

Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This slideset and associated source code may not be distributed without prior written notice

Please do not redistribute slides/source without prior written permission.

CFTT
College of Farabi Tech Talks

نشست تخصصی آشنایی با

Design Patterns

الگوهای طراحی

Mike Shah
Associate Teaching Professor of Computer Sciences
Khoury College Northeastern University
Boston, Massachusetts, USA

دانشیار علوم کامپیوتر
دانشگاه نورس ایسترن، کالج خوری
بوستون، ایالات متحده آمریکا



Computer Engineering Scientific Association of University of Tehran
انجمن علمی دانشجویی مهندسی کامپیوتر دانشکده‌گان فارابی دانشگاه تهران


Date	Time	زمان	تاریخ
Thursday January 4th	17:30 GMT	۳۱ ساعت	پنجشنبه ۱۴ دی ماه



 Cesa_ut  @Cesa_ut  @Cesa_ut

Thinking about Design with Patterns

Visitor Pattern -- in C++ with Mike Shah

Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
 **YouTube**
www.youtube.com/c/MikeShah

17:30 - 19:00 Thur, January 4, 2024

~60-90 minutes | Introductory
Audience

Originally, I was going to give a pretty 'cookie cutter' talk on design patterns -- and to some extent that holds.

However -- as I looked at some design patterns, I decided some of those patterns had more interesting trade-offs.

Mike Shah

Associate Teaching Professor of Computer Sciences
Khoury College Northeastern University
Boston, Massachusetts, USA



دانشیار علوم کامپیوتر
دانشگاه نورس ایسترن، کالج خوری
بوسطن، ایالات متحده آمریکا



Computer Engineering Scientific Association of University of Tehran, Iran
انجمن علمی دانشجویی مهندسی کامپیوتر دانشکده‌گان فارابی دانشگاه تهران

Date	Time	زمان	تاریخ
Thursday January 4th	17:30 GMT	۳۱ ساعت	پنجشنبه ۱۴ دی ماه



Cesa_ut @Cesa_ut @Cesa_ut

17:30 - 19:00 Thur, January 4, 2024

~60-90 minutes | Introductory
Audience

Thinking about Design with Patterns

Visitor Pattern -- in C++ with Mike Shah

Social: [@MichaelShah](#)

Web: [mshah.io](#)


Courses: [courses.mshah.io](#)



www.youtube.com/c/MikeShah

One of the Best Titled Talks

- (The talk is also excellent)
- But the part I want you to focus on is -- ‘The Minds of People’
 - That’s you -- the students, engineers, faculty, today
 - So anything I think about, any design choice we make, is something that we need to think about.
 - Software engineering is about making trade-offs, and engineering requires critical thinking applied to specific domains.



Cppcon 2019
The C++ Conference
cppcon.org

Timeline

- 1990s: OOP
 - Inheritance and virtuals, wheee!...
- 2000s: Generic Programming
 - Iterators and algos, wheee!...
- 2020s: Design By Introspection
 - Inspect and Customize Everything Everywhere, wheee?...

Andrei Alexandrescu

**Speed Is Found In
The Minds Of People**

Link on YouTube:
<https://www.youtube.com/watch?v=FJJTYQYB1JQ>

Your Tour Guide for Today

by Mike Shah

- **Associate Teaching Professor** at Northeastern University in Boston, Massachusetts.
 - I **love** teaching: courses in computer systems, computer graphics, geometry, and game engine development.
 - My **research** is divided into computer graphics (geometry) and software engineering (software analysis and visualization tools).
- I do **consulting** and **technical training** on modern C++, DLang, Concurrency, OpenGL, and Vulkan projects
 - Usually graphics or games related -- e.g. Building 3D application plugins
- Outside of work: guitar, running/weights, traveling and cooking are fun to talk about



Web

www.mshah.io



YouTube
<https://www.youtube.com/c/MikeShah>

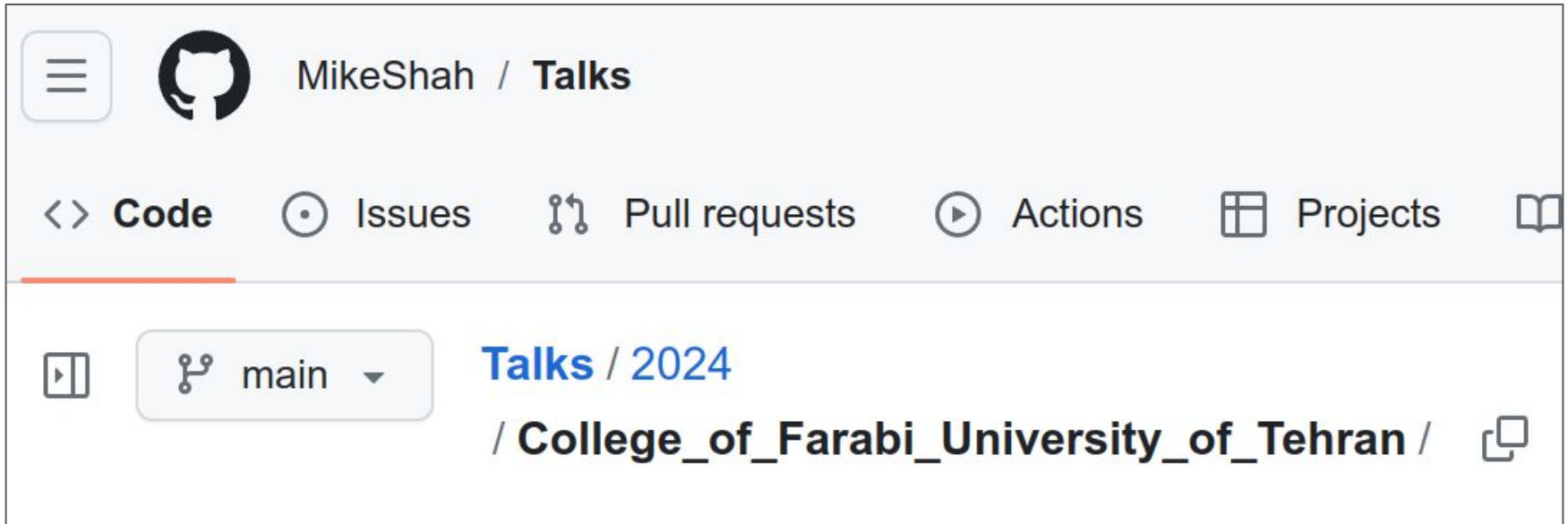
Non-Academic Courses

courses.mshah.io

Code for the talk

- Located here:

https://github.com/MikeShah/Talks/tree/main/2024/College_of_Farabi_University



The abstract that you read and enticed you to join me is here!

Abstract

Today I am going to be providing an introduction to design patterns, and going into depth on a useful behavioral design patterns--the **Visitor Pattern**.

The proper use of design patterns can greatly simplify your codebase, making your software more flexible, maintainable, and extensible.

We will also look at examples of code in open source projects to understand otherwise how these patterns are deployed.

Students will leave this talk with the knowledge to go forward and implement the both the state and the visitor pattern at a minimum, and otherwise have a better idea of when either of these patterns may provide the right trade-off in your during software development.



Quick Review of Programming Paradigms

Establishing common ground -- minimum skills needed to proceed forward

C++ as a Multi-paradigm language

- One of the great advantages of a language like C++ (and many others -- e.g. Java, DLang, etc.) is that you can use multiple programming paradigms.
 - This allows us to think about programming in different ways to solve different problems.
- Likely you have learned (or will one day learn) one of these three programming styles:
 - Procedural Programming
 - Object-Oriented Programming
 - Functional Programming
 - (Of course other paradigms exist: Generic, Data-Oriented, structured, etc.)
- Very briefly, I want to discuss a few features of each paradigm which are used in each
 - In some ways -- you can think of each paradigm as a 'design pattern' as well for how you approach computation.



<https://www.stroustrup.com/>



Procedural Programming [[wiki](#)]

A series of computational steps carried out by functions (i.e. procedures)

Basics of Programming - Procedural Style (1/2)

- Procedural Style programming
 - functions
 - Input of zero or one arguments to produce zero or more values
 - Languages also support structured data
 - e.g. 'struct'
 - e.g. 'union'
 - e.g. enum and enum class
 - e.g. arrays
 - if/else if/else conditional statements
 - switch statements
 - What's the advantage of a switch-statement versus an if-else statement?

Basics of Programming - Procedural Style (2/2)

- Example with enums (top-right, bottom-right)
 - Allow us to 'switch' on data
 - In object-oriented programming we instead use 'polymorphism' (coming up)
 - (Bottom-left) An example of 'events' that we might want to further store in an enum
 - Uses a 'union' of 'structs' to hold data.

```
18 // Special class type holding one data member at a time.
19 // Space allocated for largest type, and byte-aligned on architecture.
20 union Event{
21     // Note: 'type' always first member to ensure we can record state event
22     int type;
23     // Observe that we can 'nest' types to scope them.
24     struct mouse_event{
25         int type;
26         bool left;
27         bool middle;
28         bool right;
29     };
30     struct key_event{
31         int type;
32         char states[255];
33     };
34 };
```

```
1 /// @file procedural.cpp
2 /// g++ -std=c++20 procedural.cpp -o prog && ./prog
3 #include <iostream>
4
5 // constants represented as 'integers' types
6 enum SHAPE{SQUARE, CIRCLE, PENTAGON, END};
7
8 // Strongly typed, and scoped -- no type conversions
9 enum class STATE{RUNNING, OFF, PAUSED, ERROR};
10
11 void function(SHAPE s){
12     if(s==SQUARE){
13
14     }else if(s==CIRCLE){
15         std::cout << "Circle!\n";
16     }else{
17     }
18 }
19
20 void function(STATE s){
21     /// Switch statement -- note: compile with -Wall in case you
22     /// miss a state!
23     switch(s){
24         case STATE::RUNNING:
25             std::cout << "Running!\n";
26             break;
27         case STATE::PAUSED:
28         case STATE::OFF:
29         case STATE::ERROR:
30             default:
31                 break;
32     }
33 }
```



Object-Oriented Programming [[wiki](#)]

Objects (containing data+code) are main mechanism of computation

Basics of Programming - Object-Oriented Programming (OOP) (1/5)

- A common description of Object-Oriented language features would likely include the terms:
 - **Encapsulation**
 - Associate data (attributes) and behaviors (functions) together
 - This is how we create 'objects'
 - Thus -- **Abstraction** (Objects and Classes)
 - Other ways of managing state involve -- **Information hiding** (public/private)
 - **Inheritance**
 - Ability to derive new types with a 'is-a' relationship
 - Can use an 'interface' to derive new types.
 - **Polymorphism**
 - Not necessarily tied to inheritance hierarchies
 - Functional polymorphism (polymorphic function) -- function overloading
 - Subtyping -- Derive new types in a hierarchy

Basics of Programming - Object-Oriented Programming (OOP) (2/5)

- **Encapsulation**

- Associate data (attributes) and behaviors (functions) together
 - This is how we create 'objects'
 - Thus -- **Abstraction** (Objects and Classes)
 - Other ways of managing state involve -- **Information hiding** (public/private)

```
1 /// @file encapsulation.cpp
2 /// g++ -std=c++20 encapsulation.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5
6 class Actor{
7     public: // Access Specifier (information hiding modifiers)
8
9         // Constructor - action performed creating object
10        Actor(std::string name) : mName(name){
11
12        }
13
14        /// Destructor - action performed on destruction
15        /// (i.e. leaving scope, or explicitly deleted).
16        /// Not explicitly called otherwise.
17        ~Actor(){
18        }
19
20        /// A public member function that acts on a specific instance
21        /// of an object.
22        void PrintName(){
23            std::cout << "Actor mName=" << mName << std::endl;
24        }
25
26    private:
27        // Data associated with the 'Actor'
28        std::string mName;
29 };
30
31
32 int main(){
33     Actor s("Golshifteh Farahani");
34     s.PrintName();
35
36     return 0;
37 }
```

Basics of Programming - Object-Oriented Programming (OOP) (3/5)

- **Inheritance**

- Ability to derive new types with a 'is-a' relationship
- Can use an 'interface' to derive new types.

```
1 /// @file inheritance.cpp
2 /// g++ -std=c++20 inheritance.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5
6 class Actor{
7     public: // Access Specifier (information hiding modifiers)
8
9         Actor(std::string name) : mName(name){ }
10        /// Note: Destructor needs to be marked 'virtual' in inheritance, such
11        ///      to ensure that it is called from derived classes.
12        virtual ~Actor(){ }
13        void PrintName(){
14            std::cout << "Actor mName=" << mName << std::endl;
15        }
16    private:
17        /// Data associated with the 'Actor'
18        std::string mName;
19 };
20
21 /// - A new 'derived type' from 'Actor' (Actor serves as the Base class)
22 /// - Inherits 'public' functionality of Actor class
23 /// - Is a 'type of' Actor.
24 /// - Can expand on the functionality of 'Actor'
25 class MethodActor : public Actor{
26     public:
27         MethodActor(std::string name) : Actor(name){
28         }
29         void PracticeMethod(){
30             std::cout << "MethodActor::PracticeMethod()\n";
31         }
32 };
33
34
35 int main(){
36
37     MethodActor* s = new MethodActor("Golshifteh Farahani");
38     s->PrintName();
39     s->PracticeMethod();
40
41     delete s;
42     return 0;
43 }
```


Basics of Programming - Object-Oriented Programming (OOP) (4/5)

● Inheritance

- Ability to derive new types with a ‘**is-a**’ relationship
- Can use an ‘interface’ to derive new types.
 - Note: A function that is ‘**virtual**’ can provide a default implementation.
 - Note: A function that is ‘**purely virtual**’ must provide an implementation
 - Note: The ‘**destructor**’ of a base class should be marked ‘**virtual**’ -- in order to ensure it is called.

```
1 /// @file interface.cpp
2 /// g++ -std=c++20 interface.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5
6 /// Serves as an 'interface' for the 'Base class'
7 /// Can force functionality to be implemented
8 class IActor{
9     public: // Access Specifier (information hiding modifiers)
10
11         IActor(std::string name) : mName(name){ }
12         virtual ~IActor(){ }
13
14         /// Purely virtual function -- i.e. it must be implemented in derived class
15         virtual void Practice() = 0;
16
17     private:
18         // Data associated with the 'Actor'
19         std::string mName;
20 };
21
22 class MethodActor : public IActor{
23     public:
24         MethodActor(std::string name) : IActor(name){
25         }
26         /// NOTE: Should mark functions with 'override' to ensure that you are in fact
27         /// overriding the purely virtual function.
28         void Practice() override{
29             std::cout << "MethodActor::Practice()\n";
30         }
31 };
32
33 class ClassicalActor : public IActor{
34     public:
35         ClassicalActor(std::string name) : IActor(name){
36         }
37         /// NOTE: Should mark functions with 'override' to ensure that you are in fact
38         /// overriding the purely virtual function.
39         void Practice() override{
40             std::cout << "ClassicalActor::Practice()\n";
41         }
42 };
```

Basics of Programming - Object-Oriented Programming (OOP) (5/5)

● Polymorphism

- Not necessarily tied to inheritance hierarchies
 - Functional polymorphism
(polymorphic function) -- function overloading
- Subtyping -- Derive new types in a hierarchy
 - Purpose is to be able to 'select at type' in a hierarchy at run-time.
 - i.e. **run-time polymorphism**

```
1 /// @file polymorphism.cpp
2 /// g++ -std=c++20 polymorphism.cpp -o prog && ./prog

45 int main(){
46
47     /// IActor is the most 'generic' form of a 'type of' Actor.
48     /// So we can allocate a 'MethodActor' or 'ClassicalActor'
49     /// The 'virtual calls' will be routed correctly.
50     /// This is known as 'dynamic dispatch'
51     IActor* s = new MethodActor("Golshifteh Farahani");
52     s->Practice();
53
54     delete s;
55
56     s = new ClassicalActor("Golshifteh Farahani");
57     s->Practice();
58
59     delete s;
60
61     return 0;
62 }
```

```
mike:visitor$ g++ -std=c++20 polymorphism.cpp -o prog && ./prog
MethodActor::Practice()
ClassicalActor::Practice()
```



Functional Programming [[wiki](#)]

Programs are made by applying and composing functions

Basics of Programming - Functional Programming (FP) (1/2)

- Functions
 - lambda's
 - Higher-order functions
 - Passing functions as arguments to compose computation
 - Inputs generate the same outputs
 - potential for memoization/caching
 - value semantics -- i.e. data is copied
- Variables
 - Avoiding mutation
 - 'const' and 'constexpr'
 - Type-safety
 - e.g. [`std::variant`](#) ('tagged union' in c++ 17)
- Pattern Matching

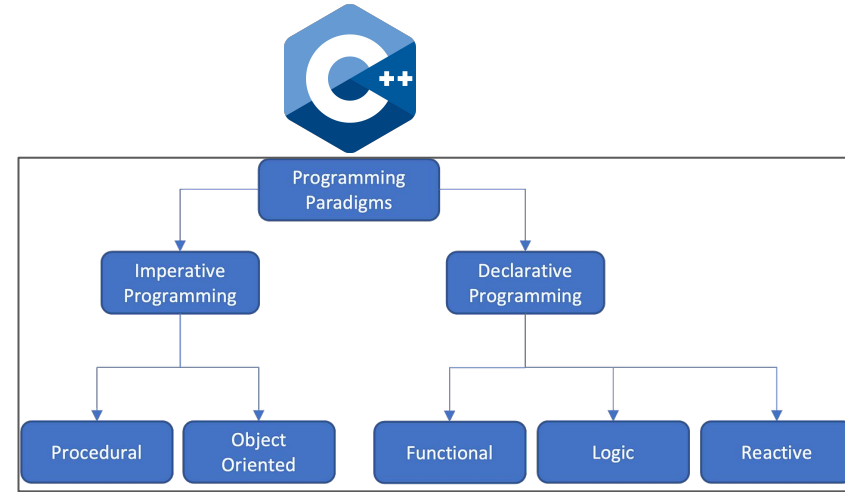
Basics of Programming - Functional Programming (FP) (2/2)

- Note: Here a few such features demonstrated
 - Functions
 - lambda's
 - Higher-order functions
 - Passing functions as arguments to compose computation
 - Inputs generate the same outputs
 - potential for memoization/caching
 - More frequently utilizes value semantics -- i.e. data is copied
 - Variables
 - Avoiding mutation
 - 'const' and 'constexpr'

```
1 // @file functional.cpp
2 // g++ -std=c++20 functional.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5 #include <functional>
6
7 struct Actor{
8     public:
9     Actor(std::string name) : mName(name){}
10    const std::string mName;
11 };
12
13 // Higher-order function
14 void Agenda(std::function<void(const Actor&)> f, const Actor& a){
15
16     // Call function with the actor argument
17     f(a);
18 }
19
20 int main(){
21
22     // Define a lambda function (Implemented as a 'functor' behind the scenes)
23     auto practice = [](const Actor& s){
24         std::cout << s.mName << " ::practice()\n";
25     };
26     auto dance = [](const Actor& s){
27         std::cout << s.mName << " ::dance()\n";
28     };
29
30     // Instantiate data
31     Actor a = Actor("Mike");
32
33     // Pass function as a parameter
34     Agenda(practice, a);
35     Agenda(dance, a);
36
37     return 0;
38 }
```

C++ as a Multi-paradigm language

- Okay -- so you've refreshed on some different 'paradigms' of programming -- specifically in C++
 - (Procedural Programming, Object-Oriented Programming, Functional Programming)
- You can use some of them as 'tools' or also 'patterns'
 - Think throughout this talk how you might implement our pattern in different styles



https://miro.medium.com/v2/resize:fit:1400/1*FFOmWawwaBGQi4hHE-sn6A.png

A Real World Application*



*Disclaimer: I did not work on this project -- I think it serves as an educational use case!

A real task -- in gaming

- Let's assume I have hired you to work at a video game company
 - You're going to be a contractor/consultant joining the project
 - I'm going to ask you to work on extending the gameplay code.

Question to Audience: (1/2)

- Who knows which game this is?



Question to Audience: (2/2)

- Who knows which game this is?
 - Lord of the Rings -- Shadows of Mordor! [\[wiki\]](#)



Game Background

- So to give you a little background, a big part of the game is combat.
 - You're of course trying to complete various missions and battling different types of enemies based



The game takes place between 'The Hobbit' and 'Lord of the Rings' Books for those who want to know :)

Enemies

- Here's are some example characters that you might encounter and battle through your journey
 - They have different combat strengths, personalities, dialogues, attributes, etc.



Vast number of enemies

- One thing I find fascinating about this game, is there are a variety of characters for which you have to battle.
 - To some degree, as previously mentioned, they have many different behaviors.



Game Development

- If you read through the [\[wiki\]](#), something interesting is that development of the project took place between 2011-2014.
 - So you can imagine this is a pretty significant development time, with hundreds of thousands or perhaps millions of lines of code



https://upload.wikimedia.org/wikipedia/en/c/cb/Shadow_of_Mordor_screenshot.jpg

(Aside) Game Development - Nemesis System

- In particular -- the 'Nemesis System' for their Artificial Intelligence was quite advanced for the time
 - https://en.wikipedia.org/wiki/Middle-earth:_Shadow_of_Mordor#Nemesis_system



https://upload.wikimedia.org/wikipedia/en/6/61/Middle-Earth_Shadow_of_Mordor_Nemesis_System.jpg

Designing an 'Enemy' Class (1/5)

- It's sort of interesting to think about designing an enemy class
 - **Question to Audience:** What attributes do you see that would be part of the class?



Designing an 'Enemy' Class (2/5)

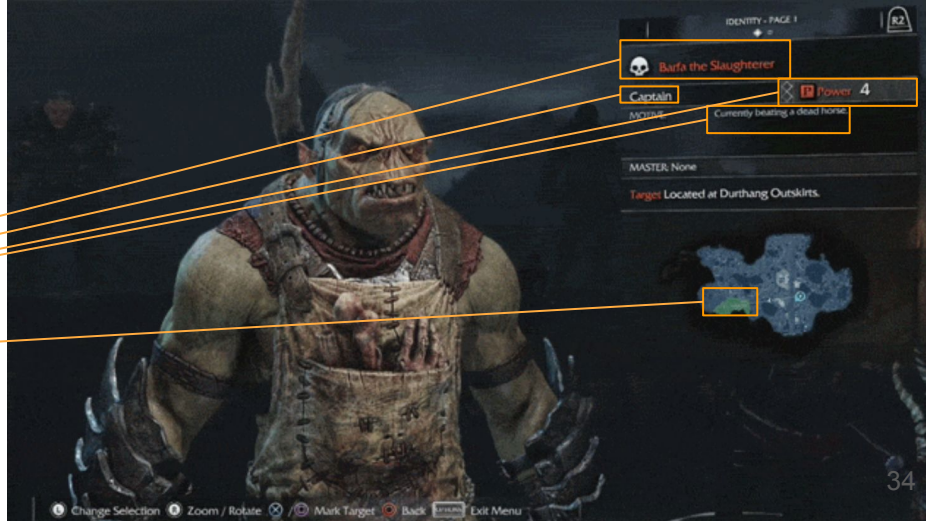
- It's sort of interesting to think about designing an enemy class
 - Would it be a class called 'Enemy'?
 - Would it be derived from something more generic? (e.g. Character)
 - What attributes do we need?
 - e.g. I see 'name', 'Power', 'rank', 'motive', 'position in world'



Designing an 'Enemy' Class (3/5)

- It's sort of interesting to think about designing an enemy class
 - Would it be a class called 'Enemy'?
 - Would it be derived from something more generic? (e.g. Character)
 - What attributes do we need?
 - e.g. I see 'name', 'Power', 'rank', 'motive', 'position in world'

```
1 // @file orc.cpp
2 // g++ -std=c++20 orc.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5
6 struct Motive{};
7
8 struct Position{
9     float x,y,z;
10 };
11
12 class orc{
13 public:
14     orc(){}
15
16 private:
17     int power;
18     std::string name;
19     int rank;
20     Motive m;
21     Position p;
22 };
23
24 int main(){
25     return 0;
26 }
```

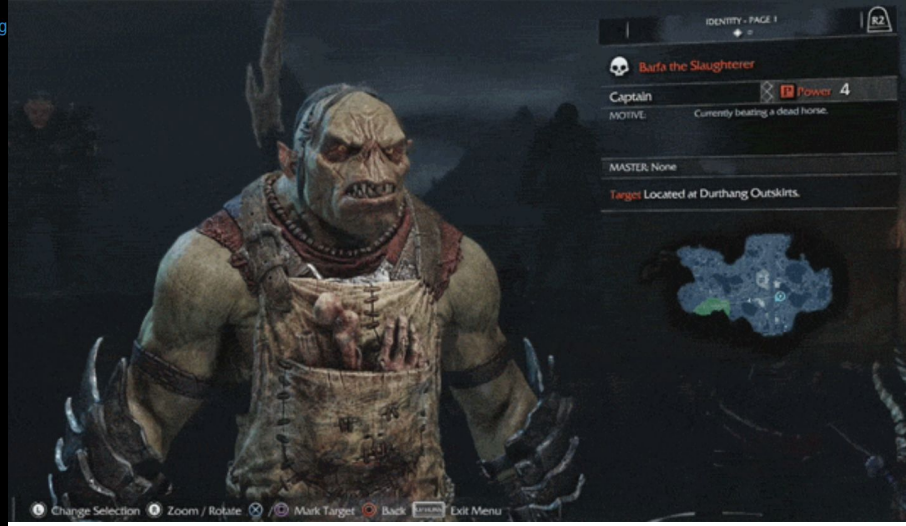
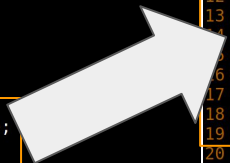


Designing an 'Enemy' Class (4/5)

- Now we can probably improve our 'orc' design by making the attributes a little more flexible over the lifetime of a project.
 - Observe we move attributes into a single struct.
 - Our 'data' becomes trivial easy to understand in the 'orc' class
 - We also open up opportunity for the pimpl idiom for ABI Stability for Orcs

```
1 // @file orc.cpp
2 // g++ -std=c++20 orc.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5
6 struct Motive{};
7
8 struct Position{
9     float x,y,z;
10 };
11
12 class orc{
13 public:
14     orc(){}
15
16 private:
17     int power;
18     std::string name;
19     int rank;
20     Motive m;
21     Position p;
22 };
23
24 int main(){
25     return 0;
26 }
```

```
1 // @file orcl.cpp
2 // g++ -std=c++20 orcl.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5
6 struct Position{
7     float x,y,z;
8 };
9
10 struct Attributes{
11     // Perhaps 'encapsulate' 'Motive'
12     struct Motive{};
13
14     int power;
15     std::string name;
16     int rank;
17     Motive m;
18     Position p;
19 };
20
21 class orc{
22 public:
23     orc(){}
24
25 private:
26     Attributes attr;
27 }
```

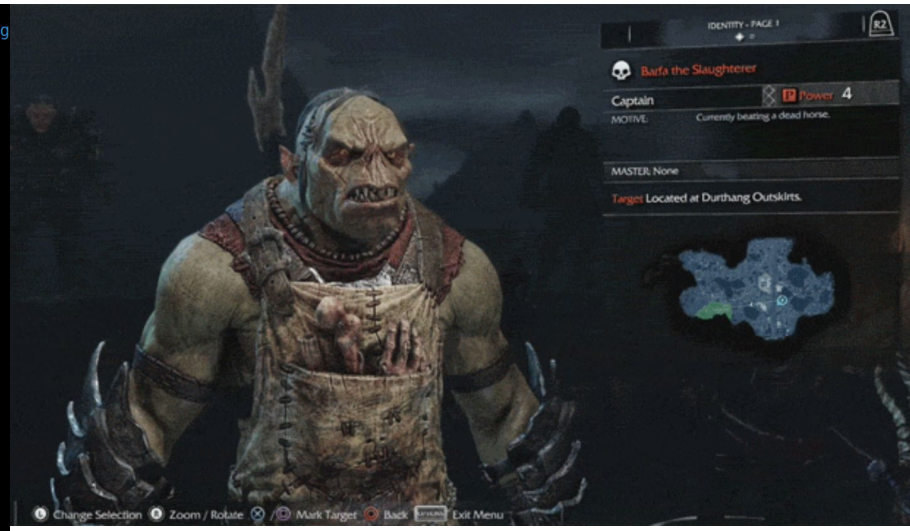


Designing an 'Enemy' Class (5/5)

- Just this simple change with the attributes should get you thinking about the 'trade-offs' of such design
 - But interestingly -- what if we could sort of do the same thing with the 'behaviors' (i.e. member functions) of this class.
 - That is -- to move or 'extract out' all or some of the behaviors? -- **hold onto that thought!**

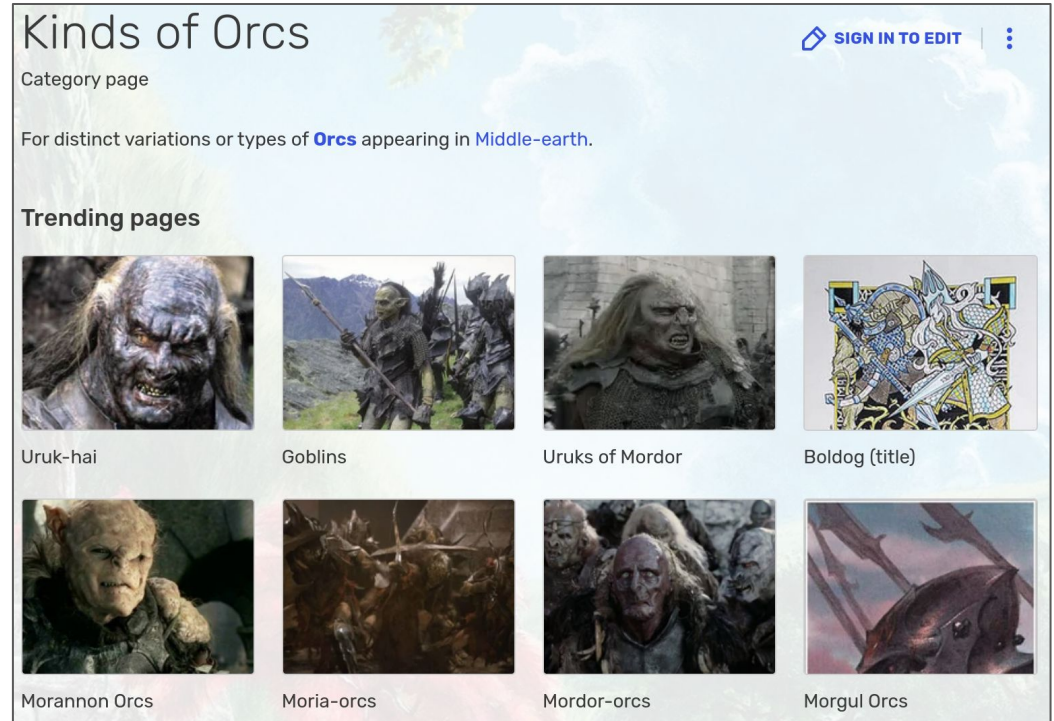
```
1 // @file orc.cpp
2 // g++ -std=c++20 orc.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5
6 struct Motive{};
7
8 struct Position{
9     float x,y,z;
10 };
11
12 class orc{
13 public:
14     orc(){}
15
16 private:
17     int power;
18     std::string name;
19     int rank;
20     Motive m;
21     Position p;
22 };
23
24 int main(){
25     return 0;
26 }
```

```
1 // @file orcl.cpp
2 // g++ -std=c++20 orcl.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5
6 struct Position{
7     float x,y,z;
8 };
9
10 struct Attributes{
11     // Perhaps 'encapsulate' 'Motive'
12     struct Motive{};
13
14     int power;
15     std::string name;
16     int rank;
17     Motive m;
18     Position p;
19 };
20
21 class orc{
22 public:
23     orc(){}
24
25 private:
26     Attributes attr;
27 }
```



Our 'Cast' of Enemies to battle with (1/2)

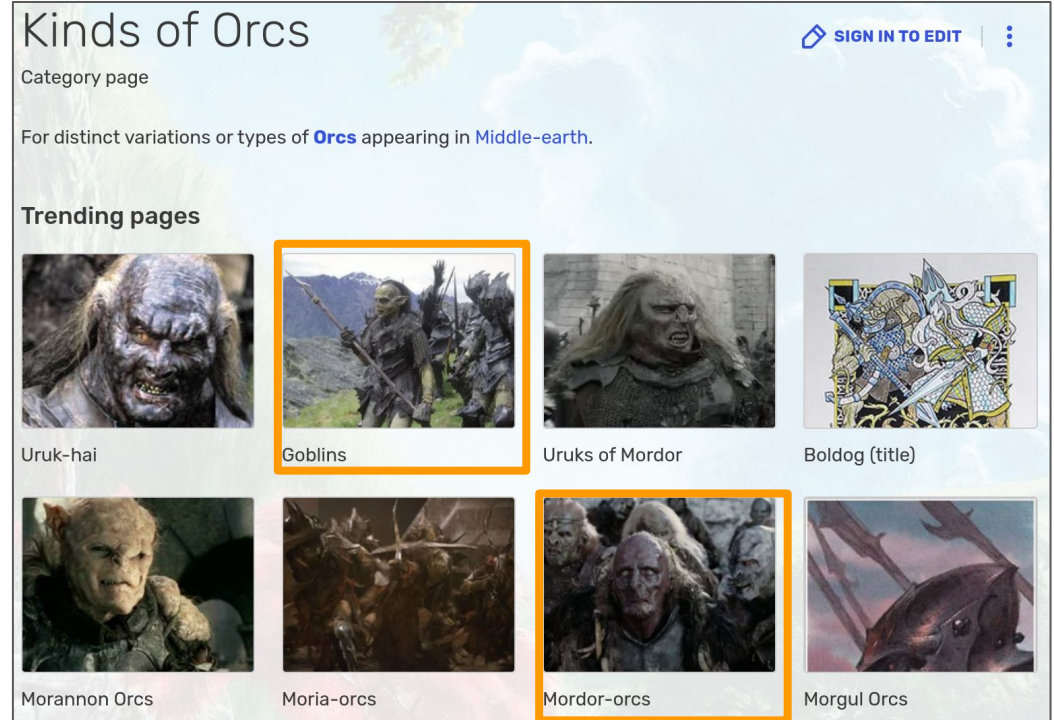
- Our design problem may extend further as there are different 'Kinds of Orcs' and enemies.
- Let's now assume we've got some 'set' of enemies here.
 - The set is 'fixed' (or closed) in that we know we have 8 'enemy' types.



https://lotr.fandom.com/wiki/Category:Kinds_of_Orcs

Our 'Cast' of Enemies to battle with (2/2)

- Our design problem may extend further as there are different 'Kinds of Orcs' and enemies.
- Let's now assume we've got some 'set' of enemies here.
 - The set is 'fixed' (or closed) in that we know we have 8 'enemy' types.
- For code illustration -- we'll focus on '2' in order to maximize the code that fits on a slide.



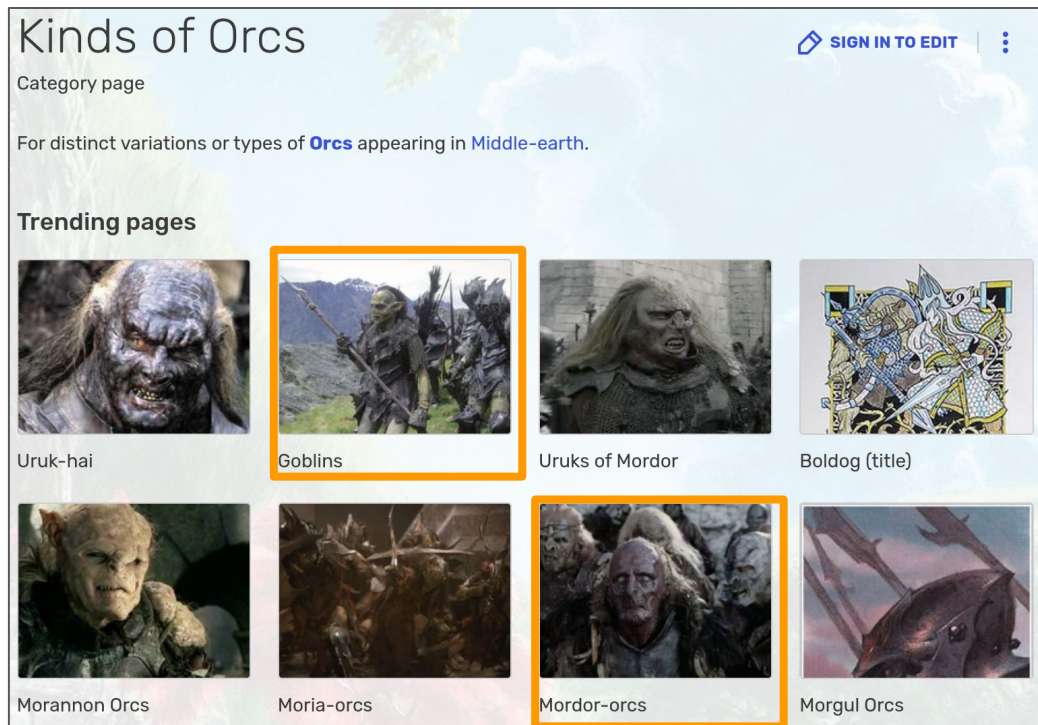
https://lotr.fandom.com/wiki/Category:Kinds_of_Orcs

What Problem Am I Trying to Solve? (1/2)

Problem: I may not know all of the behaviors that I want for these objects -- especially if they are in the same hierarchy

- A. It can be difficult to figure out what functionality I will need by the end of the project

Note: It's always worth thinking about these problems at scale -- assume we have hundreds of thousands of lines of code, and that I have been added to the team mid-project.



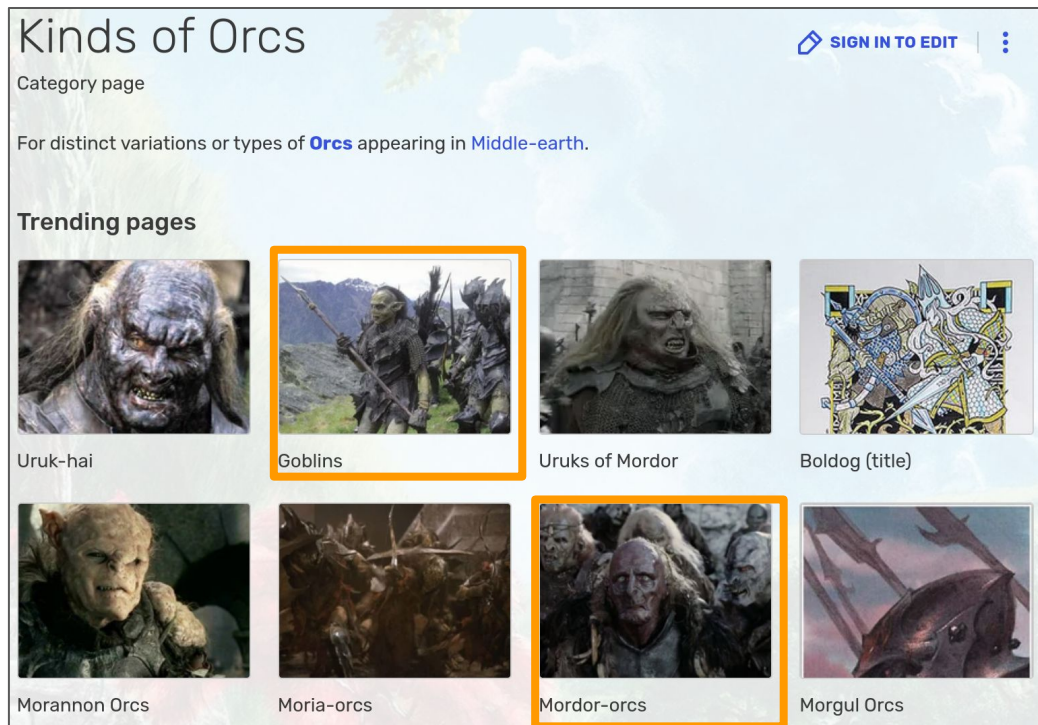
https://lotr.fandom.com/wiki/Category:Kinds_of_Orcs

What Problem Am I Trying to Solve? (2/2)

We effectively want to design solutions that are:

- ☐ Flexible
- ☐ Maintainable
- ☐ Extensible

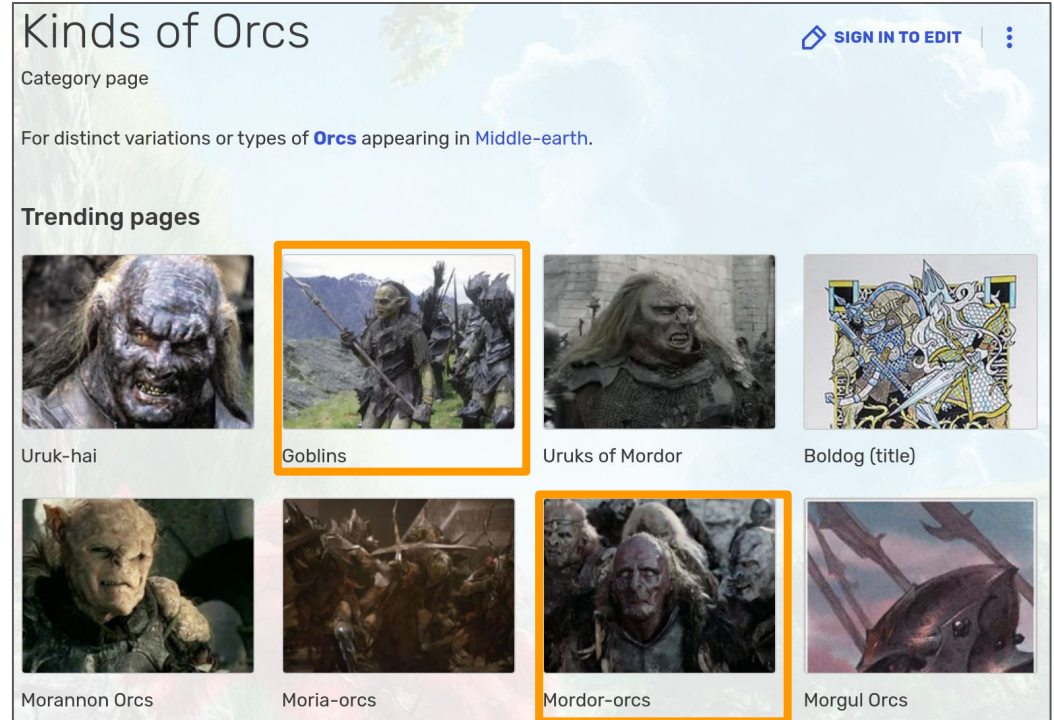
In particular -- today we will focus on the 'extensible' code -- but we'll observe a pattern that is quite flexible and maintainable to some degree as well.



https://lotr.fandom.com/wiki/Category:Kinds_of_Orcs

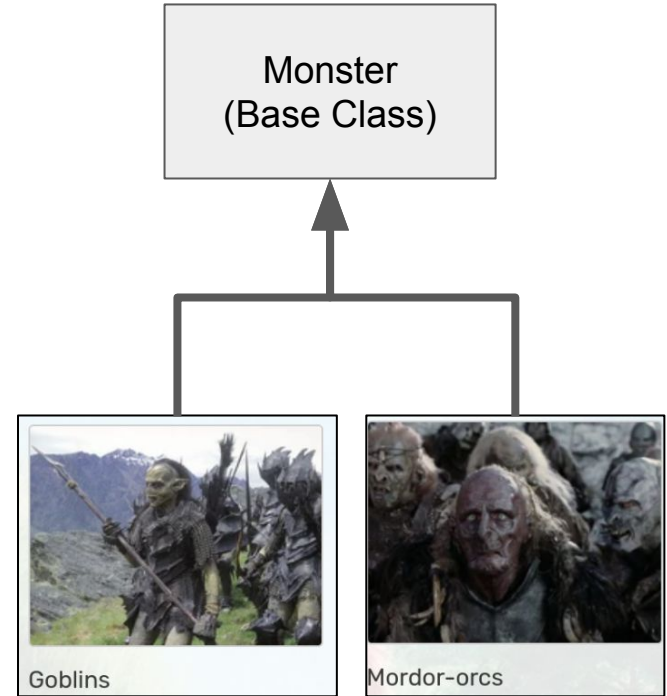
Audience Exercise:

- Take a moment to think about how you would lay out this hierarchy
 - What do your objects/functions/etc. look like?



A First Solution

Did it look something like on the right?



A First Solution (1/6)

- Question to the Audience: What do folks think of this?
 - ?

```
1 /// @file: main1.cpp
2 /// g++ -std=c++20 main1.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 /// Monster base class (for which Orc and Goblin will derive from)
7 struct Monster{
8     virtual void Sing() = 0;
9 };
10
11 /// Orc
12 struct Orc : public Monster{
13     Orc(){
14         std::cout << "Created a new Orc::Orc()\n";
15     }
16     void Sing() {
17         std::cout << "Orc::Sing()\n";
18     }
19 };
20
21 /// Goblin
22 struct Goblin : public Monster{
23     Goblin(){
24         std::cout << "Created a new Goblin::Goblin()\n";
25     }
26     void Sing() {
27         std::cout << "Orc::Sing()\n";
28     }
29 };
30
31 int main(){
32     // Create a bunch of monsters
33     std::vector<Monster*> monsters;
34     monsters.emplace_back(new Orc);
35     monsters.emplace_back(new Goblin);
36
37     for(const auto& m: monsters){
38         m->Sing();
39     }
40     // delete vector omitted for space
41     return 0;
42 }
```

A First Solution (2/6)

- **Question to the Audience:** What do folks think of this?
 - We have an interface
 - Goblins and Orcs have to Sing()
 - We've leaned into some 'object-oriented tools' to ensure functionality is supported by derived classes (pure virtual function)
 - We could have each class do different things if needed.
 - i.e. add more member functions
 - The member functions are somewhat 'piling up' into one single class.
 - We did have to repeat some code (DRY Principle -- **Don't Repeat Yourself**)
 - What about our 'loop' at line 37-39?
 - What if not all Orc's or Goblin's otherwise did not have a Sing function?
 - Perhaps different Monsters in this hierarchy have different abilities.
 - (Could dynamic_cast to check)

```
1 /// @file: main1.cpp
2 /// g++ -std=c++20 main1.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 /// Monster base class (for which Orc and Goblin will derive from)
7 struct Monster{
8     virtual void Sing() = 0;
9 };
10
11 /// Orc
12 struct Orc : public Monster{
13     Orc(){
14         std::cout << "Created a new Orc::Orc()\n";
15     }
16     void Sing() {
17         std::cout << "Orc::Sing()\n";
18     }
19 };
20
21 /// Goblin
22 struct Goblin : public Monster{
23     Goblin(){
24         std::cout << "Created a new Goblin::Goblin()\n";
25     }
26     void Sing() {
27         std::cout << "Orc::Sing()\n";
28     }
29 };
30
31 int main(){
32     // Create a bunch of monsters
33     std::vector<Monster*> monsters;
34     monsters.emplace_back(new Orc);
35     monsters.emplace_back(new Goblin);
36
37     for(const auto& m: monsters){
38         m->Sing();
39     }
40     // delete vector omitted for space
41     return 0;
42 }
```

A First Solution (3/6)

- **Question to the Audience:** What do folks think of this?
 - We have an interface
 - Goblins and Orcs have to Sing()
 - We've leaned into some 'object-oriented tools' to ensure functionality is supported by derived classes (pure virtual function)
 - We could have each class do different things if needed.
 - i.e. add more member functions
 - The member functions are somewhat 'piling up' into one single class.
 - We did have to repeat some code (DRY Principle -- **Don't Repeat Yourself**)
 - What about our 'loop' at line 37-39?
 - What if not all Orc's or Goblin's otherwise did not have a Sing function?
 - Perhaps different Monsters in this hierarchy have different abilities.
 - (Could dynamic_cast to check)

```
1 /// @file: main1.cpp
2 /// g++ -std=c++20 main1.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 /// Monster base class (for which Orc and Goblin will derive from)
7 struct Monster{
8     virtual void Sing() = 0;
9 };
10
11 /// Orc
12 struct Orc : public Monster{
13     Orc(){
14         std::cout << "Created a new Orc::Orc()\n";
15     }
16     void Sing() {
17         std::cout << "Orc::Sing()\n";
18     }
19 };
20
21 /// Goblin
22 struct Goblin : public Monster{
23     Goblin(){
24         std::cout << "Created a new Goblin::Goblin()\n";
25     }
26     void Sing() {
27         std::cout << "Orc::Sing()\n";
28     }
29 };
30
31 int main(){
32     // Create a bunch of monsters
33     std::vector<Monster*> monsters;
34     monsters.emplace_back(new Orc);
35     monsters.emplace_back(new Goblin);
36
37     for(const auto& m: monsters){
38         m->Sing();
39     }
40     // delete vector omitted for space
41     return 0;
42 }
```

A First Solution (4/6)

- **Question to the Audience:** What do folks think of this?

- We have an interface
 - Goblins and Orcs have to Sing()
 - We've leaned into some 'object-oriented tools' to ensure functionality is supported by derived classes (pure virtual function)
- We could have each class do different things if needed.
 - i.e. add more member functions
- The member functions are somewhat 'piling up' into one single class.
 - We did have to repeat some code (DRY Principle -- **Don't Repeat Yourself**)
- What about our 'loop' at line 37-39?
 - What if not all Orc's or Goblin's otherwise did not have a Sing function?

Note: This is a bit troublesome -- especially if Goblin, Orc, etc. are spread across different files

```
1 /// @file: main1.cpp
2 /// g++ -std=c++20 main1.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 /// Monster base class (for which Orc and Goblin will derive from)
7 struct Monster{
8     virtual void Sing() = 0;
9 };
10
11 /// Orc
12 struct Orc : public Monster{
13     Orc(){
14         std::cout << "Created a new Orc::Orc()\n";
15     }
16     void Sing() {
17         std::cout << "Orc::Sing()\n";
18     }
19 };
20
21 /// Goblin
22 struct Goblin : public Monster{
23     Goblin(){
24         std::cout << "Created a new Goblin::Goblin()\n";
25     }
26     void Sing() {
27         std::cout << "Orc::Sing()\n";
28     }
29 };
30
31 int main(){
32     // Create a bunch of monsters
33     std::vector<Monster*> monsters;
34     monsters.emplace_back(new Orc);
35     monsters.emplace_back(new Goblin);
36
37     for(const auto& m: monsters){
38         m->Sing();
39     }
40     // delete vector omitted for space
41     return 0;
42 }
```


A First Solution (5/6)

- **Question to the Audience:** What do folks think of this?
 - We have an interface
 - Goblins and Orcs have to Sing()
 - We've leaned into some 'object-oriented tools' to ensure functionality is supported by derived classes (pure virtual function)
 - We could have each class do different things if needed.
 - i.e. add more member functions
 - The member functions are somewhat 'piling up' into one single class.
 - We did have to repeat some code (DRY Principle -- **Don't Repeat Yourself**)
 - What about our 'loop' at line 37-39?
 - What if not all Orc's or Goblin's otherwise did not have a Sing function?
 - Perhaps different Monsters in this hierarchy have different abilities.
 - (Could dynamic_cast to check)

```
1 /// @file: main1.cpp
2 /// g++ -std=c++20 main1.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 /// Monster base class (for which Orc and Goblin will derive from)
7 struct Monster{
8     virtual void Sing() = 0;
9 };
10
11 /// Orc
12 struct Orc : public Monster{
13     Orc(){
14         std::cout << "Created a new Orc::Orc()\n";
15     }
16     void Sing() {
17         std::cout << "Orc::Sing()\n";
18     }
19 };
20
21 /// Goblin
22 struct Goblin : public Monster{
23     Goblin(){
24         std::cout << "Created a new Goblin::Goblin()\n";
25     }
26     void Sing() {
27         std::cout << "Orc::Sing()\n";
28     }
29 };
30
31 int main(){
32     // Create a bunch of monsters
33     std::vector<Monster*> monsters;
34     monsters.emplace_back(new Orc);
35     monsters.emplace_back(new Goblin);
36
37     for(const auto& m: monsters){
38         m->Sing();
39     }
40     // delete vector omitted for space
41     return 0;
42 }
```

A First Solution (6/6)

- So to be clear
 - I want to maintain the inheritance hierarchy
 - That is appropriate -- Orc's and Goblins are a 'type of' Monster
 - (is-a relationship)
 - I'm looking for a way to 'extend' this class -- again think at scale
 - A game project over the course of 3-years, we may not be able to predict all behaviors.
 - Late in the project, we may need to add functionality, and we may want to avoid adding bugs
 - There are some better defined solutions for this that *may* help.

```
1 /// @file: main1.cpp
2 /// g++ -std=c++20 main1.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 /// Monster base class (for which Orc and Goblin will derive from)
7 struct Monster{
8     virtual void Sing() = 0;
9 };
10
11 /// Orc
12 struct Orc : public Monster{
13     Orc(){
14         std::cout << "Created a new Orc::Orc()\n";
15     }
16     void Sing() {
17         std::cout << "Orc::Sing()\n";
18     }
19 };
20
21 /// Goblin
22 struct Goblin : public Monster{
23     Goblin(){
24         std::cout << "Created a new Goblin::Goblin()\n";
25     }
26     void Sing() {
27         std::cout << "Orc::Sing()\n";
28     }
29 };
30
31 int main(){
32     // Create a bunch of monsters
33     std::vector<Monster*> monsters;
34     monsters.emplace_back(new Orc);
35     monsters.emplace_back(new Goblin);
36
37     for(const auto& m: monsters){
38         m->Sing();
39     }
40     // delete vector omitted for space
41     return 0;
42 }
```


Design Pattern Goal(s) for today

What you're going to learn today

- Brief Programming Review
- What design patterns are
- One behavioral design patterns specifically:
 - Visitor Pattern
- This is not an 'expert-level' talk, but aimed more at beginners/intermediate
 - That said, I hope 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

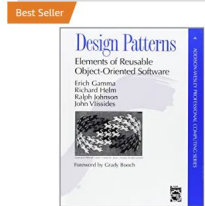
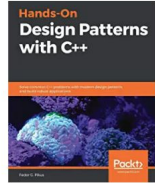
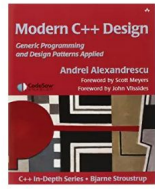


Pretend these seats are filled :)

<https://pixnio.com/free-images/2017/03/11/2017-03-11-16-47-11-550x413.jpg>

Design Patterns

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



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 :))

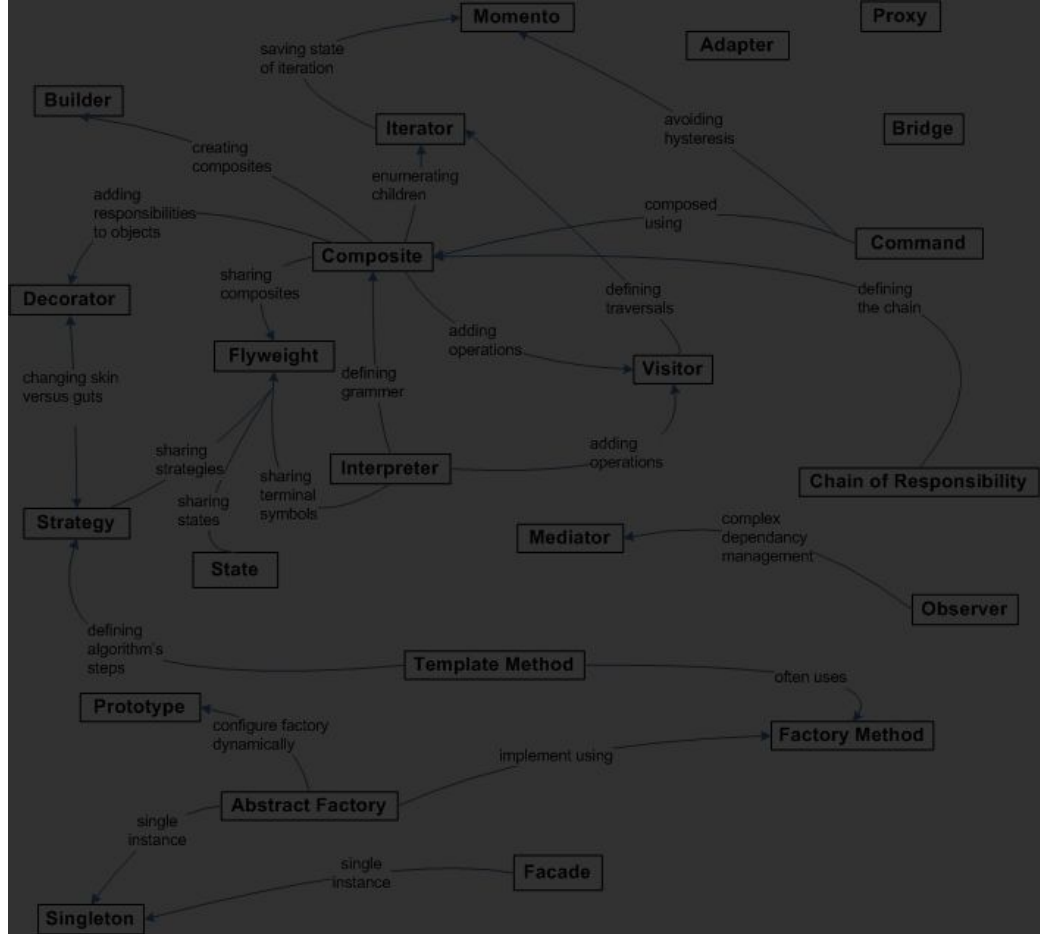
What is a Design Pattern?

- 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 (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



Design pattern relationships

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

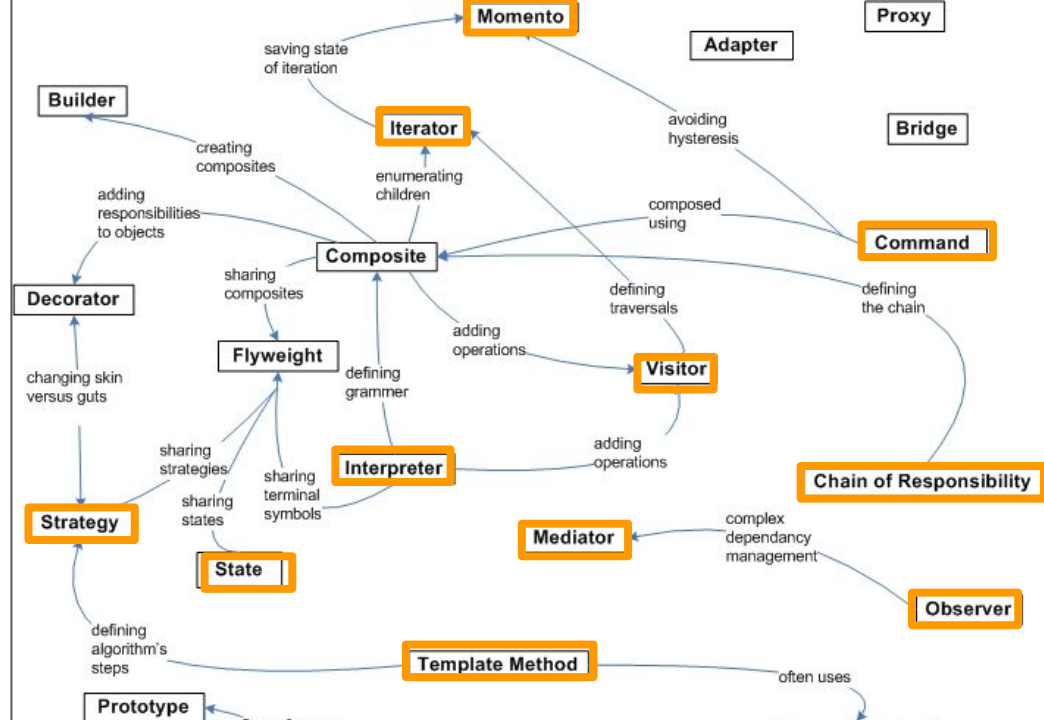
Design Patterns Book (2/2)



- 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.

Design Pattern Categories

There are '3' categories of design patterns

1. Creational

- Provide program more flexibility on how to create objects, often avoiding direct instantiation of a specific object.

2. Structural

- Focus on using inheritance to compose interfaces and define objects in a way to create new functionality.

3. Behavioral

- **Focuses on how to communicate between objects**

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

- 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

- Chain of responsibility delegates o
- Command creates objects which en
- Interpreter implements a specializ
- Iterator accesses the elements of a
- Mediator allows loose coupling betw
- Memento provides the ability to res
- Observer is a publish/subscribe pat
- State allows an object to alter its be
- Strategy allows one of a family of al
- Template method defines the skelet
- Visitor separates an algorithm from

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 factories
- [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 co
- [Command](#) creates objects which e
- [Interpreter](#) implements a specializ
- [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 be
- [Strategy](#) allows one of a family of a
- [Template method](#) defines the skele
- [Visitor](#) separates an algorithm from

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 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.

The specific pattern we'll look at today

Creational [edit]

Main article: Creational pattern

Creational patterns are ones that create

- **Abstract factory** groups object factories
- **Builder** constructs complex objects
- **Factory method** creates objects with
- **Prototype** creates objects by cloning
- **Singleton** restricts object creation to one

Structural [edit]

These concern class and object composition

- **Adapter** allows classes with incompatible interfaces to work together
- **Bridge** decouples an abstraction from its implementation
- **Composite** composes zero-or-more objects
- **Decorator** dynamically adds/removes behavior
- **Facade** provides a simplified interface to a complex system
- **Flyweight** reduces the cost of creating and storing many small objects
- **Proxy** provides a placeholder for an object

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 which 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.

The Visitor Design Pattern

A Behavioral Design Pattern

Behavior Design Patterns

- Focus on communication between objects

The Visitor Design Pattern [[wiki](#)]

- Focus on communication between objects

Definition [[edit](#)]

The [Gang of Four](#) defines the Visitor as:

Represent[ing] an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

The nature of the Visitor makes it an ideal pattern to plug into public APIs, thus allowing its clients to perform operations on a class using a "visiting" class without having to modify the source.^[2]

Resuming our game project...



Review of our First Solution

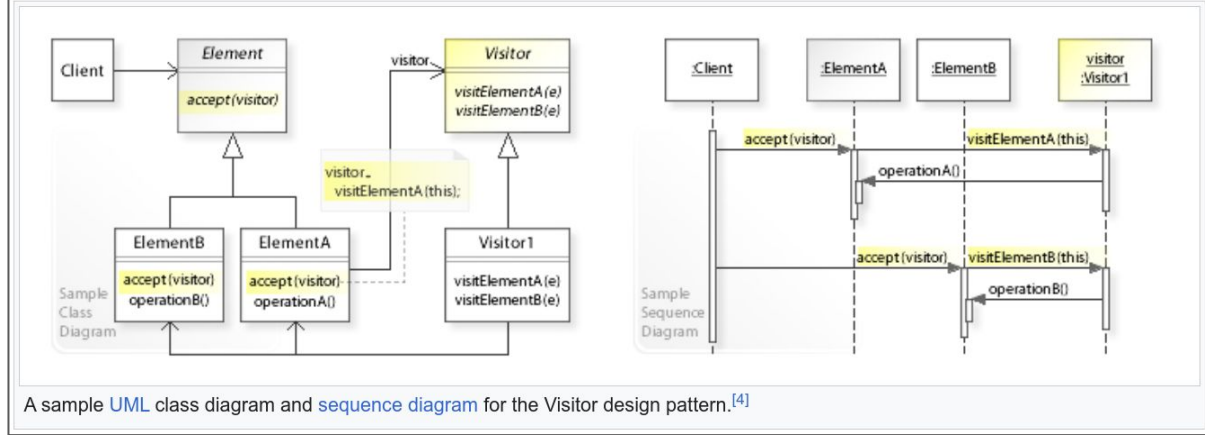
- Let's now apply the **visitor pattern** to our solution
 - In a similar way -- recall previously how we moved all the 'attributes' to its own struct, we're going to move all the member functions to a new class.

```
1 /// @file: main1.cpp
2 /// g++ -std=c++20 main1.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 /// Monster base class (for which Orc and Goblin will derive from)
7 struct Monster{
8     virtual void Sing() = 0;
9 };
10
11 /// Orc
12 struct Orc : public Monster{
13     Orc(){
14         std::cout << "Created a new Orc::Orc()\n";
15     }
16     void Sing() {
17         std::cout << "Orc::Sing()\n";
18     }
19 };
20
21 /// Goblin
22 struct Goblin : public Monster{
23     Goblin(){
24         std::cout << "Created a new Goblin::Goblin()\n";
25     }
26     void Sing() {
27         std::cout << "Orc::Sing()\n";
28     }
29 };
30
31 int main(){
32     // Create a bunch of monsters
33     std::vector<Monster*> monsters;
34     monsters.emplace_back(new Orc);
35     monsters.emplace_back(new Goblin);
36
37     for(const auto& m: monsters){
38         m->Sing();
39     }
40     // delete vector omitted for space
41     return 0;
42 }
```


(Aside) UML Diagram of Visitor Pattern

- From the wikipedia you can see a 'UML' diagram of the Visitor pattern
 - To be honest -- I think it's better if we build this up slowly with our problem rather than stare at this too long

UML class and sequence diagram [edit]

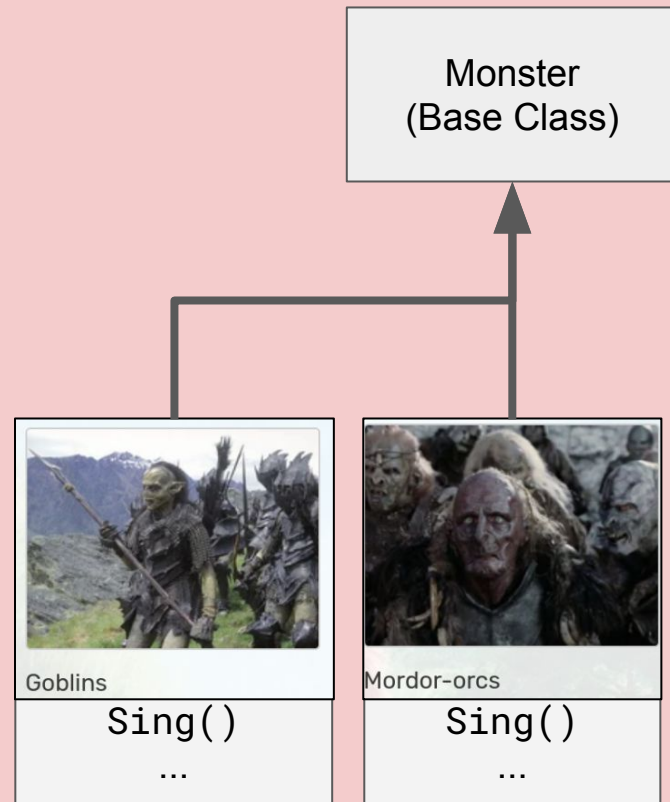


https://en.wikipedia.org/wiki/Visitor_pattern

Visitor Pattern - Extending Behaviors

- So here's our current hierarchy in our current solution

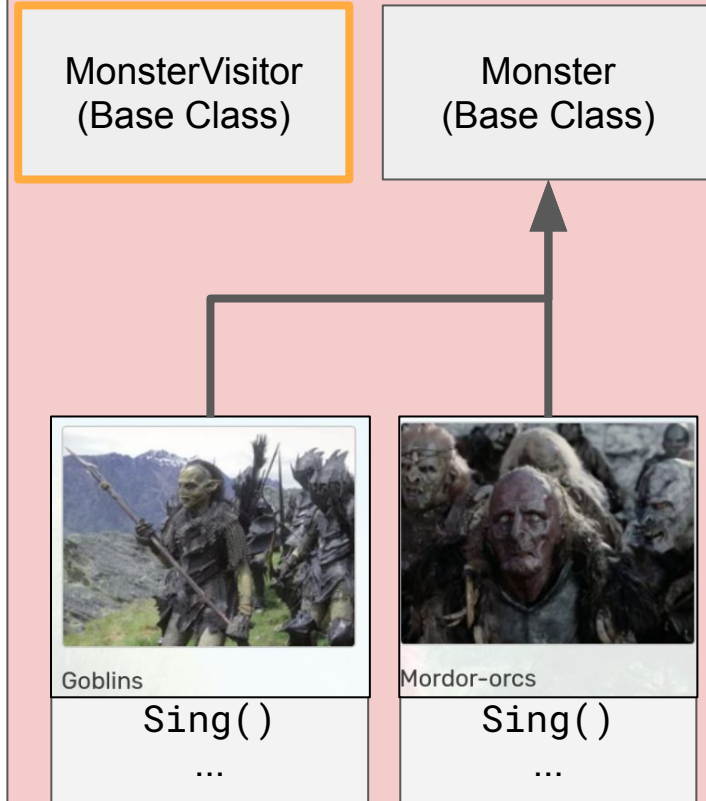
```
1 /// @file: main1.cpp
2 /// g++ -std=c++20 main1.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 /// Monster base class (for which Orc and Goblin will derive from)
7 struct Monster{
8     virtual void Sing() = 0;
9 };
10
11 /// Orc
12 struct Orc : public Monster{
13     Orc(){
14         std::cout << "Created a new Orc::Orc()\n";
15     }
16     void Sing() {
17         std::cout << "Orc::Sing()\n";
18     }
19 };
20
21 /// Goblin
22 struct Goblin : public Monster{
23     Goblin(){
24         std::cout << "Created a new Goblin::Goblin()\n";
25     }
26     void Sing() {
27         std::cout << "Orc::Sing()\n";
28     }
29 };
30
31 int main(){
32     // Create a bunch of monsters
33     std::vector<Monster*> monsters;
34     monsters.emplace_back(new Orc);
35     monsters.emplace_back(new Goblin);
36
37     for(const auto& m: monsters){
38         m->Sing();
39     }
40     // delete vector omitted for space
41     return 0;
42 }
```



Visitor Pattern - Extending Behaviors

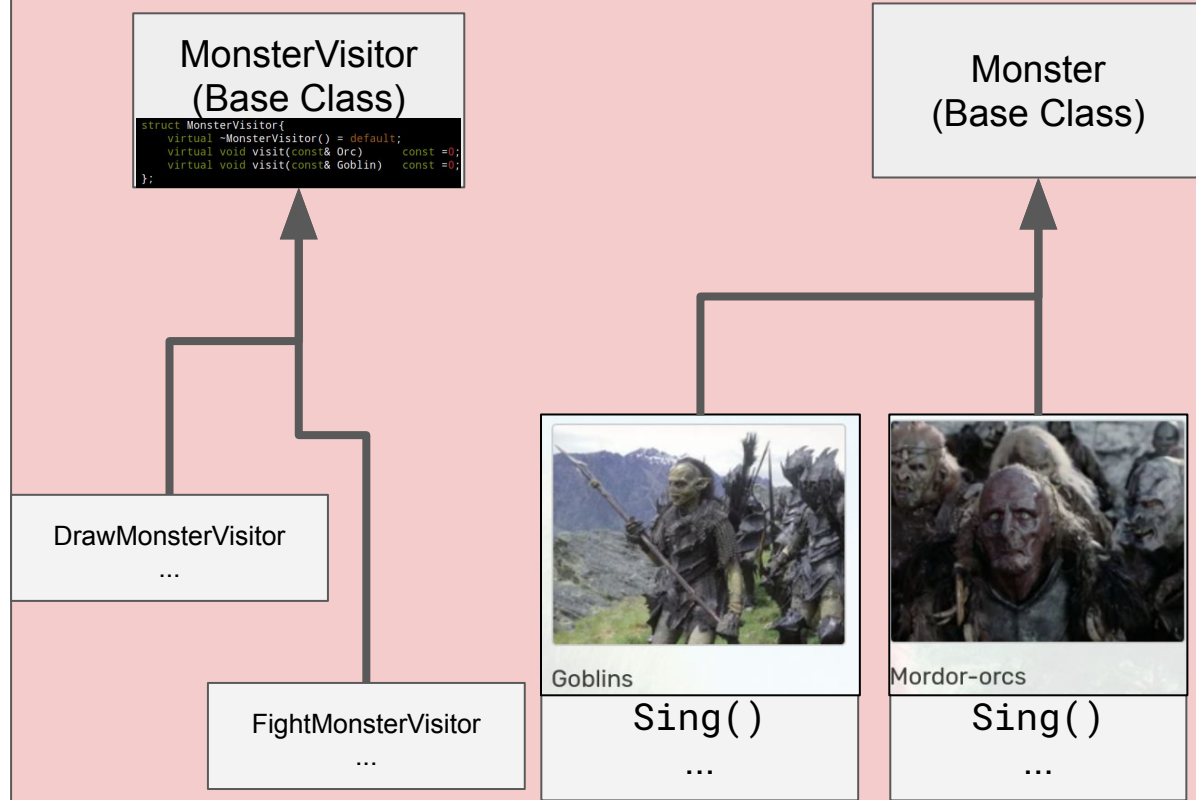
- Now part of our solution is going to be to add a 'MonsterVisitor' class
 - The purpose of this 'abstract class' is to hold behaviors (i.e. functions)
 - Thus we will avoid creating more member functions within the 'Orc' and 'Goblin' concrete classes -- new behaviors go in the 'Monster Visitor')

```
11 /// The 'MonsterVisitor' will represent how we 'extend' each of
12 /// the 'Orc', 'Goblin', etc. with more functionality.
13 /// This 'visitor class' serves as a base class for other functionality
14 /// that we will add.
15 struct MonsterVisitor{
16     virtual ~MonsterVisitor() = default;
17
18     // Per 'Monster' I need a 'visit' function that
19     // takes in the type of an object in the 'Monster hierarchy
20     // (i.e. must be 'typeof' Monster)
21     virtual void visit(const& Orc)      const =0;
22     virtual void visit(const& Goblin)   const =0;
23 };
```



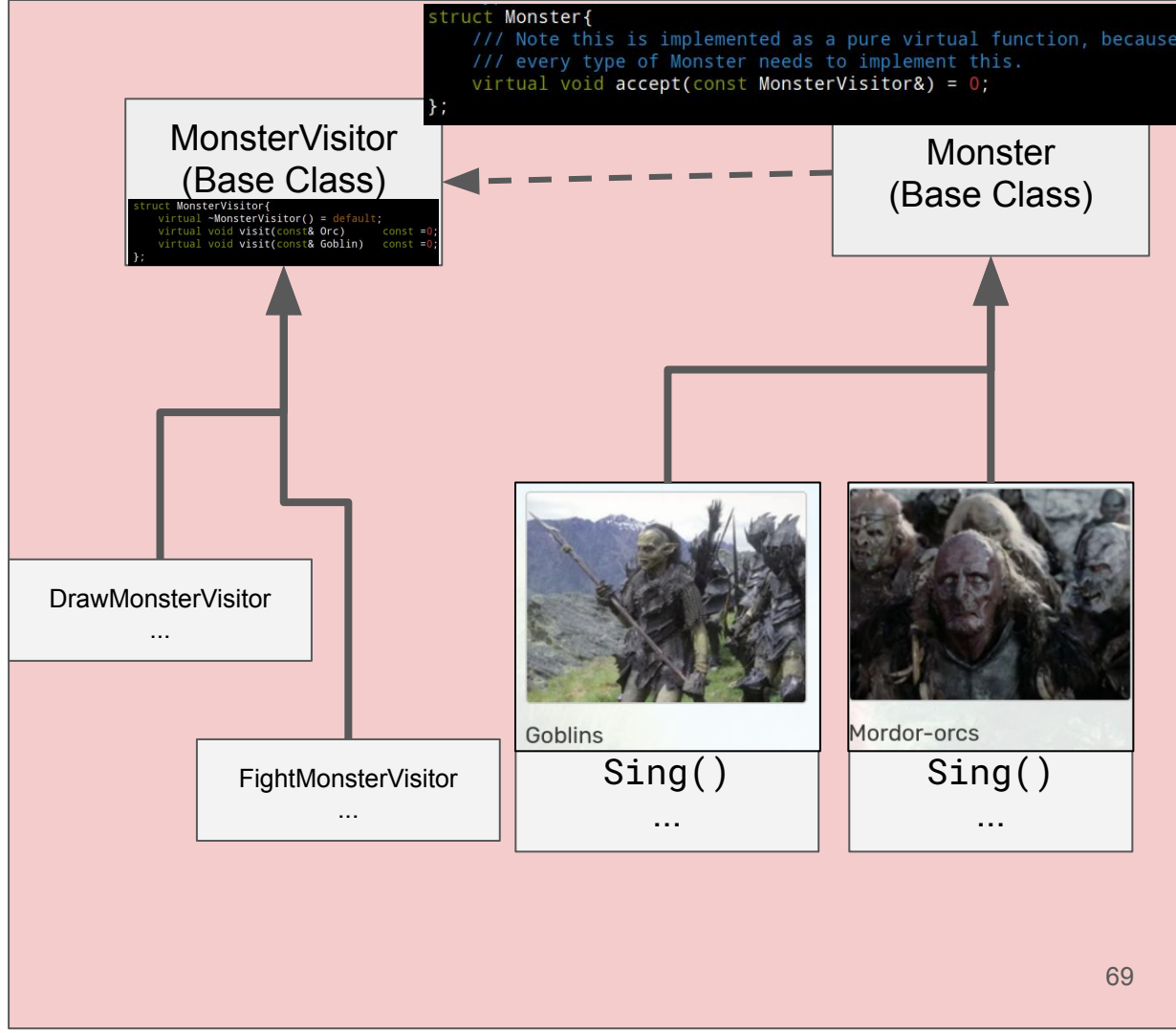
Visitor Pattern (1/8)

- Any classes derived from 'MonsterVisitor' will be the functional equivalent to adding a 'Sing()' member function.



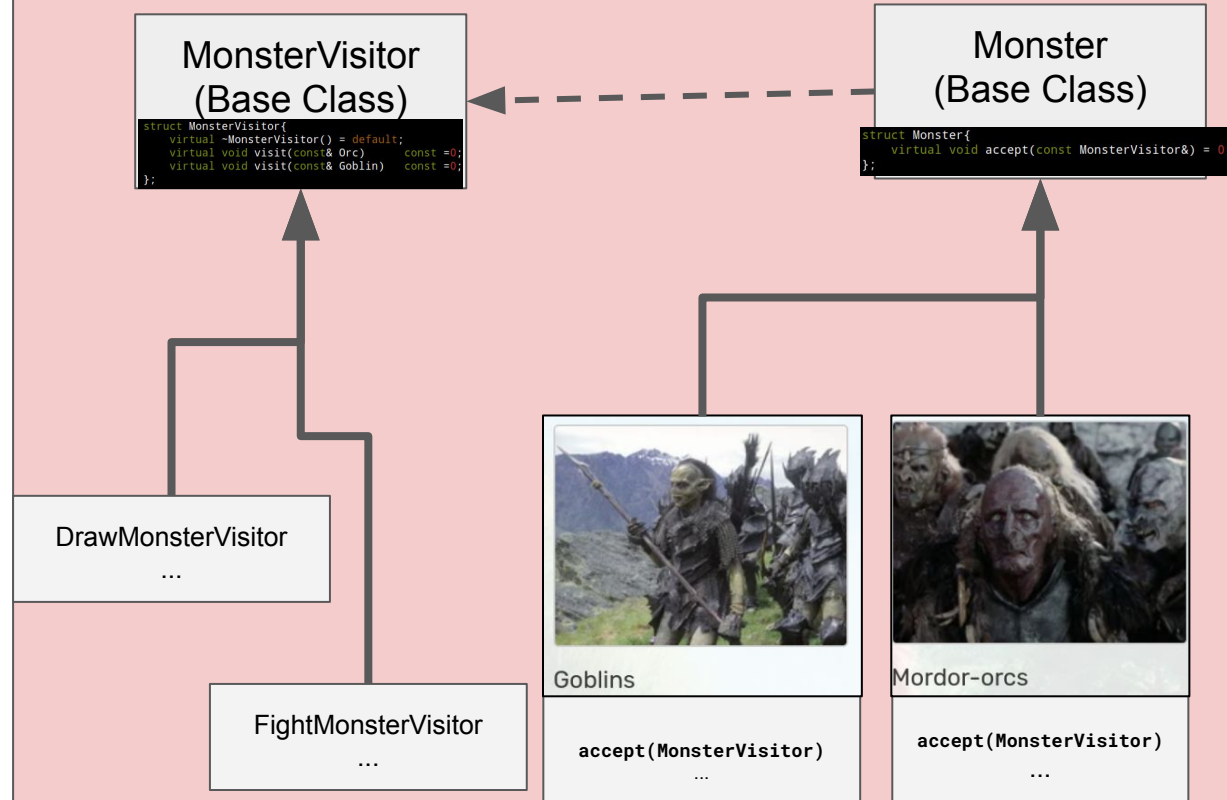
Visitor Pattern (2/8)

- Monster Interface 'connects' to the 'visitor classes', by implementing an 'accept' member function.



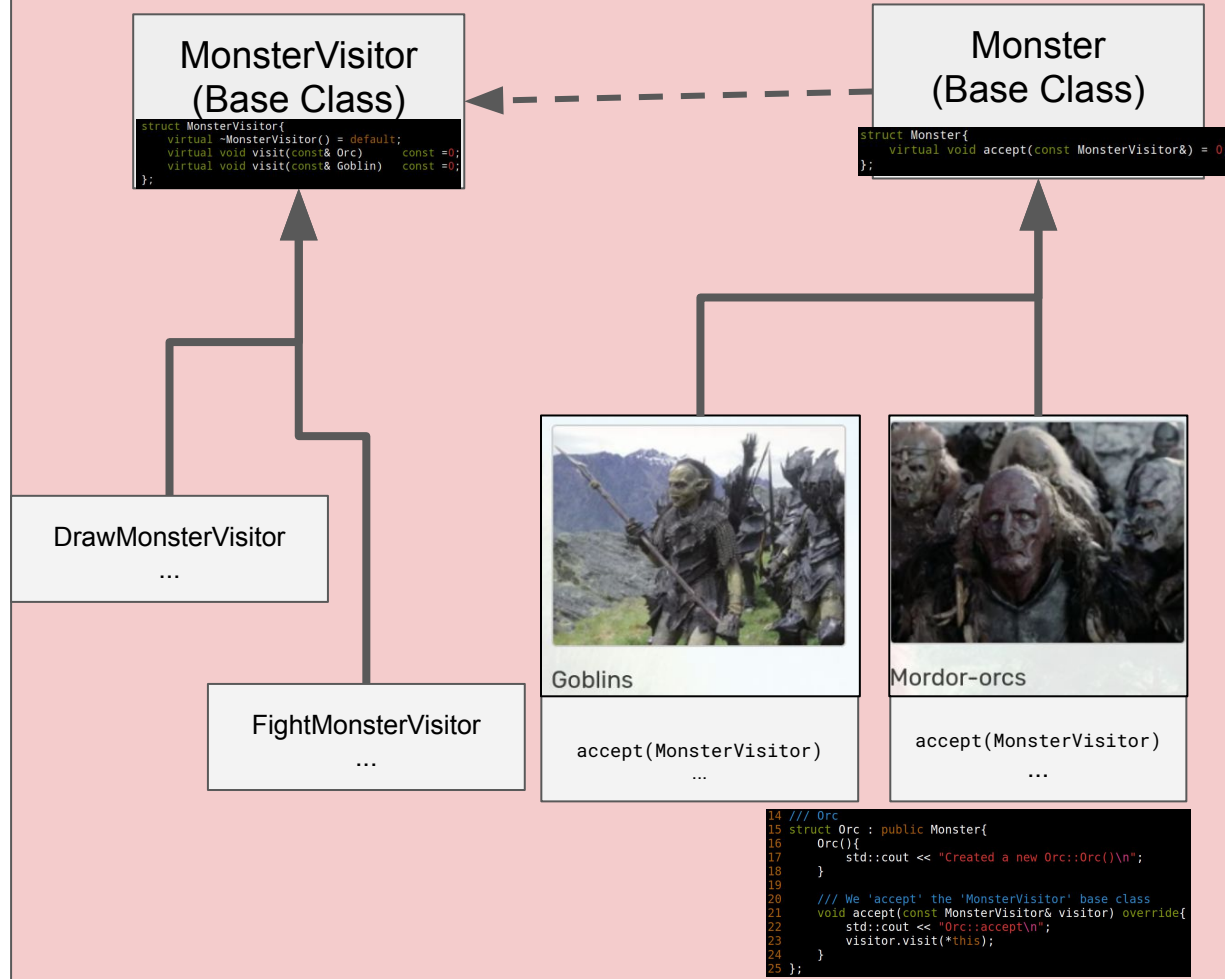
Visitor Pattern (3/8)

- Monsters 'connect' to the 'visitor classes', by implementing an 'accept' member function.



Visitor Pattern (4/8)

- Can be read as “I accept some functionality from the ‘MonsterVisitor’ hierarchy

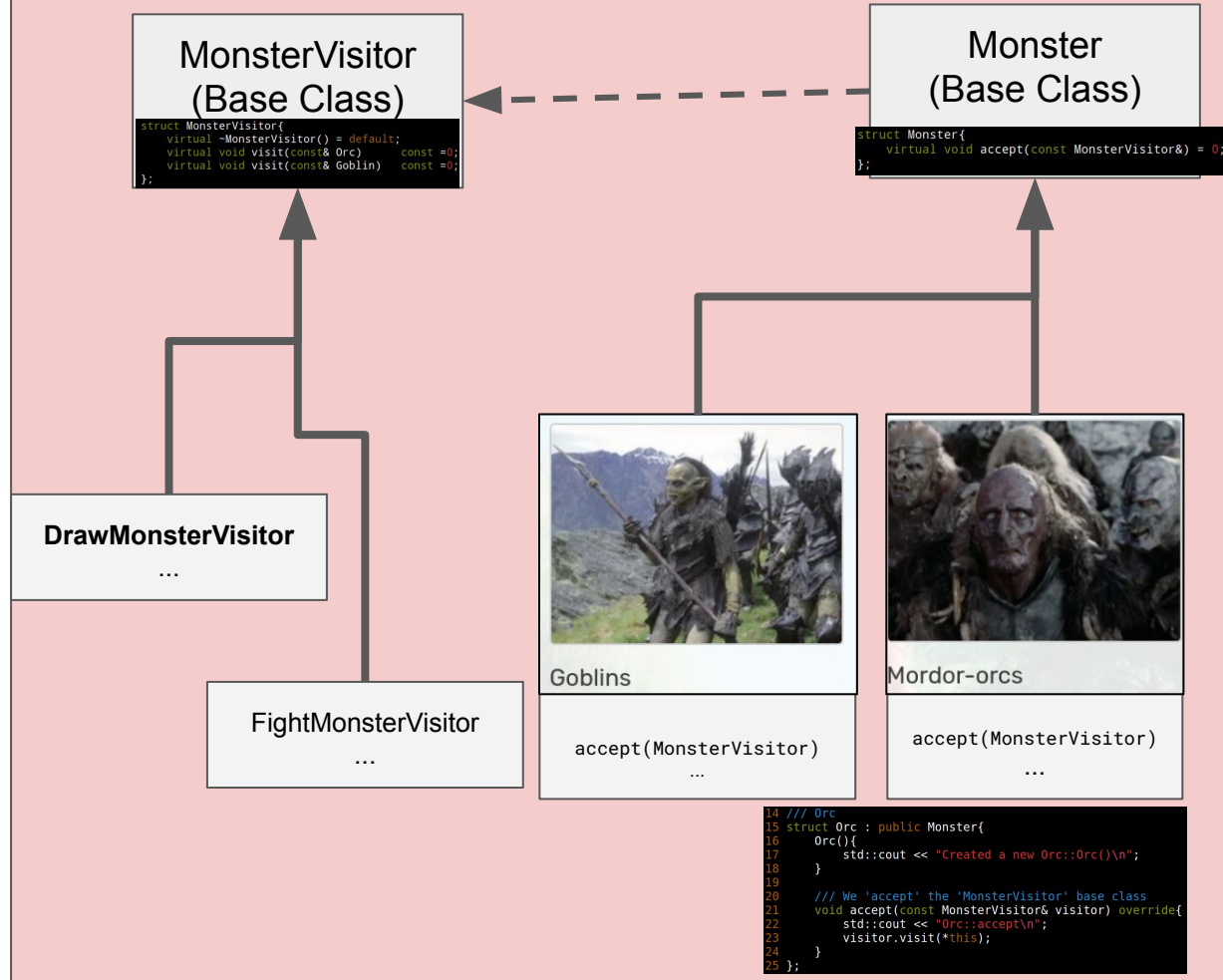


Visitor Pattern (5/8)

- Important Observation

- per-behavior that we add (e.g. **DrawMonsterVisitor**)
- I need to add one member function for every class in the 'Monster' hierarchy (i.e. Goblins, Orcs, etc.)

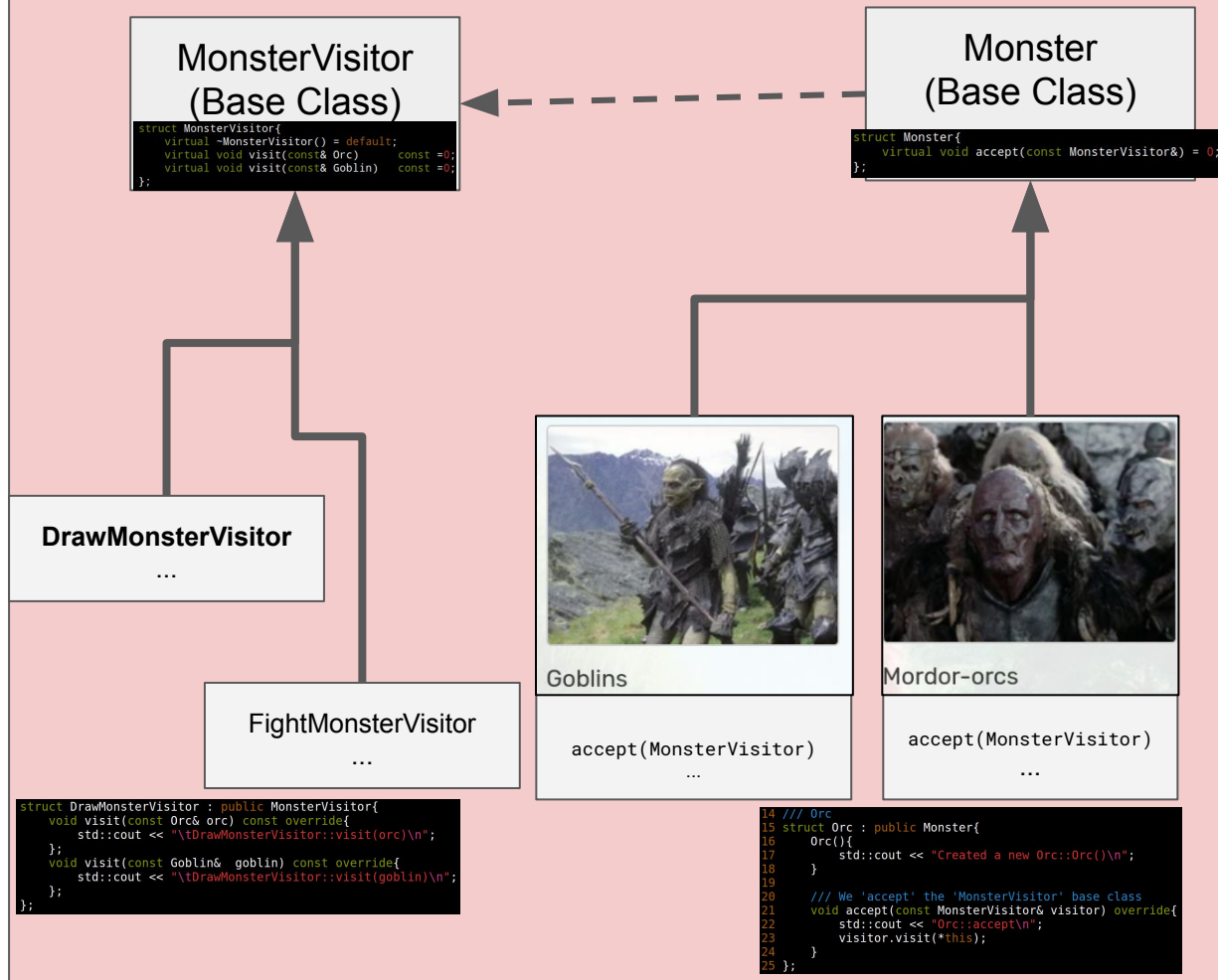
```
struct DrawMonsterVisitor : public MonsterVisitor{
    void visit(const Orc& orc) const override{
        std::cout << "\tDrawMonsterVisitor::visit(orc)\n";
    };
    void visit(const Goblin& goblin) const override{
        std::cout << "\tDrawMonsterVisitor::visit(goblin)\n";
    };
};
```



Visitor Pattern (6/8)

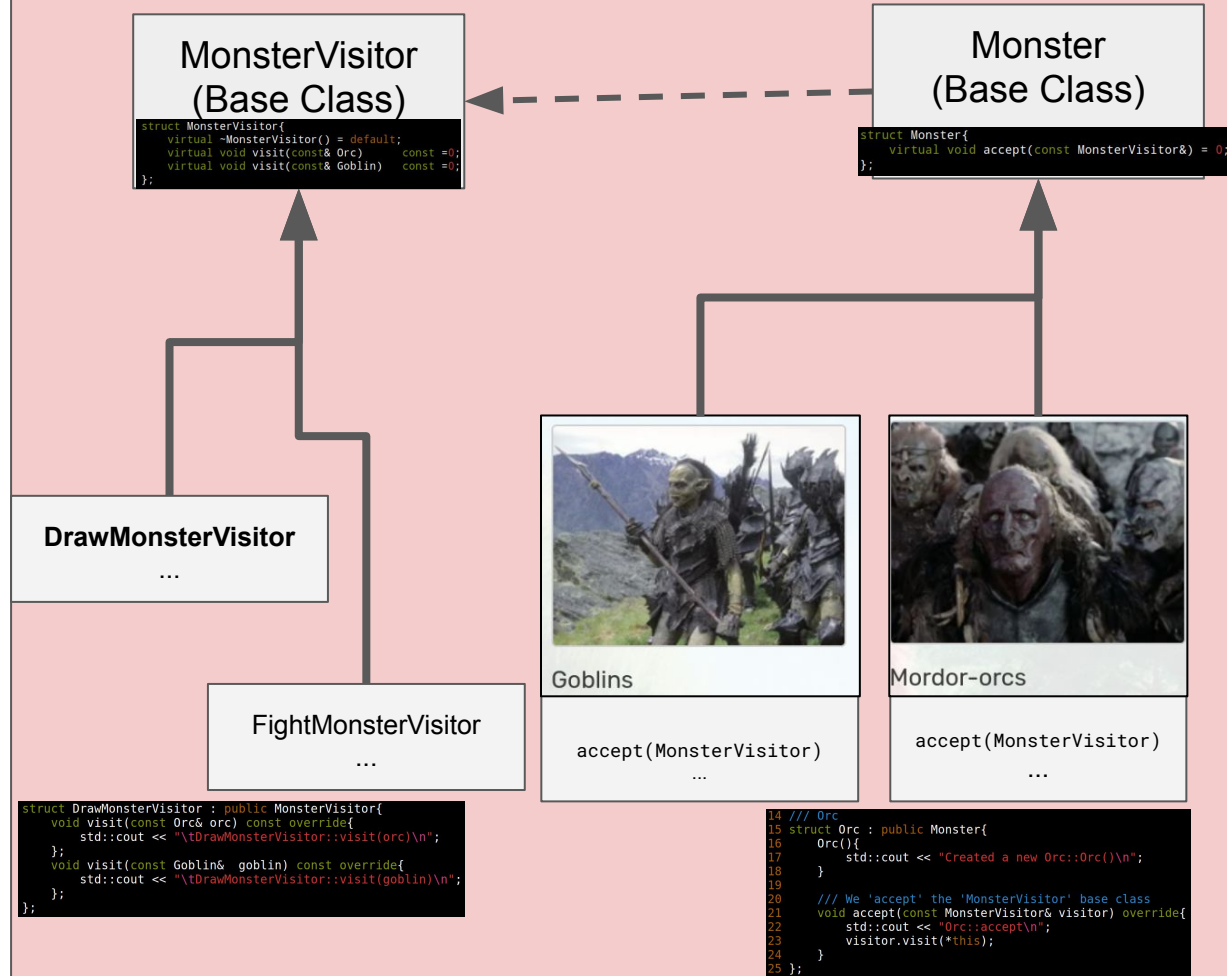
- Important Observation

- per-behavior that we add (e.g. **DrawMonsterVisitor**)
- I need to add one member function for every class in the 'Monster' hierarchy (i.e. Goblins, Orcs, etc.)



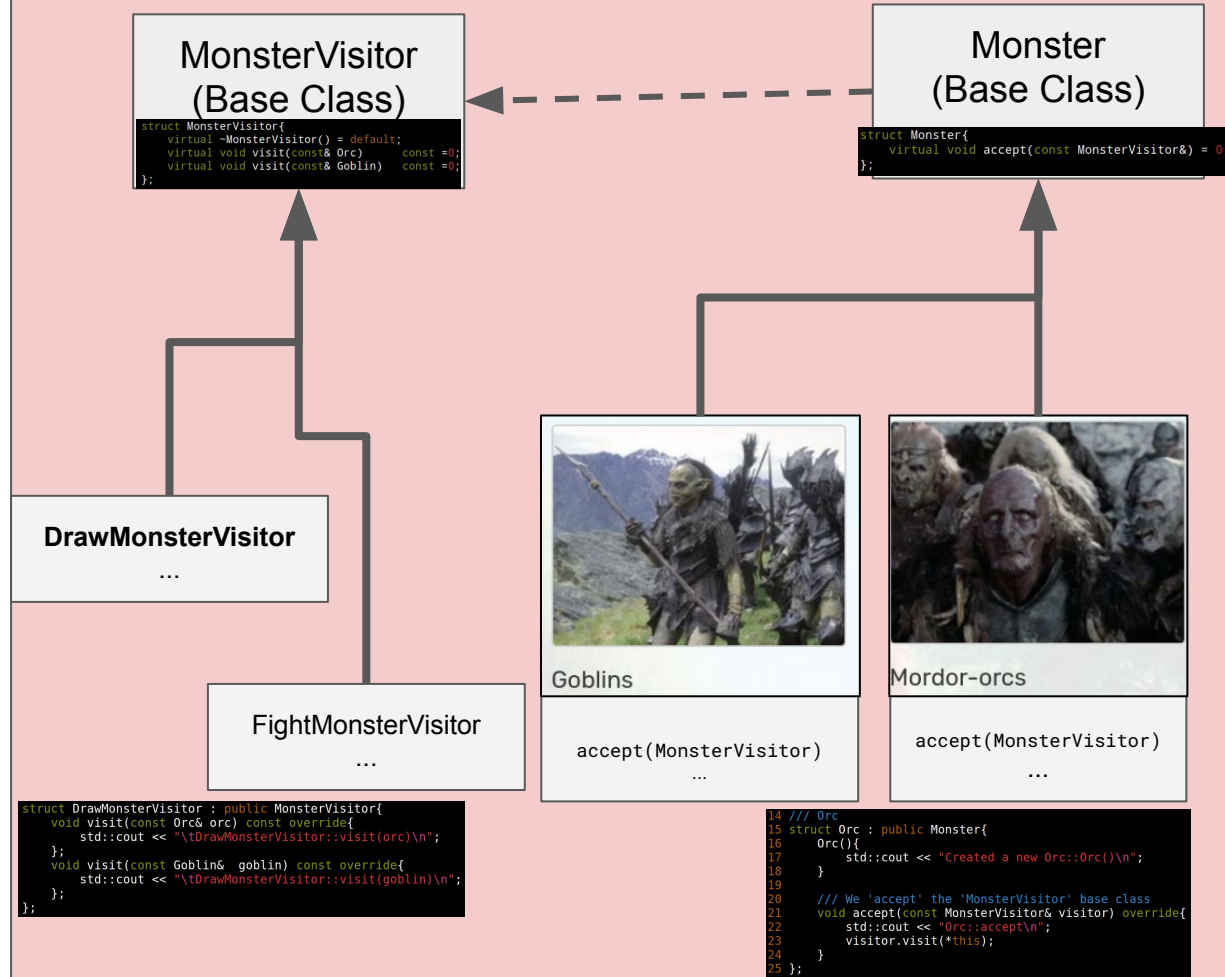
Visitor Pattern (7/8)

- Open-Closed Principle
 - One property we can observe visually is the Open-Closed Principle
 - This means we can 'add new operations to existing object (open), without modifying the structure (closed -- Monster is not changed)
- The algorithm (i.e. behavior) is thus separated from the object
 - This is a 'good thing' if our goal is 'extensible' software design.
 - As you can imagine, we can try out many different new 'MonsterVisitors' to try different behaviors.



Visitor Pattern (8/8)

- Note: However (from previous slide) -- we probably want to make sure we are not adding lots of different objects to the 'Monster hierarchy'
 - Remember, we have to implement (or remove) functionality for each object in our hierarchy -- so hopefully it is relatively stable :)



Live Code Review

- Full Code Review
- (Do ask questions at any point)

```
1 /// @file: visitor_final.cpp
2 /// g++ -std=c++20 visitor_final.cpp -o prog && ./prog
3 #include <iostream>
4 #include <vector>
5
6 // Forward Declarations
7 // of all of the Monsters that are going to exist.
8 struct Orc;
9 struct Goblin;
10
11 /// The 'MonsterVisitor' will represent how we 'extend' each of
12 /// the 'Orc', 'Goblin', etc. with more functionality.
13 /// This 'visitor class' serves as a base class for other functionality
14 /// that we will add.
15 struct MonsterVisitor{
16     virtual ~MonsterVisitor() = default;
17
18     // Per 'Monster' I need a 'visit' function that
19     // takes in the type of an object in the 'Monster hierarchy'
20     // (i.e. must be 'typeof' Monster)
21     virtual void visit(const& Orc) const =0;
22     virtual void visit(const& Goblin) const =0;
23 };
24
25 /// Monster base class (for which Orc and Goblin will derive from)
26 /// Observe that we take in a 'MonsterVisitor' -- we 'accept' that
27 /// type.
28 struct Monster{
29     /// Note this is implemented as a pure virtual function, because
30     /// every type of Monster needs to implement this.
31     virtual void accept(const MonsterVisitor&) = 0;
32 };
33
34 /// Orc
35 struct Orc : public Monster{
36     Orc(){
37         std::cout << "Created a new Orc::Orc()\n";
38     }
39
40     /// We 'accept' the 'MonsterVisitor' base class
41     void accept(const MonsterVisitor& visitor) override{
42         std::cout << "Orc::accept\n";
43         visitor.visit(*this);
44     }
45 }
```

(From Code Review) 'Visitor Classes'

- Again, here's where the functionality is extended (top-right)
- Here is an example usage on a collection (Bottom-right)
- Here is an example usage on a single object (Bottom-left)

```
Monster* myGoblin = new Goblin;
FightMonsterVisitor fmv;
myGoblin->accept(fmv);
```

```
61 // NOTE: For the visitor, could have this templated
62 //       to generate the implementations at compile-time that
63 //       are otherwise needed.
64 struct DrawMonsterVisitor : public MonsterVisitor{
65     void visit(const Orc& orc) const override{
66         std::cout << "\tDrawMonsterVisitor::visit(orc)\n";
67     };
68     void visit(const Goblin& goblin) const override{
69         std::cout << "\tDrawMonsterVisitor::visit(goblin)\n";
70     };
71 };
72
73 struct FightMonsterVisitor: public MonsterVisitor{
74     void visit(const Orc& orc) const override{
75         std::cout << "\tFightMonsterVisitor::visit(orc)\n";
76     };
77     void visit(const Goblin& goblin) const override{
78         std::cout << "\tFightMonsterVisitor::visit(goblin)\n";
79     };
80 };
81
82
83 /// Function that is called to draw all the 'Monsters' in a collection
84 /// with the correct visitor.
85 /// We are iterating through each monster and simply 'dispatching' the
86 /// behavior for the monster based on the 'visitor type'
87 void drawAllMonsters(std::vector<Monster*> const& monsters){
88     for(auto const& m : monsters){
89         m->accept(DrawMonsterVisitor{});
90     }
91 }
92
93 void fightAllMonsters(std::vector<Monster*> const& monsters){
94     for(auto const& m : monsters){
95         m->accept(FightMonsterVisitor{});
96     }
97 }
98 }
```

Design Pattern Summary

Question to Audience: Trade-offs (1/2)

- Pros
 - ??
- Neutral
 - ??
- Cons
 - ??

Question to Audience: Trade-offs (2/2)

- Pros

- Observes Open-Closed Principle (Easy to extend behaviors)
- Implementation details are isolated from class

- Neutral

- When we do have to repeat our code, at least it is in same struct/class.
 - Pure virtual functions also can ensure all classes get updated

- Cons

- Potentially makes it harder to introduce new types later in a project -- need to do lots of potential implementation
- Restricted to the public interface (or otherwise need to use getters to expose data)
- Performance may be negatively affected
 - Lots of 'polymorphism' (double dispatch)
 - Potentially more 'heap allocations' and memory management needed.

Bonus if Time

std::visit and std::variant

- Now that we know about the visitor pattern -- we can actually make use of some tools in the C++ Standard Library
 - This is a more 'functional style' in some ways of thinking about the problem.
 - We're going to use a std::variant, and implement (sort of like pattern matching) an operator to 'dispatch' on for each type in the std::variant.
- See [[Church Encoding](#)]

```
1 /// @file functional_visitor.cpp
2 /// g++ -std=c++20 functional_visitor.cpp -o prog && ./prog
3 #include <iostream>
4 #include <string>
5 #include <variant>
6
7 /// In this example we reduce our 'classes' to
8 /// simple structs with minimal behaviors
9 struct MethodActor{
10     public:
11     MethodActor() = default;
12     MethodActor(std::string name) : mName(name){}
13     const std::string mName;
14 };
15
16 struct ClassicalActor{
17     public:
18     ClassicalActor() = default;
19     ClassicalActor(std::string name): mName(name) {}
20     const std::string mName;
21 };
22
23 /// Note: Does not have to be globally scoped
24 using Actor = std::variant<MethodActor, ClassicalActor>;
25
26 /// Create a 'visitor' that handles the cases for
27 /// each type in the variant
28 struct PracticeVisitor{
29     void operator()(const MethodActor& a) const{
30         std::cout << "Practicing like a Method Actor\n";
31     }
32     void operator()(const ClassicalActor& a) const{
33         std::cout << "Practicing like a Classical Actor\n";
34     }
35 };
36
37 int main(){
38
39     // Create a variant -- effectively holding one type or
40     // the other in a 'tagged union'
41     Actor a = MethodActor("Mike");
42     // Grab the data that we need based on the type in the variant
43     std::visit(PracticeVisitor{}, a);
44
45     Actor b = ClassicalActor("Shah");
46     std::visit(PracticeVisitor{}, b);
47
48     return 0;
49 }
```

Some High Level Takeaways









- We started with ‘programming paradigms’
 - That may have been a weird place to start -- but hopefully there is some observation about moving around ‘data’ and ‘behaviors’
 - Each paradigm puts an emphasis on slightly different areas.
- We looked at a ‘real world problem’
 - In the sense that we’re ‘managing complexity’ in a system that may contain many objects and data types
- The Visitor Pattern can help manage complexity when new behaviors need to be added
 - There may be alternatives -- we may need to look at other paradigms (e.g. data-oriented in the gaming space) to otherwise think about how we process data.
- Overall -- the Visitor is a good exercise in understanding how flexible software creation can be, and the trade-offs that come with it

Further resources and training materials

- More patterns (and soon this one) added here:

<https://www.youtube.com/playlist?list=PLvv0ScY6vfd9wBfIF0f6ynIDQuaeKYzyc>

- Slides from this talk will be added to my website shortly.

	Design Patterns - Command Pattern Explanation and Implementation in C++ Mike Shah • 12K views • 2 years ago
	Design Patterns - Singleton Pattern Explanation and Implementation in C++ Mike Shah • 4.4K views • 2 years ago
	Design Patterns - Factory Method Pattern Explanation and Implementation in C++ Mike Shah • 5.6K views • 2 years ago
	Design Patterns - Factory Method Pattern Adding More Power to Count Allocated Objects in C++ Mike Shah • 1.7K views • 2 years ago
	Design Patterns - The Extensible Factory Pattern in C++ Register Objects at Runtime Mike Shah • 2K views • 2 years ago
	Design Patterns - Iterator Pattern Explanation and usage with STL in C++ Mike Shah • 1.6K views • 2 years ago
	The Observer Design Pattern in C++ - Part 1 of n - A simple implementation Mike Shah • 3.8K views • 11 months ago
	The Observer Design Pattern in C++ - Part 2 of n - Extensibility and Abstraction Mike Shah • 1.7K views • 11 months ago

CFTT
College of Farabi Tech Talks

نشست تخصصی آشنایی با

Design Patterns

الگوهای طراحی

Mike Shah
Associate Teaching Professor of Computer Sciences
Khoury College Northeastern University
Boston, Massachusetts, USA

دانشیار علوم کامپیوتر
دانشگاه نورس ایسترن، کالج خوری
بوستون، ایالات متحده آمریکا



Computer Engineering Scientific Association of University of Tehran, Iran
انجمن علمی دانشجویی مهندسی کامپیوتر دانشکده‌گان فارابی دانشگاه تهران

Date	Time	زمان	تاریخ
Thursday January 4th	17:30 GMT	ساعت ۳۱	پنجشنبه ۱۴ دی ماه



 Cesa_ut  @Cesa_ut  @Cesa_ut

Thank you College of Farabi University of Tehran and the Computer Engineering Scientific Association

Thinking about Design with Patterns

Visitor Pattern -- in C++ with Mike Shah

Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
 **YouTube**
www.youtube.com/c/MikeShah

17:30 - 19:00 Thur, January 4, 2024

~90 minutes | Introductory Audience

Thank you!

Empty Slide