The Observer Design Pattern

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

16:45 - 17:45, Thur, 15th September 2022

60 minutes | Introductory Audience

MIKE SHAH

Cppcon
The C++ Conference

2022 | September 12th-16th

Abstract

Games, desktop software, phone apps, and almost every software that a user interacts with has some sort of event handling system. In order to handle events, a common behavior design pattern known as the 'observer pattern' allows one or more objects to monitor if a change of state takes place in another object. In this talk, we are going to do a deep dive into the behavioral design pattern known as the observer. The pattern utilizes a Subject and Observer (or publisher and subscriber) model to notify when state has changed from the subject to one or more observers in order to help make our software more maintainable, extensible, and flexible.

I will show some examples of the observer in modern C++ as well as real world use cases of where observers are used for further study. Finally, I'll discuss the tradeoffs of the observer pattern, and discuss which scenarios you may not actually want to use the observer pattern. Attendees will leave this talk with the knowledge to go forward and implement the observer pattern, as well as how to spot the observer design pattern in projects they may already be working on!

Please do not redistribute slides without prior permission.

Exercise (1/3)

- Let’s take a look at a sample program.
- What types of ‘events’ do you see?
- (Or another way to state, what subsystems are involved)

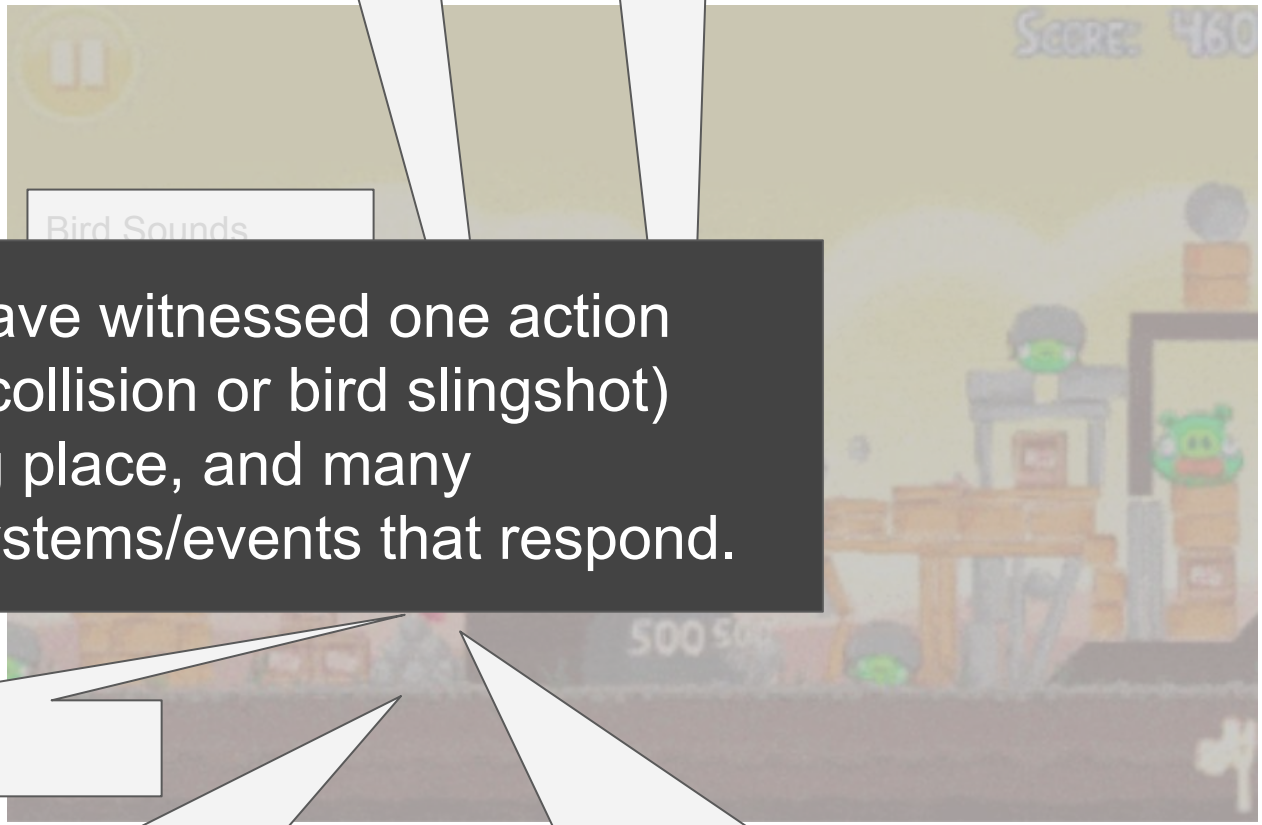
https://downloadcentral.dk/upload/datas/hvorforsucces.gif

Exercise (2/3)

- Let’s take a look at a sample program.
- What types of ‘events’ do you see?
- (Or another way to state, what subsystems are involved)

Exercise (3/3)

- Let's take a look at a sample program
- What types of 'events' do you
- (Or another way: at any state, what subsystems are involved)

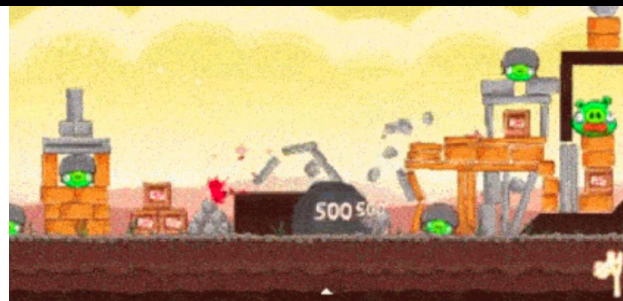


We have witnessed one action (bird collision or bird slingshot) taking place, and many subsystems/events that respond.

Code Exercise (1/3)

- Take a look at this pseudo code for how one might implement this system
  - What problem(s) do you see?

```
1  // Pseudo code -- angrybirds.cpp
2  void BirdSlingshot(){
3
>>     SimulateBirdPhysics(...)
5
6      if(CheckCollisions()){
7          PlaySoundBirdNoise()
8
9          if(HitObject Pig){
10             UpdateScore();
11
12             PlayPigSound();
13
14             SimulateObjectPhysics(pig)
15         }
16
17         LogResult();
18     }
19 }
```



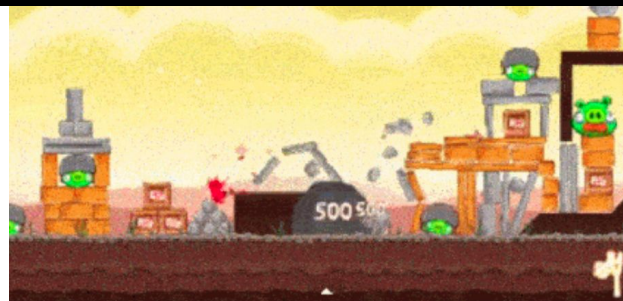
Code Exercise (2/3)

- Take a look at this pseudo code for how one might implement this system
  - What problem(s) do you see?
  -

```
1  // Pseudo code -- angrybirds.cpp
2  void BirdSlingshot(){
3
>>     SimulateBirdPhysics(...)
5
6      if(CheckCollisions()){
7          PlaySoundBirdNoise()
8
9          if(HitObject Pig){
10             UpdateScore();
11
12             PlayPigSound();
13
14             SimulateObjectPhysics(pig)
15         }
16
17         LogResult();
18     }
19 }
```

Besides the deep nesting, and perhaps complicated logic-- what I notice is how ‘coupled’ this code is.

(This is shown in a free function, but this could easily happen in a class as well.)

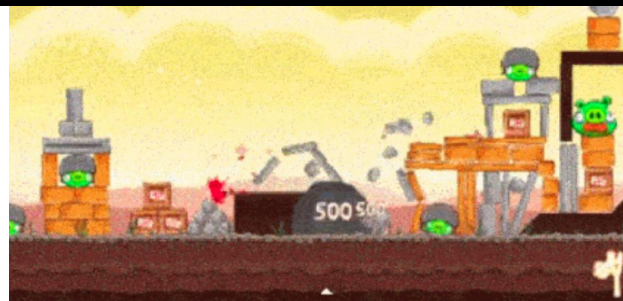


Code Exercise (3/3)

- Take a look at this pseudo code for how one might implement this system
  - What problem(s) do you see?
  -

So...with that said, let's get into a talk about software design where we can perhaps 'fix' this system.

```
1  // Pseudo code -- angrybirds.cpp
2  void BirdSlingshot(){
3
>>     SimulateBirdPhysics(...)
5
6      if(CheckCollisions()){
7          PlaySoundBirdNoise()
8
9          if(HitObject Pig){
10             UpdateScore();
11
12             PlayPigSound();
13
14             SimulateObjectPhysics(pig)
15         }
16
17         LogResult();
18     }
19 }
```



My expectations and why this talk exists

- This talk is part of the Software Design Track at Cppcon
  - Part of this track Klaus and I (the co-chairs) thought would be good to have some ‘tutorial like’ or ‘more fundamental’ (i.e. like the back to the basics) talks on Design Patterns since 2021.
    - (Perhaps 1 or 2 talks like this a year--stay tuned and submit to future Cppcons!)
- So this probably is not an ‘expert-level’ talk, but aimed more at beginner level C++ programmers
  - That said, I hope intermediate/experts will derive some value for looking at today’s pattern.
    - Or otherwise, be able to refresh and point out some tradeoffs with today’s pattern
- Design patterns talks in my opinion are about ‘trade-offs’ and are not 100% solutions
  - They come up often enough, that it’s worth knowing some of the popular ones

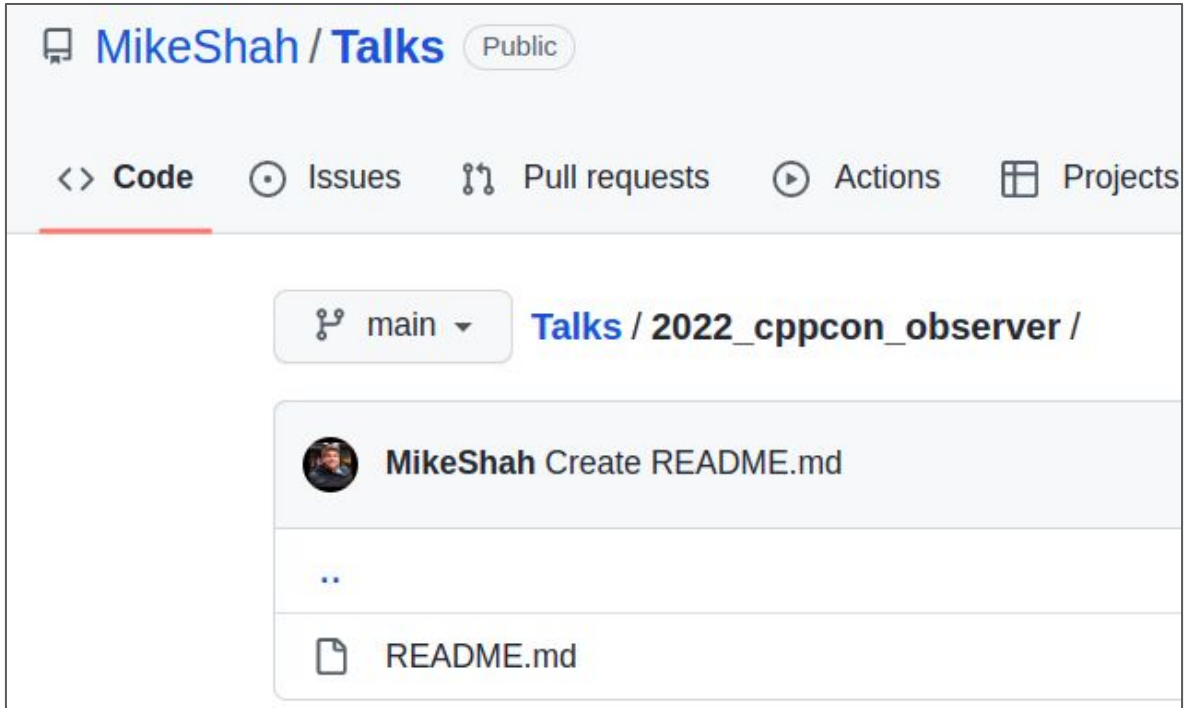
Your Tour Guide for Today

by Mike Shah (he/him)

- Associate Teaching Professor at Northeastern University in Boston, Massachusetts.
  - I teach courses in computer systems, computer graphics, and game engine development.
  - My research in program analysis is related to performance building static/dynamic analysis and software visualization tools.
- I do consulting and technical training on modern C++, Concurrency, OpenGL, and Vulkan projects
  - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of computer graphics, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io
- More online training at courses.mshah.io

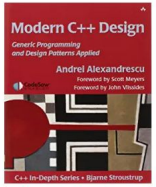
Code for the talk

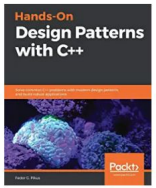
Available here: https://github.com/MikeShah/Talks/tree/main/2022_cppcon_observer



Design Patterns

‘templates’ or ‘flexible blueprints’ for developing software.





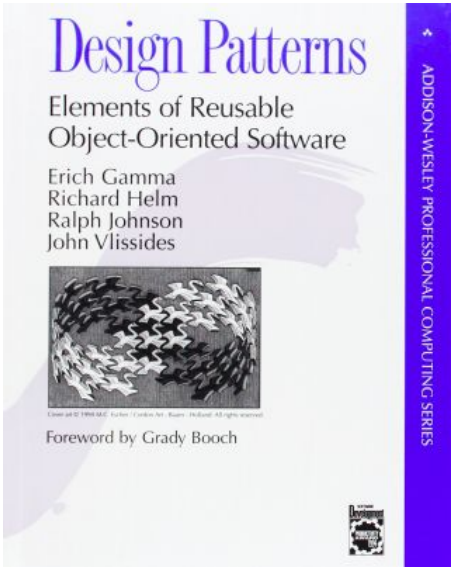
Best Seller

What is a Design Pattern?

- A common repeatable solution for solving problems.
  - Thus, Design Patterns can serve as ‘templates’ or ‘flexible blueprints’ for developing software.
- Design patterns can help make programs more:
  - Flexible
  - Maintainable
  - Extensible
  - (A good pattern helps satisfy these criteria)

Design Patterns Book

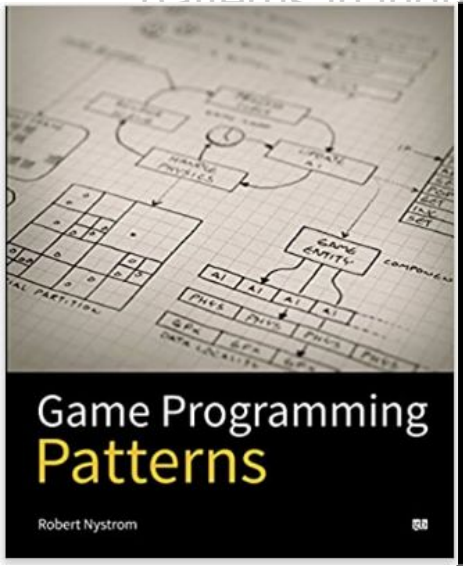
- In 1994 a book came out collecting heavily used patterns in industry titled “Design Patterns”
  - It had four authors, and is dubbed the “Gang of Four” book (GoF).
  - The book is popular enough to have it’s own wikipedia page: https://en.wikipedia.org/wiki/Design_Patterns
  - C++ code samples included, but can be applied in many languages.
  - This book is a good starting point on design patterns for object-oriented programming



* See also the 1977 book “A Pattern Language: Towns, Buildings, Construction” by Christopher Alexander et al. where *I believe* the term design pattern was coined.

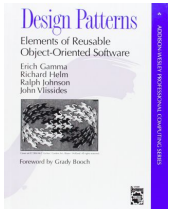
Design Patterns Book * Brief Aside *

- In 1994 a book came out collecting heavily used patterns in industry titled “Design Patterns”



- I really enjoyed this book (as a graphics programmer) for learning design patterns.
  - There’s a free web version here: https://gameprogrammingpatterns.com/
  - I also bought a physical copy to keep on my desk
  - (I am not commissioned to tell you this :))

Design Patterns Book (1/2)

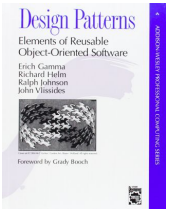


- So design patterns are reusable templates that can help us solve problems that occur in software
  - One (of the many) nice thing the Design Patterns Gang of Four (GoF) book does is organize the 23* presented design patterns into three categories:
    - Creational
    - Structural
    - Behavioral

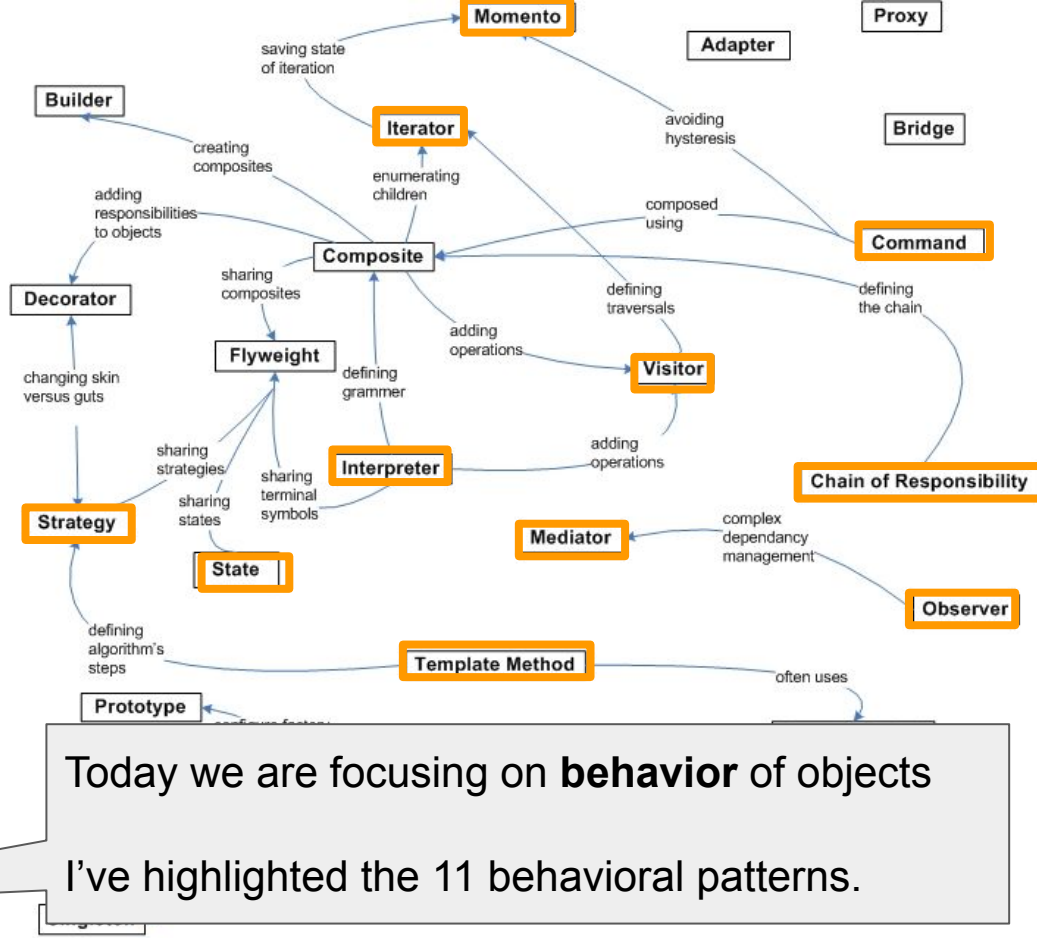
*Keep in mind there are more than 23 design patterns in the world

Design pattern relationships

Design Patterns Book (2/2)



- So design patterns are reusable templates that can help us solve problems that occur in software
  - One (of the many) *nice* thing the Design Patterns Gang of Four (GoF) book does is organize the 23* presented design patterns into three categories:
    - Creational
    - Structural
    - Behavioral



Today we are focusing on **behavior** of objects

I've highlighted the 11 behavioral patterns.

*Keep in mind there are more than 23 design patterns in the world

Design pattern relationships

Behavioral Design Patterns (1/2)

- “Most of these design patterns are specifically concerned with communication between objects.” [wiki]

Behavioral [edit]

Most of these design patterns are specifically concerned with communication between objects.

- Chain of responsibility delegates commands to a chain of processing objects.
- Command creates objects that encapsulate actions and parameters.
- Interpreter implements a specialized language.
- Iterator accesses the elements of an object sequentially without exposing its underlying representation.
- Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- Memento provides the ability to restore an object to its previous state (undo).
- Observer is a publish/subscribe pattern, which allows a number of observer objects to see an event.
- State allows an object to alter its behavior when its internal state changes.
- Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
- Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Creational [edit]

Main article: Creational pattern

Creational patterns are ones that create

- Abstract factory groups object facto
- Builder constructs complex objects
- Factory method creates objects with
- Prototype creates objects by cloning
- Singleton restricts object creation fo

Structural [edit]

These concern class and object compo

- Adapter allows classes with incomp
- Bridge decouples an abstraction fro
- Composite composes zero-or-more
- Decorator dynamically adds/overrid
- Facade provides a simplified interfa
- Flyweight reduces the cost of creati
- Proxy provides a placeholder for an

Behavioral [edit]

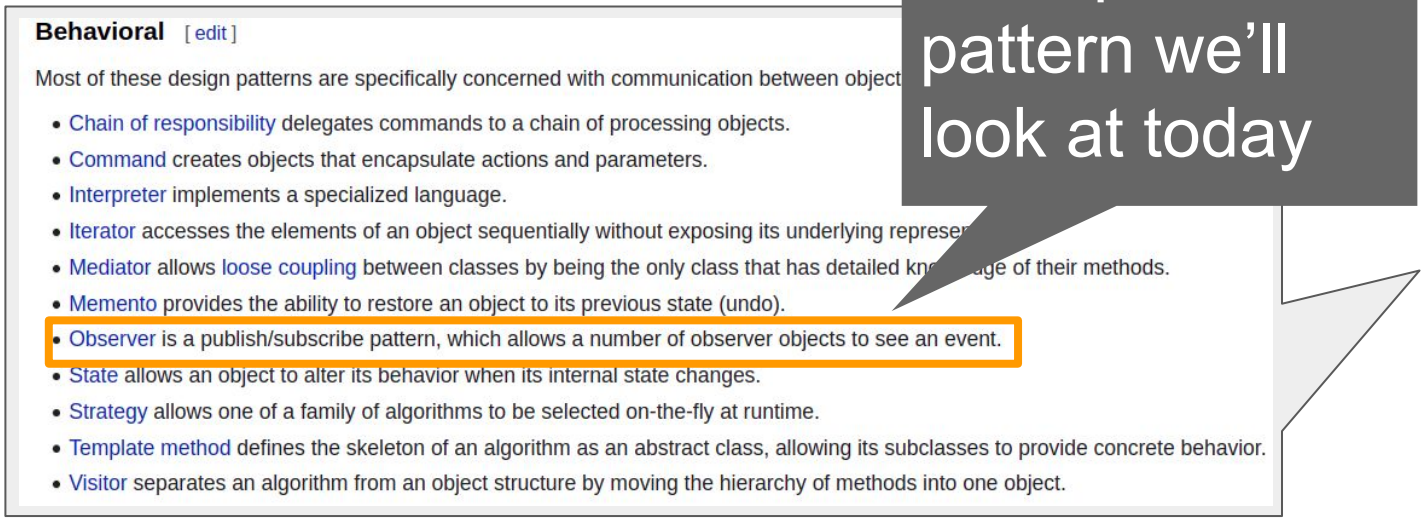
Most of these design patterns are spec

- Chain of responsibility delegates c
- Command creates objects which e
- Interpreter implements a specialize
- Iterator accesses the elements of a
- Mediator allows loose coupling bet
- Memento provides the ability to res
- Observer is a publish/subscribe pa
- State allows an object to alter its b
- Strategy allows one of a family of a
- Template method defines the skele
- Visitor separates an algorithm from

https://en.wikipedia.org/wiki/Design_Patterns#Patterns_by_type

Behavioral Design Patterns (2/2)

- “Most of these design patterns are specifically concerned with communication between objects.” [wiki]



Behavioral [edit]

Most of these design patterns are specifically concerned with communication between object

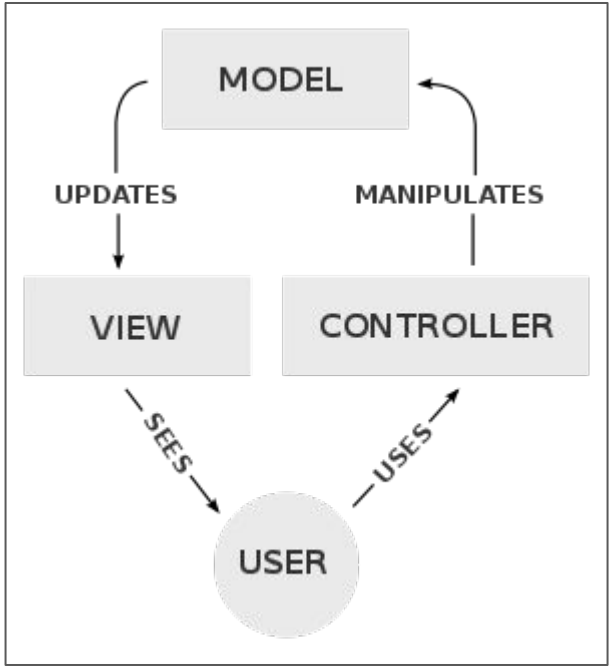
- Chain of responsibility delegates commands to a chain of processing objects.
- Command creates objects that encapsulate actions and parameters.
- Interpreter implements a specialized language.
- Iterator accesses the elements of an object sequentially without exposing its underlying represen
- Mediator allows loose coupling between classes by being the only class that has detailed kn ge of their methods.
- Memento provides the ability to restore an object to its previous state (undo).
- Observer is a publish/subscribe pattern, which allows a number of observer objects to see an event.
- State allows an object to alter its behavior when its internal state changes.
- Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
- Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object.

https://en.wikipedia.org/wiki/Design_Patterns#Patterns_by_type

Model, View, Controller (MVC)

Model, View, Controller (MVC)

- The Model, View, Controller architecture fits quite well with the ‘observer pattern’ we are going to discuss
  - The idea is to separate the application into three main components. [wiki]



Model

The central component of the pattern. It is the application's dynamic data structure, independent of the user interface.[14] It directly manages the data, logic and rules of the application.

View

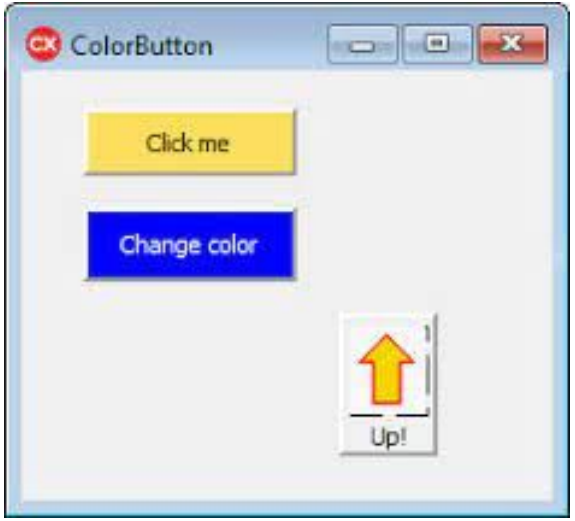
Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

Controller

Accepts input and converts it to commands for the model or view.[15]

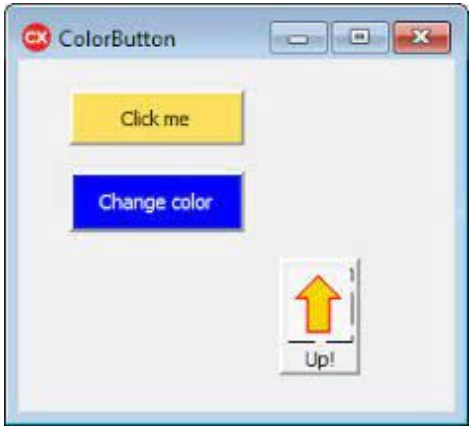
Model, View, Controller (MVC) - Example

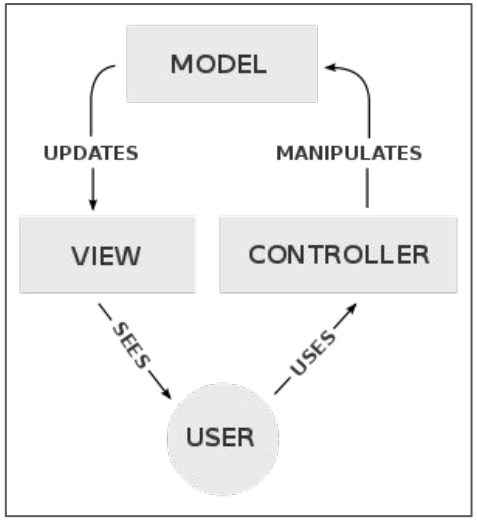
- The ideal example for MVC would be for something like a mouse-click event.
  - Click the mouse (Input, i.e. the controller)
  - Update some data in the model based on the mouse click
  - Update the View based on the model
- A mouseclick could for instance activate several ‘functions’ to take place.



Observer Pattern

- *“The observer pattern is a software design pattern in which an object, named 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.” [wiki]*
- So again, thinking about a simple example, we want to be able to have a ‘mouse-click’ trigger 1 to many events.
    - When the mouse clicks (our subject), a series of events are triggered (by our observers)



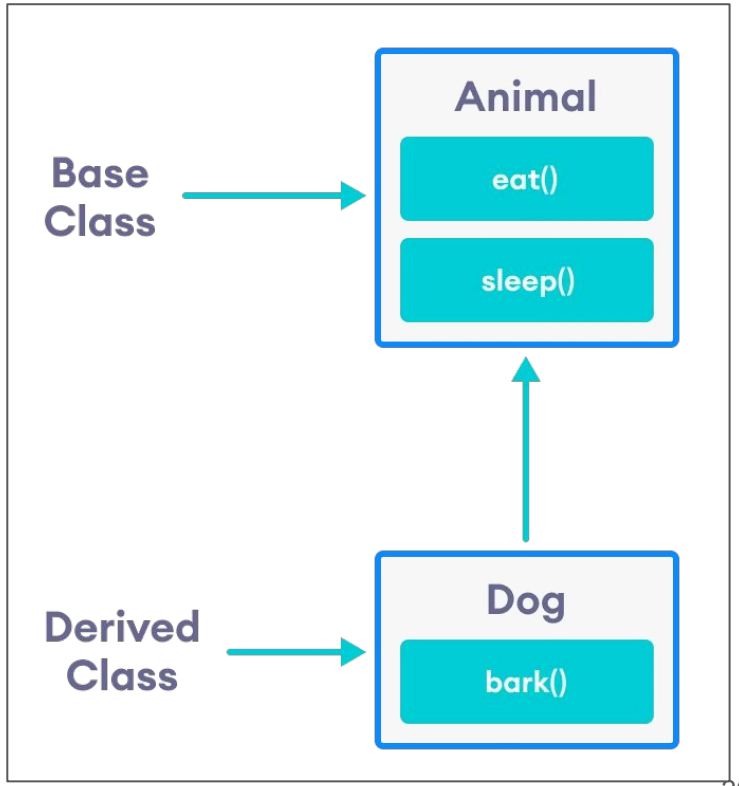


Observer Pattern

First Implementation

Quick Refresh: Object-Oriented Programming Toolbox

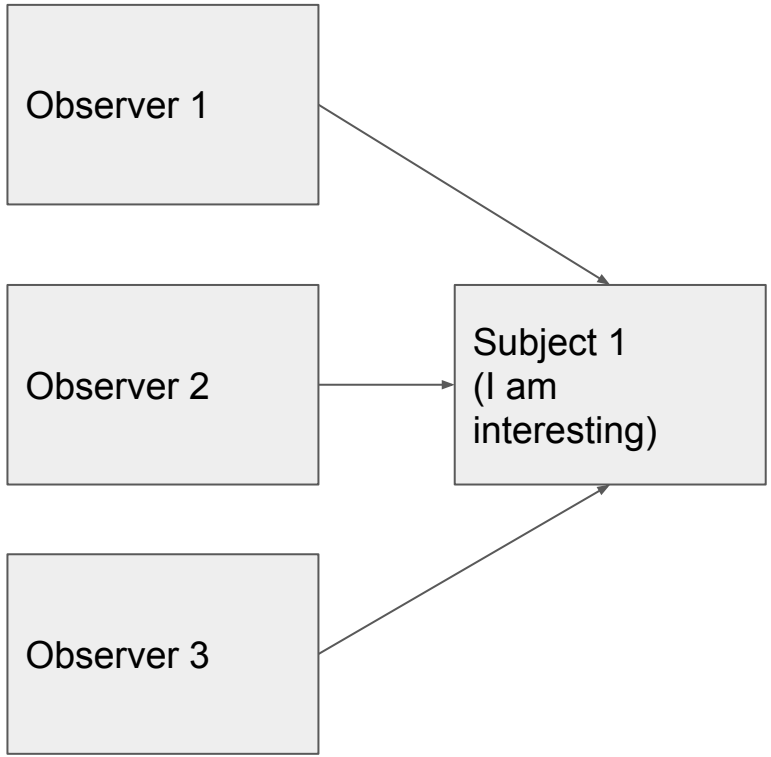
- One of our tools that we can utilize is inheritance
  - This is a mechanism where we create an *is-a* relationship between two types
    - The relationship is a parent-child relationship
    - (e.g., on right, we see that a 'Dog' *is-an* 'Animal'
- Now, I can use the '*is-a*' relationship to my advantage and utilize polymorphism
  - (i.e., inheritance based polymorphism)



https://cdn.programiz.com/sites/tutorial2program/files/cpp-inheritance.png

Observer Pattern 1st implementation

- I want to start with what is probably the simplest version of an Observer Pattern that I can think of.
  - We are going to have two classes
    - **Subject** - This is the ‘thing’ of interest that we want to keep track of any interesting state changes
    - **Observer** - These are the objects we want to react based on the subject
- A *typical* use case, is to have a 1-many relationship between Subject and observer
  - (i.e. 1 subject, and at least 1 observer watching that subject)



Observer Pattern Class Names (and meanings)

- **Subject** (Sometimes also called "*Publisher*" or "*Observable*")
  - 'Subject' or 'observable' is the 'thing' of interest.
  - Can also think of it as a 'publisher' because it will 'notify' of interesting events.
- **Observer** (Sometimes also called "*Subscriber*")
  - 'observer' because it is waiting to be notified of something interesting
  - 'subscriber' you can think of as 'I subscribe to Netflix and am notified on my app when a new movie comes out)

Observer Pattern 1st implementation - Subject

- Here's what the subject looks like
  - Remember, the subject is the 'interesting' thing.
    - (e.g. a celebrity in Hollywood is interesting)
  - So that means we are adding 'observers' to it.
  - The Subjects responsibility is to 'Notify' all of the observers when something

```
19 // example0/main.cpp
20 class Subject{
21     public:
22         void AddObserver(Observer* observer){
23             mObservers.push_front(observer);
24         }
25
26         void RemoveObserver(Observer* observer){
27             mObservers.remove(observer);
28         }
29
30         void Notify(){
31             for(auto& o: mObservers){
32                 o->OnNotify();
33             }
34         }
35
36     private:
37         std::forward_list<Observer*> mObservers;
38 };
```

Observer Pattern 1st implementation - Observer

- Here's what the observer looks like
  - The 'OnNotify' member function is what we implement.
  - This is the member function called when the 'Subject' does something interesting.

```
 6 // example0/main.cpp
 7 class Observer{
 8     public:
 9         Observer(std::string name) : mName(name){
10         }
11
12         void OnNotify(){
13             std::cout << mName << "-hello!" << std::endl;
14         }
15     private:
16         std::string mName;
17 };
```

Observer Pattern 1st implementation - Subject & Observer

Observer 1

```
6  // example0/main.cpp
7  class Observer{
8      public:
9          Observer(std::string name) : mName(name){
10         }
11
12         void OnNotify(){
13             std::cout << mName << "-hello!" << std::endl;
14         }
15     private:
16         std::string mName;
17 };
```

Observer 2

```
6  // example0/main.cpp
7  class Observer{
8      public:
9          Observer(std::string name) : mName(name){
10         }
11
12         void OnNotify(){
13             std::cout << mName << "-hello!" << std::endl;
14         }
15     private:
16         std::string mName;
17 };
```

Observer 3

```
6  // example0/main.cpp
7  class Observer{
8      public:
9          Observer(std::string name) : mName(name){
10         }
11
12         void OnNotify(){
13             std::cout << mName << "-hello!" << std::endl;
14         }
15     private:
16         std::string mName;
17 };
```

Subject 1
(I am interesting)

```
19 // example0/main.cpp
20 class Subject{
21     public:
22         void AddObserver(Observer* observer){
23             mObservers.push_front(observer);
24         }
25
26         void RemoveObserver(Observer* observer){
27             mObservers.remove(observer);
28         }
29
30         void Notify(){
31             for(auto& o: mObservers){
32                 o->OnNotify();
33             }
34         }
35
36     private:
37         std::forward_list<Observer*> mObservers;
38 };
```

- So this is effectively what we want to setup in our system.
- Let's go ahead and implement this (next slide)

Observer Pattern 1st implementation - Usage (1/3)

- Here's the usage and output.
  - First we create our subject, create some observers, and then setup the 1 to many relationship.

```
41 // example0/main.cpp
42 int main(){
43
44     Subject subject;
45     Observer observer1("observer1");
46     Observer observer2("observer2");
47     Observer observer3("observer3");
48
49     subject.AddObserver(&observer1);
50     subject.AddObserver(&observer2);
51     subject.AddObserver(&observer3);
52
53     subject.Notify();
54
55     std::cout << std::endl;
56     subject.RemoveObserver(&observer1);
57
58     subject.Notify();
59
60     return 0;
61 }
```

Observer Pattern 1st implementation - Usage (2/3)

- Here's the usage and output.
  - Here we call 'Notify' on our subject
  - We see that all of the 'observers' perform an their 'OnNotify' action

```
mike:example0$ g++ -g main.cpp -o prog
mike:example0$ ./prog
observer3-hello!
observer2-hello!
observer1-hello!
```

```
41 // example0/main.cpp
42 int main(){
43
44     Subject subject;
45     Observer observer1("observer1");
46     Observer observer2("observer2");
47     Observer observer3("observer3");
48
49     subject.AddObserver(&observer1);
50     subject.AddObserver(&observer2);
       subject.AddObserver(&observer3);
52
53     subject.Notify();
54
55     std::cout << std::endl;
56     subject.RemoveObserver(&observer1);
57
58     subject.Notify();
59
60     return 0;
61 }
```

Observer Pattern 1st implementation - Usage (3/3)

- Here's the usage and output.
  - Now we modify our observers for 'subject'
    - And we see 2 of our observers

```
41 // example0/main.cpp
42 int main(){
43
44     Subject subject;
45     Observer observer1("observer1");
46     Observer observer2("observer2");
47     Observer observer3("observer3");
48
49     subject.AddObserver(&observer1);
50     subject.AddObserver(&observer2);
       subject.AddObserver(&observer3);
52
53     subject.Notify();
54
55     std::cout << std::endl;
56     subject.RemoveObserver(&observer1);
57
58     subject.Notify();
59
60     return 0;
61 }
```

```
mike:example0$ g++ -g main.cpp -o prog
mike:example0$ ./prog
observer3-hello!
observer2-hello!
observer1-hello!

observer3-hello!
observer2-hello!
```

Discussion of our First Try

1st implementation - Pros and Cons? (1/2)

- So, no design pattern is **perfect**, computer science is about trade-offs.
  - (Question to the audience)
    - Is this pattern:
      - Flexible
      - Maintainable
      - Extensible

```cpp
// example0/main.cpp
#include <string>
#include <iostream>
#include <forward_list>

// example0/main.cpp
class Observer{
    public:
        Observer(std::string name) : mName(name){
        }

        void OnNotify(){
            std::cout << mName << "-hello!" << std::endl;
        }
    private:
        std::string mName;
};

// example0/main.cpp
class Subject{
    public:
        void AddObserver(Observer* observer){
            mObservers.push_front(observer);
        }

        void RemoveObserver(Observer* observer){
            mObservers.remove(observer);
        }

        void Notify(){
            for(auto& o: mObservers){
                o->OnNotify();
            }
        }

    private:
        std::forward_list<Observer*> mObservers;
};


// example0/main.cpp
int main(){

    Subject subject;
    Observer observer1("observer1");
    Observer observer2("observer2");
    Observer observer3("observer3");

    subject.AddObserver(&observer1);
    subject.AddObserver(&observer2);
    subject.AddObserver(&observer3);

    subject.Notify();

    std::cout << std::endl;
    subject.RemoveObserver(&observer1);

    subject.Notify();

    return 0;
}
```

1st implementation - Pros and Cons? (2/2)

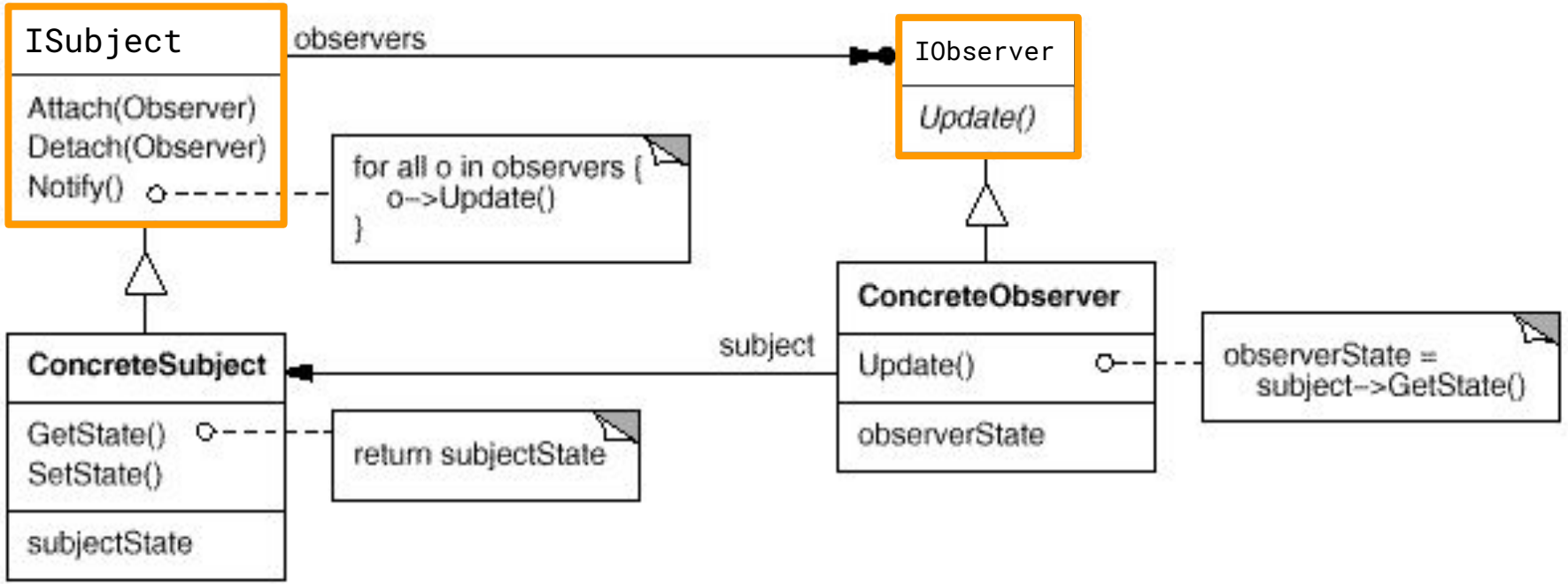
- So, no design pattern is **perfect**, computer science is about trade-offs.
  - (Question to the audience)
    - Is this pattern:
      - Flexible - Not really as it stands
      - Maintainable - Maybe?
      - Extensible - Not really
- **So we haven't really done our job yet -- I've shown you the 'idea' of a Subject with observers**
  - **Let's enhance this a bit.**

```cpp
// example0/main.cpp
#include <string>
#include <iostream>
#include <forward_list>

// example0/main.cpp
class Observer{
    public:
        Observer(std::string name) : mName(name){
        }

        void OnNotify(){
            std::cout << mName << "-hello!" << std::endl;
        }
    private:
        std::string mName;
};

// example0/main.cpp
class Subject{
    public:
        void AddObserver(Observer* observer){
            mObservers.push_front(observer);
        }

        void RemoveObserver(Observer* observer){
            mObservers.remove(observer);
        }

        void Notify(){
            for(auto& o: mObservers){
                o->OnNotify();
            }
        }

    private:
        std::forward_list<Observer*> mObservers;
};


// example0/main.cpp
int main(){

    Subject subject;
    Observer observer1("observer1");
    Observer observer2("observer2");
    Observer observer3("observer3");

    subject.AddObserver(&observer1);
    subject.AddObserver(&observer2);
    subject.AddObserver(&observer3);

    subject.Notify();

    std::cout << std::endl;
    subject.RemoveObserver(&observer1);

    subject.Notify();

    return 0;
}
```

Observer Pattern - 2nd Try

Utilizing Interfaces

ISubject and IObserver Interfaces



https://www.bogotobogo.com/DesignPatterns/images/observer/observer_pattern.gif

Observer Pattern 2nd implementation - ISubject (1/2)

- We've modified our 'Subject' to now be 'ISubject'
  - (next slide)

```
28 // example1/main.cpp
29 // Now we have converted 'Subject' to 'ISubject'
30 // 'ISubject' now works with our IObserver interface
31 class ISubject{
32     public:
33         ISubject() {};
34         virtual ~ISubject() {}
35
36         virtual void AddObserver(IObserver* observer){
37             mObservers.push_front(observer);
38         }
39
40         virtual void RemoveObserver(IObserver* observer){
41             mObservers.remove(observer);
42         }
43
44         virtual void Notify(){
45             for(auto& o: mObservers){
46                 o->OnNotify();
47             }
48         }
49
50     private:
51         std::forward_list<IObserver*> mObservers;
52 };
53
54 // example1/main.cpp
55 class SomeSubject : public ISubject{
56         public:
57
58 };
```

Observer Pattern 2nd implementation - ISubject (2/2)

- We've modified our 'Subject' to now be 'ISubject'
  - ISubject also will take in 'IObserver' (IObserver on next slide), so that anything like 'SomeSubject' can use any IObservable

```
28 // example1/main.cpp
29 // Now we have converted 'Subject' to 'ISubject'
30 // 'ISubject' now works with our IObserver interface
31 class ISubject{
32     public:
33         ISubject() {};
34         virtual ~ISubject() {}
35
36         virtual void AddObserver(IObserver* observer){
37             mObservers.push_front(observer);
38         }
39
40         virtual void RemoveObserver(IObserver* observer){
41             mObservers.remove(observer);
42         }
43
44         virtual void Notify(){
45             for(auto& o: mObservers){
46                 o->OnNotify();
47             }
48         }
49
50     private:
51         std::forward_list<IObserver*> mObservers;
52 };
53
54 // example1/main.cpp
55 class SomeSubject : public ISubject{
56         public:
57
58 };
```

Observer Pattern 2nd implementation - IObserver

- We really want an `IObserver` interface so we can have any object derive from this class.
- ‘Watcher’ is an example that implements the interface.
  - ‘Watcher’ can now ‘observer/subscribe/attach’ to any subject.

```
// example1/main.cpp
// Abstract class, the idea is we will inherit from
// this class
class IObserver{
    public:
        virtual ~IObserver() {}
        virtual void OnNotify()=0;
};

// Watcher is derived from IObserver
class Watcher : public IObserver{
    public:
        explicit Watcher(const std::string& name) : mName(name){
        }

        void OnNotify(){
            std::cout << mName << "-hello!" << std::endl;
        }
    private:
        std::string mName;
};
```

Observer Pattern 2nd implementation - Usage

- Here's the actual implementation making use of the derived classes.
  - This produces the same result as before.

```
60 // example1/main.cpp
61 int main(){
62
63     SomeSubject subject;
64     Watcher watcher1("watcher1");
65     Watcher watcher2("watcher2");
66     Watcher watcher3("watcher3");
67
68     subject.AddObserver(&watcher1);
69     subject.AddObserver(&watcher2);
70     subject.AddObserver(&watcher3);
71
72     subject.Notify();
73
74     std::cout << std::endl;
75     subject.RemoveObserver(&watcher1);
76
77     subject.Notify();
78
79     return 0;
80 }
```

2nd implementation - Pros and Cons? (1/2)

- So, no design pattern is **perfect**, computer science is about trade-offs.
  - (Question to the audience)
    - Is this pattern:
      - Flexible
      - Maintainable
      - Extensible

```cpp
// example1/main.cpp
#include <string>
#include <iostream>
#include <forward_list>

// example1/main.cpp
// Abstract class, the idea is we will inherit from
// this class
class IObserver{
    public:
        virtual ~IObserver() {}
        virtual void OnNotify()=0;
};

// Watcher is derived from IObserver
class Watcher : public IObserver{
    public:
        Watcher(std::string name) : mName(name){
        }

        void OnNotify(){
            std::cout << mName << "-hello!" << std::endl;
        }
    private:
        std::string mName;
};

// example1/main.cpp
// Now we have converted 'Subject' to 'ISubject'
// 'ISubject' now works with our IObserver interface
class ISubject{
    public:
        ISubject() {};
        virtual ~ISubject() {}

        virtual void AddObserver(IObserver* observer){
            mObservers.push_front(observer);
        }

        virtual void RemoveObserver(IObserver* observer){
            mObservers.remove(observer);
        }

        virtual void Notify(){
            for(auto& o: mObservers){
                o->OnNotify();
            }
        }

    private:
        std::forward_list<IObserver*> mObservers;
};

// example1/main.cpp
class SomeSubject : public ISubject{
        public:

};

// example1/main.cpp
int main(){

    SomeSubject subject;
    Watcher watcher1("watcher1");
    Watcher watcher2("watcher2");
    Watcher watcher3("watcher3");

    subject.AddObserver(&watcher1);
    subject.AddObserver(&watcher2);
    subject.AddObserver(&watcher3);

    subject.Notify();

    std::cout << std::endl;
    subject.RemoveObserver(&watcher1);

    subject.Notify();

    return 0;
}
```

Yes, I know you can't read this. You can download and read this later :)

2nd implementation - Pros and Cons? (2/2)

- So, no design pattern is **perfect**, computer science is about trade-offs.
  - (Question to the audience)
    - Is this pattern:
      - Flexible - Needs more power, not quite there yet
      - Maintainable - More so, just need to keep the interfaces abstract. Otherwise, changes are made in derived classes mostly on the ‘Watcher’ end
      - Extensible - Utilizing inheritance we can make use of this pattern
- **Okay, still more to do on the flexibility, and think in particular the ‘subject’ needs more power.**

Yes, I know you can’t read this. You can download and read this later :)

```cpp
// example1/main.cpp
#include <string>
#include <iostream>
#include <forward_list>

// example1/main.cpp
// Abstract class, the idea is we will inherit from
// this class
class IObserver{
    public:
        virtual ~IObserver() {}
        virtual void OnNotify()=0;
};

// Watcher is derived from IObserver
class Watcher : public IObserver{
    public:
        Watcher(std::string name) : mName(name){
        }

        void OnNotify(){
            std::cout << mName << "-hello!" << std::endl;
        }
    private:
        std::string mName;
};

// example1/main.cpp
// Now we have converted 'Subject' to 'ISubject'
// 'ISubject' now works with our IObserver interface
class ISubject{
    public:
        ISubject() {};
        virtual ~ISubject() {}

        virtual void AddObserver(IObserver* observer){
            mObservers.push_front(observer);
        }

        virtual void RemoveObserver(IObserver* observer){
            mObservers.remove(observer);
        }

        virtual void Notify(){
            for(auto& o: mObservers){
                o->OnNotify();
            }
        }

    private:
        std::forward_list<IObserver*> mObservers;
};

// example1/main.cpp
class SomeSubject : public ISubject{
        public:

};

// example1/main.cpp
int main(){

    SomeSubject subject;
    Watcher watcher1("watcher1");
    Watcher watcher2("watcher2");
    Watcher watcher3("watcher3");

    subject.AddObserver(&watcher1);
    subject.AddObserver(&watcher2);
    subject.AddObserver(&watcher3);

    subject.Notify();

    std::cout << std::endl;
    subject.RemoveObserver(&watcher1);

    subject.Notify();

    return 0;
}
```

Observer Pattern - 3rd Round

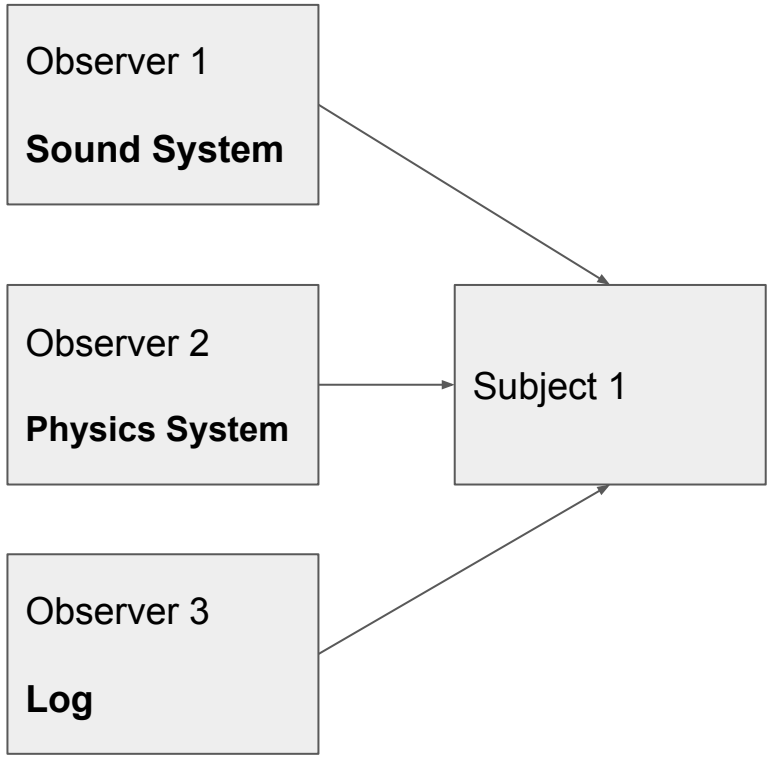
Making our `ISubject` more powerful

Observer Pattern 3rd - Improvements

- Sometimes we want to be a little more specific with the types of things we are ‘subscribing to’
  - i.e. Our observers should handle specific events, perhaps in specific subsystems

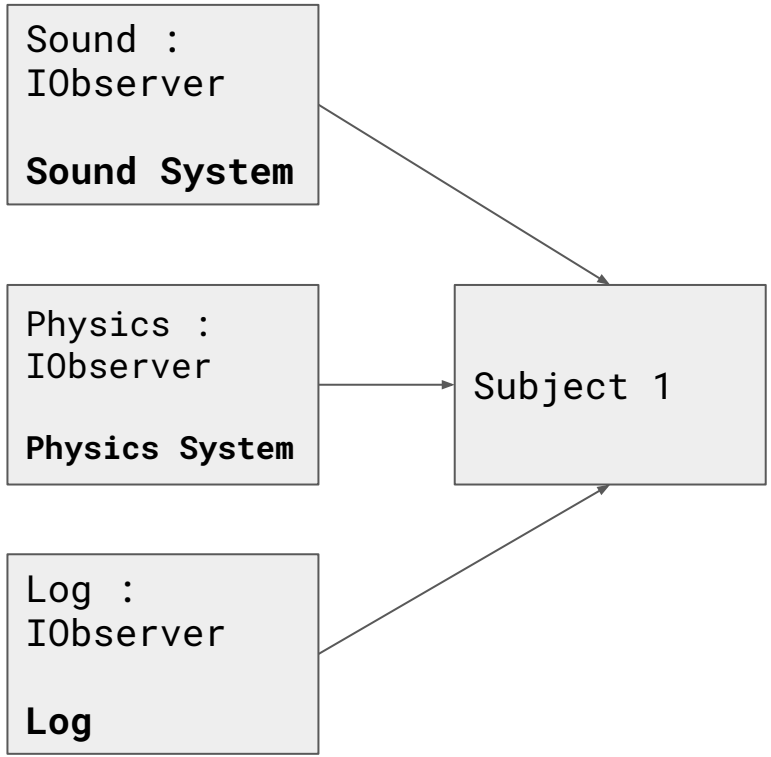
Design (1/3)

- Notice now that each of our ‘Observers’ could be part of a system.



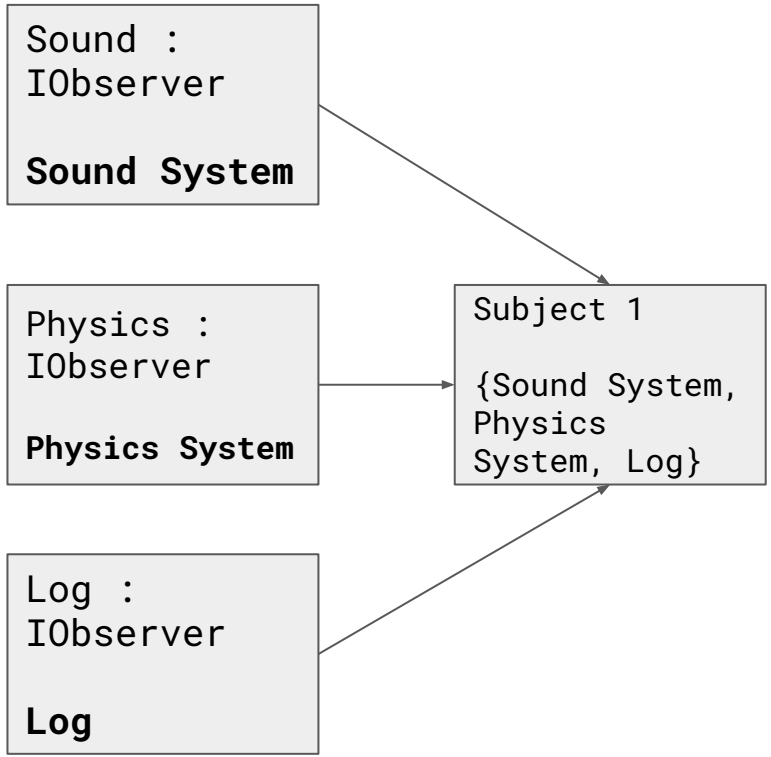
Design (2/3)

- Notice now that each of our ‘Observers’ could be part of a system.
  - That is to say, ‘Observer 1’ can be for sounds
  - ‘Observer 2’ might be deriving from some IObserver to handle Physics
  - ‘Observer 3’ for logging
  - etc.



Design (3/3)

- Notice now that each of our ‘Observers’ could be part of a system.
  - That is to say, ‘Observer 1’ can be for sounds
  - ‘Observer 2’ might be deriving from some IObserver to handle Physics
  - ‘Observer 3’ for logging
  - etc.
- Notice: Subject 1 will be able to ‘store’ different types of observers (Sound System, System Physics, Log, etc.)



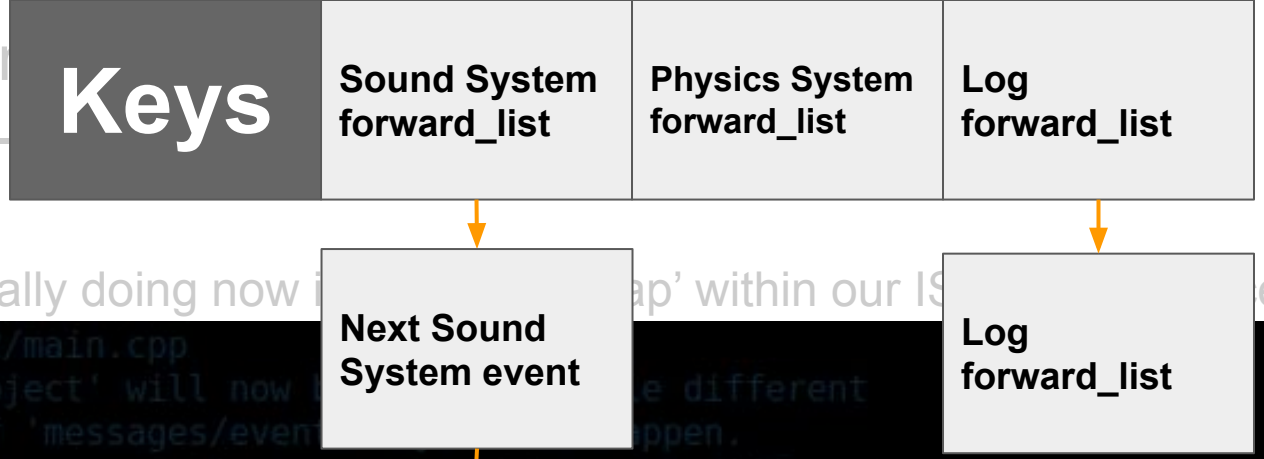
Observer Pattern 3rd implementation - ISubject

(1/2)

- So what we're really doing now is making a 'map' within our ISubject interface

```
33 // example2/main.cpp
34 // Our 'subject' will now be able to handle different
35 // types of 'messages/events/keys' that happen.
36 // This opens our subject up to working with different
37 // subsystems, but still staying decoupled.
38 class ISubject{
39     public:
40
41     private:
42         // Two typedefs here, one for the list of observers (same as before)
43         typedef std::forward_list<IObserver*>   ObserversList;
44         // The map is going to store 'int' as the key.
45         // The key is for each of the 'messages'
46         // Then, the value in our map is a forward_list as previous
47         typedef std::map<int, ObserversList>        ObserversMap;
48         ObserversMap mObservers;
49 };
```

Observer Pattern implementation - (2/2)



- So what we're really doing now i ap' within our IS e

```
// example2/main.cpp
// Our 'subject' will now  e different
// types of 'messages/event  ppen.
// This opens our subject up to w king with different
// subsystems, but still s .
class ISubject{
    public:

    private:
        // Two typedefs here, one for the list of observers (same as before)
        typedef std::forward_list<IObserver*>   ObserversList;
        // The map is going to store 'int' as the key.
        // The key is for each of the 'messages'
        // Then, the value in our map is a forward_list as previous
        typedef std::map<int, ObserversList>        ObserversMap;
        ObserversMap mObservers;
};
```

Observer Pattern 3rd implementation - ‘MessageTypes’

- The derived class from our ISubject, which in this case is ‘SomeSubject’ will determine the Message Types.
  - **Note:** `enum class > enum` [See core guideline] (I’m being simple here)
  - **Note**: Instead of an enum, we could instead add some sort of ‘AddType’ member function and use a different data structure to more dynamically handle this.

```
103 // example2/main.cpp
104 class SomeSubject : public ISubject{
105     public:
106         enum MessageTypes{PLAYSOUND, HANDLEPHYSICS, LOG};
107 };
```

Observer Pattern 3rd implementation - Adding Specific Observers

- So I'll show you how this changes how we add new observers.
  - Each 'message' is added to the 'correct' bucket in our map

```
43          // Message is the 'key' or 'event' that we want to handle.
44          // 'observer' is the observer we'll add to the events
45          virtual void AddObserver(int message, IObserver* observer){
46              // Check to see if our 'key' or 'event type' is in our map
47              auto it = mObservers.find(message);
48              if(it==mObservers.end()){
49                  // Since we did not find our message, we need
50                  // to construct a new list
51                  mObservers[message] = ObserversList();
52              }
53              // Add our observer to the appropriate 'bucket' of events.
54              mObservers[message].push_front(observer);
55          }
```

Observer Pattern 3rd implementation - Subject Notify

- I’ve added the ability to ‘NotifyAll()’
  - This iterates through every observer
- Notify now takes a parameter to a specific observer.
  - This time iterating through the entire map, or a specific key in our map.

```
105         // example2/main.cpp
106         // New function to notify everything
107         virtual void NotifyAll(){
108             // Search through every message type (our keys)
109             for(ObserversMap::iterator it = mObservers.begin(); it!=mObservers.end(); ++it){
110                 for(auto& o : mObservers[it->first]){
111                     o->OnNotify();
112                 }
113             }
114         }
115         // Notify a specific set of observers
116         virtual void Notify(int message){
117             for(auto& o: mObservers[message]){
118                 o->OnNotify();
119             }
120         }
```

Observer Pattern 3rd implementation - Multiple Observers

- Highlighted are two examples of different derived classes from IObserver
  - These can further be specialized to perform an action or activate a specific subsystem within ‘OnNotify’

```
18 // SoundEvent is derived from IObserver
19 class SoundEvent : public IObserver{
20     public:
21         explicit SoundEvent(const std::string& name) : mName(name){
22         }
23         // Do something 'physics like'
⚠          void OnNotify(){
25             std::cout << mName << " Sound engine did something" << std::endl;
26         }
27
28         std::string GetName() const override { return mName; }
29     private:
30         std::string mName;
31 };
32 // PhysicsEvent is derived from IObserver
33 class PhysicsEvent : public IObserver{
34     public:
35         explicit PhysicsEvent(const std::string& name) : mName(name){
36         }
37         // Do something 'physics like'
⚠          void OnNotify(){
39             std::cout << mName << " physics engine did something" << std::endl
40         }
41
42         std::string GetName() const override { return mName; }
43     private:
44         std::string mName;
45 };
```

Observer Pattern 3rd implementation - Usage (1/2)

- Now we can see the actual ‘usage’ of our observer.
  - Multiple types of observers (SoundEvent, PhysicsEvent, LogEvent) can subscribe(i.e. observe) to a ‘subject’
- When something important happens in our system, we can NotifyAll() or Notify a specific class of events.

```
138 // example2/main.cpp
139 int main(){
140
141     SomeSubject subject;
142     // Each of our 'watchers' are derived from IObserver.
143     // They are each event types that can 'subscribe' to
144     // a subject, and when they're notified, do something
145     // specific to that subsystem, without being coupled.
146     SoundEvent      watcher1("watcher1");
147     PhysicsEvent    watcher2("watcher2");
148     LogEvent        watcher3("watcher3");
149
150     subject.AddObserver(SomeSubject::PLAYSOUND,     &watcher1);
151     subject.AddObserver(SomeSubject::HANDLEPHYSICS, &watcher2);
152     subject.AddObserver(SomeSubject::LOG,           &watcher3);
153
154
155     // Send messages to every observer for this subject, regardless
156     // of the 'MessageTypes'
157     subject.NotifyAll();
158
159     // Notify any observer that has to do with 'sound'
160     subject.Notify(SomeSubject::PLAYSOUND);
161     std::cout << std::endl;
162     subject.RemoveObserver(SomeSubject::PLAYSOUND, &watcher1);
163
164     // Notice, no messages sent now
165     subject.Notify(SomeSubject::PLAYSOUND);
166
167     return 0;
168 }
```

Observer Pattern 3rd implementation - Usage (2/2)

-
```
mike:example2$ g++ -g main.cpp -o prog
mike:example2$ ./prog
watcher1 Sound engine did something
watcher2 physics engine did something
watcher3 log did something
watcher1 Sound engine did something

Goodbye watcher1
```

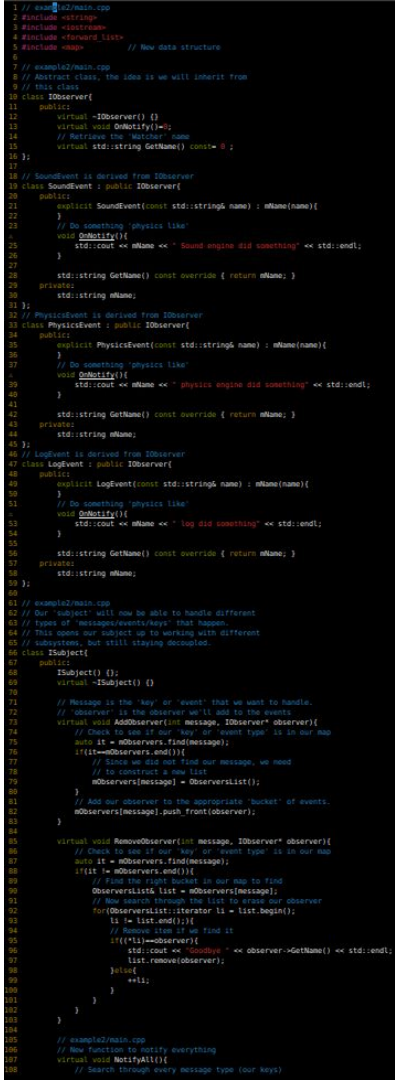
observe) to a subject

- When something important happens in our system, we can NotifyAll() or Notify a specific class of events.

```
138 // example2/main.cpp
139 int main(){
140
141     SomeSubject subject;
142     // Each of our 'watchers' are derived from IObserver.
143     // They are each event types that can 'subscribe' to
144     // a subject, and when they're notified, do something
145     // specific to that subsystem, without being coupled.
146     SoundEvent      watcher1("watcher1");
147     PhysicsEvent    watcher2("watcher2");
148     LogEvent        watcher3("watcher3");
149
150     subject.AddObserver(SomeSubject::PLAYSOUND,     &watcher1);
151     subject.AddObserver(SomeSubject::HANDLEPHYSICS, &watcher2);
152     subject.AddObserver(SomeSubject::LOG,           &watcher3);
153
154
155     // Send messages to every observer for this subject, regardless
156     // of the 'MessageTypes'
157     subject.NotifyAll();
158
159     // Notify any observer that has to do with 'sound'
160     subject.Notify(SomeSubject::PLAYSOUND);
161     std::cout << std::endl;
162     subject.RemoveObserver(SomeSubject::PLAYSOUND, &watcher1);
163
164     // Notice, no messages sent now
165     subject.Notify(SomeSubject::PLAYSOUND);
166
167     return 0;
168 }
```

3rd implementation - Pros and Cons? (1/2)

- So, no design pattern is **perfect**, computer science is about trade-offs.
  - (Question to the audience)
    - Is this pattern:
      - Flexible
      - Maintainable
      - Extensible

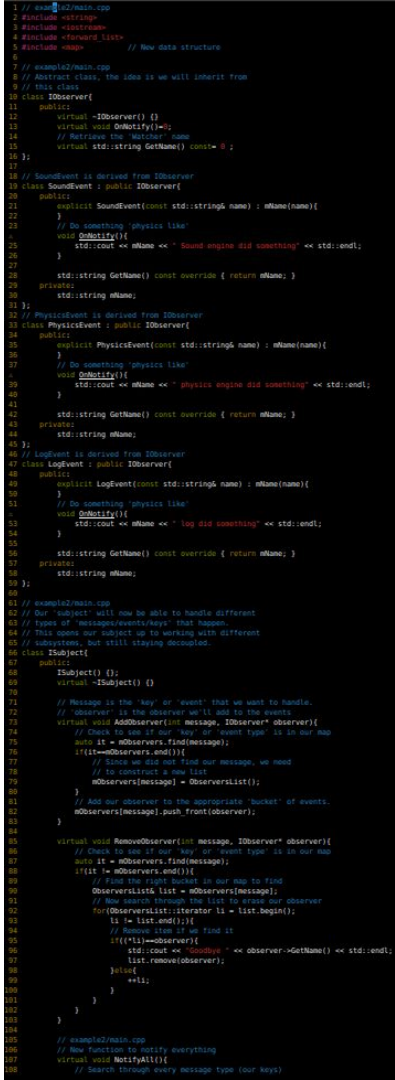


This one really does not fit on our screen, see the repo to see all of the code!

3rd implementation - Pros and Cons? (2/2)

- So, no design pattern is **perfect**, computer science is about trade-offs.
  - (Question to the audience)
    - Is this pattern:
      - Flexible -
        - Yes (e.g. NotifyAll(), Notify(specific subsystem), we can easily derive new IObserver classes)
      - Maintainable -
        - Yes, the heavy lifting is abstracted to each derivation of IObserver.
          - Subject just ‘iterates through’ the right set of Observers.
      - Extensible -
        - Yes, we showed this with the different types of IObservers

This one really does not fit on our screen, see the repo to see all of the code!



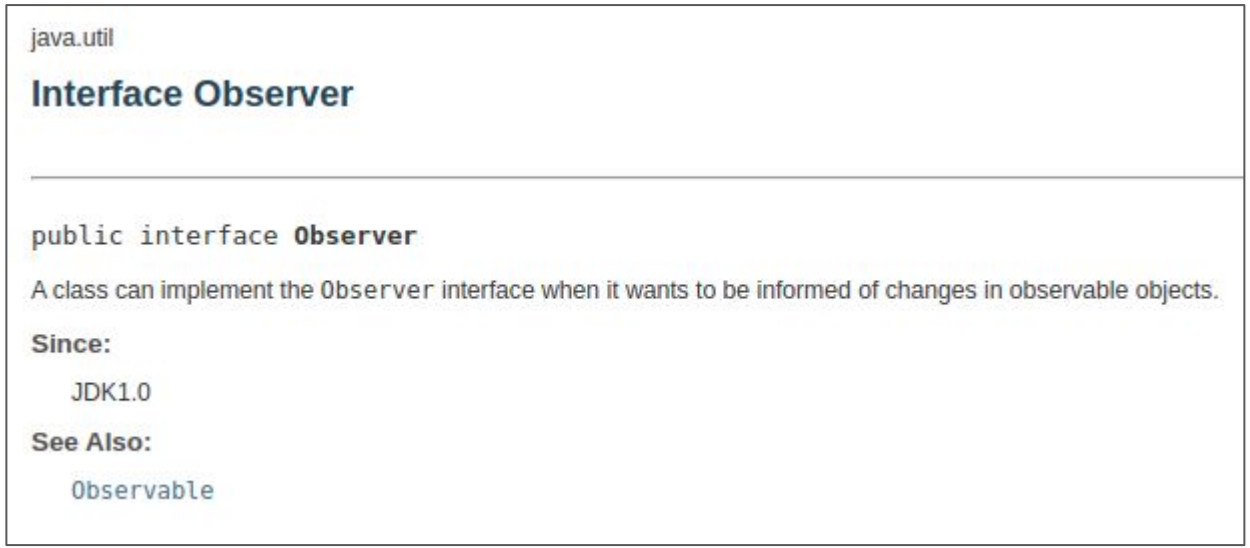
Is the pattern actually used?

Observer Pattern Usage

- I dug around for places that may be interesting for you to study usage of Observer -- so here's a few:
  - `grep -irn "observer" .`

Observer Pattern Usage

- Java !?!?
- Since JDK 1.0, Java.util has had observer because the pattern is so pervasive
- It may be worth just looking at the API if you're building a library



java.util

Interface Observer

`public interface Observer`

A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.

Since:

JDK1.0

See Also:

`Observable`

https://docs.oracle.com/javase/7/docs/api/java/util/Observer.html

Observer Pattern Usage

- Godot engine
  - Now my understanding is their implementation actually uses a ‘signals’ and ‘slots’ version of the observer pattern
  - https://docs.godotengine.org/en/3.1/getting_started/step_by_step/signals.html

```
platform/ios/joypad_ios.mm:48:          [observer finishObserving];
platform/ios/joypad_ios.mm:49:          observer = nil;
platform/ios/joypad_ios.mm:54:  if (observer) {
platform/ios/joypad_ios.mm:55:          [observer startProcessing];
platform/ios/joypad_ios.mm:59:@interface JoypadIOSObserver ()
platform/ios/joypad_ios.mm:108:                 addObserver:self
platform/ios/joypad_ios.mm:115:                 addObserver:self
platform/ios/joypad_ios.mm:123:         [[NSNotificationCenter defaultCenter] removeObserver:self];
platform/macos/os_macos.h:54:   CFRunLoopObserverRef pre_wait_observer;
platform/macos/os_macos.h:61:   static void pre_wait_observer_cb(CFRunLoopObserverRef p_observer, CFRunLoopActivity p_activiy, void *p_context);
platform/macos/display_server_macos.mm:475:void DisplayServerMacOS::_keyboard_layout_changed(CFNotificationCenterRef center, void *observer, CFStringRef name, const
ct, CFDictionaryRef user_info) {
platform/macos/display_server_macos.mm:3492:    CFNotificationCenterAddObserver(CFNotificationCenterGetDistributedCenter(),
platform/macos/display_server_macos.mm:3643:    CFNotificationCenterRemoveObserver(CFNotificationCenterGetDistributedCenter(), nullptr, kTISNotifySelectedKeyboardIn
anged, nullptr);
platform/macos/os_macos.mm:58:void OS_MacOS::pre_wait_observer_cb(CFRunLoopObserverRef p_observer, CFRunLoopActivity p_activiy, void *p_context) {
platform/macos/os_macos.mm:572: pre_wait_observer = CFRunLoopObserverCreate(kCFAllocatorDefault, kCFRunLoopBeforeWaiting, true, 0, &pre_wait_observer_cb, nullptr);
platform/macos/os_macos.mm:573: CFRunLoopAddObserver(CFRunLoopGetCurrent(), pre_wait_observer, kCFRunLoopCommonModes);
platform/macos/os_macos.mm:594: CFRunLoopRemoveObserver(CFRunLoopGetCurrent(), pre_wait_observer, kCFRunLoopCommonModes);
platform/macos/os_macos.mm:595: CFRelease(pre_wait_observer);
platform/macos/display_server_macos.h:195:      static void _keyboard_layout_changed(CFNotificationCenterRef center, void *observer, CFStringRef name, const void *c
```

Observer Pattern Usage

- Blender3D
  - https://github.com/blender/blender

```
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:118:    // \brief ObserverBase is the base class for the observers.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:120:    // ObserverBase is the abstract base class for the observers.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:124:    // The observer interface contains some pure virtual functions
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:128:    // The build() and clear() members are to notify the observer
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:131:    class ObserverBase {
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:139:      // Default constructor for ObserverBase.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:140:      ObserverBase() : _notifier(0) {}
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:142:      // \brief Constructor which attach the observer into notifier.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:144:      // Constructor which attach the observer into notifier.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:145:      ObserverBase(AlterationNotifier& nf) {
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:152:      // the other observer is attached to.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:153:      ObserverBase(const ObserverBase& copy) {
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:160:      virtual ~ObserverBase() {
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:166:      // \brief Attaches the observer into an AlterationNotifier.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:168:      // This member attaches the observer into an AlterationNotifier.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:173:      // \brief Detaches the observer into an AlterationNotifier.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:175:      // This member detaches the observer from an AlterationNotifier.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:187:      // Gives back true when the observer is attached into a notifier.
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:192:      ObserverBase& operator=(const ObserverBase& copy);
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:197:      typename std::list<ObserverBase*>::iterator _index;
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:199:      // \brief The member function to notificate the observer about an
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:202:      // The add() member function notificates the observer about an item
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:207:      // \brief The member function to notificate the observer about
/extern/quadriflow/3rd/lemon-1.3.1/lemon/bits/alteration_notifier.h:210:      // The add() member function notificates the observer about more item
```

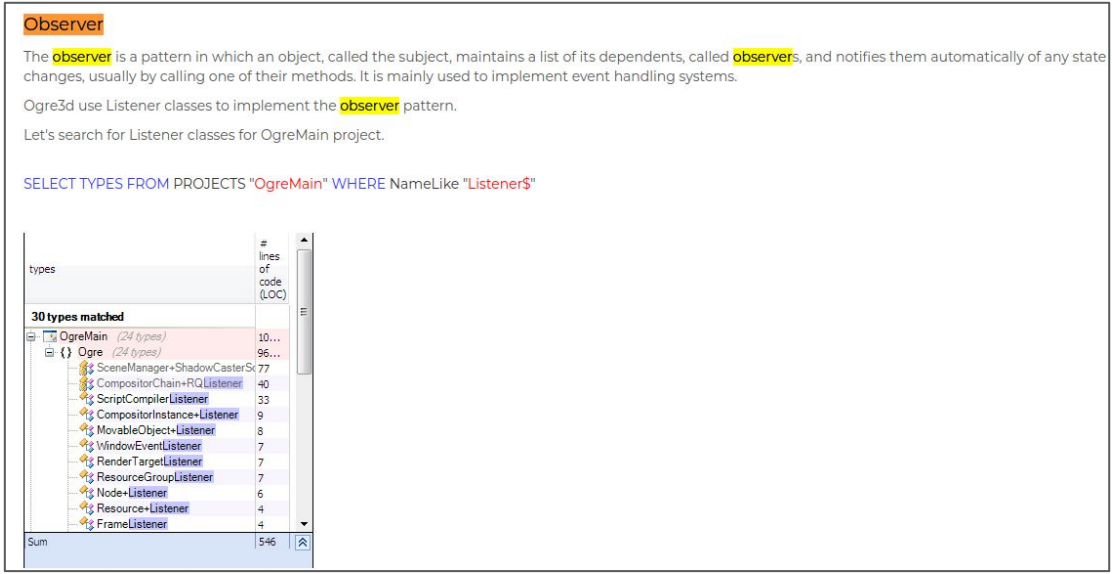
Observer Pattern Usage

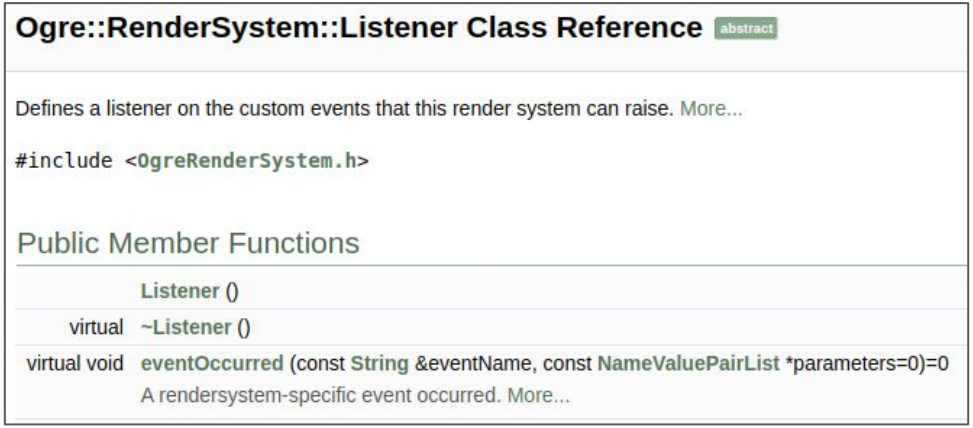
- Maya3D
  - https://download.autodesk.com/us/maya/2009help/API/_observer_8h-example.html
  - (2009) Little older example, but was interesting to see the implementation using ‘states’

```
//
//
class stateObserver : IObserveFX {
  public:

    stateObserver() : m_state(NULL), m_bs(NULL), m_ds(NULL), m_ss(NULL), m_ps(NULL), m_as(NULL), m_cs(NULL), m_fs(NULL), m_ptS(NULL) {};
    ~stateObserver() {};

    //these are the interface functions used by Ashli
    void setLightState(LightState state, int handle, const char* value);
    void setMaterialState(MaterialState state, int handle, const char* value);
    void setVertexRenderState(VertexRenderState state, int handle, const char* value);
    void setPixelRenderState(PixelRenderState state, int handle, const char* value);
    void setSamplerState(SamplerState state, int handle, const char* value);
    void setVertexShaderState(VertexShaderState state, int handle, const char* value);
    void setPixelShaderState(PixelShaderState state, int handle, const char* value);
    void setTextureState(TextureState state, int handle, const char* value);
    void setTransformState(TransformState state, int handle, const char* value);

    //this is used to configure a pass monitor
    void setPassMonitor( passState *state);
    void finalizePassMonitor();

  protected:

    //These functions parse the values and convert them to GL values
    bool isTrue( const char* value);
    bool isFalse( const char* value);
    GLenum compareFunc( const char* value);
    GLenum blendFactor( const char* value);
    GLenum stencilOp( const char* value);
    GLenum blendOp( const char* value);
    GLenum polyMode( const char* value);

    passState *m_state;

    blendStateItem *m_bs;
    depthStateItem *m_ds;
    stencilStateItem *m_ss;
    primitiveStateItem *m_ps;
    alphaStateItem *m_as;
    colorStateItem *m_cs;
    fogStateItem *m_fs;
    pointStateItem *m_ptS;
};
```

Observer Pattern Usage

- Ogre3D (Graphics Engine)
  - Neat study here using cppdepend
  - https://www.cppdepend.com/ogre3d
    - Implements with a ‘listener’ object.
    - https://www.ogre3d.org/docs/api/1.8/class_ogre_1_1_render_system_1_1_listener.html





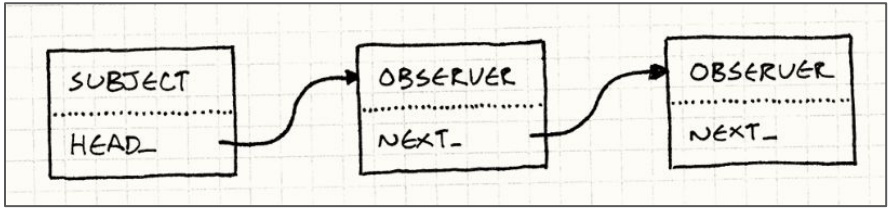
More Ideas on Observer Pattern

Some other *neat* ideas -

- Some C++ improvements
  - Replace raw pointers with `smart_ptr's`
    - This can actually become very important as the observers may be part of different subsystems that have to shutdown or restart.
  - As mentioned before, use enum class instead of enum
  - ***I have an example3.cpp**** in the repo showing this cleanup
- Consider using other data structures than `std::forward_list`
  - Observers may need to be ordered based on priority (i.e. use priority_queue)
  - Consider using '`unordered_map`' instead of map as shown in my examples for constant time average lookup.

Game Prog. Patterns Book

- Consider avoiding dynamic memory allocation at runtime with a fixed size of observers
  - Or otherwise consider a ‘linked’ implementation
    - See Game Programming patterns Book
    - Notice Subject and Observer each just store a pointer to the next observer



To implement this, first we’ll get rid of the array in `Subject` and replace it with a pointer to the head of the list of observers:

```
class Subject
{
  Subject()
  : head_(NULL)
  {}

  // Methods...
private:
  Observer* head_;
};
```

Then we’ll extend `Observer` with a pointer to the next observer in the list:

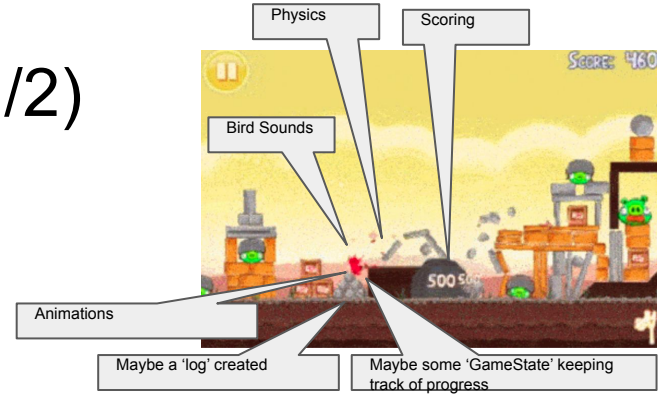
```
class Observer
{
  friend class Subject;

public:
  Observer()
  : next_(NULL)
  {}

  // Other stuff...
private:
  Observer* next_;
};
```

Revisiting Angry Birds

A Summary of what we have learned

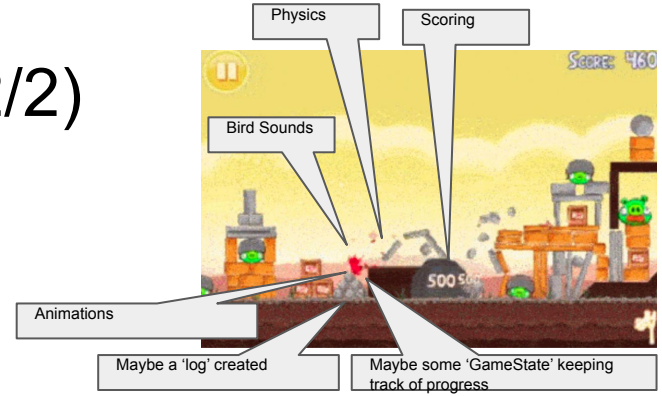
(Question for Audience) Thoughts? (1/2)

- I’m certain that the Observer pattern could help us here.
  - But what do folks think about the ‘scaling’ in this context?



(Question for Audience) Thoughts? (2/2)

- I’m certain that the Observer pattern could help us here.
  - But what do folks think about the ‘scaling’ in this context?
  - We do have a problem if too many events are registered per object -- i.e. we have to iterate one at a time through our objects.
    - Possible Solution: Push each event into some queue, maybe handle that queue in another thread or asynchronously depending on event type
      - Here’s a link to event queue if you can totally decouple subject from observer.
      - (Also may open up to threads more)
    - Possible Solution: Perhaps other models, e.g. Actor Model [More]



Conclusion

A Summary of what we have learned

Summary of what we have learned and should learn next

- We’ve built out the observer pattern from a very simple implementation to something reasonably useable
- The observer pattern isn’t perfect, but it’s a good candidate for 1 to many interactions.
    - The pattern is extensible, maintainable, and flexible -- there’s a reason it’s quite popular.
- There exist many variations of the observer pattern--so take a look at some codebases to see different implementations

Going Further

Some things that may be useful for learning more design patterns

example3/main.cpp

- I've included in the repo a more modernized code using smart pointers and usage of enum class
  - Used `std::unordered_map` instead of `std::map` (assume we don't care about order)
  - I used shared_ptr<IObserver> instead of raw pointers
    - **Note**: You'll need to think a bit about where you want the ownership to be (i.e. you could possibly move ownership and use a unique_ptr instead)
    - **Note**: There will be some CoreCPP talk from 2022 talking about the dangers of shared_ptr and performance.

```
126     private:
127         // Two typedefs here, one for the list of observers (same as before)
128         typedef std::forward_list<std::shared_ptr<IObserver>>   ObserversList;
129         // The unordered_map is going to store 'int' as the key.
130         // The key is for each of the 'messages'
131         // Then, the value in our unordered_map is a forward_list as previous
132         typedef std::unordered_map<MessageTypes, ObserversList>         ObserversMap;
133         ObserversMap mObservers;
```

Resources

- (Note: These are listed in the order that I would watch these)
- Game Programming Patterns part 7.1 - (Reading) Observer Pattern
  - https://www.youtube.com/watch?v=ryBEm0_5Y-g
- https://apibook.com/
  - Check out Martin Reddy's book for another nice implementation
  - I learned a few tricks and extended the implementation.
- Tony Van Eerd: Thread-safe Observer Pattern - You're doing it wrong
  - https://www.youtube.com/watch?v=RVvVQpIy6zc

More General Design Patterns

- Videos
  - C++ Design Patterns: From C++03 to C++17 - Fedor Pikus - CppCon 2019
    - Overview of evolution of design patterns
  - Introduction to Design Patterns (Back to Basics Track CPPCON 2020)
  - The Factory Pattern (Software Design Track) - Mike Shah - CppCon 2021
  - And many more!
    - https://www.youtube.com/results?search_query=cppcon+design+patterns
- Books
  - API Design for C++
  - Game Programming Patterns
  - Modern C++ Design

Thank you!

The Observer Design Pattern

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

16:45 - 17:45, Thur, 15th September 2022

60 minutes | Introductory Audience

MIKE SHAH

Cppcon
The C++ Conference

2022 September 12th-16th