Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #9

Interfacing

Interfacing

- Abstract interface
- Interfacing with
  - Library architecture
  - Plugin
  - Patch

Abstract interface

- Primary objective of designing classes
  - represent some concept, *e.g.* a player
  - while hiding irrelevant implementations from the users
- Ideally, only the interface describing public members for a class should be exposed
  - and more ideally only function members (get/set methods to access data members)

Abstract interface

- Unfortunately, C++ exposes a lot more than that in the header file
  - Pre-processor directives
  - Includes necessary for the file to compile correctly
  - Declarations of all private and protected members
  - Inline functions and template implementations
- Extra information are not necessarily problematic as the use of non public members is shielded by the compiler

Abstract interface

- But sometimes we need a better decoupling between a class implementation and the program
  - to change class implementations without modifying other code parts, only re-compile/re-link necessary
    - *e.g.* re-implement AI strategy using different algorithms
  - to change class implementation at run-time
    - *e.g.* select a different rendering system at run-time
  - to add new implementations after the program has been released
    - *e.g.* add new levels or game entities

Abstract interface

- Decoupling in general is a good idea
  - Cleaner, easier-to-maintain code
  - Better structural overview of the different parts of the code
  - Several programmers can work separately on related pieces of code
- A very important tool for decoupling is the use of abstract interfaces
  - to separate the declaration and the implementation of a C++ class

Abstract interface

- An abstract interface is a class that only has pure virtual function members
  - no implementation (except empty destructor)
  - no data members

```
class IAbstractInterfacePlayer {
   public:
        virtual ~IAbstractInterfacePlayer() {};
        virtual void moveTo(const float, const float) = 0;
        virtual int shootAt(Player) = 0;
        virtual bool isAlive() const = 0;
};
```

Abstract interface

- There is no need for a body file (.cpp) as there is no implementation, only a header (.h) file exists
- By convention the class name is prefixed with the letter I to indicate that it is an abstract interface
- An implementation based on this interface inherits from it and provides the implementations for all the virtual functions

Abstract interface

- Example: implementation of the abstract interface IAbstractInterfacePlayer

```
#include “IAbstractInterfacePlayer.h”                              Player.h

class Player : public IAbstractInterfacePlayer {
   public:
        virtual void moveTo(const float, const float);
        virtual int shootAt(Player);
        virtual bool isAlive() const;
};
```

  – function members are not pure virtual anymore

Abstract interface

- Example: implementation of the abstract interface IAbstractInterfacePlayer

```
#include "Player.h"                                          Player.cpp

void Player::moveTo(const float x, const float y) {
   // ...
}

int Player::shootAt(Player p) {
   // ...
}

bool Player::isAlive() const {
   // ...
}
```

Abstract interface

- Example: implementation of the abstract interface IAbstractInterfacePlayer
  - we can now create an IAbstractInterfacePlayer pointer instantiated with a derived Player object

```
// ...                                                   main.cpp
IAbstractInterfacePlayer * player = new Player();
// ...
player->moveTo(2.0,4.6);
int amnoleft = player->shootAt(player2);
if (player->isAlive()) {
   // ...
}
// ...
```

Abstract interface

- We can use the abstract interface as a barrier
  - Example with graphics renderer classes

```
class IGraphicsRenderer {                               IGraphicsRenderer.h
  public:
      virtual ~IGraphicsRenderer() {};
      virtual void initialize() = 0;
      virtual void setWorld(const Matrix& m) = 0;
      virtual void renderMesh(const Mesh& m) = 0;
      // ...
};
```

Abstract interface

- Different implementations can be provided
  - *e.g.* Direct3D and OpenGL

```
#include "IGraphicsRenderer.h"                GraphicsRendererOGL.h
#include <gl.h>

class GraphicsRendererOGL: public IGraphicsRenderer {
  public:
        virtual void initialize();
        virtual void setWorld(const Matrix& m);
        virtual void renderMesh(const Mesh& m);
        // ...
};
```

Abstract interface

- Different implementations can be provided
  - *e.g.* Direct3D and OpenGL

```
#include "IGraphicsRenderer.h"                  GraphicsRendererD3D.h
#include <d3d.h>

class GraphicsRendererD3D: public IGraphicsRenderer {
  public:
        virtual void initialize();
        virtual void setWorld(const Matrix& m);
        virtual void renderMesh(const Mesh& m);
        // ...
};
```

Abstract interface

- The abstract renderer interface
  - allows us to change the renderer at run-time
  - hides the specific renderer to the rest of the program

```
IGraphicsRenderer * pRenderer = new GraphicsRendererD3D();
// ...

// methods of IGraphicsRenderer can be used whatever the
// specific implementation used to create pRenderer
pRenderer->initialize();
pRenderer->setWorld(coordinateSystemMatrix);
pRenderer->renderMesh(playerMesh);
```

Abstract interface

- Abstract interfaces can be very useful in combination with the factory pattern

```
GraphicsRendererFactory rendererfactory;
IGraphicsRenderer * pRenderer;

// This creates an OpenGL renderer
pRenderer = rendererfactory.createRenderer("OpenGL");

// This creates a Direct3D renderer
pRenderer = rendererfactory.createRenderer("Direct3D");
```

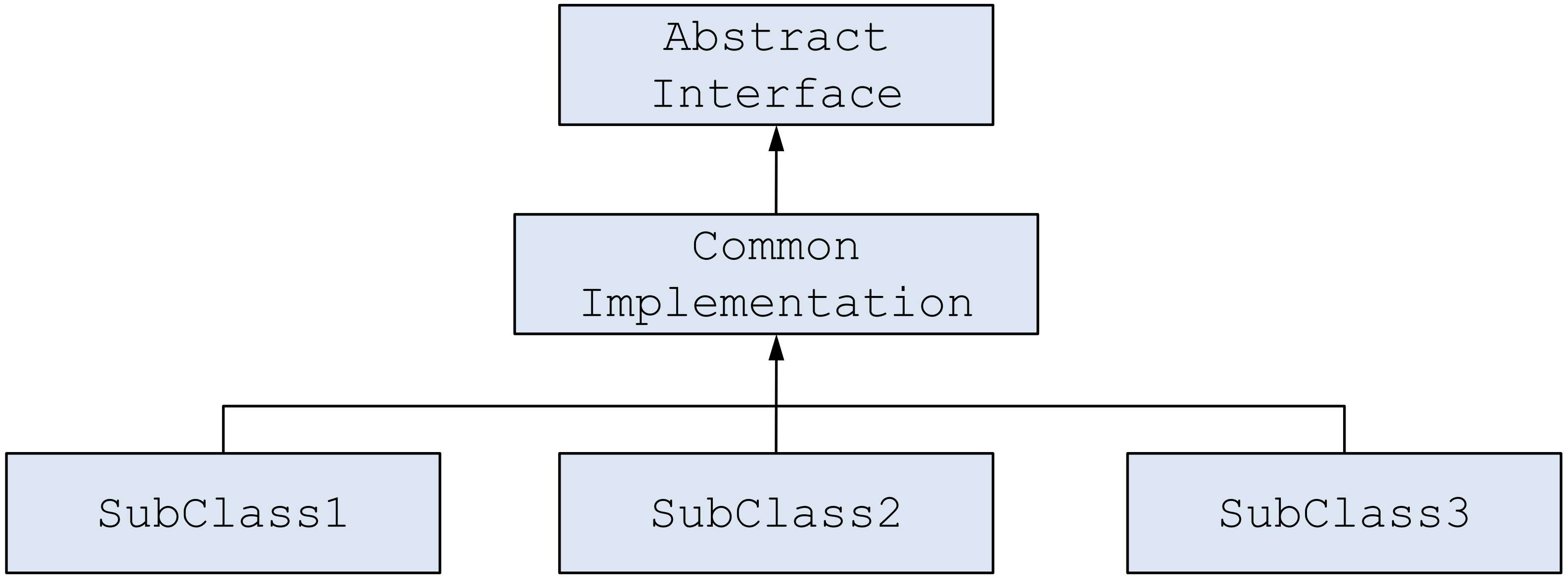
Abstract interface

- Can an abstract interface provide some partial implementation?
  – acceptable, however the more functions we add, the less “abstract” the interface becomes

```
class IGraphicsRenderer {                                   IGraphicsRenderer.h
   public:
         virtual ~IGraphicsRenderer() {};
         virtual void setWorld(const Matrix& m) = 0;
         virtual void renderMesh(const Mesh& m) = 0;
         void renderAllMeshes() {
                  // for each mesh call renderMesh
         }
         // ...
};
```

Abstract interface

- Can an abstract interface provide some partial implementation?
  - acceptable, however the more functions we add, the less "abstract" the interface becomes
  - better solution:



Abstract interface

```
class IGraphicsRenderer {
   public:
        virtual ~IGraphicsRenderer() {};
        virtual void setWorld(const Matrix& m) = 0;
        virtual void renderMesh(const Mesh& m) = 0;
        virtual void renderAllMeshes() = 0;
};
```

```
class CommonGraphicsRenderer : public IGraphicsRenderer {
   public:
        // ...
        void renderAllMeshes() {
                // for each mesh call renderMesh
        }
};
```

```
class GraphicsRendererOGL: public CommonGraphicsRenderer
```

```
class GraphicsRendererD3D: public CommonGraphicsRenderer
```

Abstract interface

- Abstract interfaces are also used to design characteristic
  - *e.g.* to add rendering or serialization capabilities

```
class IDrawable {
   public:
         virtual ~IDrawable() {}
         virtual bool draw() = 0;
};
```

```
class ISerializable {
   public:
         virtual ~ISerializable() {};
         virtual void read() = 0;
         virtual bool write() = 0;
};
```

Abstract interface

- Abstract interfaces are also used to design characteristic
  - *e.g.* to add rendering or serialization capabilities

```
class Player :
         public IAbstractInterfacePlayer ,
         public IDrawable ,
         public ISerializable {
 public:
   // member functions from IAbstractInterfacePlayer
   // member functions from IDrawable
   // member functions from ISerializable
   // ...
};
```

Abstract interface

- This involves multiple inheritance, but we avoid most of the potential problems because we inherit from abstract interfaces
  - usually no DoD or ambiguous members
- In order to use the IDrawable or ISerializable interface on an IAbstractInterfacePlayer object (or any other parent of Player), we can check if the object implements the interface
  - easier way is to check by type casting
  - if ok, use the IDrawable or ISerializable functions

Abstract interface

- This can be written in a QueryInterface function in the Player class

```
void* Player::QueryInterface (InterfaceID i) {
   if (i == IDRAWABLE) {
        IDrawable* pDraw = static_cast<IDrawable*>(this);
        return (void*)(pDraw);
   }
   if (i == ISERIALIZABLE) {
        ISerializable* pSeri = static_cast<ISerializable*>(this);
        return (void*)(pSeri);
   }
   return NULL;
}
```

Abstract interface

- Using the interfaces

```
void renderAndSave() {
   for ( /* each object (including Players) in the world */ ) {
        void* pIntR = object.QueryInterface(IDRAWABLE);
        if (pIntR != NULL) {
               IDrawable* pDraw = (IDrawable*) pIntR;
               pDraw->draw();
        }
        void* pIntS = object.QueryInterface(ISERIALIZABLE);
        if (pIntS != NULL) {
               ISerializable* pSeri = (ISerializable*) pIntS;
               pSeri->write();
        }
   }
}
```

Abstract interface

- Every class that inherits from an abstract interface needs to implement this QueryInterface() function
- A list of unique identifiers for each interface is given
  - here the identifiers are IDRAWABLE and ISERIALIZABLE
  - use an enumeration type

Interfacing

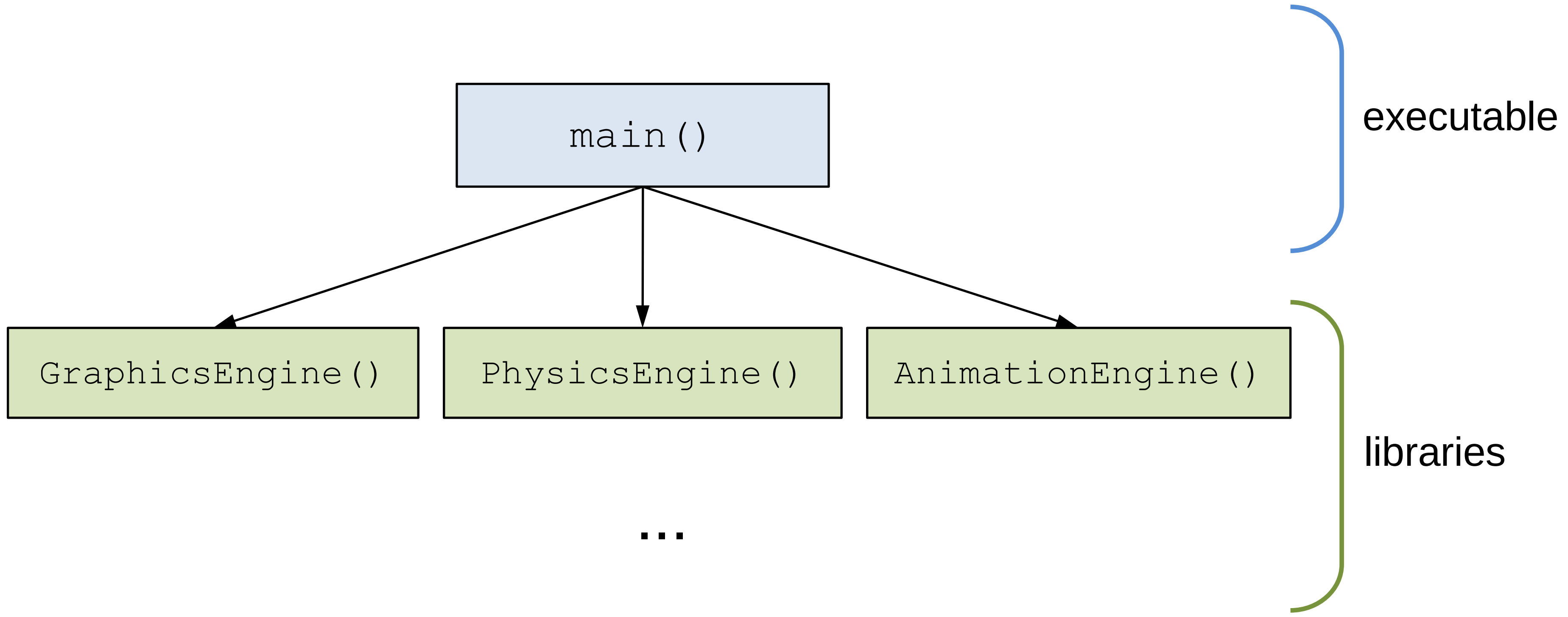
- Extending the game
  - to add data files (levels, characters, items ...)
    - ➢load them at run-time along with the others (through a resource manager)
  - to partially update code possibly with new data
    - ➢library architecture (or through library loader)
  - to release new functionalities as components, possibly with new data
    - ➢plugin (no updated executable)
  - to update executable possibly with new data
    - ➢patch (updated executable)

Library architecture

- Releasing new code without updating executable can be done using dynamic libraries
  - Linking with the code is done at run-time
  - We can tell the compiler that a project is building a library and not an executable file
  - We need to decide which functions to export (abstract interface)

Library architecture

- Main file of game engine almost empty
  - – calls to game loop components



Library architecture

- Library compilation is platform-dependent
  - dll (dynamic) and lib (static) in Windows
  - here we want dynamic library (not append to the code)
  - be aware that linking with libraries
    - needs debug dll to debug
    - decreases run-time performances
    - decreases readability as ‘separated projects’

Library architecture

- In Windows, exported functions are declared as follows (usually abstract interface member functions)

```
__declspec(ddlexport) returntype functionName (parameters);
```

  - use of namespace is strongly recommended
- Implementations of the exported functions as usual in the body file
- Header files are provided along with the dll
  - for the external programs to compile
  - if header changes, executable needs a rebuild

Plugin

- Using a plugin architecture in combination with dynamic libraries gives us a way to extend the application in a convenient manner
  - to avoid rebuild of executable when adding functionalities
  - header file ‘never’ changed as only derived from a predefined abstract interface
  - every plugin has the same exported function(s)

Plugin

- The abstract interface contains the functions that the program uses to manipulate the plugin
  - example: plugin to export Player data

```
class IPluginPlayerExport {
   public:
         virtual ~IPluginPlayerExport(){};
         virtual bool export(Player *) = 0;
}
```

Plugin

- Initialization and shutdown of plugins are done in constructor and destructor
- Each plugin can implement the *export* function differently
  - *e.g.* same data exported in different formats
- A ‘version’ function can also be useful to keep track of the plugin version
  - as plugin and main program are independent

Plugin

- Definition of Player exporter plugins

```
class PluginPlayerExportXML : public IPluginPlayerExport {
   public:
         PluginPlayerExportXML (std::string& filename);
         ~PluginPlayerExportXML (){};
         bool export(Player *);
   // ...
}
```

```
class PluginPlayerExportDefault : public IPluginPlayerExport {
   public:
         PluginPlayerExportDefault ();
         ~PluginPlayerExportDefault (){};
         bool export(Player *);
   // ...
}
```

Plugin

- A game engine often needs more than one type of plugin
  - exporter, importer, viewer, extensions *etc.*
- Types of plugin are organized through inheritance
  - using an abstract interface containing the common functions
    - such as initialization and shutdown of the plugin, get name and versioning functions

Plugin

- An abstract interface for different types of plugin

```
class IPlugin {
   public:
        virtual ~IPlugin (){};
        virtual const std::string& getPluginName() const = 0;
        virtual const VersionInfo& getVersion() const = 0;
        virtual bool initialize() = 0;
        virtual void shutdown() = 0;
}
```

Plugin

- Creation of different interfaces of plugin
  - Player exporter plugin

```
class IPluginPlayerExporter : public IPlugin {
  public:
        virtual ~IPluginPlayerExporter(){};
        virtual const std::string& getPluginName() const = 0;
        virtual const VersionInfo& getVersion() const = 0;
        virtual bool initialize() = 0;
        virtual void shutdown() = 0;
        virtual bool export(Player *) = 0;
}
```

Universiteit Utrecht

Plugin

- Creation of different interfaces of plugin
  - Player importer plugin

```
class IPluginPlayerImporter : public IPlugin {
  public:
      virtual ~IPluginPlayerImporter(){};
      virtual const std::string& getPluginName() const = 0;
      virtual const VersionInfo& getVersion() const = 0;
      virtual bool initialize() = 0;
      virtual void shutdown() = 0;
      virtual bool import(Player *) = 0;
}
```

Plugin

- Creation of different interfaces of plugin
  – Player viewer plugin

```
class IPluginPlayerViewer : public IPlugin {
   public:
        virtual ~IPluginPlayerViewer (){};
        virtual const std::string& getPluginName() const = 0;
        virtual const VersionInfo& getVersion() const = 0;
        virtual bool initialize() = 0;
        virtual void shutdown() = 0;
        virtual bool view(Player *) = 0;
}
```

Plugin

- Specialization of a plugin interface
  - XML Player exporter plugin

```
class PluginPlayerExporterXML : public IPluginPlayerExporter {
  public:
        virtual ~PluginPlayerExporterXML(){};
        virtual const std::string& getPluginName() const ;
        virtual const VersionInfo& getVersion() const ;
        virtual bool initialize();
        virtual void shutdown();
        virtual bool export(Player *);
}
```

Plugin

- Plugins are loaded at run-time
  - they are not part of the ‘main’ code that uses them
  - they are compiled separately and loaded on the fly
- Plugins are loaded through dynamic libraries
  - differ the link with the code inside the library until run-time

Plugin

- Instead of exporting the plugin class itself, we export a global factory function creating an instance of the plugin class

```
#define PLUGINEXPORT __declspec(dllexport)
// ...
extern “C” PLUGINEXPORT IPlugin* createPlugin(PluginManager& mgr);
```

  – every plugin DLL needs to provide an implementation of *createPlugin*

```
PLUGINEXPORT IPlugin* createPlugin(PluginManager& mgr) {
   return new PluginPlayerExporterXML();
}
```

Plugin

- The plugin manager deals with the incoming plugins at run-time
  - by scanning a folder for existing DLLs, detecting the ones with a plugin interface (*createPlugin* function)
  - by reading a file (*e.g.* xml or cfg) specifying which plugins to load
  - by an explicit (GUI) plugin loading procedure
- Uses the LoadLibrary / FreeLibrary / GetProcAddress API calls from Windows

Plugin

- Plugins are loaded by

```
#include <windows.h>
// ...
HMODULE handler = LoadLibrary(pluginFileName);
```

  – returns NULL if load failed, handler to library otherwise

Plugin

- The exported function is accessible via

```
#define PLUGINIMPORT __declspec(dllimport)

extern “C” PLUGINIMPORT IPlugin* createPlugin(PluginManager& mgr);

typedef IPlugin* (*PCREATEFUNC)(PluginManager&);

PCREATEFUNC pfunc =
   (PCREATEFUNC)::GetProcAddress(handler,“createPlugin”);
```

  - returns NULL if function not found in library
  - N.B. we can use the plain name “createPlugin” because exported as *extern “C”*

Plugin

- As *createPlugin* cannot have both signature
  - PLUGINIMPORT __declspec(dllimport)
    - in manager
  - PLUGINEXPORT __declspec(dllexport)
    - in plugin
- Common definition in manager and selection at compile time through pre-processor directives

Plugin

```
#ifdef PLUGINEXPORT                                        PluginManager.h
   #define PLUGINUSE __declspec(dllexport)
#else
   #define PLUGINUSE __declspec(dllimport)
#endif

extern "C" PLUGINUSE IPlugin* createPlugin(PluginManager& mgr);
```

```
#define PLUGINEXPORT 1                             PluginPlayerExporterXML.h
#include "PluginManager.h"

extern "C" PLUGINUSE IPlugin* createPlugin(PluginManager& mgr) {
   return new PluginPlayerExporterXML();
}
```

Plugin

- The plugin manager finally creates the plugin by calling the *createPlugin* function

```
IPlugin* pPlugin = pfunc(*this);
```

  - as every plugin inherits from IPlugin, the manager does not know the exact type of *pPlugin*
  - **this* is the reference to the manager (optionally used in the plugin)
  - the manager usually stores the plugin pointers, handlers and names

Plugin

- At shutdown, the manager
  - delete the object *pPlugin* (if owner)
  - unload the DLL by calling *FreeLibrary(handler)*
  - removes the plugin from the lists

Plugin

- We can load/shutdown different plugins without having to change any original code
  - creation of types of plugin using inheritance (importers, exporters, viewers *etc.*)
  - use templates to generate interfaces
- The plugin manager gives us direct access to the loaded plugins
  - *e.g.* setup of different environments with different loader contexts (set of loaded plugins)

Plugin

- Communication with plugins is not always convenient as we have to pass through the plugin manager to (un)load them
- Due to dependencies between plugins
  - Order in which they are loaded can produce a crash
  - Functionalities can mismatched as plugins are provided separately
    - need to add additional checking for conflicts / versioning

Patch

- When the main code needs to be released along with new data, the executable has itself to be either
  – replaced (new complete distribution)
  – patched (modification of the exe file)
    - very difficult, requires ‘reverse engineering’
    - however possible to read exe file as hexadecimal file and manipulate data from code offset
    - but almost impossible to make modifications of code sequence

Interfacing

- Games usually contain
  - One ‘small’ main exe file with
    - splash screen
    - few options saved on HD (configuration files)
    - versioning updater and/or plugin manager
  - Several dynamic libraries for internal and external game-related components *etc.*
  - Directories with resources
- Update and extension with either DLL update, plugin manager and/or patcher

End of lecture #9

Next lecture

Resource and object sharing