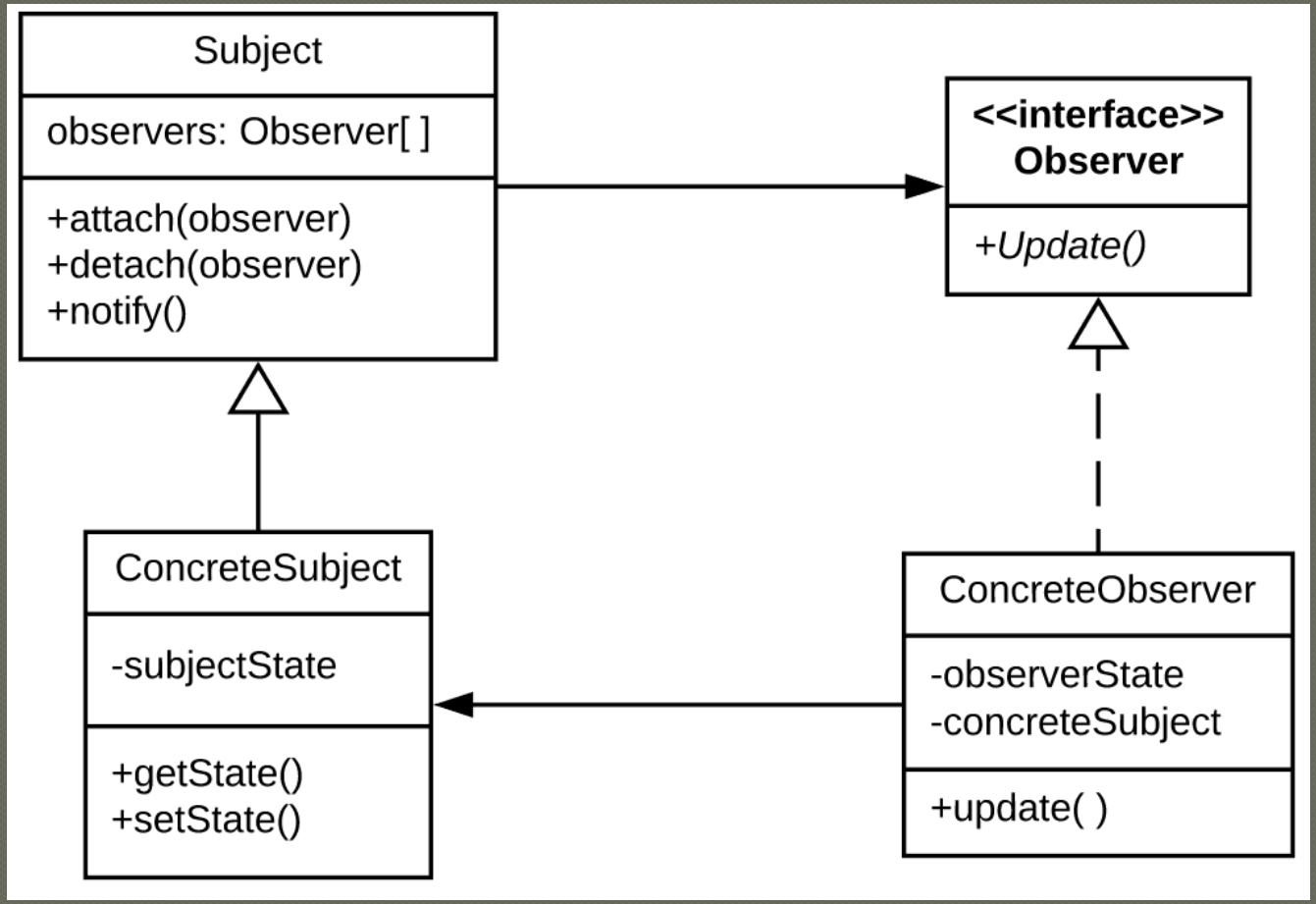


- The activation tells you how long the object is needed
- More than 4 objects: sequence diagrams start to become unwieldy

Are Events a Kind of Strategy?

- In strategy: we have one Behavior interface, and several mutually exclusive implementers of that interface
- In events: the implementors are not mutually exclusive
  - They are all invoked

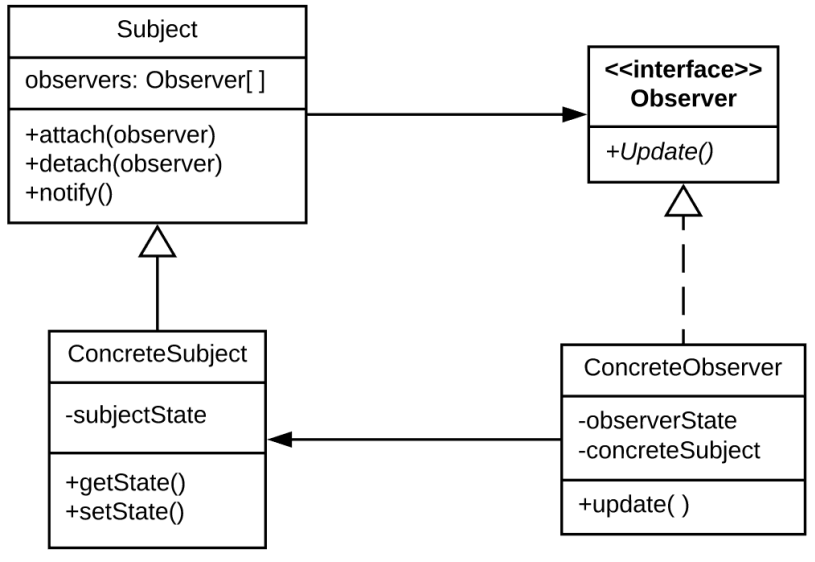
Back to Observer: Generic Diagram



Two Variants:

- 1-way variant: Subject passes arguments to observer in `notify()`
  - Observer does not need a reference to subject
  - This is the CrashReporter example
- 2-way variant (official): Observer retrieves state
  - Benefits: Can be specialized for different concrete subjects
  - Drawbacks: Much higher coupling

Notes



- Subject does not have to be abstract
- None of its methods are abstract
- State here does not mean the state pattern
  - It means the private fields of the object

One More Example:

- Java ActionListeners:
  - https://docs.oracle.com/javase/tutorial/uiswing/events/intro.html

Project Discussion

- How can you apply the observer pattern to your project?