Back to Basics:
The Factory Pattern

MIKE SHAH

Please do not redistribute slides without prior permission.

Software Design: Factory Pattern

Mike Shah, Ph.D.
@MichaelShah | mshah.io | www.youtube.com/c/MikeShah

2:00 pm MDT, Wed. October 27

60 minutes | Introductory Audience

Abstract

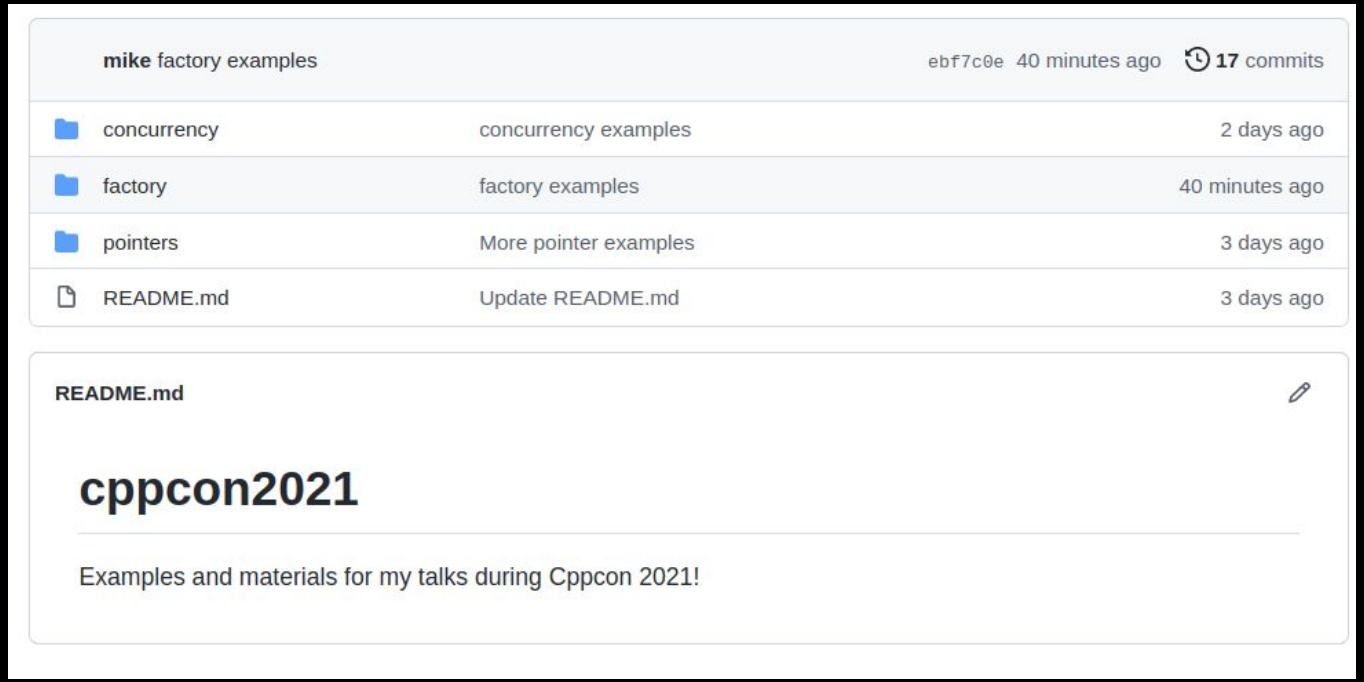
The abstract that you read and enticed you to join me is here!

C++ programs that are dynamic in nature have to create objects at some time during run-time. New objects can be created by explicitly calling ‘new’ and then the data type of that object. However, this requires that a programmer knows at ‘compile-time’ what object should be created. What we would like, is to have a layer of abstraction, or someway to create objects at run-time to reflect the dynamic nature of some C++ programs. Luckily, there is a common pattern that can help solve this problem--the factory design pattern.

In this talk, we are going to discuss a creational design pattern known as a factory. The pattern can be as simple as a function, or take on other forms as a distributed factory, or an abstract factory. We’ll show some basic examples of a factory in modern C++ as well as real world use cases of where factories occur for further study. Finally, we’ll discuss the tradeoffs of the factory pattern, and discuss which scenarios you may not actually want to use a factory. Attendees will leave this talk with the knowledge to go forward and implement the factory pattern, as well as how to spot the factory pattern in projects they may already be working on!

Code for the talk

Available here: https://github.com/MikeShah/cppcon2021



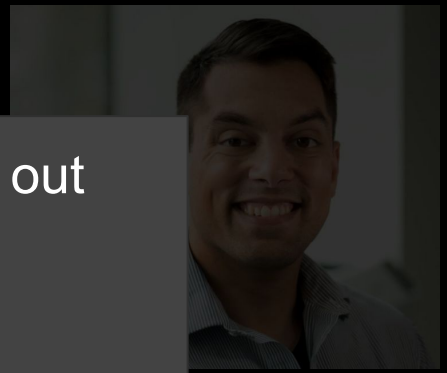
Who Am I?

by Mike Shah

- Assistant 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

Who Am I?

by Mike Shah



- Assistant Teaching Professor at Northeastern University in Boston, Massachusetts
  -
  -
- I do
  Cond
  - (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

This will be an interactive session, please shout out answers

(Note: I never call on audience members randomly, I did not like that as a student)

I was asking a few folks at the conference earlier a question:

I was asking a few folks at the conference earlier a question:

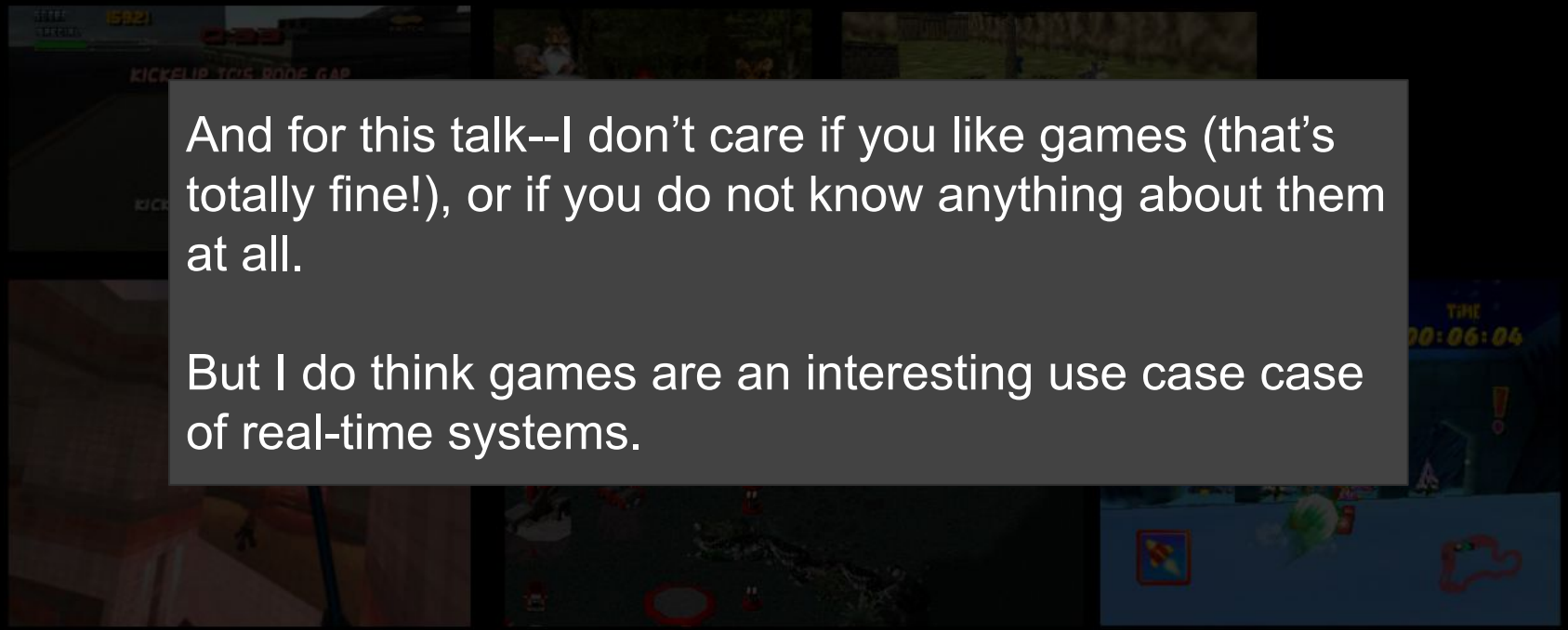
How did you get your start in Programming?

(Or rather--what domain peaked your interest?)

Video Games! (That's my answer) (1/2)

(Here are some favorites from around the time I was learning how to program, can you name them all?)

Video Games! (That's my answer) (2/2)



And for this talk--I don't care if you like games (that's totally fine!), or if you do not know anything about them at all.

But I do think games are an interesting use case case of real-time systems.

(Here are some favorites from around the time I was learning how to program, can you name them all?)

Real time systems (1/2)

- So just looking at the animation on the right--there is a lot going on.
  - There are many different types of objects
    - Some of these objects are being destroyed
    - New objects are being created
  - Objects are moving around
  - Artificial intelligence (path finding and other decision making)
  - In-game resource management is taking place

Command and Conquer Red Alert

Real time systems (2/2)

- So just looking at the animation on the
  - For today's talk, I want us to think about software design for a real time application--I'll use a 'game' as an example
  - other decision making)
  - In-game resource management is taking place

Command and Conquer Red Alert

My expectations and why this talk exists (1/2)

- This talk is part of the Software Design Track at Cppcon
  - Klaus Iglberger and I (Klaus doing the majority of the work!) put together the software design track
    - Part of this track we thought would be good to have some ‘tutorial like’ or ‘more fundamental’ (i.e. like the back to the basics) talks on Design Patterns.
    - (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 beginners
  - 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
- The design pattern of today is....
  - But before I spoil the pattern (even though it’s the name of the talk) let’s think about what our goals are

My expectations and why this talk exists (2/2)

- This talk is part of the Software Design Track at Cppcon
  - Klaus Iglberger and I (Klaus doing the majority of the work!) put together the software design track
    - Part of this track we thought would be good to have some ‘tutorial like’ or ‘more fundamental’ (i.e. like the back to the basics) talks on Design Patterns.
    - (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 beginners
  - 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
- The design pattern of today is....
  - But before I spoil the pattern (even though it’s the name of the talk) let’s think about what our goals are

A user-driven application

(e.g., a video game)

Question to Audience:

- If I have a user-driven application (e.g., a game)
- And part of that game is the users ability to ‘create’ objects at run-time.
  - How well do you think I as a developer can predict at compile-time what objects to create?
- What are your thoughts?

(Observe the user creating a new object)

Thought Process (1/16)

- *How well do you think I as a developer can predict at <u>compile-time</u> what objects to create? (in a very dynamic program...)*

(Observe the user creating a new object)

Thought Process (2/16)

- *How well do you think I as a developer can predict at compile-time what objects to create? (in a very dynamic program...)*
  - Maybe we could guess 100 of each type of our objects?
  - Maybe we have played our game a few times and ‘100’ feels like a good number (or otherwise you have empirical evidence).
    - What happens if a player goes over?
    - Maybe we restrict them? Maybe reallocate

```
1  // @file compiletime1.cpp
2  // g++ -std=c++17 compiletime1.cpp
3
4  class ObjectType1;
5  class ObjectType2;
6  class ObjectType3;
7
8  int main(){
9
10     // Create your units ahead of time.
11     ObjectType1 units1[100];
12     ObjectType2 units2[100];
13     ObjectType3 units3[100];
14
15
16     while(true){
17         // Run your game here
18
19         // Iterate through your units
20         // update them, run logic, etc.
21     }
22
23     return 0;
24 }
```

TI 3/16)

In a video game, let's assume these objects are potentially very large

We may run out of stack space here! (Click here to review stack memory)

- as a

compile-time what objects to create?

(in a very dynamic program...)

  - Maybe we could guess 100 of each type of our objects?
  - Maybe we have played our game a few times and '100' feels like a good number (or otherwise you have empirical evidence).
    - What happens if a player goes over?
    - Maybe we restrict them? Maybe reallocate

```cpp
// @file compiletime1.cpp
// g++ -std=c++17 compiletime1.cpp

class ObjectType1;
class ObjectType2;
class ObjectType3;

int main(){

    // Create your units ahead of time.
    ObjectType1 units1[100];
    ObjectType2 units2[100];
    ObjectType3 units3[100];


    while(true){
        // Run your game here

        // Iterate through your units
        // update them, run logic, etc.
    }

    return 0;
}
```

Thought Process (4/16)

- *How well do you think I as a developer can predict at compile-time what objects to create? (in a very dynamic program...)*
  - So, let's allocate our memory on the heap.
    - We can also resize if the user goes over the limit.
      - What's the problem here?

```cpp
// @file compiletime2.cpp
// g++ -std=c++17 compiletime2.cpp

class ObjectType1;
class ObjectType2;
class ObjectType3;

int main(){

    // Create your units ahead of time.
    ObjectType1* units1 = new ObjectType1[100];
    ObjectType2* units2 = new ObjectType2[100];
    ObjectType3* units3 = new ObjectType3[100];


    while(true){
        // Run your game here

        // Iterate through your units
        // update them, run logic, etc.
    }
    delete[] units1;
    delete[] units2;
    delete[] units3;

    return 0;
}
```

Well, if I need 200 units, that copy takes O(n) time.

(Or--even if I shrink the array, I have to reallocate.)

```
// Delete old array and copy in data
ObjectType1* ResizeObjectType1(ObjectType1* array,
                               size_t oldsize,
                               size_t newsize)
{
    ObjectType1* newArray = new ObjectType1[newsize];
    for(size_t i=0; i < oldsize; i++){
        newArray[i] = array[i];
    }
    delete[] array;
    return newArray;
}
```

- So, let's allocate our memory on the heap.
  - We can also resize if the user goes over the limit.
    - What's the problem here?

```
// @file compiletime2.cpp
// g++ -std=c++17 compiletime2.cpp

class ObjectType1;
class ObjectType2;
class ObjectType3;

int main(){

    // Create your units ahead of time.
    ObjectType1* units1 = new ObjectType1[100];
    ObjectType2* units2 = new ObjectType2[100];
    ObjectType3* units3 = new ObjectType3[100];


    while(true){
        // Run your game here

        // Iterate through your units
        // update them, run logic, etc.
    }
    delete[] units1;
    delete[] units2;
    delete[] units3;

    return 0;
}
```

Thought Process (6/16)

- *How well do you think I as a developer can predict at compile-time what objects to create? (in a very dynamic program...)*
  - So I don't want to manage resizing, so I'll just use a data structure.
    - I could also preallocate some objects (say 100 again) if I know that's reasonable for the game.
      - (Often games prefer to preallocate (i.e., give you a load screen))

```cpp
// @file compiletime4.cpp
// g++ -std=c++17 compiletime4.cpp
#include <vector>

class ObjectType1{};
class ObjectType2{};
class ObjectType3{};

int main(){

    // Create your units ahead of time.
    std::vector<ObjectType1> units1;
    std::vector<ObjectType2> units2;
    std::vector<ObjectType3> units3;


    while(true){
        // Run your game here

        // Iterate through your units
        // update them, run logic, etc.
    }

    return 0;
}
```

Thought Process (7/16)

- *How well do you think I as a developer can predict at compile-time what objects to create? (in a very dynamic program...)*
  - Now, what if half-way through development the team decides we don't want 'ObjectType3'--it does not make the game fun?
    - I have to delete everywhere.

```cpp
// @file compiletime4.cpp
// g++ -std=c++17 compiletime4.cpp
#include <vector>

class ObjectType1{};
class ObjectType2{};
class ObjectType3{};

int main(){

    // Create your units ahead of time.
    std::vector<ObjectType1> units1;
    std::vector<ObjectType2> units2;
    std::vector<ObjectType3> units3;


    while(true){
        // Run your game here

        // Iterate through your units
        // update them, run logic, etc.
    }

    return 0;
}
```

Thought Process (8/16)

- *How well do you think I as a developer can predict at compile-time what objects to create? (in a very dynamic program...)*
  - Okay, ObjectType3 is gone.
    - I still need to create my 100 objects, and perhaps some of them are going to be different (meaning created using different constructors)...

```cpp
// @file compiletime5.cpp
// g++ -std=c++17 compiletime5.cpp
#include <vector>

class ObjectType1{
    ObjectType1(){}
    ObjectType1(int x,int y){ // do some work}
};

class ObjectType2{};

int main(){

    // Create your units ahead of time.
    std::vector<ObjectType1> units1;
    std::vector<ObjectType2> units2;

    // Wait, which constructor do I use?
    ObjectType1 myObject1;
    ObjectType1 myObject2(10,20);
    // Do I care....does it matter, do I know?
    units1.push_back(myObject1);
    units2.push_back(myObject2);
    // ...
```

Thought Process (9/16)

- *How well do you think I as a developer can predict at compile-time what objects to create? (in a very dynamic program...)*
  - Wait...
  - How do I even construct my objects?
    - What if there are multiple constructors?
    - Do I think I'll be able to guess at compile-time which one to use?

```
1  // @file compiletime5.cpp
2  // g++ -std=c++17 compiletime5.cpp
3  #include <vector>
4
5  class ObjectType1{
6      ObjectType1(){}
7      ObjectType1(int x,int y){ // do some work}
8  };
9
10 class ObjectType2{};
11
12 int main(){
13
14     // Create your units ahead of time.
15     std::vector<ObjectType1> units1;
16     std::vector<ObjectType2> units2;
17
18     // Wait, which constructor do I use?
19     ObjectType1 myObject1;
20     ObjectType1 myObject2(10,20);
21     // Do I care....does it matter, do I know?
22     units1.push_back(myObject1);
23     units2.push_back(myObject2);
24     // ...
```

Thought Process (10/16)

- *How well do you think I as a developer can predict at compile-tim what objects to create? (in a very dynamic program...)*
  - Wait...
  - How do I even construct my objects?
    - What if there are multiple constructors?
    - Do I think I'll be able to guess at compile-time which one to use?
      - Okay, I can again try to fix that...let's just have one function (and I need to do this for each type)

```
// Try to handle creation of objects here
void makeObject1AndPushToVector(std::vector<ObjectType1>& units1Vector,
                                int x,
                                int y){
    ObjectType1 newObject(x,y);
    units1Vector.push_back(newObject);
}


int main(){

    // Create your units ahead of time.
    std::vector<ObjectType1> units1;
    std::vector<ObjectType2> units2;

    // Wait, which constructor do I use?
    makeObject1AndPushToVector(units1,0,0);
    makeObject1AndPushToVector(units1,10,20);
```

Thought Process (11/16)

And maybe I could generalize this function to handle the creation of multiple objects.

Certainly that could be a parameter!

If there are different numbers of arguments....well, I'll just use the maximum number (and guess the types that may be used...)

Hmm, I probably should be careful about passing in my std::vector as a parameter--but by reference is okay...

- Okay, I can again try to fix that...let's just have one function (and I need to do this for each type)

```cpp
// @file compiletime5.cpp
// g++ -std=c++17 compiletime5.cpp
#include <vector>

// Try to handle creation of objects here
void makeObject1AndPushToVector(std::vector<ObjectType1>& units1Vector,
                                int x,
                                int y){
    ObjectType1 newObject(x,y);
    units1Vector.push_back(newObject);
}

int main(){

    // Create your units ahead of time.
    std::vector<ObjectType1> units1;
    std::vector<ObjectType2> units2;

    // Wait, which constructor do I use?
    makeObject1AndPushToVector(units1,0,0);
    makeObject1AndPushToVector(units1,10,20);
    // Create your units ahead of time.
    std::vector<ObjectType1> units1;
    std::vector<ObjectType2> units2;

    // Wait, which constructor do I use?
    makeObject1AndPushToVector(units1,0,0);
    makeObject1AndPushToVector(units1,10,20);
    // ...
    while(true){
        // Run your game here

        // Iterate through your units
        // update them, run logic, etc.
    }

    return 0;
}
```

Thought Process (12/16)

So we're slowly getting closer to something interesting here.

I like that we can create all of our objects in one place.

The parameters and their types that may (or may not be used) is concerning to get correct.

I also don't particularly like that we made units1 and units2 global to simplify this function

And, I'm starting to think about where these objects will live (right now in a global vector...probably a bad idea)

- Okay, I can again try to fix that...let's just have one function (and I need to do this for each type)

```
// Create your units ahead of time.
std::vector<ObjectType1> units1;
std::vector<ObjectType2> units2;

// Try to handle creation of objects here
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}


int main(){

    makeObject(1,0,0);
    makeObject(1,10,20);
    // Look different object types!
    makeObject(2,10,20);

    // ...
```

So... this works.

Question to Audience: How do I know when I'm done?
(Your thoughts)

```
// Create your units ahead of time.
std::vector<ObjectType1> units1;
std::vector<ObjectType2> units2;

// Try to handle creation of objects here
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}


int main(){

    makeObject(1,0,0);
    makeObject(1,10,20);
    // Look different object types!
    makeObject(2,10,20);

    // ...
```

So... this works.

Question to Audience: How do I know when I'm done?
(Your thoughts)

- Maybe a deadline hits, maybe it's *good enough*, maybe we are tired...
- How about some better criteria--where I can check off three boxes
  - When we're convinced our solution (especially if the code is going to live a long time) is:
    - ❑ Flexible
    - ❑ Maintainable
    - ❑ Extensible

```
// Create your units ahead of time.
std::vector<ObjectType1> units1;
std::vector<ObjectType2> units2;

// Try to handle creation of objects here
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}


int main(){

    makeObject(1,0,0);
    makeObject(1,10,20);
    // Look different object types!
    makeObject(2,10,20);

    // ...
```

- So is this code (‘makeObject’ specifically)
  - Flexible?
    - ❏ (your thoughts?)
  - Maintainable
    - ❏ (your thoughts?)
  - Extensible
    - ❏ (your thoughts?)

```
// Create your units ahead of time.
std::vector<ObjectType1> units1;
std::vector<ObjectType2> units2;

// Try to handle creation of objects here
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}


int main(){

    makeObject(1,0,0);
    makeObject(1,10,20);
    // Look different object types!
    makeObject(2,10,20);

    // ...
```

- So is this code ('makeObject' specifically)
  - Flexible?
    - sort-of, we can extend our list of *objectTypes*
    - But how many params should we add?
      - Maybe we can have a separate ObjParamsType and use inheritance for the types (okay interesting idea...needs to be worked out more)
  - Maintainable?
    - Not really
      - If I remove an object type, do I renumber?
      - What about the parameters?
      - How much knowledge do I need to add to this function--it seems like a lot!
  - Extensible?
    - I suppose--we can add ObjectTypes
    - BUT, we have to figure out which vector to add to...
- **I'll give this score a 1 out of 3--we can probably do better.**

```
// Create your units ahead of time.
std::vector<ObjectType1> units1;
std::vector<ObjectType2> units2;

// Try to handle creation of objects here
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}


int main(){

    makeObject(1,0,0);
    makeObject(1,10,20);
    // Look different object types!
    makeObject(2,10,20);

    // ...
```

What Problem Am I Trying to Solve?

Claim: If I have a user-driven application (e.g., a game)

A. It can be difficult to figure out how to create objects of different types
    a. (And we probably cannot do this well at compile-time)
B. It can be difficult to figure out ‘where’ to create objects
    a. (i.e., If I have lots of free functions)

And it’s worth thinking about this problem at scale--where I have 10 different types, or even 100 different object types.

So let's think about this game (1/2)

- There are many different types of objects
  - How do we *create the different objects* in this real-time application such that our code design is:
    - flexible
    - maintainable
    - and extensible
  - In other words--what is the right pattern?

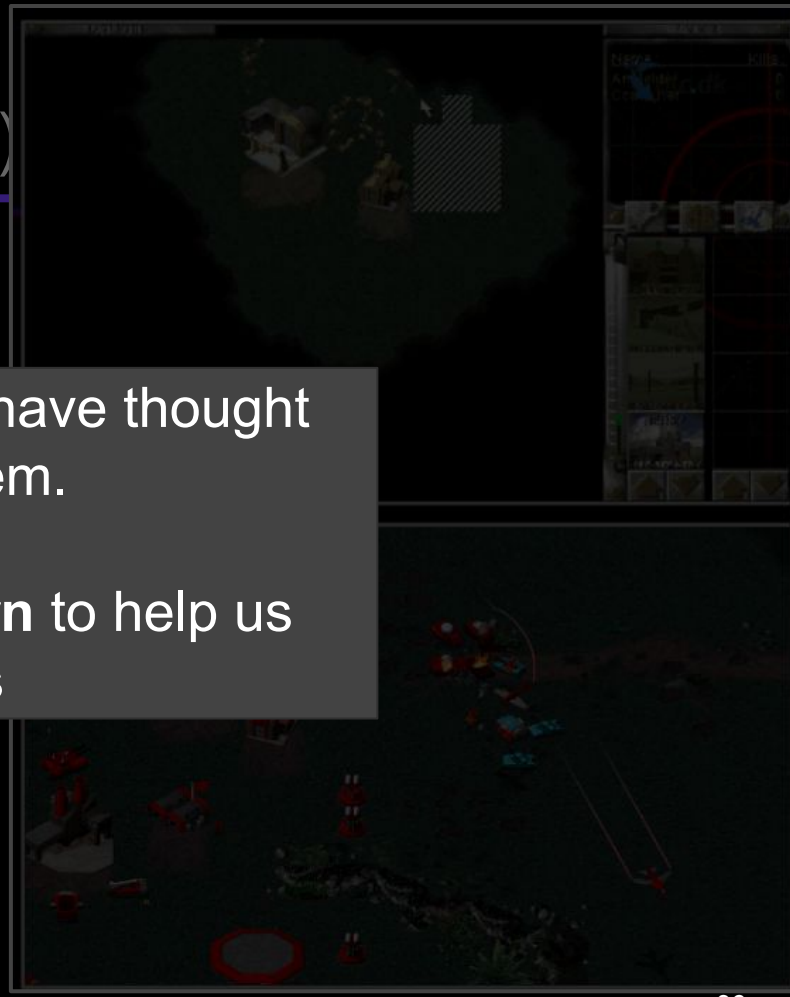
Command and Conquer Red Alert

So let's think about this game (2/2)

- There are many different types of objects
  - How do we *create the different objects* in this real-time
    is:
    - fle
    - ma
    - an
  - In other

Luckily, some smart folks have thought about this problem.

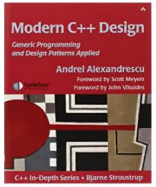
We have a **design pattern** to help us create objects

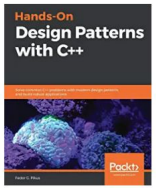


Command and Conquer Red Alert

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
  - (Recall, these are our three criteria we’d like to satisfy)

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.

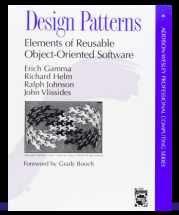
Design Patterns Book * Brief Aside *

- 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

Design Patterns Book (1/2)

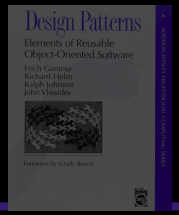


- So design patterns are reusable templates that can help us solve problems that occur in software
  - One (of the many) *nice* thing the Design Patterns Gang of Four (GoF) book does is organize the 23* presented design patterns into three categories:
    - Creational
    - Structural
    - Behavioral

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

Design pattern relationships

Design Patterns Book (2/2)



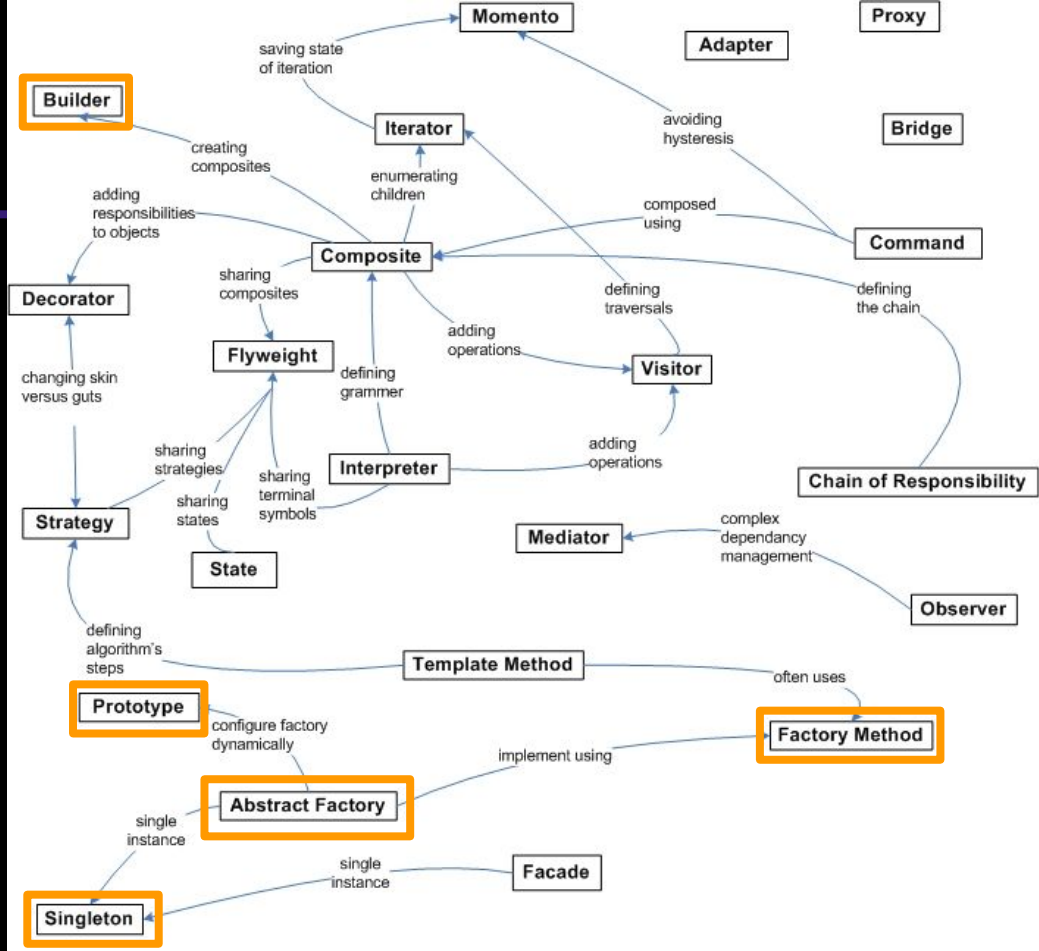
Today we are focusing on ‘creation’ of objects

I’ve highlighted the 5 creational patterns.

Design Patterns Gang of Four (GoF) book does is organize the 23* presented design patterns into three categories:

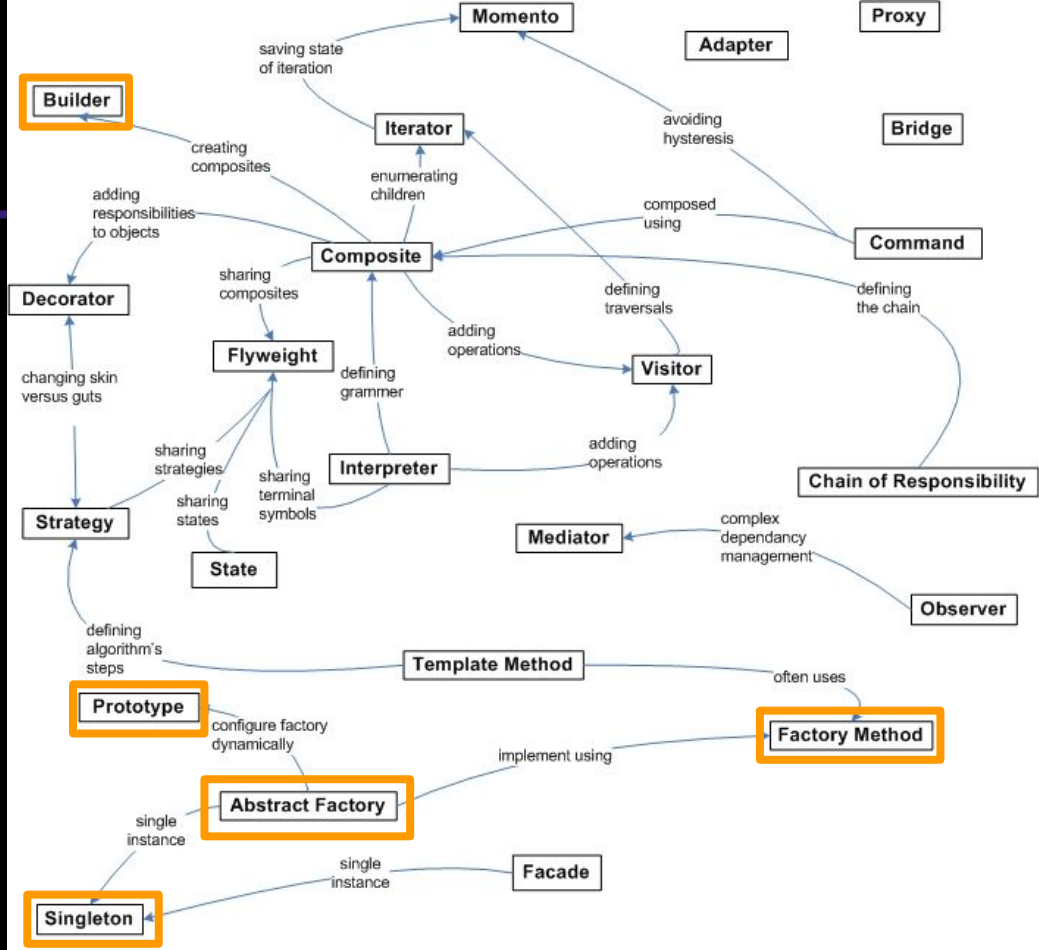
- Creational
- Structural
- Behavioral

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



Design pattern relationships

Creational Design Patterns



Design pattern relationships

Creational Design Patterns

- Provide program more flexibility on how to create objects, often avoiding direct instantiation of a specific object.
  - So this means:
    - We try to **avoid directly creating instances of objects** in our code:
      - `ObjectType1 myObject = new ObjectType1;`
    - We prefer instead to encapsulate how an object is created

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

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 speci

- Chain of responsibility delegates co
- Command creates objects which en
- Interpreter implements a specialized
- Iterator accesses the elements of a
- Mediator allows loose coupling betw
- Memento provides the ability to rest
- Observer is a publish/subscribe patt
- 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

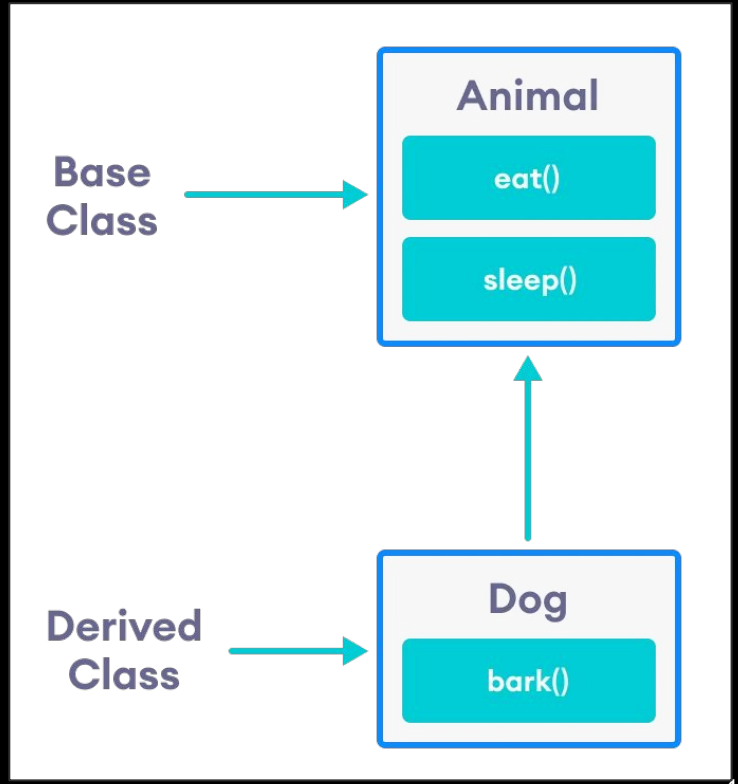
We are close to a creational pattern here

- We are *somewhat* encapsulating how we create our objects
  - It's just not very robust
    - What if a user types in an 'int' for the wrong objectType
    - Or we otherwise remove objectType's
      - (We also need to remove our vector)
  - We need to clean this up--and it will require thinking about our 'ObjectType' with a little more structure.

```
// Try to handle creation of objects here
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}
```

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

We were close in solving our problem

Our Object Inheritance Hierarchy (1/4)

- So to start, we're going to want some common Interface for which our different game objects can inherit from
  - This is probably a good idea to enforce (with the pure virtual member functions) properties of each Game Object.
  - Second, we can inherit from any `IGameObject` from this common interface to help ease our construction of different types of objects.
    - (again leveraging inheritance-based polymorphism)

```cpp
// @file hierarchy.cpp
// g++ -std=c++17 hierarchy.cpp
#include <vector>

class IGameObject{
    public:
        // Ensure derived classes call
        // the correct destructor (i.e., top of the chain)
        virtual ~IGameObject() {}
        // Pure virtual functions that must be
        // implemented by derived class
        virtual void ObjectPlayDefaultAnimation() = 0;
        virtual void ObjectMoveInGame() = 0;
        virtual void Update() = 0;
        virtual void Render() = 0;
};
```

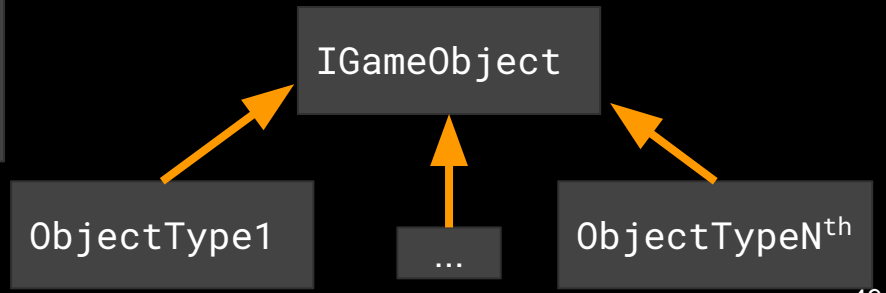
Our Object Inheritance Hierarchy (2/4)

- So observe to the bottom-right our inheritance hierarchy we want to establish.

  `ObjectType1 is-a IGameObject`
  `ObjectType2 is-a IGameObject`

```cpp
// @file hierarchy.cpp
// g++ -std=c++17 hierarchy.cpp
#include <vector>

class IGameObject{
    public:
        // Ensure derived classes call
        // the correct destructor (i.e., top of the chain)
        virtual ~IGameObject() {}
        // Pure virtual functions that must be
        // implemented by derived class
        virtual void ObjectPlayDefaultAnimation() = 0;
        virtual void ObjectMoveInGame() = 0;
        virtual void Update() = 0;
        virtual void Render() = 0;
};
```

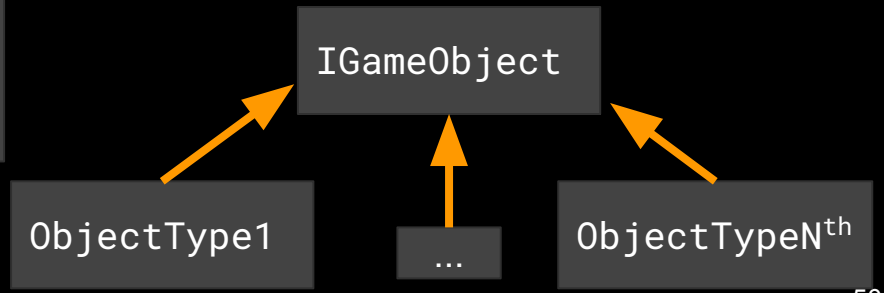


Our Object Inheritance Hierarchy (3/4)

- (Code) So we'll have something like this for each different ObjectType that we create

```
18 class ObjectType1 : IGameObject{
19     public:
20     ObjectType1(int x,int y){ /* ... */ }
21     void ObjectPlayDefaultAnimation() { /* ... */}
22     void ObjectMoveInGame() { /* ... */}
23     void Update() { /* ... */}
24     void Render() { /* ... */}
25 };
26
27 class ObjectType2 : IGameObject{
28     public:
29     ObjectType2(int x,int y){ /* ... */ }
30     void ObjectPlayDefaultAnimation() { /* ... */}
31     void ObjectMoveInGame() { /* ... */}
32     void Update() { /* ... */}
33     void Render() { /* ... */}
34 };
```

```
1  // @file hierarchy.cpp
2  // g++ -std=c++17 hierarchy.cpp
3  #include <vector>
4
5  class IGameObject{
       public:
           // Ensure derived classes call
8          // the correct destructor (i.e., top of the chain)
9          virtual ~IGameObject() {}
10         // Pure virtual functions that must be
11         // implemented by derived class
12         virtual void ObjectPlayDefaultAnimation() = 0;
13         virtual void ObjectMoveInGame() = 0;
14         virtual void Update() = 0;
15         virtual void Render() = 0;
16 };
```

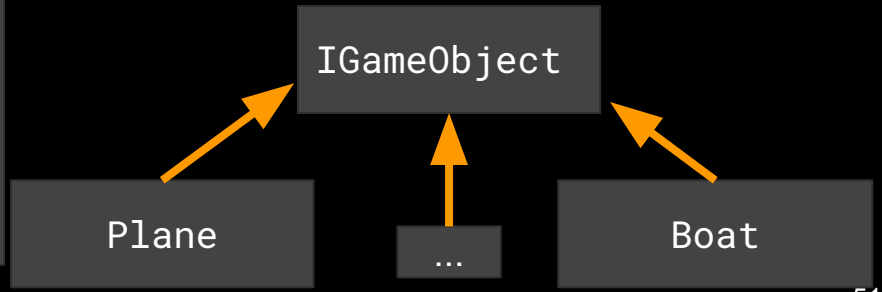


Our Object Inheritance Hierarchy (4/4)

```
18 class Plane : public IGameObject{
19     public:
20     Plane(int x,int y){ /* ... */ }
21     void ObjectPlayDefaultAnimation() { /* ... */}
22     void ObjectMoveInGame() { /* ... */}
23     void Update() { /* ... */}
24     void Render() { /* ... */}
25 };
26
27 class Boat: public IGameObject{
28     public:
29     Boat(int x,int y){ /* ... */ }
30     void ObjectPlayDefaultAnimation() { /* ... */}
31     void ObjectMoveInGame() { /* ... */}
32     void Update() { /* ... */}
33     void Render() { /* ... */}
34 };
```

- Two subtle changes, let's give more specific names to our objects (`Plane` and `Boat`), and make sure we're inheriting publicly `IGameObject`.

```
1  // @file hierarchy.cpp
2  // g++ -std=c++17 hierarchy.cpp
3  #include <vector>
4
5  class IGameObject{
6      public:
           // Ensure derived classes call
           // the correct destructor (i.e., top of the chain)
9          virtual ~IGameObject() {}
10         // Pure virtual functions that must be
11         // implemented by derived class
12         virtual void ObjectPlayDefaultAnimation() = 0;
13         virtual void ObjectMoveInGame() = 0;
14         virtual void Update() = 0;
15         virtual void Render() = 0;
16 };
```

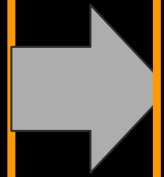


Updated ‘function’ to create objects (1/4)

- So next I have updated our ‘creation’ function shown on the right

```
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}
```

Before



```
37 // Enum class so we will create a type
38 enum class ObjectType{PLANE, BOAT};
39
40 // One single function to create our object types
41 // The object will just 'do the right thing'
42 IGameObject* CreateObjectFactory(ObjectType o){
43     switch(o){
44         case ObjectType::PLANE:
45             return new Plane(0,0);
46         case ObjectType::BOAT:
47             return new Boat(0,0);
48     }
49 }
```

After

Updated ‘function’ to create objects (2/4)

- The first major change is that I have an `enum class` for the different types of objects
  - (I could have also done this on the left)
  - This ensures we’ll create the correct object type (i.e., better than using a plain ‘int’)

```
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}
```

Before

```
37 // Enum class so we will create a type
38 enum class ObjectType{PLANE, BOAT};
39
40 // One single function to create our object types
41 // The object will just 'do the right thing'
42 IGameObject* CreateObjectFactory(ObjectType o){
43     switch(o){
44         case ObjectType::PLANE:
45             return new Plane(0,0);
46         case ObjectType::BOAT:
47             return new Boat(0,0);
48     }
49 }
```

After

- The second major idea, is that I have simplified the function function
  - We just return an `*IGameObject`
  - This is much cleaner that what we were doing previously
    - (The managing of which collection to push to is gone!)
  - We're also *moving towards* the 'Single Responsibility Principle' where I could create all of my objects in `CreateObjectFactory`

```
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}
```

Before

```
37 // Enu          so we will create a type
38 enum class ObjectType{PLANE, BOAT};
39
40 // One single function to create our object types
41 // The object will just 'do the right thing'
42 IGameObject* CreateObjectFactory(ObjectType o){
43     switch(o){
44         case ObjectType::PLANE:
45             return new Plane(0,0);
46         case ObjectType::BOAT:
47             return new Boat(0,0);
48     }
49 }
```

After

Updated ‘function’ to create objects (4/4)

- Another small change, is to slightly modify our return type to keep with the modern times :)
  - I recommend shared_ptr for this game example.
  - In a game, we might have multiple pointers to the same resource
    - e.g. Objects may share resources (e.g., pixel data, texture, etc.)

```
void makeObject(int objectType,
                int param1,
                int param2){
    if(1 == objectType){
        ObjectType1 newObject(param1,param2);
        units1Vector.push_back(newObject);
    }
    if(2 == objectType){
        ObjectType2 newObject;
        units2Vector.push_back(newObject);
    }
}
```

Before

```
38 // Enum class so we will create a type
39 enum class ObjectType{PLANE, BOAT};
40
41 // One single function to create our object types
42 // The object will just 'do the right thing'
43 std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o){
44     switch(o){
45         case ObjectType::PLANE:
46             return std::make_shared<Plane>(0,0);
47         case ObjectType::BOAT:
48             return std::make_shared<Boat>(0,0);
49     }
50 }
```

After

Usage in Main Loop (1/3)

- Here's the creation of our two different object types (Boat's and Planes)
  - Notice we only now have 1 collection (std::vector) to store our types
    - (Due to our abstraction layer for `IGameObject`)

```
53 int main(){
54     // Formerly our units1 and units2
55     std::vector<std::shared_ptr<IGameObject>> gameObjectCollection;
56     // Add the correct object to our collection
57     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
58     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
59     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
60     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::BOAT));
61     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::BOAT));
62
63     // Main Game Loop
64     while(true){
65         // Iterate through your game object
66         // update them, run logic, etc.
67         for(auto& e: gameObjectCollection){
68             e->Update();
69             e->Render();
70         }
71     }
72
73     return 0;
74 }
```

Usage in Main Loop (2/3)

- Here's the creation of our two different object types (Boat's and Planes)
  - Notice we only now have 1 collection (std::vector) to store our types
    - (Due to our abstraction layer for `IGameObject`)
  - Additionally notice the 'main game loop' is simplified
    - We only have to iterate through one collection

```
53 int main(){
54     // Formerly our units1 and units2
55     std::vector<std::shared_ptr<IGameObject>> gameObjectCollection;
56     // Add the correct object to our collection
57     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
58     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
59     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
60     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::BOAT));
61     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::BOAT));
62
63     // Main Game Loop
64     while(true){
65         // Iterate through your game object
66         // update them, run logic, etc.
67         for(auto& e: gameObjectCollection){
68             e->Update();
69             e->Render();
70         }
71     }
72
73     return 0;
74 }
```

Usage in Main Loop (3/3)

- Here's the creation of our two different object types (Boat's and Planes)
  - Notice we only now have 1 collection (std::vector) to store our types
    - (Due to our abstraction layer for `IGameObject`)
  - Additionally notice the 'main game loop' is simplified
    - We only have to iterate through one collection

```
53 int main(){
54     // Formerly our units1 and units2
55     std::vector<std::shared_ptr<IGameObject>> gameObjectCollection;
56     // Add the correct object to our collection
57     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
58     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
59     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
60     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::BOAT));
61     gameObjectCollection.push_back(CreateObjectFactory(ObjectType::BOAT));
62
63     // Main Game Loop
64     while(true){
65         // Iterate through your game object
66         // update them, run logic, etc.
67         for(auto& e: gameObjectCollection){
68             e->Update();
69             e->Render();
70         }
71     }
72
73     return 0;
74 }
```

Note: To experts--we can refactor for performance and a more 'data-oriented' approach. That is a separate talk--this is fine for our needs for now.

We have implemented

The Factory Method

(A creational design pattern)

The Factory Method (1/5)

The Factory Method pattern provides a generalized way to create instances of an object and can be a great way to hide implementation details for derived class

```
38 // Enum class so we will create a type
39 enum class ObjectType{PLANE, BOAT};
40
41 // One single function to create our object types
42 // The object will just 'do the right thing'
43 std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o){
44     switch(o){
45         case ObjectType::PLANE:
46             return std::make_shared<Plane>(0,0);
47         case ObjectType::BOAT:
48             return std::make_shared<Boat>(0,0);
49     }
50 }
```

Here is our factory--and perhaps we should also add a ‘default’ case which returns nullptr.

The Factory Method (2/5)

The Factory Method pattern provides a generalized way ***to create instances of an object*** *and can be a great way to hide implementation details for derived class*

- Yes, we have that!
  - We can add new types to our enum class and function easily.

```
38 // Enum class so we will create a type
39 enum class ObjectType{PLANE, BOAT};
40
41 // One single function to create our object types
42 // The object will just 'do the right thing'
43 std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o){
44     switch(o){
45         case ObjectType::PLANE:
46             return std::make_shared<Plane>(0,0);
47         case ObjectType::BOAT:
48             return std::make_shared<Boat>(0,0);
49     }
50 }
```

Here is our factory--and perhaps we should also add a ‘default’ case which returns nullptr.

The Factory Method (3/5)

The Factory Method pattern provides a generalized way ***to create instances of an object*** *and can be a great way to hide implementation details for derived class*

- We could extend our enum class creatively as well
  - e.g., PLANE_IN_AIR
    - This handles constructing the same types with different parameters or setup functions

```
38 // Enum class so we will create a type
39 enum class ObjectType{PLANE, BOAT};
40
41 // One single function to create our object types
42 // The object will just 'do the right thing'
43 std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o){
44     switch(o){
45         case ObjectType::PLANE:
46             return std::make_shared<Plane>(0,0);
47         case ObjectType::BOAT:
48             return std::make_shared<Boat>(0,0);
49     }
50 }
```

```
38 // Enum class so we will create a type
39 enum class ObjectType{PLANE,PLANE_IN_AIR, BOAT};
40
41 // One single function to create our object types
42 // The object will just 'do the right thing'
43 std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o){
44     switch(o){
45         case ObjectType::PLANE:
46             return std::make_shared<Plane>(0,0);
47         case ObjectType::PLANE_IN_AIR:
48             return std::make_shared<Plane>(0,100000); // (x,y)
49         case ObjectType::BOAT:
50             return std::make_shared<Boat>(0,0);
51     }
52 }
```

The Factory Method (4/5)

The Factory Method pattern provides a generalized way to create instances of an object and can be a great way to hide implementation details for derived class

```
38 // Enum class so we will create a type
39 enum class ObjectType{PLANE,PLANE_IN_AIR, BOAT};
40
41 // One single function to create our object types
42 // The object will just 'do the right thing'
43 std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o){
44     switch(o){
45         case ObjectType::PLANE:
46             return std::make_shared<Plane>(0,0);
47         case ObjectType::PLANE_IN_AIR:
48             return std::make_shared<Plane>(0,100000); // (x,y)
49         case ObjectType::BOAT:
50             return std::make_shared<Boat>(0,0);
51     }
52 }
```

Updated with another object type (PLANE_IN_AIR) that we can create

The Factory Method (5/5)

The Factory Method pattern provides a generalized way to create instances of an object and can be a ***great way to hide implementation details for derived class***

- This we did not talk about, but we can hide our implementation to clients of our API fairly well
  - Client really only needs to know that they can create IGameObject's

```
#ifndef FACTORY_HPP
#define FACTORY_HPP
// @directory /simplefactory
// g++ -std=c++17 main.cpp Factory.cpp
#include <memory>

// Declare our one interface type
class IGameObject;

// Enum class so we will create a type
// Could list it in the header here so clients
// know what types they can create
enum class ObjectType{PLANE,PLANE_IN_AIR, BOAT};

// One single function to create our object types
// The object will just 'do the right thing'
std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o);


#endif
```

This is the Factory.hpp
Here I'm only exposing the enum class (which could also be hidden) to the client of our API.

(Full example in ./simplefactory -- Not 100% optimal, but shows how to setup your Factory in a header file)

The Factory Method - Pros and Cons? (1/2)

- So, no design pattern is **perfect**, computer science is about trade-offs.
  - What do we like about this?
    - (i.e., the pros)
  - (Question to the audience)
    - Is this pattern:
      - Flexible
      - Maintainable
      - Extensible
  - What do we dislike?
    - (i.e., the cons)

```
38 // Enum class so we will create a type
39 enum class ObjectType{PLANE,PLANE_IN_AIR, BOAT};
40
41 // One single function to create our object types
42 // The object will just 'do the right thing'
43 std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o){
44     switch(o){
45         case ObjectType::PLANE:
46             return std::make_shared<Plane>(0,0);
47         case ObjectType::PLANE_IN_AIR:
48             return std::make_shared<Plane>(0,100000); // (x,y)
49         case ObjectType::BOAT:
50             return std::make_shared<Boat>(0,0);
51     }
52 }
```

Here is our factory--and perhaps we should also add a ‘default’ case which returns nullptr.

The Factory Method - Pros and Cons? (2/2)

- Pros
  - Flexible
    - Relatively flexible
  - Maintainable
    - 1 update to the enum class, and one update to the switch statement--not too bad.
  - Extensible
    - Creating new object types is done through inheritance is easy
- Cons
  - May need several factories for different hierarchies
  - Still potentially two ‘updates’ in two places of our code (i.e. the enum class and then in our actual function)
    - So potentially over-engineered for a very small project
- Other thoughts
  - You should probably think more about if you want to use shared_ptr or unique_ptr for your domain
    - (i.e., think about your ownership)
  - Probably need a way to ‘destroy’ all objects.

```
38 // Enum class so we will create a type
39 enum class ObjectType{PLANE, BOAT};
40
41 // One single function to create our object types
42 // The object will just 'do the right thing'
43 std::shared_ptr<IGameObject> CreateObjectFactory(ObjectType o){
44     switch(o){
45         case ObjectType::PLANE:
46             return std::make_shared<Plane>(0,0);
47         case ObjectType::BOAT:
48             return std::make_shared<Boat>(0,0);
49     }
50 }
```

Here is our factory--and perhaps we should also add a ‘default’ case which returns nullptr.

Some other *neat* ideas - Example of loading objects

- We can make our application data-driven using some simple configuration file
  - Much more intuitive in our pattern

```
1 plane
2 plane
3 boat
4 boat
5 plane_in_air
```

level1.txt

```cpp
// @directory data-driven
// g++ -std=c++17 main.cpp Factory.cpp -I./
#include <vector>
#include <memory>
#include <fstream>
#include <string>

#include "Factory.hpp"
#include "GameObjects.hpp"

int main(){
    std::vector<std::shared_ptr<IGameObject>> gameObjectCollection;
    // Add the correct object to our collection
    // based on a .txt file
    std::string line;
    std::ifstream myFile("level1.txt");
    if(myFile.is_open()){
        while(std::getline(myFile,line)){
            if(line=="plane"){
                gameObjectCollection.push_back(CreateObjectFactory(ObjectType::PLANE));
            }
            /* You get the idea...
            else if(line==""){

            }
            */
        }
    }
```

Some other *neat* ideas - Tracking Object Counts

- We may also want to manage object counts.
  - Several ways to do so
    - Could do it directly in each of our game objects
- Remember one of our earlier questions:
  - *How well do you think I as a developer can predict at compile-time what objects to create? (slide 17)*
  - *(The answer might be--an exact number)*

```
17 // @directory ./tracking
18 // GameObjects.hpp
19 class Plane : public IGameObject{
20     public:
21     Plane(int x,int y){
22         ObjectsCreated++;
23     }
24     void ObjectPlayDefaultAnimation() { /* ... */}
25     void ObjectMoveInGame() { /* ... */}
26     void Update() { /* ... */}
27     void Render() { /* ... */}
28     private:
29         static int ObjectsCreated;
30 };
```

```
1 // @file GameObjects.cpp
2 #include "GameObjects.hpp"
3
4 int Plane::ObjectsCreated = 0;
```

Let’s make our factory more extendable

Extensible Factory (Alexandrescu, 2001 in Modern C++ Design)
(Example based on Martin Reddy’s API Design for C++)

The Goal is to allow us at run-time to create new types (1/4)

- And this makes sense for a game, or some system that may be long running (and we want flexibility)
  - So I am going to create a ‘MyGameObjectFactory’
  - My class has all static member functions for now...I want to keep things simple
    - Pros/Cons of that can be discussed.

```
// One change is that I have removed our 'enum class'
// This is because during run-time I want to be able to
// create different types
class MyGameObjectFactory{
    public:
        // Callback function for creating an object
        typedef IGameObject *(*CreateObjectCallback)();
        // Register a new user created object type
        // Again, we also have to pass in how to 'create' an object.
        static void RegisterObject(const std::string& type, CreateObjectCallback cb){
            s_Objects[type] = cb;
        }
        // Unregister a user created object type
        // Remove from our map
        static void UnregisterObject(const std::string& type){
            s_Objects.erase(type);
        }
        // Our Previous 'Factory Method'
        //
        static IGameObject* CreateSingleObject(const std::string& type){
            CallBackMap::iterator it = s_Objects.find(type);
            if(it!=s_Objects.end()){
                // Call the callback function
                return (it->second)();
            }
            return nullptr;
        }
    private:
        // Convenience typedef
        typedef std::map<std::string, CreateObjectCallback> CallBackMap;
        // Map of all the different objects that we can create
        static CallBackMap s_Objects;
};
```

The Goal is to allow us at run-time to create new types (2/4)

- So the key component is the ability to ‘register’ and ‘unregister’ object types.

  Our types will now be stored in a std::map

  functions for now...I want to keep things simple
  - Pros/Cons of that can be discussed.

```
// One change is that I have removed our 'enum class'
// This is because during run-time I want to be able to
// create different types
class MyGameObjectFactory{
    public:
        // Callback function for creating an object
        typedef IGameObject *(*CreateObjectCallback)();
        // Register a new user created object type
        // Again, we also have to pass in how to 'create' an object.
        static void RegisterObject(const std::string& type, CreateObjectCallback cb){
            s_Objects[type] = cb;
        }
        // Unregister a user created object type
        // Remove from our map
        static void UnregisterObject(const std::string& type){
            s_Objects.erase(type);
        }
        // Our Previous 'Factory Method'
        //
        static IGameObject* CreateSingleObject(const std::string& type){
            CallBackMap::iterator it = s_Objects.find(type);
            if(it!=s_Objects.end()){
                // Call the callback function
                return (it->second)();
            }
            return nullptr;
        }
    private:
        // Convenience typedef
        typedef std::map<std::string, CreateObjectCallback> CallBackMap;
        // Map of all the different objects that we can create
        static CallBackMap s_Objects;
};
```

The Goal is to allow us at run-time to create new types (3/4)

- So for whatever type we are creating, we'll pass in a callback function for that type.

  This is one way you could implement a 'plugin system' to your software.

  functions for now...I want to keep things simple
  - Pros/Cons of that can be discussed.

```
11 // One change is that I have removed our 'enum class'
12 // This is because during run-time I want to be able to
13 // create different types
14 class MyGameObjectFactory{
15     public:
16         // Callback function for creating an object
17         typedef IGameObject *(*CreateObjectCallback)();
18         // Register a new user created object type
19         // Again, we also have to pass in how to 'create' an object.
20         static void RegisterObject(const std::string& type, CreateObjectCallback cb){
21             s_Objects[type] = cb;
22         }
23         // Unregister a user created object type
24         // Remove from our map
25         static void UnregisterObject(const std::string& type){
26             s_Objects.erase(type);
27         }
28         // Our Previous 'Factory Method'
29         //
30         static IGameObject* CreateSingleObject(const std::string& type){
31             CallBackMap::iterator it = s_Objects.find(type);
32             if(it!=s_Objects.end()){
33                 // Call the callback function
34                 return (it->second)();
35             }
36             return nullptr;
37         }
38     private:
39         // Convenience typedef
40         typedef std::map<std::string, CreateObjectCallback> CallBackMap;
41         // Map of all the different objects that we can create
42         static CallBackMap s_Objects;
43 };
```

The Goal is to allow us at run-time to create new types (4/4)

- Our previous ‘Factory Pattern’ is almost the same.

  The difference is we have to search for the type (as new types could be registered at run-time at any time)

  (Maybe this is not as fast or direct--as always, patterns have trade-offs)

discussed.

```
11 // One change is that I have removed our 'enum class'
12 // This is because during run-time I want to be able to
13 // create different types
14 class MyGameObjectFactory{
15     public:
16         // Callback function for creating an object
17         typedef IGameObject *(*CreateObjectCallback)();
18         // Register a new user created object type
19         // Again, we also have to pass in how to 'create' an object.
20         static void RegisterObject(const std::string& type, CreateObjectCallback cb){
21             s_Objects[type] = cb;
22         }
23         // Unregister a user created object type
24         // Remove from our map
25         static void UnregisterObject(const std::string& type){
             s_Objects.erase(type);
           }
           // Our Previous 'Factory Method'
           //
30         static IGameObject* CreateSingleObject(const std::string& type){
31             CallBackMap::iterator it = s_Objects.find(type);
32             if(it!=s_Objects.end()){
33                 // Call the callback function
34                 return (it->second)();
35             }
36             return nullptr;
37         }
38     private:
39         // Convenience typedef
40         typedef std::map<std::string, CreateObjectCallback> CallBackMap;
41         // Map of all the different objects that we can create
42         static CallBackMap s_Objects;
43 };
```

Creating our Previous Types (1/2)

- To the right we can see how to create our previous types: plane and boat.

```
32 int main(){
33     // Register a Different type
34     MyGameObjectFactory::RegisterObject("plane",Plane::Create);
35     MyGameObjectFactory::RegisterObject("boat",Boat::Create);
36
37     std::vector<IGameObject*> gameObjectCollection;
38     // Add the correct object to our collection
39     // based on a .txt file
40     std::string line;
41     std::ifstream myFile("level1.txt");
42     if(myFile.is_open()){
43         while(std::getline(myFile,line)){
44             // TODO: We'll have to iterate through 'all objects', but we can just
45             //       match on the string of our objects from our configuration now.
46             // FIXME: An exercise for the reader/viewer :)
47             if(line=="plane"){
48                 gameObjectCollection.push_back(MyGameObjectFactory::CreateSingleObject("plane"));
49             }
50             /* You get the idea...
51             else if(line==""){
52
53             }
54             */
```

Creating our Previous Types (2/2)

- To the right we can see how to create our previous types: plane and boat.

```
13 // For fun, create a new type
14 class Ant : public IGameObject{
15     public:
16     Ant(int x,int y){
17         ObjectsCreated++;
18     }
19     void ObjectPlayDefaultAnimation() { /* ... */}
20     void ObjectMoveInGame() { /* ... */}
21     void Update() { /* ... */}
22     void Render() { /* ... */}
23     static IGameObject* Create() {
24         return new Ant(0,0);
25     }
26     private:
27         static int ObjectsCreated;
28 };
29
30 int Ant::ObjectsCreated = 0;
```

```
32 int main(){
33     // Register a Different type
34     MyGameObjectFactory::RegisterObject("plane",Plane::Create);
35     MyGameObjectFactory::RegisterObject("boat",Boat::Create);
36
37     std::vector<IGameObject*> gameObjectCollection;
38     // Add the correct object to our collection
39     // based on a .txt file
40     std::string line;
41     std::ifstream myFile("level1.txt");
42     if(myFile.is_open()){
43         while(std::getline(myFile,line)){
44             // TODO: We'll have to iterate through 'all objects', but we can just
45             //       match on the string of our objects from our configuration now.
46             // FIXME: An exercise for the reader/viewer :)
47             if(line=="plane"){
48                 gameObjectCollection.push_back(MyGameObjectFactory::CreateSingleObject("plane"));
49             }
50             /* You get the idea...
51             else if(line==""){
52
53             }
54             */
```

```
36     MyGameObjectFactory::RegisterObject("ant",Ant::Create);
```

And here is our new data type (created by a user), and then later registered by the user

Is this pattern actually used?

Factory Method/Pattern Usage (1/6)

- I dug around a few open source projects to see if the factory pattern is actually used
  - `grep -irn “factory” .`
- The answer is yes!

Factory Method/Pattern Usag

- I dug around a few open source projects actually used
  - `grep -irn "factory" .`
- The answer is yes!
- https://github.com/horde3d/Horde3D


```
Horde3DEditor/src/GameEnginePlugIn/QGameEntityNode.h:61:         * To be able to use a factory, we need a static method as a callback that can be registered in the factory.
Horde3DEditor/src/GameEnginePlugIn/QGameEntityNode.h:62:         * This method will be registerd in the PlugInManager as an Extra node factory by the GameControllerAttachment using
Horde3DEditor/src/GameEnginePlugIn/ExtraTreeModel.h:50: ExtraTreeModel(PlugInManager* factory, const QDomElement& extrasRoot, QObject* parent = 0);
Horde3DEditor/src/GameEnginePlugIn/ExtraTreeModel.h:61: PlugInManager* nodeFactory() const {return m_extraNodeFactory;}
Horde3DEditor/src/GameEnginePlugIn/ExtraTreeModel.h:65: PlugInManager*  m_extraNodeFactory;
Horde3DEditor/src/GameEnginePlugIn/QExtraNode.cpp:50:   QXmlTreeNode* childItem = static_cast<ExtraTreeModel*>(m_model)->nodeFactory()->loadExtraNode(childNode, row, m_model, this);
Horde3DEditor/src/GameEnginePlugIn/ExtraTreeModel.cpp:32:ExtraTreeModel::ExtraTreeModel(PlugInManager* factory, const QDomElement& extrasRoot, QObject* parent /*=0*/) : QXmlTreeModel(parent),
Horde3DEditor/src/GameEnginePlugIn/ExtraTreeModel.cpp:33:m_extraNodeFactory(factory)
Horde3DEditor/src/HordeSceneEditorCore/ExtSceneNodePlugIn.h:62: virtual void setPlugInManager(PlugInManager* factory) = 0;
Horde3DEditor/src/HordeSceneEditorCore/SceneTreeModel.cpp:38:SceneTreeModel::SceneTreeModel(PlugInManager* factory, const QDomElement &node, QSceneNode* parentNode) : QXmlTreeModel(),
Horde3DEditor/src/HordeSceneEditorCore/SceneTreeModel.cpp:39:m_parentNode(parentNode), m_sceneNodeFactory(factory)
Horde3DEditor/src/HordeSceneEditorCore/SceneTreeModel.cpp:41:   m_rootItem = m_sceneNodeFactory->loadSceneNode(node, 0, this, parentNode);
Horde3DEditor/src/HordeSceneEditorCore/SceneTreeModel.h:44:     SceneTreeModel(PlugInManager* factory, const QDomElement &node, QSceneNode* parentNode);
Horde3DEditor/src/HordeSceneEditorCore/SceneTreeModel.h:60:     PlugInManager* nodeFactory() const {return m_sceneNodeFactory;}
Horde3DEditor/src/HordeSceneEditorCore/SceneTreeModel.h:74:     PlugInManager* m_sceneNodeFactory;
Horde3DEditor/src/HordeSceneEditorCore/QSceneNode.cpp:42:               m_knownNodeNames = model->nodeFactory()->sceneNodeNames();
Horde3DEditor/src/HordeSceneEditorCore/QSceneNode.cpp:76:               AttachmentPlugIn* plugIn = model->nodeFactory()->attachmentPlugIn();
Horde3DEditor/src/HordeSceneEditorCore/QSceneNode.cpp:87:       QSceneNode* childItem = model->nodeFactory()->loadSceneNode(childNode, row, model, this);
Horde3D/Source/Horde3DEngine/egMain.cpp:772:    SceneNode *sn = Modules::sceneMan().findType( SceneNodeTypes::Group )->factoryFunc( tpl );
Horde3D/Source/Horde3DEngine/egMain.cpp:786:    SceneNode *sn = Modules::sceneMan().findType( SceneNodeTypes::Model )->factoryFunc( tpl );
Horde3D/Source/Horde3DEngine/egMain.cpp:855:    SceneNode *sn = Modules::sceneMan().findType( SceneNodeTypes::Mesh )->factoryFunc( tpl );
Horde3D/Source/Horde3DEngine/egMain.cpp:867:    SceneNode *sn = Modules::sceneMan().findType( SceneNodeTypes::Joint )->factoryFunc( tpl );
Horde3D/Source/Horde3DEngine/egMain.cpp:886:    SceneNode *sn = Modules::sceneMan().findType( SceneNodeTypes::Light )->factoryFunc( tpl );
Horde3D/Source/Horde3DEngine/egMain.cpp:900:    SceneNode *sn = Modules::sceneMan().findType( SceneNodeTypes::Camera )->factoryFunc( tpl );
```

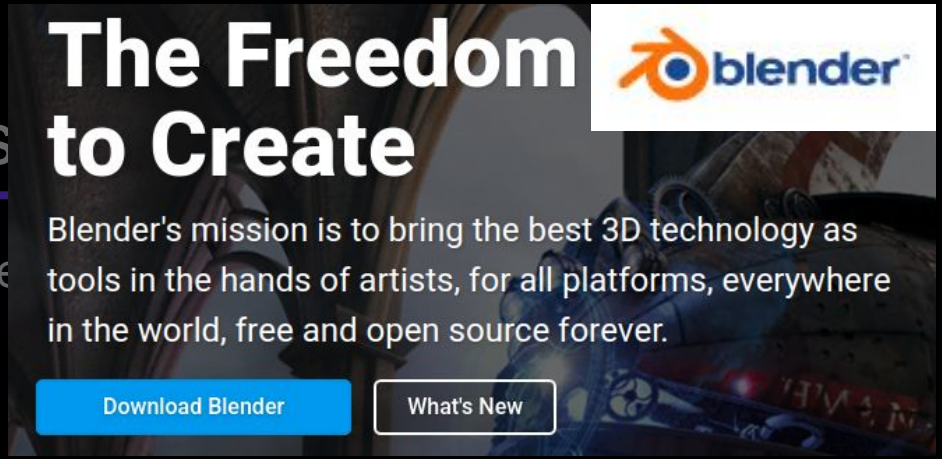
Factory Method/Pattern Usag

- I dug around a few open source projects actually used
  - `grep -irn “factory” .`
- The answer is yes!
- https://github.com/OGRECave/ogre

```
Components/Overlay/include/OgreOverlay.i:33:%include factory.i
Components/Overlay/include/OgreOverlay.i:65:%feature("director") Ogre::OverlayElementFactory;
Components/Overlay/include/OgreOverlay.i:66:%include "OgreOverlayElementFactory.h"
Components/Overlay/include/OgreOverlay.i:72:%factory(Ogre::OverlayElement* Ogre::OverlayManager::createOverlayElement, Ogre::OverlayContainer);
Components/Bites/src/OgreApplicationContextAndroid.cpp:107:    Ogre::ArchiveManager::getSingleton().addArchiveFactory( new Ogre::APKFileSystemArchiveFactory(mAAssetMgr) );
Components/Bites/src/OgreApplicationContextAndroid.cpp:108:    Ogre::ArchiveManager::getSingleton().addArchiveFactory( new Ogre::APKZipArchiveFactory(mAAssetMgr) );
Components/Volume/include/OgreVolumeOctreeNode.h:128:        /** Factory method to create octree nodes.
Components/Volume/include/OgreVolumeChunk.h:415:        /** Overridable factory method.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:146:A factory that enables creation of FFPFog instances.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:147:@remarks Sub class of SubRenderStateFactory
Components/RTShaderSystem/src/OgreShaderFFPFog.h:149:class FFPFogFactory : public SubRenderStateFactory
Components/RTShaderSystem/src/OgreShaderFFPFog.h:154:    @see SubRenderStateFactory::getType.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:159:    @see SubRenderStateFactory::createInstance.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:164:    @see SubRenderStateFactory::writeInstance.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:172:    @see SubRenderStateFactory::createInstanceImpl.
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:130:    mProgramWriterFactories.push_back(OGRE_NEW ShaderProgramWriterCGFactory());
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:131:    mProgramWriterFactories.push_back(OGRE_NEW ShaderProgramWriterGLSLFactory());
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:132:    mProgramWriterFactories.push_back(OGRE_NEW ShaderProgramWriterHLSLFactory());
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:134:    mProgramWriterFactories.push_back(OGRE_NEW ShaderProgramWriterGLSLESFactory());
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:138:        ProgramWriterManager::getSingletonPtr()->addFactory(mProgramWriterFactories[i]);
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:147:        ProgramWriterManager::getSingletonPtr()->removeFactory(mProgramWriterFactories[i]);
Components/RTShaderSystem/src/OgreShaderProgramWriterManager.cpp:60:void ProgramWriterManager::addFactory(ProgramWriterFactory* factory)
Components/RTShaderSystem/src/OgreShaderProgramWriterManager.cpp:62:    mFactories[factory->getTargetLanguage()] = factory;
Components/RTShaderSystem/src/OgreShaderProgramWriterManager.cpp:65:void ProgramWriterManager::removeFactory(ProgramWriterFactory* factory)
```

Factory Method/Pattern Us

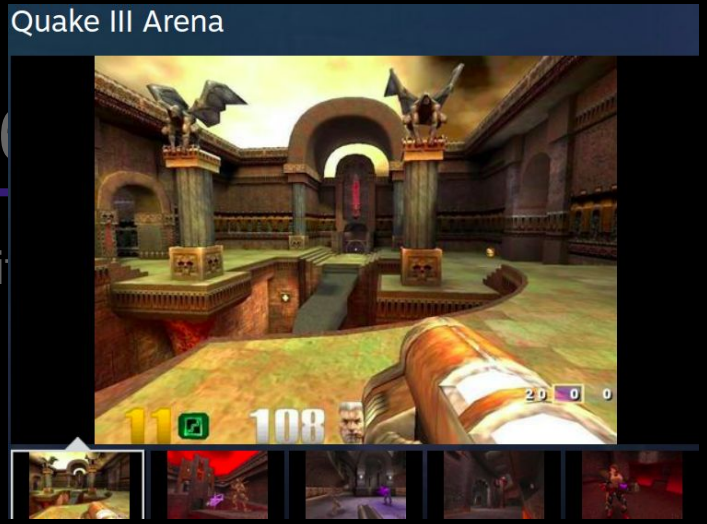
- I dug around a few open source proje actually used
  - `grep -irn “factory” .`
- The answer is yes!
- https://github.com/blender/blender




```
Components/Overlay/include/OgreOverlay.i:33:%include factory.i
Components/Overlay/include/OgreOverlay.i:65:%feature("director") Ogre::OverlayElementFactory;
Components/Overlay/include/OgreOverlay.i:66:%include "OgreOverlayElementFactory.h"
Components/Overlay/include/OgreOverlay.i:72:%factory(Ogre::OverlayElement* Ogre::OverlayManager::createOverlayElement, Ogre::OverlayContainer);
Components/Bites/src/OgreApplicationContextAndroid.cpp:107:    Ogre::ArchiveManager::getSingleton().addArchiveFactory( new Ogre::APKFileSystemArchiveFactory(mAAssetMgr) );
Components/Bites/src/OgreApplicationContextAndroid.cpp:108:    Ogre::ArchiveManager::getSingleton().addArchiveFactory( new Ogre::APKZipArchiveFactory(mAAssetMgr) );
Components/Volume/include/OgreVolumeOctreeNode.h:128:        /** Factory method to create octree nodes.
Components/Volume/include/OgreVolumeChunk.h:415:        /** Overridable factory method.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:146:A factory that enables creation of FFPFog instances.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:147:@remarks Sub class of SubRenderStateFactory
Components/RTShaderSystem/src/OgreShaderFFPFog.h:149:class FFPFogFactory : public SubRenderStateFactory
Components/RTShaderSystem/src/OgreShaderFFPFog.h:154:    @see SubRenderStateFactory::getType.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:159:    @see SubRenderStateFactory::createInstance.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:164:    @see SubRenderStateFactory::writeInstance.
Components/RTShaderSystem/src/OgreShaderFFPFog.h:172:    @see SubRenderStateFactory::createInstanceImpl.
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:130:    mProgramWriterFactories.push_back(OGRE_NEW ShaderProgramWriterCGFactory());
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:131:    mProgramWriterFactories.push_back(OGRE_NEW ShaderProgramWriterGLSLFactory());
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:132:    mProgramWriterFactories.push_back(OGRE_NEW ShaderProgramWriterHLSLFactory());
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:134:    mProgramWriterFactories.push_back(OGRE_NEW ShaderProgramWriterGLSLESFactory());
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:138:        ProgramWriterManager::getSingletonPtr()->addFactory(mProgramWriterFactories[i]);
Components/RTShaderSystem/src/OgreShaderProgramManager.cpp:147:        ProgramWriterManager::getSingletonPtr()->removeFactory(mProgramWriterFactories[i]);
Components/RTShaderSystem/src/OgreShaderProgramWriterManager.cpp:60:void ProgramWriterManager::addFactory(ProgramWriterFactory* factory)
Components/RTShaderSystem/src/OgreShaderProgramWriterManager.cpp:62:    mFactories[factory->getTargetLanguage()] = factory;
Components/RTShaderSystem/src/OgreShaderProgramWriterManager.cpp:65:void ProgramWriterManager::removeFactory(ProgramWriterFactory* factory)
```

Factory Method/Pattern Usage (5/

- I dug around a few open source projects to see i
  actually used
  - `grep -irn “factory” .`
- The answer is yes!
- https://github.com/id-Software/Quake-III-Arena




```
mike:Quake-III-Arena-master$ grep -irn "factory" .
./q3radiant/PlugIn.h:54:        _QERPlugEntitiesFactory* m_pQERPlugEntitiesFactory;
./q3radiant/IPluginEntities.h:60:// _QERPlugEntitiesFactory is a set of commands Radiant uses to instanciate plugin entities
./q3radiant/IPluginEntities.h:94:static const GUID QERPlugEntitiesFactory_GUID =
./q3radiant/IPluginEntities.h:99:struct _QERPlugEntitiesFactory
./q3radiant/PlugIn.cpp:43:  m_pQERPlugEntitiesFactory = NULL;
./q3radiant/PlugIn.cpp:48:      if (m_pQERPlugEntitiesFactory)
./q3radiant/PlugIn.cpp:49:              delete m_pQERPlugEntitiesFactory;
./q3radiant/PlugIn.cpp:218:             // resquest a _QERPlugEntitiesFactory
./q3radiant/PlugIn.cpp:221:                     m_pQERPlugEntitiesFactory = new _QERPlugEntitiesFactory;
./q3radiant/PlugIn.cpp:222:                     m_pQERPlugEntitiesFactory->m_nSize = sizeof(_QERPlugEntitiesFactory);
./q3radiant/PlugIn.cpp:223:                     if (m_pfnRequestInterface( QERPlugEntitiesFactory_GUID, m_pQERPlugEntitiesFactory ))
./q3radiant/PlugIn.cpp:230:                             Sys_Printf( "WARNING: failed to request QERPlugEntitiesFactory from plugin %s\n"
etBuffer(0) );
./q3radiant/PlugIn.cpp:285:     if (m_pQERPlugEntitiesFactory)
./q3radiant/PlugIn.cpp:289:             IPluginEntity *pEnt = m_pQERPlugEntitiesFactory->m_pfnCreateEntity( e->eclass, pEp );
./q3radiant/PlugIn.cpp:295:     Sys_Printf("WARNING: unexpected m_pQERPlugEntitiesFactory is NULL in CPlugin::CreatePluginEntity\n");
./code/ui/ui_main.c:71: {"Weapons Factory Arena", "wfa" },
./code/unix/pcons-2.3.1:4849:# factory that creates packages like sig::md5::debug, etc., on the fly.
./code/unix/cons:3775:# factory that creates packages like sig::md5::debug, etc., on the fly.
./code/unix/README.Linux:65:minimum requirements, you are unlikely to be able to run a satisfactory
./lcc/x86/linux/tst/paranoia.1bk:182:The arithmetic diagnosed seems Satisfactory though flawed.
./lcc/tst/paranoia.c:1794:                      printf("Satisfactory though flawed.\n");
./lcc/tst/paranoia.c:1814:                      printf("The arithmetic diagnosed seems Satisfactory.\n");
```

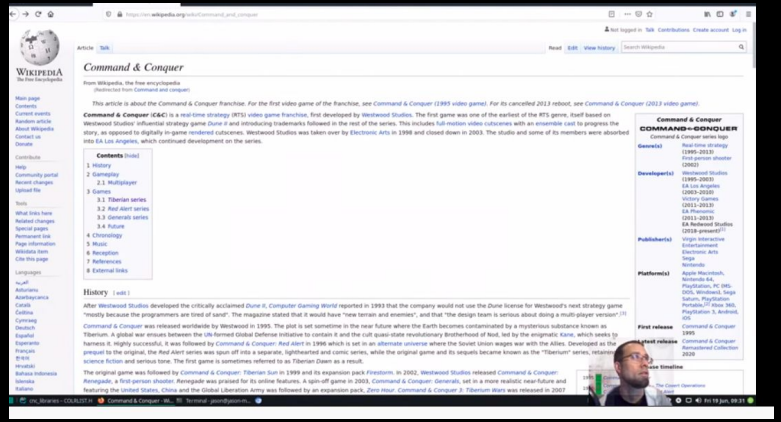
Factory Method/Pattern Usage (6/6)

- And of course--command and conquer
- https://github.com/electronicarts/CnC_Remastered_Collection
  - (Aside, that there is a type called a ‘Factory, that is literally a ‘factory’ in the game -- not to be confused with the pattern!)

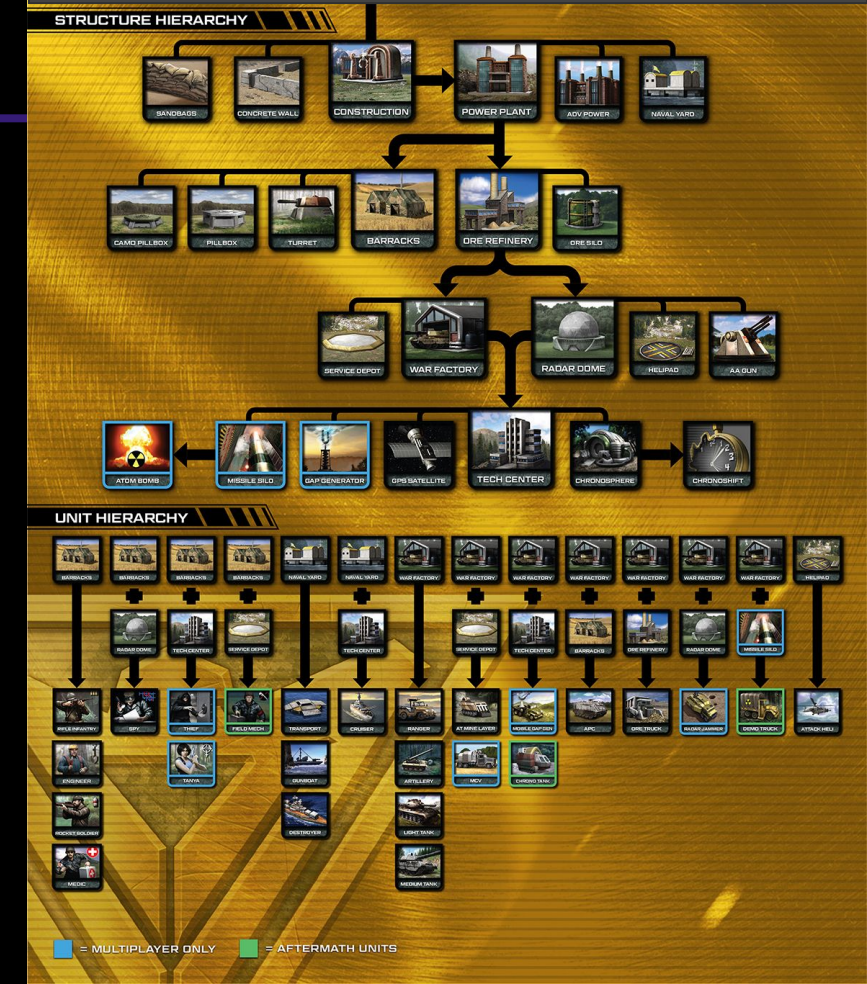

```
./REDALERT/DLLInterface.cpp:5847:                                                FactoryClass * factory =
./REDALERT/DLLInterface.cpp:5945:                                                        FactoryClass * f
./REDALERT/DLLInterface.cpp:7898:                                              FactoryClass* factory = Factories.Ptr(index);
./REDALERT/SCENARIO.CPP:834:    FactoryClass::Init();
./REDALERT/SIDEBARGlyphx.CPP:415:                                    FactoryClass * factory = Factories.Raw_Ptr(factoryid);
./REDALERT/FACTORY.H:40:class FactoryClass : private StageClass
./REDALERT/FACTORY.H:47:                FactoryClass(void);
./REDALERT/FACTORY.H:48:                FactoryClass(NoInitClass const & x) : StageClass(x) {};
./REDALERT/FACTORY.H:49:                ~FactoryClass(void);
./REDALERT/BUILDING.H:68:               CCPtr<FactoryClass> Factory;
./REDALERT/HOUSE.H:45:class FactoryClass;
./REDALERT/HOUSE.H:714:         FactoryClass * Fetch_Factory(RTTIType rtti) const;
./REDALERT/HOUSE.H:715:         void Set_Factory(RTTIType rtti, FactoryClass * factory);
./REDALERT/IOOBJ.CPP:37: *   FactoryClass::Code_Pointers -- codes class's pointers for load/save                    *
./REDALERT/IOOBJ.CPP:38: *   FactoryClass::Decode_Pointers -- decodes pointers for load/save                        *
./REDALERT/IOOBJ.CPP:317: * FactoryClass::Code_Pointers -- codes class's pointers for load/save                        *
./REDALERT/IOOBJ.CPP:336:void FactoryClass::Code_Pointers(void)
./REDALERT/IOOBJ.CPP:347: * FactoryClass::Decode_Pointers -- decodes pointers for load/save                        *
./REDALERT/IOOBJ.CPP:364:void FactoryClass::Decode_Pointers(void)
./REDALERT/HEAP.CPP:71:template class TFixedIHeapClass<FactoryClass>;
./REDALERT/CCPTR.CPP:49:template class CCPtr<FactoryClass>;
./REDALERT/BUILDING.CPP:426:            mono->Printf("%s %d%%", Factory->Get_Object()->Class_Of().IniName, (100*Factory->Completion())/FactoryClass::STEP_COUNT);
./REDALERT/BUILDING.CPP:524:            FactoryClass * factory = NULL;
./REDALERT/BUILDING.CPP:1676:   delete (FactoryClass *)Factory;
```

Actual Hierarchy of Objects

- (An aside for those that are interested)
  - I also thought Jason Turners Review of the source was interesting!
    - https://www.youtube.com/watch?v=Oee7gje-XRc



Unit and Structure Hierarchy



No Design Pattern is perfect -- recap

- Trade offs
  - Pros
    - Can hide lots of implementation details (only need to know type)
    - Can be extensible
  - Cons
    - Still need to document how to create the different types and what is available.
      - (Maybe this is in text documentation, or maybe the factory can print a listing for you)
    - Perhaps some performance issue if we have lots of inheritance
      - (Needs to be measured, potentially able to be optimized away--I have no empirical evidence for this specific talk)

‘Mike careful calling it Factory Pattern’

Factory Method Pattern (What we have largely discussed) is different and exist several various for Factory Pattern

e.g., abstract factory, extensible factory, distributed factories, etc.
(I’m providing some key words here for you to continue forward)

Conclusion

A Summary of what we have learned

Summary of what we have learned and should learn next

- We have learned about the ‘Factory Method Pattern’
  - We have thought a bit about some of the pros and cons.
- We have learned about an *extensible Factory* Pattern
- We did not talk about creating multiple factories
  - (We could have used one single Templated Factory for this)
- There are several alternations of the factory pattern as well
  - The Abstract Factory Pattern is likely the most popular (and in the Gang of Four book)
    - Multiple interfaces for each of the products that you want to build.

Going Further

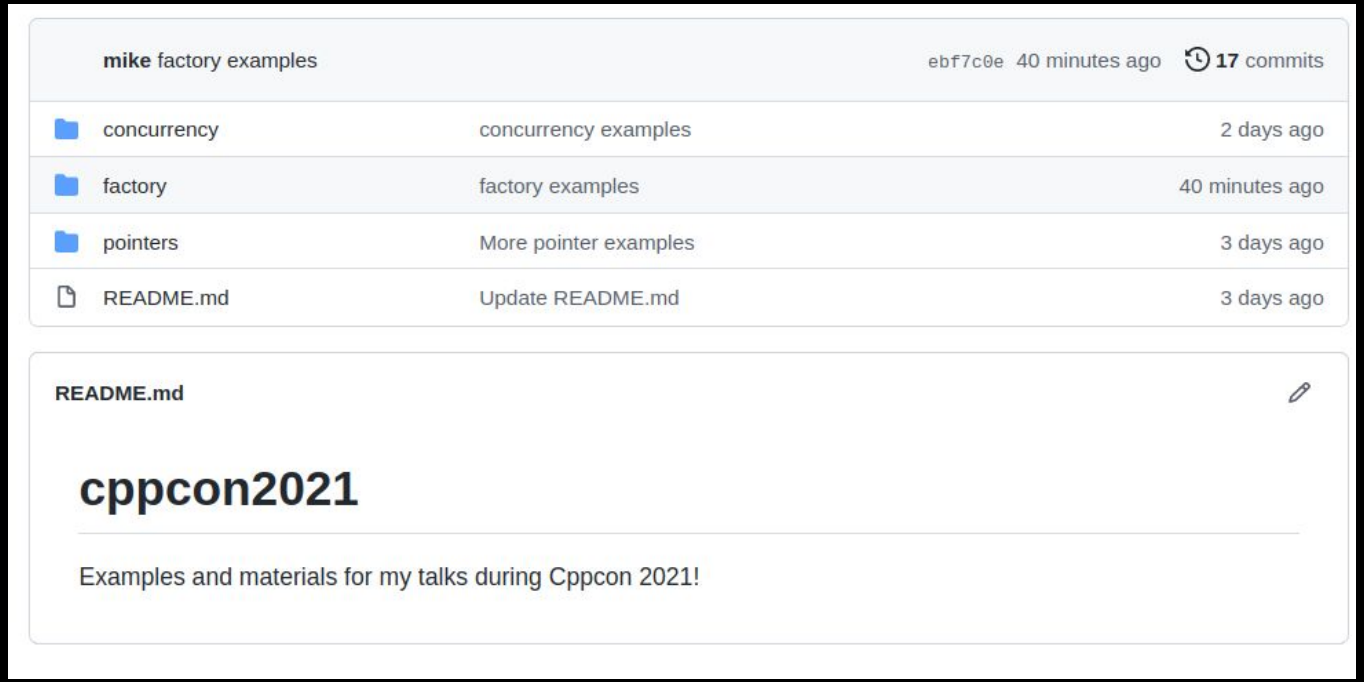
Some things that may be useful for learning more design patterns

Some References

- 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)
    - (I give an overview and 3 patterns)
      - (Some folks aren't going to like Singleton!)
  - And many more!
    - https://www.youtube.com/results?search_query=cppcon+design+patterns
- Books
  - API Design for C++
  - Game Programming Patterns
  - Modern C++ Design

Code for the talk

Available here: https://github.com/MikeShah/cppcon2021



Software Design: Factory Pattern

Mike Shah, Ph.D.
@MichaelShah | mshah.io | www.youtube.com/c/MikeShah

Thank you Cppcon attendees, reviewers, chairs!

Thank you!