

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.



The Canadian C++ Conference

July 21-24, 2024 • Toronto, Canada

A Study of Plugin Architecture for Supporting Extensible Software

-- in C++

with Mike Shah

11:00 - 12:00 EDT Mon, July 22, 2024

60 minutes with Q&A
Introductory/Intermediate Audience

Social: [@MichaelShah](#)

Web: [mshah.io](#)

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



[www.youtube.com/c/MikeShah](#)

[http://tinyurl.com/mike-talks](#)

The Canadian C++ Conference

July 21-24, 2024 • Toronto, Canada

So this will be an introductory software architecture talk

A Study of Plugin Architecture

for Supporting Extensible Software

-- in C++

with Mike Shah

11:00 - 12:00 EDT Mon, July 22, 2024

60 minutes with Q&A
Introductory/Intermediate Audience

Social: [@MichaelShah](#)

Web: [mshah.io](#)

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



[www.youtube.com/c/MikeShah](#)

<http://tinyurl.com/mike-talks>

The Canadian C++ Conference

July 21-24, 2024 • Toronto, Canada

Let's say this advice applies when you start running into issues of scale -- maybe that's 50k or 100k+ lines of code in your domain.

A Study of Plugin Architecture for Supporting Extensible Software

-- in C++

with Mike Shah

11:00 - 12:00 EDT Mon, July 22, 2024

60 minutes with Q&A
Introductory/Intermediate Audience

Social: [@MichaelShah](#)
Web: [mshah.io](#)
Courses: [courses.mshah.io](#)
 [YouTube](#)
[www.youtube.com/c/MikeShah](#)
[http://tinyurl.com/mike-talks](#)

The Canadian C++ Conference

July 21-24, 2024 • Toronto, Canada

My explicit goal today, is to introduce you to how a plugin architecture may be useful -- so thanks for having me, and let's begin!

A Study of Plugin Architecture for Supporting Extensible Software

-- in C++

with Mike Shah

11:00 - 12:00 EDT Mon, July 22, 2024

60 minutes with Q&A
Introductory/Intermediate Audience

Social: [@MichaelShah](#)

Web: [mshah.io](#)

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

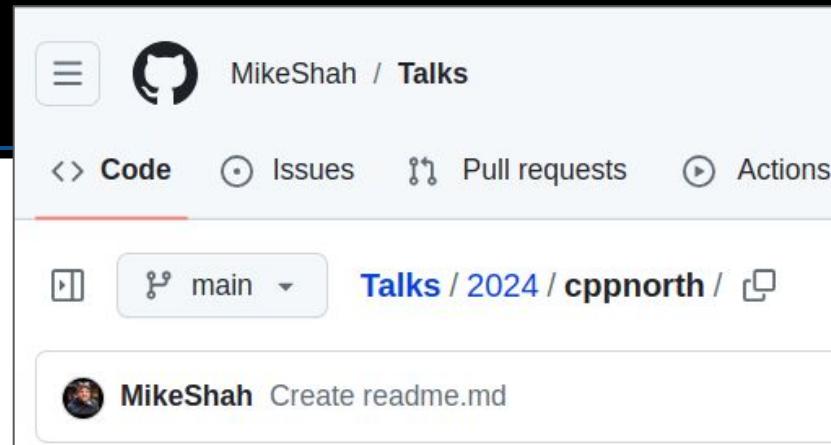


[www.youtube.com/c/MikeShah](#)

<http://tinyurl.com/mike-talks>

Code and Slides for the talk

- Code Located here:
<https://github.com/MikeShah/Talks/tree/main/2024/cppnorth>
- Slides posted after conference at:
 - www.mshah.io
- Live coding the examples from this (if any) posted at:
 - www.youtube.com/c/MikeShah



Talk Abstract: Building extensible software is a goal and often a metric of good software design. It is becoming more and more common for users to also contribute to the development of the software that they use--especially in the domains of computer graphics and gaming. Terms like 'modding' software have been around since *at least* the early 90s when the popular game Doom allowed for users to create their own content and modify the behavior of the program. Behind these programs there thus must be a mechanism for allowing users to 'hook' into the main program. In this talk, I will be showing several software developer kits including Autodesk Maya 3D (C++), Unity3D (C#), Unreal Engine (C++), and QT Modeler(C), and present a case study of how they are designed. At the end of the design discussion I will present how to get started building your own plugin system, and what considerations must be taken in mind (e.g. does the application or plugin manage resources, what should be exposed in the API, how do you embed a scripting language, and how should you distribute your plugins). Attendees will leave the presentation with practical knowledge on how to build software that can be extended by their user base.

Your Tour Guide for Today

Mike Shah

- **Current Role:** Teaching Faculty at [Yale University](#)

(Previously Teaching Faculty at Northeastern University)

- **Teach/Research:** computer systems, graphics, geometry, game engine development, and software engineering.

- **Available for:**

- **Contract work** in Gaming/Graphics Domains
 - e.g. tool building, plugins, code review
- **Technical training** (virtual or onsite) in Modern C++, D, and topics in Performance or Graphics APIs

- **Fun:**

- Guitar, running/weights, traveling, video games, and cooking are fun to talk to me about!



Web

www.mshah.io

 **YouTube**

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

Non-Academic Courses

courses.mshah.io

Conference Talks

<http://tinyurl.com/mike-talks>

Your Tour Guide for Today

Mike Shah

- **C**onference Talks
(PPTs, Slides, etc.)
We will talk about some video
games today -- stay tuned!
- **A**cademic Papers
So let's get started!
- **T**echnical training (Virtual Reality, C++, D, and topics in Performance)
- **F**un:
 - Guitar, running/weights, traveling, **video games**, and cooking are fun to talk to me about!



Web
www.mshah.io

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

Non-Academic Courses
courses.mshah.io

Conference Talks
<http://tinyurl.com/mike-talks>



The neat part of software is...





The neat part of software is...



Right in front of
us!

Software versus Hardware

- My friend and colleague Klaus reminded me of the definition of *software* in his 2020 talk from the origin of the names:
 - ‘hardware’ being *not really readily changeable, and*
 - ‘software’ being [*much more*] *easily modifiable.*

Klaus Iglberger

Breaking Dependencies: The SOLID Principles

because probably all talks at this conference are about software

▶ Software

Breaking Dependencies: The SOLID Principles - Klaus Iglberger - CppCon 2020
<https://www.youtube.com/watch?v=Ntraj80qN2k>

Software is easily modify

- So the **neat** thing for software engineers is (relative to hardware) it is:
 - Easier to change code over time (e.g. add/remove features, improve performance of an algorithm)
 - Easier to fix bugs on deployed products



certain expectation in software

Soft

=

Easy to change and extend

Breaking Dependencies: The SOLID Principles - Klaus Iglberger - CppCon 2020
<https://www.youtube.com/watch?v=Ntraj80qN2k>

5

Software is maybe too easy to modify (1/3)

- So the **neat** thing for software engineers is (relative to hardware) it is:
 - Easier to change code over time (e.g. add/remove features, improve performance of an algorithm)
 - Easier to fix bugs on deployed products
- **Careful** however -- relative to hardware:
 - Almost too easy to update software
 - We might actually add breaking features, bugs, and other regressions!



Klaus Iglberger
Breaking Dependencies:
The SOLID Principles

certain expectation in software

5

Breaking Dependencies: The SOLID Principles - Klaus Iglberger - CppCon 2020
<https://www.youtube.com/watch?v=Ntraj80qN2k>

Software is maybe too easy

- So the **neat** thing for software engineers is (relative to hardware) it is:
 - Easier to change code over time (e.g. add/remove features, improve performance of an algorithm)
 - Easier to fix bugs on deployed products
- **Careful** however -- relative to hardware:
 - Almost too easy to update software
 - We might actually add breaking features, bugs, and other regressions!

So what we're going to start working on today is:

1. **Develop software with a ‘solid and stable core’ (like hardware)**
2. **Maintain the flexibility and ease in which we can extend software**

Software is maybe too easy

- So the **neat** thing for software engineers is (relative to hardware) it is:
 - Easier to change code over time (e.g. add/remove features, improve performance of an algorithm)
 - Easier to fix bugs on deployed products
- **Careful** however -- relative to hardware:
 - Almost too easy to update software
 - We might actually add breaking features, bugs, and other regressions!

Oh, and just to be clear --

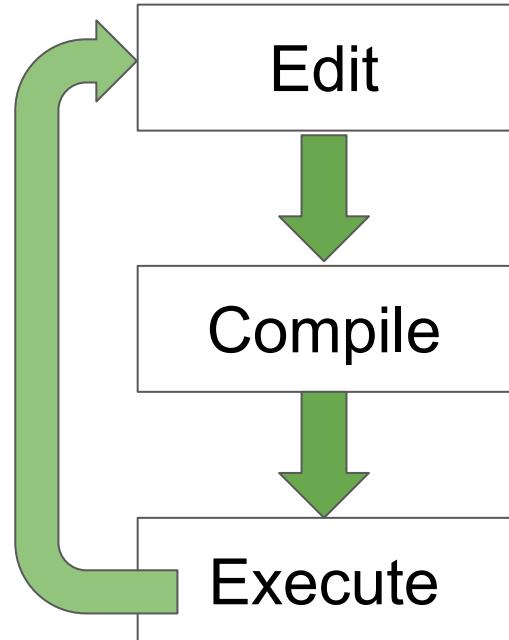
It's wonderful that we can modify software easily, but we are going to talk about software architecture here at scale with some examples later -- stay with me for a bit!

Rewinding a Bit --

A Typical Software Cycle

How do we change software? (1/2)

- A typical Software Engineering workflow is some form of:
 - Edit the code
 - Compile the code
 - Execute the Code

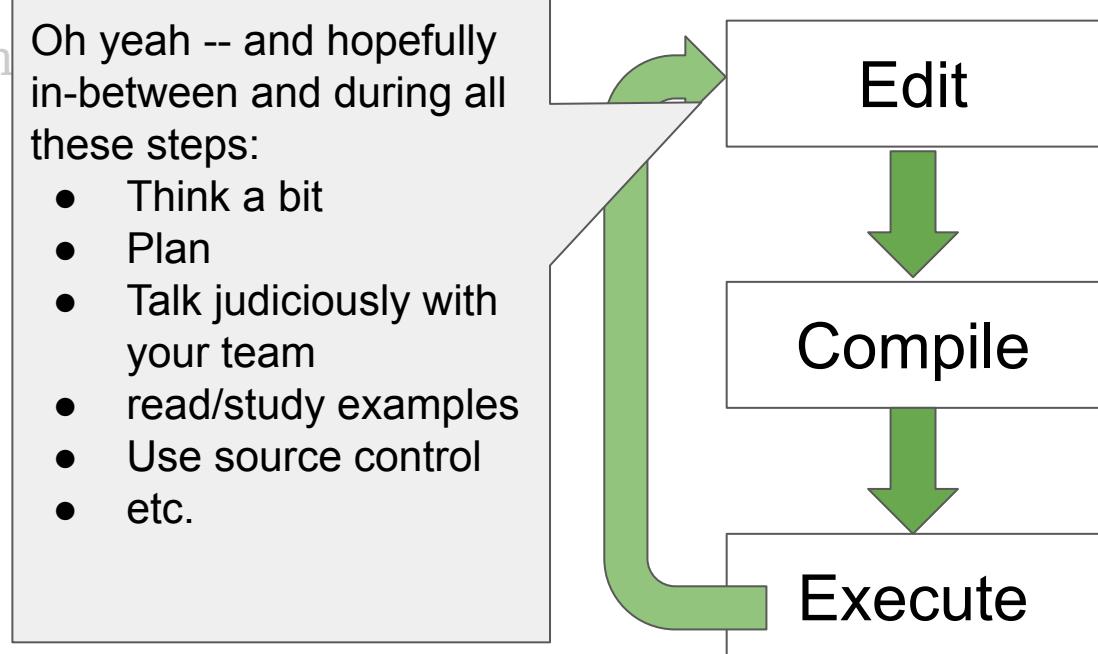


How do we change software? (2/2)

- A typical Software Engineering process follows some form of:
 - Edit the code
 - Compile the code
 - Execute the Code

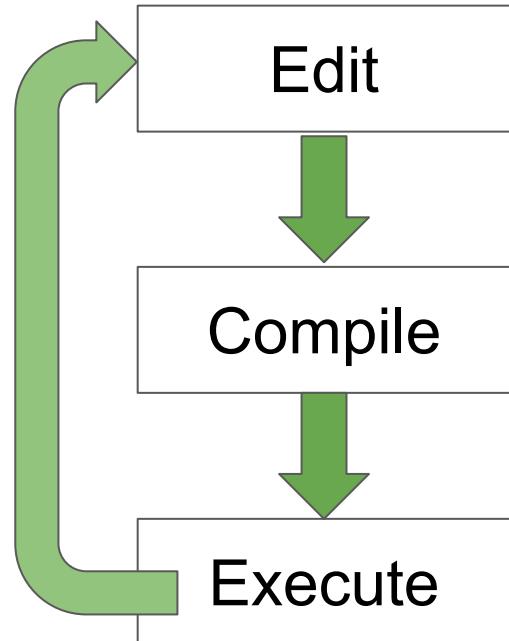
Oh yeah -- and hopefully in-between and during all these steps:

- Think a bit
- Plan
- Talk judiciously with your team
- read/study examples
- Use source control
- etc.



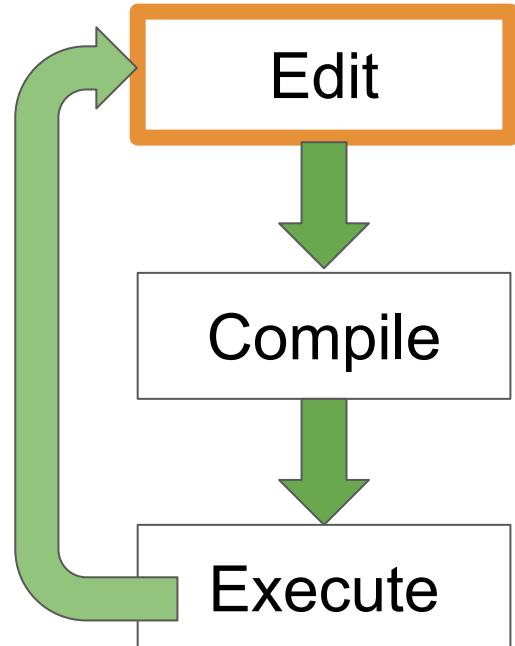
Software Challenges (especially at scale)

- So what are some of the challenges for us -- this seems relatively simple?
 - **(next slide)**



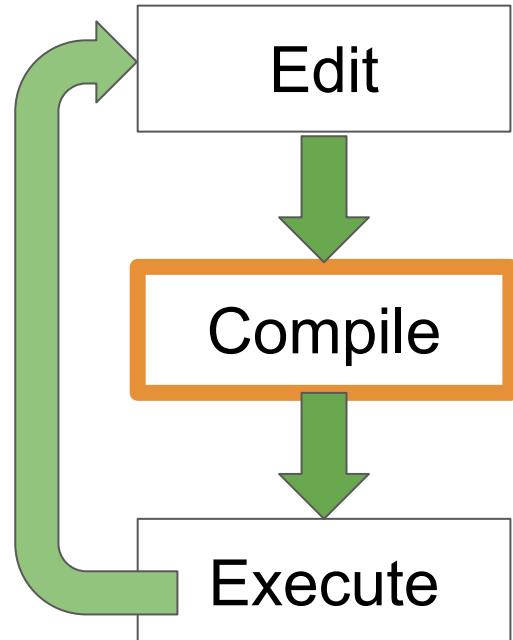
Challenge #1 - Huge Codebase

- ‘Edit’ could be a big step in a monolithic codebase -- especially if my codebase is 100k or 1+ million lines of code.
 - Lots to understand
 - (Do I understand more than my part?)
 - Perhaps other developers to coordinate with
 - Code may be ‘brittle’
 - i.e. programmers are afraid to touch certain parts



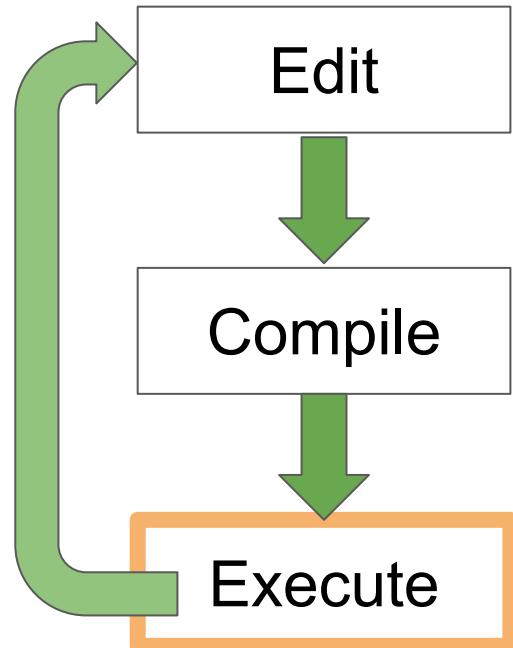
Challenge #2 - Huge Compile Times

- Perhaps that ‘compile step’ is quite large.
 - Perhaps it takes a lot of time to compile a million lines of code (lots of code, templates, compile-time execution, etc.)



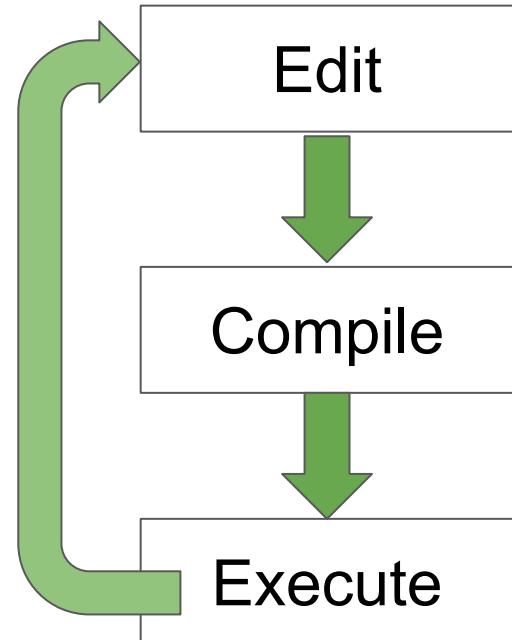
Challenge #3 - Tricky to Deploy New Code

- Maybe that ‘execute’ step also is hard to get to.
 - Execute might involve redeployment:
 - To the cloud, a piece of hardware, or a system that is already running which is less trivial.



Software Challenges Summary

1. Huge codebase that's hard to understand
2. Huge compile times
3. Deploying to an already running system is hard

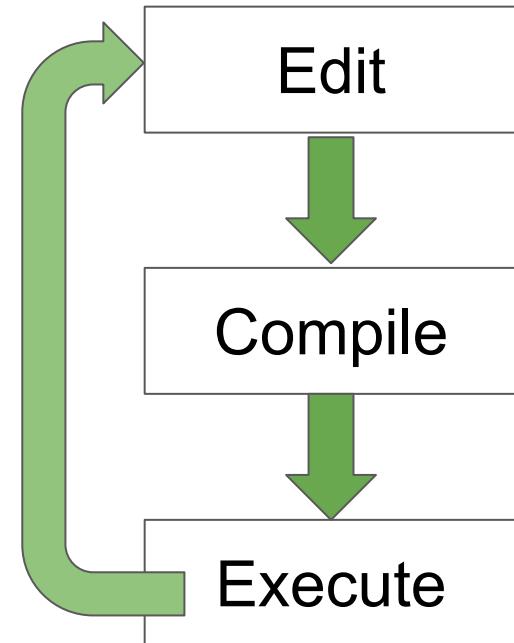


Software Challenges Summary - Reminder of our goals

1. Huge codebase that's hard to understand
2. Huge compile times
3. Deploying to an already running system is hard

Reminder of our goal(s):

1. Develop software with a ‘solid and stable core’ (like hardware)
2. Maintain flexibility and ease at which we can extend software



Let's get inspired and see what happens when we achieve these two goals

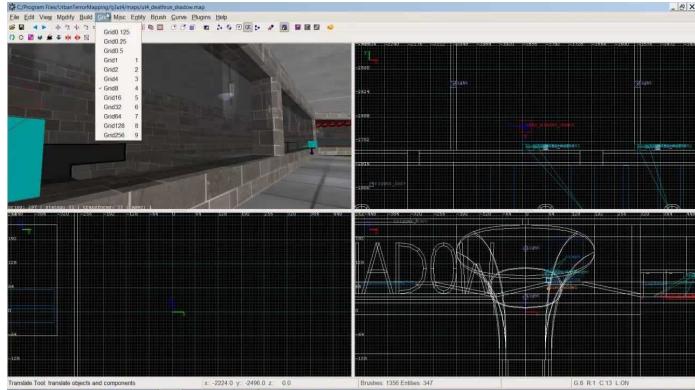


Reminder of our goal(s):

1. Develop software with a ‘solid and stable core’ (like hardware)
2. Maintain flexibility and ease at which we can extend software

Modding Video Games

(A Brief History of the ‘mod’)



Enter the ‘mod’

- Video game modding ([mods](#)) became popular in the early 90s
 - Specifically with game engines (idTech) from id Software which were either licensed or otherwise modified by users/studios.
- The idea being that you only needed to modify the assets and gameplay code
 - The main game framework (the game engine) took care of the low level details of managing resources, memory, I/O, displaying graphics, etc.
 - Mods usually loaded as a new ‘data file’ instead of the normal game package



The Early Days of id Software - GDC Europe 2016
<https://www.youtube.com/watch?v=E2M1pi8plvY>
https://en.wikipedia.org/wiki/Wolfenstein_3D

Game ‘Mods’ (A few examples 1990s - early 2000s)

- Basically all of these games came with tools (or data files) that allowed extension -- they became a sandbox for creativity!
 - Wolfenstein 3D (1992)
 - Doom (1993)
 - Quake I, II, III (1996 - 1999)
 - Source Engine (From Valve, 1998)
 - Team Fortress (1999)
 - Counterstrike (2000)
 - Neverwinter Nights Aurora Engine (2002)
 - Warcraft 3 (2002)
 - Morrowind editor (2002)



Gary’s Mod - A Physics Sandbox

https://miro.medium.com/v2/resize:fit:1400/0*fwqqFJ4wo3EieSo.jpg



Wintermaul - Warcraft 3 Tower defense

<https://www.hiveworkshop.com/data/gallery-images/112/112170-e79d22c97a1faa6b4d7b9e9ddfb3ac15.jpg>

The ‘scale’ of mods

- The Elder Scrolls V Skyrim [[wiki](#)] from Bethesda released in 2011 powered by the [Creation Engine](#)
 - By 2016 -- Skyrim had 40,000+ mods [[source](#)]
 - By 2024 -- Skyrim had still increased nearly 70,000 mods [[source](#)] after being 13 years old!
- The longevity of such projects and sheer amount of projects in the ecosystem is impressive!



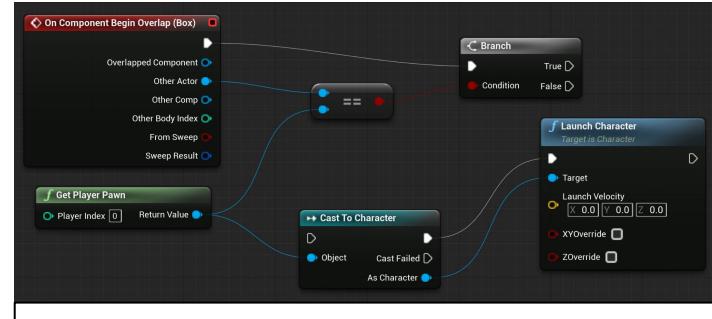
<https://www.gametrader.com/the-forgotten-city-is-the-skyrim-mod-that-became-a-crazed-time-loop-mystery-and-you-can-play-it-now/>

How a Lawyer Sacrificed his Career to
Redevelop his Skyrim Mod | The
Forgotten City Documentary

<https://www.youtube.com/watch?v=CSqHTxgcXil>

Why Allow your software to be extended by users? (1/2)

1. Software reuse for you and your users
 - a. Note: Many cases in the game world where these ‘users’ become developers later on!
2. Can build systems that programmers and non-programmers can use
 - a. e.g. Unreal Blueprints (Visual Scripting) require less programming to build games

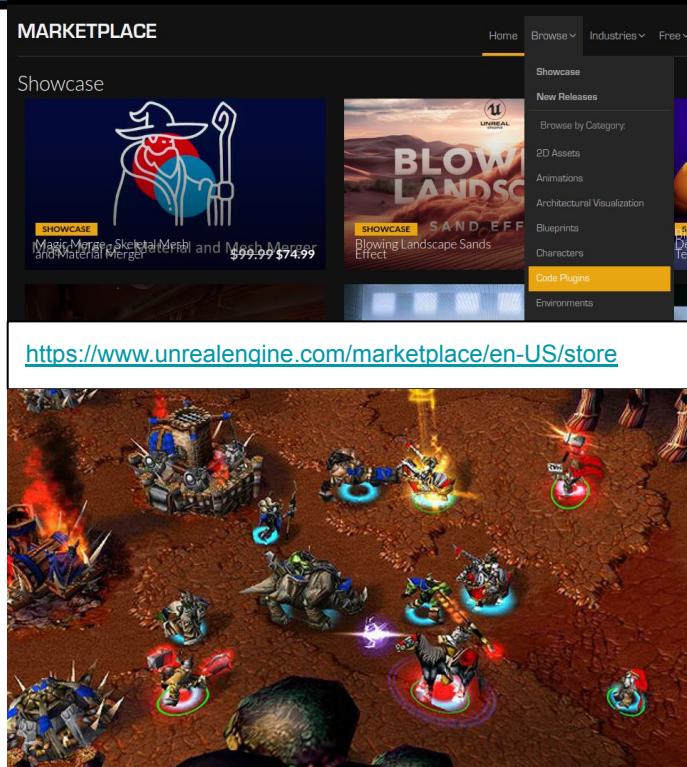


Unreal Engines ‘Visual Scripting’

https://docs.unrealengine.com/4.27/Images/ProgrammingAndScripting/Blueprints/QuickStart/BPQS_6_Step4.png

Why Allow your software to be extended by users? (2/2)

1. Economic benefits
 - a. license fees for your technology
 - b. marketplace for various extensions
2. Good karma
 - a. Some mods become more popular than the actual games!
 - b. And this can result again in continued sales and community building for the original product
 - i. Should be a win-win for devs and users



The screenshot shows the Unreal Engine Marketplace interface. On the left, there's a sidebar with categories like Home, Browse, Industries, Showcase, New Releases, and various asset types. The main area displays a 'Showcase' section with two items: 'Magic Masher - Skeletal Mesh and Material Masher' and 'Blowing Landscape Sands Effect'. Below this is a preview of a Warcraft 3 mod titled 'Defense of the Ancients (2003)', which features a chaotic battle scene with multiple units and effects. A URL at the bottom of the screenshot provides the link to the mod page.

<https://www.unrealengine.com/marketplace/en-US/store>

Defense of the Ancients (2003) is a Warcraft 3 mod that became the popular 'DOTA'

https://media.moddb.com/images/downloads/1/166/165889/w3_featovr.jpg

Why **Not** To Allow your platform to be extended by users?

1. Could expose internals that you may not want
 - a. Whether directly exposed in your API or indirectly by accident
2. What happens if you allow talking across programming language boundaries?
 - a. i.e. interop with a language with different safety guarantees.
3. Once you have a critical mass of users, you have to be careful of how frequently you change APIs and break code that is depended on
 - a. i.e. You now need to support such features
 - b. i.e. [Hyrum's law](#) (*With a sufficient number of users of an API, it does not matter what you promise in the contract: all observable behaviors of your system will be depended on by somebody.*)

Back to our Main Goals

- To some degree, all of the games we looked at (from a software perspective) must have met (to some degree) our 2 goals.
 - (As always -- in any software talk -- there are trade-offs!)



Reminder of our goal(s):

- 1. Develop software with a ‘solid and stable core’ (like hardware)**
- 2. Maintain flexibility and ease at which we can extend software**

Let's talk about the first goal

3 Ideas of how to **Modify** Software



Reminder of our goal(s):

- 1. Develop software with a ‘solid and stable core’ (like hardware)**
- Maintain flexibility and ease at which we can extend software

3 Ideas of how to **Modify** Software (1/2)

1. Just edit the source code (Editing Code)
2. Edit source with discipline (Editing Code)
3. Live Patching (Editing Code)

3 Ideas of how to **Modify** Software (2/2)

1. Just edit the source code (Editing Code)
2. Edit source with discipline (Editing Code)
3. Live Patching (Editing Code)



Think to yourself if these strategies meet the below goal:

- 1. Develop software with a ‘solid and stable core’ (like hardware)**
2. Maintain flexibility and ease at which we can extend software

Idea #1 - Just edit the source code (Editing Code)

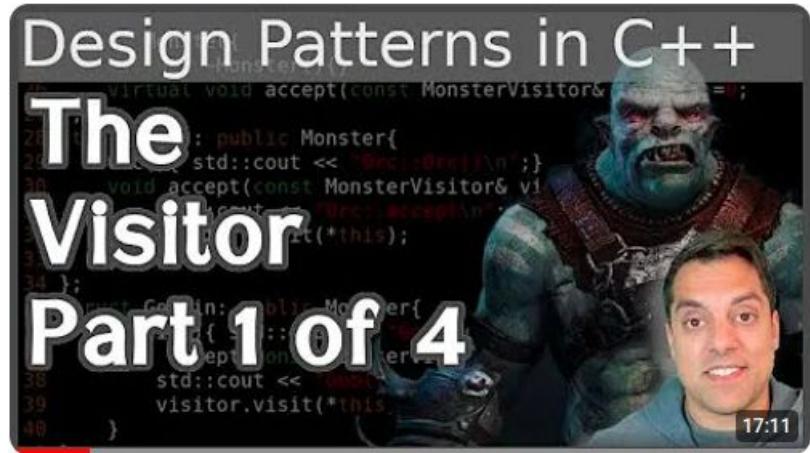
- So idea 1 is to just literally modify the code
 - You always have this option
 - But do we think this is a good way to get 70,000+ ‘mods’ distributed for your product?



<https://compote.slate.com/images/8a5bf959-9321-4a83-b960-dad1120144ac.jpeg?width=1200&rect=1560x1040&offset=0x0>

Idea #2 - Edit source with discipline (Editing Code)

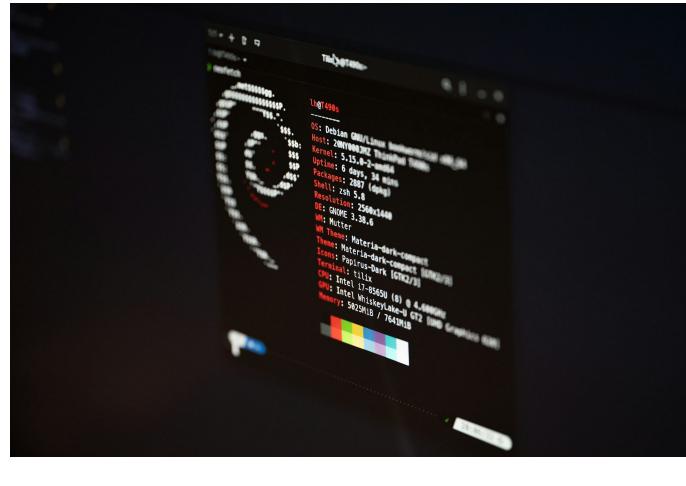
- With careful use, we can use programming language features like inheritance to extend or otherwise override behaviors.
 - This requires your programmers and users to understand subsystems they work on.
 - And perhaps further discipline to follow the perhaps the Open-Closed Principle (from SOLID)
 - Some patterns like the ‘visitor pattern’ may be useful,



C++ Visitor Design Pattern - Part 1 of 4 - Programming Paradigms
<https://www.youtube.com/watch?v=9cNnh2BmTOU&t=2s&pp=ygUZbWlrZSBzaGFolHZpc2l0b3IgcGF0dGVybq%3D%3D>

Idea #3 - Live Patching (Editing Code)

- Another strategy is to actually modify a binary by live/hot/monkey patching in new code.
 - This is a bit more risky, but certainly doable.
 - BUT -- we really need to understand how the system operates.
 - Some domains (e.g. an operating system) may require this strategy
- Frameworks like Intel's Pin may be of interest
 - <https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html>
- However -- this probably is not the appropriate mechanism for an easily extensible system -- we're changing the core at the binary level!



3 Ideas of how to Modify Software

1. Just edit the source code (Editing Code)
2. Edit source with discipline (Editing Code)
3. Live Patching (Editing Code)

Think to yourself if these strategies meet the below goals:

1. **Develop software with a ‘solid and stable core’ (like hardware)**
2. Maintain flexibility and ease at which we can extend software

So to answer the question:

- You can *maybe* have a strong and stable codebase -- but you need discipline and good abstractions if you are going to freely modify it.
 - programming tools constraints/contracts/concepts/interfaces may help
- So let’s see if we can focus on designing ‘extensible systems’ and controlling a bit where new code enters.

Let's talk about the second goal

Some Ideas of how to **Extend** Software



Reminder of our goal(s):

1. Develop software with a 'solid and stable core' (like hardware)
2. **Maintain flexibility and ease at which we can extend software**

4 Ideas of how to **Extend** Software (1/2)

1. Configuration Files (Data-Driven)
2. Command Driven Language (Data-Driven)
3. Interpreter/Script Language (Data-Driven)
4. Plugin System (shared libraries) (Data-Driven)

4 Ideas of how to Extend Software (2/2)

1. Configuration Files (Data-Driven)
2. Command Driven Language (Data-Driven)
3. Interpreter/Script Language (Data-Driven)
4. Plugin System (shared libraries) (Data-Driven)



Think to yourself if these strategies meet the below goal:

1. Develop software with a ‘solid and stable core’ (like hardware)
2. **Maintain flexibility and ease at which we can extend software**

Idea #1 - Configuration Files (Data-Driven) (1/2)

- We can make our software data-driven using ‘configuration files’
 - In our game modding examples -- the ‘art assets’ are an example of something data-driven
 - Other configuration like ‘settings’ and ‘rules/constraints’ could be controlled from configuration files
 - You can even setup different data types with the Type Object pattern:
 - <https://gameprogrammingpatterns.com/type-object.html>

```
5 int main(int argc, char* argv[]){  
6     // Program arguments  
7     if(argc < 2){  
8         std::cout << "Please specify a file name"  
9             << std::endl  
10            << "e.g. ./prog filename.txt"  
11                << std::endl;  
12    }  
13    return 0;  
14}  
15  
16 std::ifstream inputFile;  
17 inputFile.open(argv[1]);  
18 if(inputFile.is_open()){  
19  
20     std::string line;  
21     std::getline(inputFile,line);  
22     std::cout << line;  
23     std::getline(inputFile,line);  
24     std::cout << line;  
25     std::getline(inputFile,line);  
26     std::cout << line;  
27 }  
28 }
```

<https://www.youtube.com/watch?v=CAqX8YT4IHI>
Read, write, and parse files(fstream, string, & stringstream)

Idea #1 - Configuration Files (Data-Driven) (2/2)

- Note: for any data-driven strategy consider:
 - Is the file loaded at build system (compile-time) or run-time?
 - If it's 'run-time' you have to decide:
 - Does loading configuration files happen when the program starts?
 - Or could loading happen at an arbitrary point while the software is running?
 - (i.e. user clicks a button, or system listens for changes)

Idea #2 - Command Driven Language (Data-Driven)

- Your ‘configuration’ files could be a ‘command-driven language’
 - Can effectively build a small Domain Specific Language (DSL) or Virtual Machine
 - e.g.
 - Events that take place when a new game level is setup (see top-right)
 - Image a painting application where a script has several ‘effects’ or ‘transformations’ that happen successively

```
ShowBitmap "Gfx/LevelLoading.bmp"
LoadLevel "Levels/Level4.lev"
FadeBGMusicOut
PlaySound "Sounds/LevelLoaded.wav"
LoadBGMusic "Music/Level4.mp3"
FadeBGMusicIn
```

A series of commands (no loops or conditions) is executed
Game Scripting Mastery - Chapter 2

Idea #3 - Interpreter/Script Language (Data-Driven) (1/3)

- And in fact, if you do not want to create your own little language or interpreter, then you could just embed a scripting language
 - Python (boost python)
 - pybind11 [[example videos](#)]
 - lua
 - ruby
 - C#, etc.
 - QuakeC [[wiki](#)]
- You'll need to decide how much of your API to otherwise expose

```
local EventFrame = CreateFrame("frame", "EventFrame")
EventFrame:RegisterEvent("PLAYER_ENTERING_WORLD")

EventFrame:SetScript("OnEvent", function(self, event, ...)
    if(event == "PLAYER_ENTERING_WORLD") then
        ChatFrame1:Hide()
    end
end)
```

Lua is a popular scripting language for games in general -- pictured is a lua script for World of Warcraft
<https://wowpedia.fandom.com/wiki/Lua>

Idea #3 - Interpreter/Script Language (Data-Driven) (2/3)

- Here's an example with pybind11, which wraps a 'component' class for use in a shared library
- In the **next slide** -- I'll show using the shared library within Python

```
8 class Component{
9     public:
10         // Constructor
11         Component() {}
12         // Setter function
13         void SetData(T _data){
14             data = _data;
15         }
16         // Return the type
17         T GetData(){
18             return data;
19         }
20     private:
21         // Templated field
22         T data;
23 };
24
25 PYBIND11_MODULE(library, m){
26     m.doc() = "Templated class library with some exposed types";
27
28     // Notice the templated type here
29     py::class_<Component<int>>(m, "ComponentInt")
30         .def(py::init())
31         .def("SetData", &Component<int>::SetData)
32         .def("GetData", &Component<int>::GetData);
33         // Each function above uses the template argument
34         // Notice the templated type here
```

Idea #3 - Interpreter/Script Language (Data-Driven) (3/3)

- On the bottom I have the compilation command for building a library with pybind11
- On the right using that library within Python
 - Note: We can even embed Python within C++, then import our library there.
 - Now users are simply writing Python scripts using our exposed API to change behaviors :)

```
>>> import library
>>> help(library)

>>> c = library.ComponentString()
>>> c.SetData("mike")
>>> c.GetData()
'mike'
```

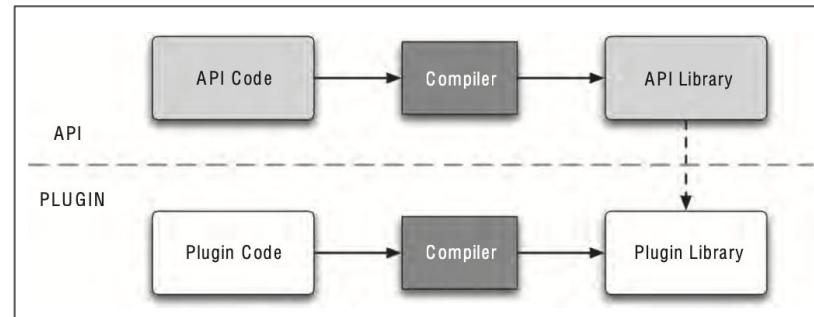
```
1 # build the library that will be used in pybind
2 clang++ -shared -fPIC -std=c++17 -I./pybind11/include/ `python3.6 -m pybind11 --includes` *.cpp -o library.so `python3.6-config --ld
flags`
```

Full example video of building a Python plugin

https://www.youtube.com/watch?v=XSKGSnMmTNw&list=PLvv0ScY6vfd90tV5g_wzbkfCZ8iR9qSMK&index=6

Idea #4 - Plugin System (shared libraries) (Data-Driven)

- We can actually compile our code as a library -- now we get the benefit of speed of compiled C++!
 - Then we load that library at compile or run-time
 - Note: In this talk I'll focus on run-time loading
 - This allows us to work in our native language (or any language with a C Foreign Function Interface)
- Allowing code extension by plugins does require us to now create a plugin interface!



The idea here is users write separate 'Plugins' from whatever API we expose. A plugin manager will handle loading our code into our systems.

API Design for C++ Ch. 12

4 Ideas of how to Extend Software

1. Configuration Files (Data-Driven)
2. Command Driven Language (Data-Driven)
3. Interpreter/Script Language (Data-Driven)
4. Plugin System (shared libraries) (Data-Driven)

Think to yourself if these strategies meet the below goal:

1. Develop software with a ‘solid and stable core’ (like hardware)
2. **Maintain flexibility and ease at which we can extend software**

So to answer the question:

- All of these options are ‘data-driven’ and extend our system in some way.
 - They naturally require us to have some ‘core’ to build an API around
- The plugin system may be the easiest solution for allowing anyone to hook into our system once setup.

One Solution -- Plugin Architecture

-- allow for extension while addressing code scale issues

Challenges

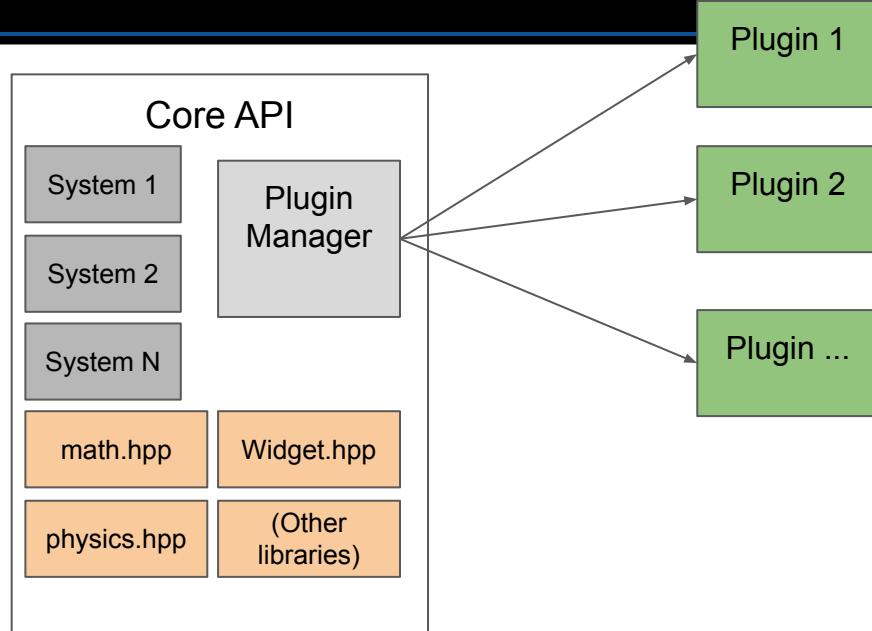
1. Huge codebase that's hard to understand
2. Huge compile times
3. Deploying to an already running system is hard

Objectives/Goals

1. Develop software with a 'solid and stable core' (like hardware)
2. Maintain flexibility and ease at which we can extend software

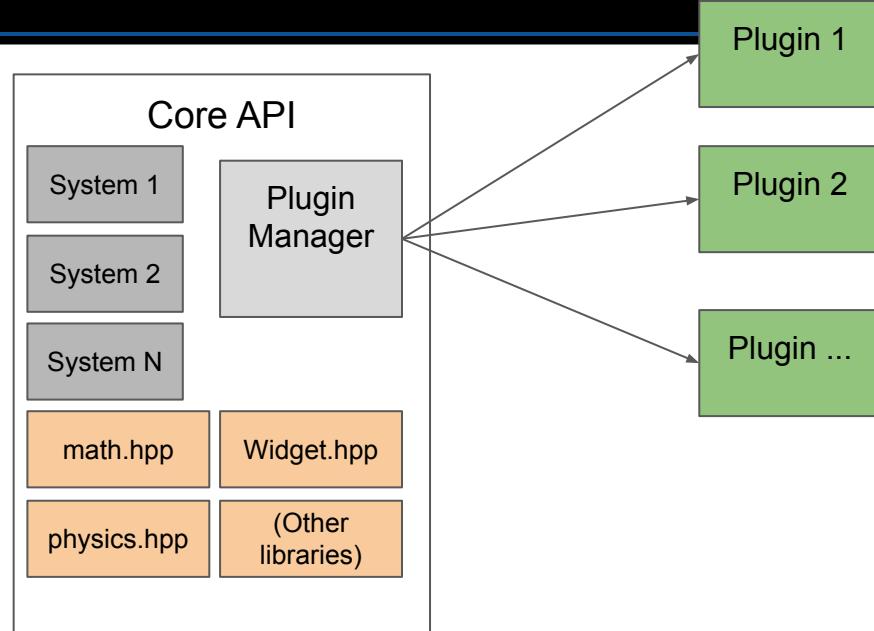
Plugin Architecture (1/2)

- The software architecture for our system will look something like this
- Our ‘Core API’ is relatively firm
 - Users are not really changing source in the Core API to change behaviors
 - Instead they’re building features through the exposed core API as plugins



Plugin Architecture (2/2)

- We almost want to think of this ‘Core API’ as a black box -- like a ‘hardened software’ that is changed by trusted engineers
 - Thus we extend our system through ‘plugins’
- Note:
 - We do have two competing stakeholders (plugin developers and core engineers) --
 - ‘contracts’ in the API or otherwise a stable API/Versioning system become important!



Plugin Manager mechanism

- On Linux the mechanism is the **dlopen** function for opening a library.
 - **dlsym** specifically reads in the function names of a loaded library
- Note: Your plugin manager needs some way to read the functions exported (e.g. reading a file of exported symbol names or otherwise ‘registering them’)

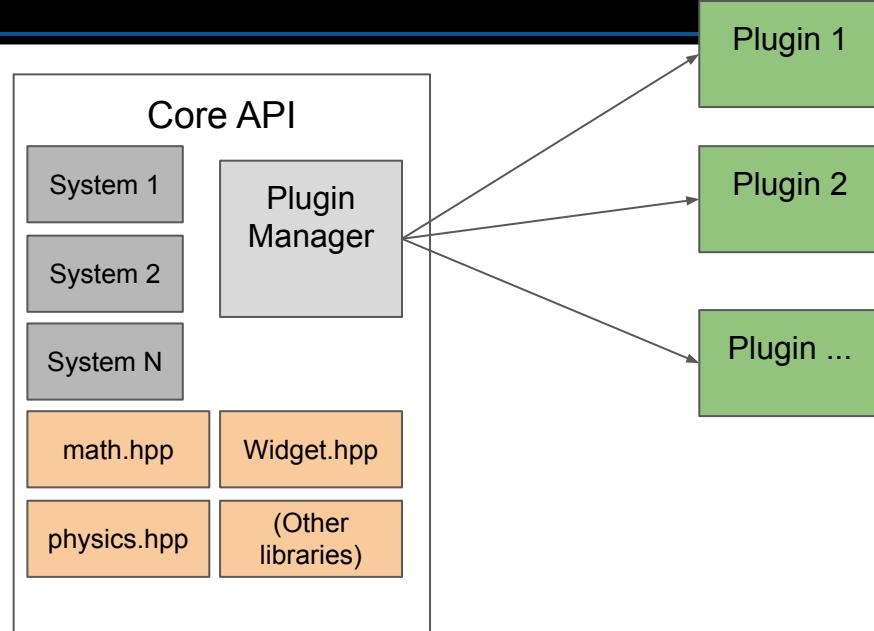
```
1 // Compile with: g++ -o prog library.cpp -ldl
2 //
3 // What is -ldl, well that is linking in the dynamic loading library of course
4 !
5 #include <iostream>
6 #include <cstdlib>
7 #include <dlfcn.h>
8
9 typedef double(*double_double)(double);
10
11 int main(){
12     // Here we get a pointer into our dynamically linked library
13     void* handle;
14     // Here we open up a specific library, in this case
15     // the math library that already exists.
16     handle = dlopen("/usr/lib/x86_64-linux-gnu/libm.so.6", RTLD_LAZY);
17     // Output an error if we are not able to open our shared library
18     // In our program, we will just exit
19     if(!handle){
20         std::cout << "Decide how you want to handle errors!\n";
21         exit(1);
22     }
23     // This is what is known as a function pointer
24     // It is a pointer to a named function, with the argument following
25     // In this case, our function is called 'cosine', and it takes in a double.
26     double (*cosine)(double);
27     // We find our symbol in the library symbol table
28     cosine = (double_double)dlsym(handle, "cos");
29     // Actually use our function
30     std::cout << "loaded cosine, and calculated cosine(45.0)=" << (*cosine)(45)
31     << std::endl;
32     // Finally we have to close access to our dynamic library
33     // (Similar to how we always close files after opening them)
34     dlclose(handle);
35     return 0;
36 }
```

```
nike@system76-pc:~/Talks/2024/cppnorth$ g++ -o prog library.cpp -ldl
nike@system76-pc:~/Talks/2024/cppnorth$ ./prog
loaded cosine, and calculated cosine(45.0)=0.525322
```

Note: On windows there are equivalent functions to dl* functions like: LoadLibraryA and GetProcAddress

Plugin Architecture Review

- So the plugin architecture seems to hit our goals!
- We can focus on building, testing, and compiling ‘smaller’ plugin components:
 - We can localize complexity within our plugins
 - Avoid recompiling the entire codebase (only the plugins if that’s where changes are made)
 - And even load (or reload) plugins to a running system
- (Or -- if you’re on the ‘core side’ -- you can just focus on building the infrastructure that others plug into -- either way, less code to manage)



Example Enabling of a Plugin System - Manager

- Have a single manager for managing plugins
 - This is perhaps a use case for a Singleton Design
 - Note: You may additionally consider concurrency and error logging as needed.

```
3 // In 'Core' Application or 'engine'
4 class PluginManager {
5     public:
6         static PluginManager &GetInstance();
7         bool LoadAll();
8         bool Load(const std::string &name);
9         bool UnloadAll();
10        bool Unload(const std::string &name);
11        std::vector<PluginInstance *> GetAllPlugins();
12     private:
13         PluginManager();
14         ~PluginManager();
15         PluginManager(const PluginManager &);
16         const PluginManager &operator=(const PluginManager &);
17         std::vector<PluginInstance *> mPlugins;
18 }
```

Single Entry point and manager of all plugins
<https://www.youtube.com/watch?v=eLAvry56vLU>

Example Enabling of a Plugin System - Interfaces

- Add whatever functionality that you think you need
 - e.g. ‘Load’, ‘Update’, ‘Render’
- Since we’re dynamically loading the module, we rely on run-time polymorphism for functionality
 - Thus -- the ‘virtual’ keyword

```
20 // Provided by Engine Maintainers are some 'base class'
21 // from which you can override behavior and 'hook' into systems.
22 class IPlugin{
23     public:
24         virtual ~IPlugin() {}
25         virtual void Load(const char* file) =0;
26         virtual void Update() =0;
27         virtual void Render() =0;
28 };
```

Example of a Custom Plugin

- Example implementation of a custom plugin implementing the IPlugin interface
 - Observe the ‘C’ based API that exposes out the factory functions to ‘create’ and ‘destroy’ a new instance
 - Note: Most of your ‘factory’ functions need to be ‘extern C’ to avoid mangling.
 - Windows and other platforms may need to handle dllexport

```
30 /// Your C++ Implementation
31 class MyCustomPlugin : public IPlugin {
32     public:
33         MyCustomPlugin() {}
34         override void Load(const char* file){}
35         override void Update(){}
36         override void Render(){}
37     };
38
39 /// C API for exposing some functionality
40 //
41 extern "C" IPlugin *CreateInstanceMyCustomPlugin() {
42     return new MyCustomPlugin();
43 }
44 extern "C" void DestroyMyCustomPlugin(MyCustomPlugin *p) {
45     delete p;
46 }

49 /// Interface to your 'Plugin Manager' to register and name your plugin
50 extern "C" int Plugin_Registry()
51 {
52     RegisterPlugin("my_custom_plugin", CreateInstanceMyCustomPlugin, DestroyMyCustomPlugin);
53     return 0;
54 }
55
```

Case Studies

A Brief Look at a few Real World Systems

Maya

- Maya is a 3D modeling and animation package used in games and motion pictures
- Just looking at the user interface, it's screaming at me for a plugin architecture
 - i.e. Every button can be a plugin that performs some unique action



Image from:

https://www.aptech.ae/uploads/thumbnails/course_thumbnails/course_image_default_27.jpg

Maya is now owned by AutoDesl -- learn more of its history here:

https://en.wikipedia.org/wiki/Wavefront_Technologies#Acquisitions_and_mergers

Maya - Hello World Plugin (1/3)

- We can observe this follows a very similar pattern to our plugin architecture previously explained
 - (next slide)

```
#include <stdio.h>
#include <maya/MString.h>
#include <maya/MArgList.h>
#include <maya/MFnPlugin.h>
#include <maya/MPxCommand.h>
#include <maya/MIOStream.h>

class helloWorld : public MPxCommand
{
public:
    MStatus doIt( const MArgList& args );
    static void* creator();
};

MStatus helloWorld::doIt( const MArgList& args ) {
    cout << "Hello World " << args.asString( 0 ).asChar() << endl;
    return MS::kSuccess;
}

void* helloWorld::creator() {
    return new helloWorld;
}

MStatus initializePlugin( MObject obj ) {
    MFnPlugin plugin( obj, "Autodesk", "1.0", "Any" );
    plugin.registerCommand( "HelloWorld", helloWorld::creator );
    return MS::kSuccess;
}

MStatus uninitializedPlugin( MObject obj ) {
    MFnPlugin plugin( obj );
    plugin.deregisterCommand( "HelloWorld" );
    return MS::kSuccess;
}
```

Maya - Hello World Plugin (2/3)

- We can observe this follows a very similar pattern to our plugin architecture previously explained
 - Here is our new command ‘helloWorld’ that inherits from a base class.

```
#include <stdio.h>
#include <maya/MString.h>
#include <maya/MArgList.h>
#include <maya/MFnPlugin.h>
#include <maya/MPxCommand.h>
#include <maya/MIOStream.h>

class helloWorld : public MPxCommand
{
public:
    MStatus doIt( const MArgList& args );
    static void* creator();
};

MStatus helloWorld::doIt( const MArgList& args ) {
    cout << "Hello World " << args.asString( 0 ).asChar() << endl;
    return MS::kSuccess;
}

void* helloWorld::creator() {
    return new helloWorld;
}

MStatus initializePlugin( MObject obj ) {
    MFnPlugin plugin( obj, "Autodesk", "1.0", "Any" );
    plugin.registerCommand( "HelloWorld", helloWorld::creator );
    return MS::kSuccess;
}

MStatus uninitializedPlugin( MObject obj ) {
    MFnPlugin plugin( obj );
    plugin.deregisterCommand( "HelloWorld" );
    return MS::kSuccess;
}
```

Maya - Hello World Plugin (3/3)

- We can observe this follows a very similar pattern to our plugin architecture previously explained
 - Below observe are our set of creator function and set of functions for initializing the plugin in our plugin manager

```
#include <stdio.h>
#include <maya/MString.h>
#include <maya/MArgList.h>
#include <maya/MFnPlugin.h>
#include <maya/MPxCommand.h>
#include <maya/MIOStream.h>

class helloWorld : public MPxCommand
{
public:
    MStatus doIt( const MArgList& args );
    static void* creator();
};

MStatus helloWorld::doIt( const MArgList& args ) {
    cout << "Hello World " << args.asString( 0 ).asChar() << endl;
    return MS::kSuccess;
}

void* helloWorld::creator() {
    return new helloWorld;
}

MStatus initializePlugin( MObject obj ) {
    MFnPlugin plugin( obj, "Autodesk", "1.0", "Any" );
    plugin.registerCommand( "HelloWorld", helloWorld::creator );
    return MS::kSuccess;
}

MStatus uninitializedPlugin( MObject obj ) {
    MFnPlugin plugin( obj );
    plugin.deregisterCommand( "HelloWorld" );
    return MS::kSuccess;
}
```

Maya - Errors

- We haven't talked too much about errors in plugins -- but observe in Maya there is an `MStatus` returned from actions
 - We need some way to query in both our Core, and our plugin if the event succeeds, so this is one strategy

```
#include <stdio.h>
#include <maya/MString.h>
#include <maya/MArgList.h>
#include <maya/MFnPlugin.h>
#include <maya/MPxCommand.h>
#include <maya/MIOStream.h>

class helloWorld : public MPxCommand
{
public:
    MStatus doit( const MArgList& args );
    static void* creator();
};

MStatus helloWorld::doIt( const MArgList& args ) {
    cout << "Hello World " << args.asString( 0 ).asChar() << endl;
    return MS::kSuccess;
}

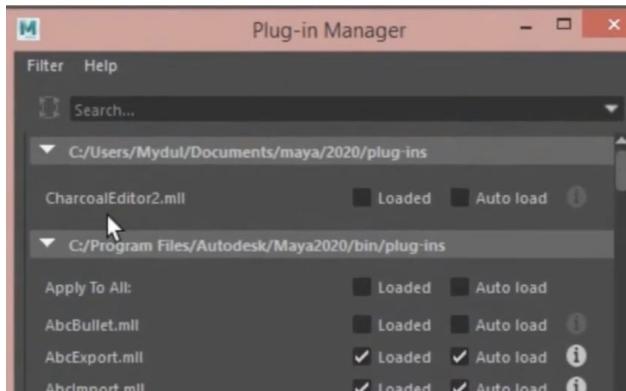
void* helloWorld::creator() {
    return new helloWorld;
}

MStatus initializePlugin( MObject obj ) {
    MFnPlugin plugin( obj, "Autodesk", "1.0", "Any" );
    plugin.registerCommand( "HelloWorld", helloWorld::creator );
    return MS::kSuccess;
}

MStatus uninitializedPlugin( MObject obj ) {
    MFnPlugin plugin( obj );
    plugin.deregisterCommand( "HelloWorld" );
    return MS::kSuccess;
}
```

Maya - Hello World Plugin

- From my own experience I can tell you in Maya this works well -- Maya also provides many plugins in this manner [\[list\]](#)
 - Effectively creating new tools off the shelf is relatively easy
- Plugins can be reloaded through a simple dialog once created [\[2025 docs on plugin manager\]](#)



List of plug-ins included with Maya

AbcBullet.mll

Lets you cache the results of a Bullet simulation, including the collision volumes. The results are saved in an animation-only Alembic file. See [Alembic Caching](#).

AbcExport.mll

Lets you export all or selected objects and animations to Alembic files in the Alembic folder of the current Maya project. See [Alembic Caching](#).

AbcImport.mll

Lets you load data from an Alembic cache file into a Maya scene.

animImportExport.mll

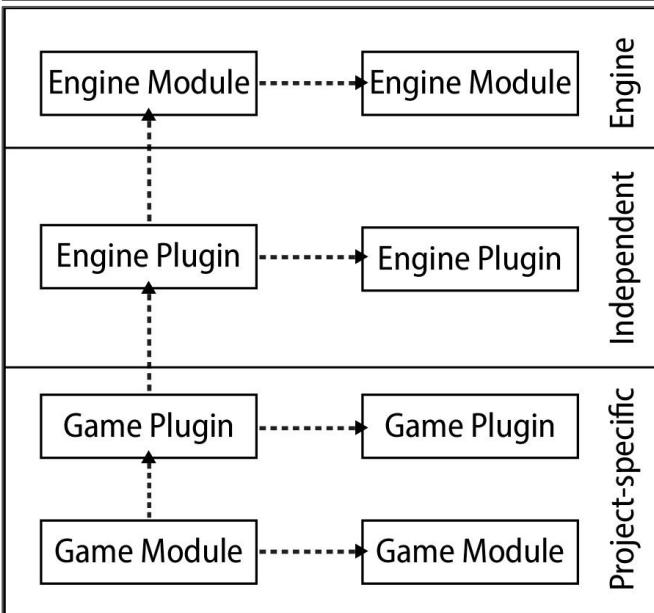
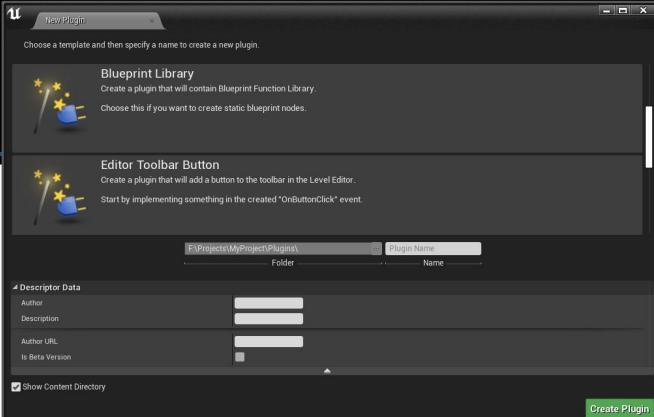
Lets you import or export anim curves to the API clipboard. See [Maya documentation archive](#) for more information.

ArubaTessellator.mll

Lets you tessellate NURBS in Viewport 2.0 while providing geometry. See [Working in Viewport 2.0](#).

Moving to Unreal Engine

- A few different types of plugins (for C++ and blueprint) depending on which ‘core system’ you are modifying
- Unreal engine guides you through what type of plugin you’d like to make
 - <https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/Plugins/>
 - <https://dev.epicgames.com/community/learning/tutorials/qz93/unreal-engine-building-plugins>



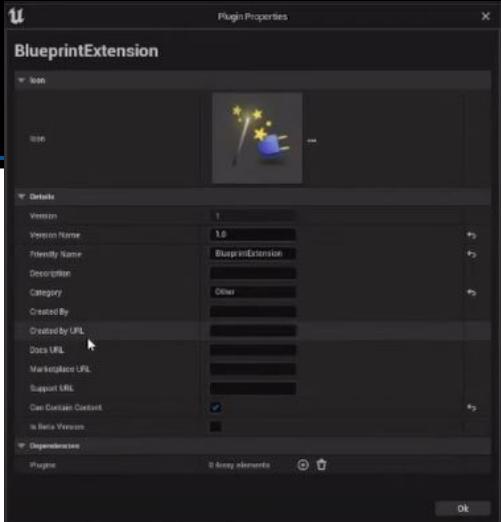
Unreal Engine

- Hmm -- I'm seeing a pattern here
 - Some functions for 'startup/shutdown'
 - Some functions for something that adheres to the plugin interface
 - (i.e. PluginButtonClicked in this example)

```
1 // Copyright Epic Games, Inc. All Rights Reserved.
2
3 #pragma once
4
5 #include "CoreMinimal.h"
6 [ #include "Modules/ModuleManager.h"
7
8 class FToolBarBuilder;
9 class FMenuBuilder;
10
11 class FNightButtonModule : public IModuleInterface
12 {
13 public:
14
15     /** IModuleInterface implementation */
16     virtual void StartupModule() override;
17     virtual void ShutdownModule() override;
18
19     /** This function will be bound to Command. */
20     void PluginButtonClicked();
21
22 private:
23
24     void RegisterMenus();
25
26
27 private:
28     TSharedPtr<class FUICommandList> PluginCommands;
29 };
30
```

Unreal Engine

- Typically additional metadata attached with each plugin for users to know about plugin.



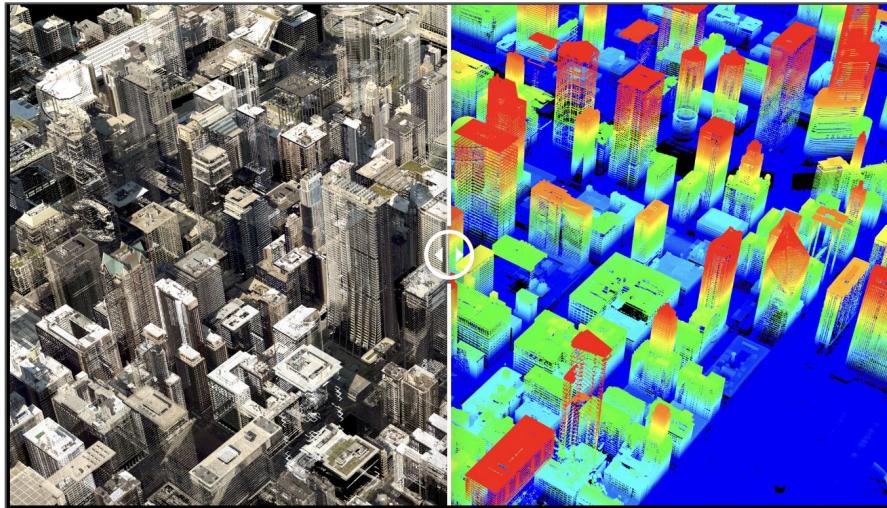
Descriptor File Example

This example plugin descriptor is from the Engine's [UObjectPlugin](#).

```
{  
    "FileVersion" : 3,  
    "Version" : 1,  
    "VersionName" : "1.0",  
    "FriendlyName" : "UObject Example Plugin",  
    "Description" : "An example of a plugin which declaratively defines its components.",  
    "Category" : "Examples",  
    "CreatedBy" : "Epic Games, Inc.",  
    "CreatedByURL" : "http://epicgames.com",  
    "DocsURL" : "",  
    "MarketplaceURL" : "",  
    "SupportURL" : "",  
    "EnabledByDefault" : true,  
    "CanContainContent" : false,  
    "IsBetaVersion" : false,  
    "Installed" : false,  
    "Modules" :  
}
```

QT Modeler

- A terrain modeler with a ‘C’ based API
 - Effectively the plugins were distributed as .h files with a list of .c functions and structs that were publicly available
 - Developer releases new binary exposing the ‘plugins’ available.



<https://appliedimagery.com/features/>

Wrap Up and More Resources

Localize Complexity

- Plugin-based architectures can help ‘localize complexity’
 - Note: The phrase ‘localize complexity’ was drilled into my head after reading ‘The Rules of Programming’ Book by Chris Zimmerman
 - I recommend this book

O'REILLY®

The Rules of
Programming

HOW TO WRITE BETTER CODE



Summary

- Today we talked about why a plugin architecture might solve software engineering problems at scale
 - There are benefits otherwise to software being open to modification to the wider world (mods to games or tools as an example)
- I hope you can think about otherwise breaking your software into chunks
 - The first part should really be ‘core systems’ -- hardened, with specific contracts for functionality like hardware
 - The second part ‘open for extension’ to help manage complexity and enable developers

More Thoughts (if time left after Q&A) (1/2)

- Q: What if my (or someone else's) plugin leaks memory?
 - A: Tools like valgrind on linux (and possibly leaks on Mac), and other memory sanitizers may likely help here
- Q: What if my plugin crashes?
 - A: It's probably ideal 'not' to crash the rest of the system -- you may consider a '[service locator](#)' pattern for your plugin system if applicable to have 'nullptr' versions available.
 - If you otherwise have a crash in your plugin that brings the whole system down, consider the cause and if that can be mitigated (e.g. resources only handed out by core system and plugin interface can only request them).
- Q: What if I compiled my code with some 'new' or 'old' version of a compiler, or something that's not ABI compatible?
 - A: Hmm, probably just need to update your compiler and try to build with the same compiler.

More Thoughts (if time left after Q&A) (2/2)

- Q: What if I want a ‘C’ based interface from my Object-Oriented C++
 - A: That’s probably a good idea.
 - Try to avoid ‘C++’ specific things that are not in C (e.g. exceptions, use const char* instead of std::string, etc.)
- Q: Should I use a plugin system or embed a scripting language?
 - A: You can actually use both if you need
 - Just make sure to load your plugins first before scripts are run, then your scripts could call into your plugins as well.
- Q: I don’t like dynamically loaded libraries because of run-time overhead or load-time overhead
 - A: It’s possible you lose some performance from not having whole program optimizations -- measure first.
 - If you are using plugins for ‘critical core systems’ (and own the source) consider making that functionality part of the core.
 - A: As far as loading



The Canadian C++ Conference

July 21-24, 2024 • Toronto, Canada

A Study of Plugin Architecture for Supporting Extensible Software

Thank you
C++ North!

-- in C++

with Mike Shah

Social: [@MichaelShah](#)

Web: [mshah.io](#)

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



[www.youtube.com/c/MikeShah](#)

[http://tinyurl.com/mike-talks](#)

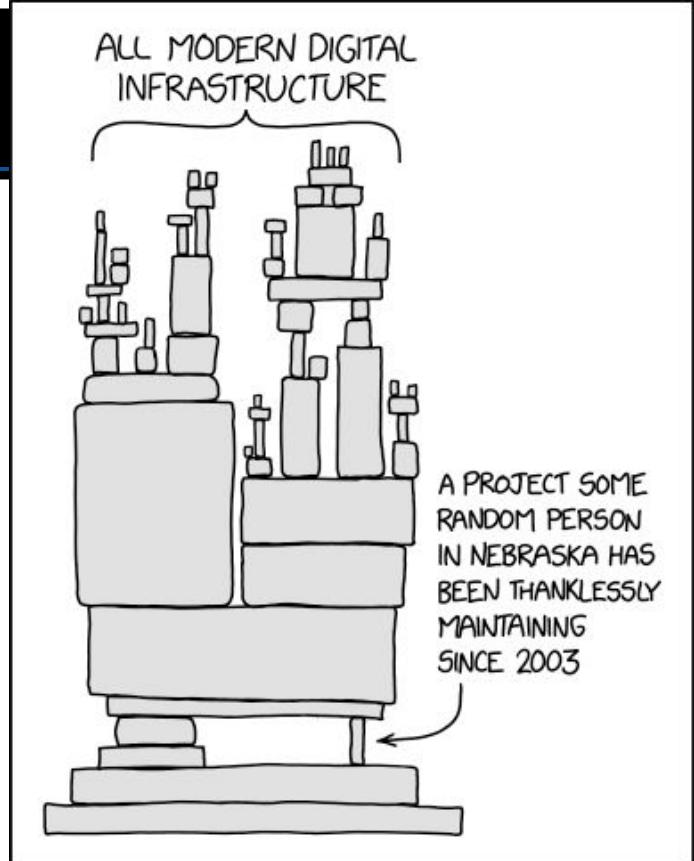
13:00 - 14:00 EDT Mon, July 22, 2024

60 minutes with Q&A
Introductory/Intermediate Audience

Extra

Audience Question (1/2)

- What do folks think of this?



Hover over title: *Someday ImageMagick will finally break for good and we'll have a long period of scrambling as we try to reassemble civilization from the rubble.*

<https://imgs.xkcd.com/comics/dependency.png>