Informatikprojekt

Concept DocUmentation

Lars schlömer, Fabian Friederichs

2018

Table of Contents

[General Concept 2](#_Toc516217786)

[Injector 3](#_Toc516217787)

[What is dependency injection, inversion of control, dependency inversion? 3](#_Toc516217788)

[Inversion of Control (IoC): 3](#_Toc516217789)

[Dependency Inversion Principle (DIP): 3](#_Toc516217790)

[Dependency Injection (DI): 3](#_Toc516217791)

[Why choose DI, what is accomplished with it, and what alternatives were explored? 4](#_Toc516217792)

[The dependency injector 4](#_Toc516217793)

[What functionality does the DI require? 4](#_Toc516217794)

[What functionality does the module interface require? 5](#_Toc516217795)

[Module Interface proposal 5](#_Toc516217796)

[Core 8](#_Toc516217797)

[The necessity of a core 8](#_Toc516217798)

[Core responsibilities 9](#_Toc516217799)

[Platform abstraction 9](#_Toc516217800)

[Memory management 10](#_Toc516217801)

[Multi-threading support 11](#_Toc516217802)

[Core type library 12](#_Toc516217803)

[Debugging/Monitoring 12](#_Toc516217804)

[Scheduling 12](#_Toc516217805)

[Console 13](#_Toc516217806)

[Messaging 14](#_Toc516217807)

[Configuration 14](#_Toc516217808)

[Core operation 15](#_Toc516217809)

[Core linking 15](#_Toc516217810)

[Application Base 16](#_Toc516217811)

[Project Structure proposal 16](#_Toc516217812)

# General Concept

All the big game engines available to students, e.g. Unreal Engine or Unity, seem to be very monolithic and make it difficult to explore some of the more low-level/"under the hood" components. Although, according to the respective documentation, modular design has been used in those engines for some time now, they clearly emerged from the basic idea of a complete game engine. Plugins, that are available to these engines represent small sets of additional functionality, but replacing major parts of the baseline functionality seems to be complicated. This is, from our point of view, because nearly all that functionality is implemented in a mandatory, core-like part. You get all that core functionality even with a minimal configuration. We call this kind of design fat core.

This makes it strenuous to prototype or otherwise experiment with such components in the environment of a fully functional game, because there is a high degree of coupling between them. Even if some of the core components could be replaced, the interface that has to be implemented would be quite restrictive.

It appears to be highly desireable for teaching and experimental work to have a framework which makes it easy to rapidly prototype low-level components like render pipelines, simulation etc. without having to worry about breaking compatibility with any other parts of the engine.

To make this possible a framework enabling a plugin-based architecture is proposed. This framework should take care of a lot of tasks like connecting different task-specific code, which we will be calling modules from now on, and providing a common set of tools. Such tools would include a lowest common denominator of data types and frequently used functionality that every module has access to, so that many low-level tasks such as memory management, filesystem I/O and configuration management don't have to be reimplemented by every single module. This is desireable because modules by many different authors might be working together. These tools should be encapsulated in a single entity called the IPCore. Every module would get access to the same IPCore instance.   
Furthermore, a way to dynamically link modules at runtime, without pushing this onto module developers, will be needed. The use of an inversion of control design principle would fulfill these requirements.

In short, the concept is the following:

* There is a very lightweight core that realizes a common baseline for the system
* All actual engine functionality is implemented in the form of (in the best case very orthogonal) plugins, more specifically called **modules** for the rest of this document
* A dependency injection system glues everything together

# Injector

## What is dependency injection, inversion of control, dependency inversion?

### Inversion of Control (IoC):

“Don’t call us, we call you!” A very broad design principle in which flow of control is reversed. You might call into our framework to access some core functionality like the platform abstraction layer, however, we construct and initialize you, as well as define the broader flow of control. This is done by configuring how objects are ‘wired’ at start up, e.g. by reading in a file that describes relationships between task-specific code. These objects will be called **modules** for sake of simplicity in what follows.   
This design principle generally makes an application quite flexible and encourages modularity.  
A couple of IoC patterns make it possible to use **Dependency Inversion**.

### Dependency Inversion Principle (DIP):

To make high-level code, typically control flow defining, less dependent of low-level code, generally utility code like glm or something like a data structure parser, an abstract interface is put between the two, thus making both depend on that abstraction instead. Designing an interface around the desired interaction between high-level and low-level code, instead of separately, as is usually the case, results in less coupled modules, since they are less implementation dependent, and therefore make them easier to change, test and reuse. Being the author of both sides is not always possible, but even done only for one side this approach is superior in the above aspects.

### Dependency Injection (DI):

This is a pattern used in **IoC** that enables **DIP**.   
Henceforth a module that depends on another module and implements an abstract module interface will be called **dependent**. Such an object has its dependencies resolved by an **injector**. This way modules do not need to worry about loading any other module at runtime or statically binding any at compile time. All that is needed is the respective header file defining the interface.  
A dependent can have any number of dependencies. A **dependency** is any abstract interface that a module depends on and can be satisfied by any module that implements aforementioned interface.  
Using this pattern it is possible to effectively implement modules that follow the **DIP**. Since a dependent is not allowed to know anything about the implementation of a dependency it needs access to either some kind of factory that creates an instance of the dependency it needs or be given such an instance from an external source.  
The injector will load a dependency graph modeling the relationship between modules and inject the necessary modules into each other’s dependencies accordingly. All modules will have a function the injector can call to set module instances in their dependencies. This is commonly called setter injection. Using DI will allow for easier creation of modules that follow DIP, as an easy to use, abstract interface will be inherited by every module. All that is then required to do is model the possible interactions between modules according to DIP and recreate this as an interface that inherits the basic module interface. This basic module interface will take care of things like up- or downcasting pointers to the dependencies so that dependencies can be used effortlessly and as already mentioned, find, load and initialize the modules that will fill those dependencies. Relationships between dependent and dependencies are then defined in the dependency graph and everything is taken care of by the framework.

### Why choose DI, what is accomplished with it, and what alternatives were explored?

**DI** enables the use of modules that follow the **DIP** which in turn fulfill the requirements of being highly decoupled of one another, specialized and easy to replace. This also means that rapid prototyping is made possible as required.  
Thus providing a DI framework that makes the DIP easy to follow fulfills the project requirements. A well thought out best-practice guide needs to be created to ensure consistent code with this unusual methodology.  
An alternative implementation that was explored is a mix of the **Blackboard Pattern** and **Reflection** in which modules store their data along with metadata. This metadata describes functions and other information so other modules can interact with it. One publication regarding a combination of these two patterns was found, but the presented solution seemed too convoluted, possibly slow and difficult to implement both the framework as well as individual modules so it was not further investigated.

## The dependency injector

### What functionality does the DI require?

* A data structure, e.g. a graph, which models the dependencies of individual modules. The graph should be chosen so that it is easy to find out by how many and which other modules a module is used as a dependency.
* It needs to load and save such a graph from an external source like a file.
* It should be able to browse a file directory to find and load shared libraries which contain modules. This should work platform independent.
* It has to be able to resolve whether a module A assigned to the dependency of another module B fulfills any of the following requirements set in module Bs implementation
  + Most specialized interface
  + Version
* It has to know whether a dependency is mandatory or optional whenever this dependency is changed. So it can output errors if a dependency cannot be fulfilled.
* It has to know whether a dependency of a module can be reassigned after initialization.
* It needs a way of initializing modules after their dependencies have been fulfilled
* Provide a common interface to all modules that they use, so that it can access a modules dependencies.
* It should report errors when loading, initializing and injecting back to some higher entity in a non-terminating way. Worst case no modules are started. The injector must not cause the program to crash.
* It needs to be able to start and stop modules.
* If a module A is stopped then all modules that have A in any of their dependencies have to be notified of this change. If such a dependency is mandatory then the dependent has to either stop too or get a replacement before A can be stopped.
* It needs to register the following commands in the IPCore
  + Reassigning a modules dependency.
  + Start and stop modules.
  + Query a list of all loaded modules
  + Get detailed information of a module such as the interfaces it implements, the version and what dependencies or extensions it has.
* A way to gracefully shut down all modules down. The order of shut downs as to not cause dead-locks is important.
* It should support extensions. Extensions are planned to be slimmed down modules that cannot act autonomously. They will be registered in extension points which are defined inside a modules implementation. This means they are even further specialized and are only guaranteed to work with the specific module they were designed for, not any module implementing an interface. When a module calls such an extension point all extensions registered there will be called with a set of parameters defined in the extension point. The extension point host does not know about any of the extensions registered, but the extensions know of the host. This adds further flexibility to modules, if desired by a module author.

### What functionality does the module interface require?

* It needs dependency metadata to define whether a dependency is mandatory or optional, whether it can be updated after startup or not and what interface the dependency requires.
* It requires an identifier for the module that is identical to what is used in the injectors graph.
* It has to know what interfaces the module implements.
* The modules version.
* It needs the file path to the shared library it was loaded from.
* The interface further needs a container that maps dependency names to the pointers of modules who fulfill those dependencies. It should also handle up/downcasting to the desired interface.
* Also a function the injector can access to start and stop the module.
* Some way to tell the injector whether it has been initialized successfully.
* A pointer to the programs IPCore object

A way to define extension points in which extensions can be registered.

## Module Interface proposal

Class **Module\_API**:

* **Friend** of **injector** so that protected fields can be accessed by it.
* A **public** function to retrieve a **ModuleInformation** object.
* A **private** **Boolean** **isStartUp** initialized to false which will tell whether the module has been initialized/started up.
* **Private pure virtual** functions **\_startup** in which a module will handle any necessary initialization **after** all of its dependencies have been injected.
* **Private** **pure virtual** function **\_shutdown** in which a module will handle any necessary clean up **before** any of its dependencies have been removed/invalidated.
* **Protected** pointer to the applications **IPCore** object that the **Injector** will assign and the **module** will use to access any core functionality.
* Protectedfunction **startUp** which calls **\_startup** and assigns the return value to **isStartUp** and returns this value.
* Protectedfunction **shutDown** which calls **\_shutdown** and assigns the negated return value to **isStartUp** and returns this value.
* A virtual function **dependencyUpdated** which is optional to implement. This will be called by the **Injector** upon successfully changing any of this modules dependencies. This will need a parameter to tell what dependency has been changed and a pointer to the object previously assigned to this dependency. Any necessary clean up and initialization should be handled here.

Class **ModuleInformation**:

* Following public **string** members: (Maybe string is too simple for some of these)
  + iam – This will contain the names of the abstract module interfaces it implements. Delimited by dots. E.g. “Module\_API.MyFancyFooBar\_API”.  
    This would be used to check whether a dependency fulfills the possible type requirement some modules might want.
  + identifier – This is the identifier (The name) of this module. These should be non-conflicting with other modules as this will be used extensively by the framework to access it within containers. Something akin to what is done with package names in java.
  + version – This would be used to check whether a module fulfills a dependencies possible version requirement. It is very rudimentary. Using a custom data type might be necessary when this is actually used by modules as a string is a lot of unnecessary work to compare versions.
  + dlibpath – This would be the absolute file path to the dll/so this module was loaded from. Use unclear as of now, but might be useful someday.
* Some kind of **public hash\_map** **depinfo** which will use the dependency identifier as key and have a **DependencyInformation** as value. This will hold optional metadata a module can assign to its dependencies like necessary version, whether this dependency is mandatory or optional or whether it can be updated at run time or not.
* A **public DependencyContainer** object that holds pointers to modules assigned to the modules dependencies. This custom container should handle up/down casting of the stored pointers to the requested API so that it is as simple as possible to retrieve the dependencies implementation.
* A public **ExtensionReceptor** object. This will be similar to the DependencyContainer but for **Extensions**. These use a slightly different base interface than modules and several can be assigned to a single “dependency”.

Class **DependencyInformation**:

* An object of this type should be created inside a modules constructor which will be called upon loading the dll/so and long before any injection or start up. It should also be immutable.
* A **const bool isMandatory** which is false by default. Self explanatory?
* A **const bool isUpdateable**  which is false by default. Self explanatory?
* A **const string ModuleType** which contains the name of the most specialized Module\_API (child) that this dependency requires.
* A **const (string/version) ModuleVersion** same as **ModuleType** but an implementations version.

Class **DependencyContainer:**

* **Injector** needs to be **friends** with this class too.
* A **private** **hash\_map** **dependencies** which stores **Module\_API** pointers (The assigned modules assigned to specific dependencies). The key is the dependencies name as described in the dependency graph loaded by the Injector.
* **Protected** function **assignDependency** which takes a dependencies name and the pointer to a module inserts these into **dependencies**.
* **Protected** function **removeDependency** which takes a dependencies name and removes it from **dependencies.**
* **Public** function **size** to report back the amount of dependencies stored.
* **Public** function **exists** which takes a dependencies name and returns whether this dependency has been assigned.
* **Public** template function **getDep** which takes a dependencies name and returns the module assigned to it as the type specified in the template parameter. A **nullptr** will be returned if the dependency doesn’t exist or the cast fails.

Class **ExtensionReceptor:**

* **Public hash\_map** **expoints** which assigns **arrays** of **Extension\_API** to extension point identifiers (strings)
* **Public** function **execute** which executes all extensions assigned to a given extension point with a given set of arguments passed as parameters. This will probably need some kind of **any** object to pass arbitrary objects.
* **Public** function **isActive** to query whether a given extension points extension at a given position is active.
* **Public** function **setActive** to set an extension inside a given extension point to active or inactive.
* **Public** function **assignExtension** which inserts a given **Extension\_API** object into the given extension point (at the given location)

Class **Extension\_API:**

* **Identical** to **Module\_API** but missing the startup and shutdown functionality.
* Also gains a **Boolean isActive** member. This would be used to decide whether an extension of an extension point should be executed or not when the extension point is executed.

# Core

The core is a central part in our architecture as it helps orchestrating the communication between the modules, provides convenient access to resources and essentially drives the whole system.

This part explains the overall concept and purpose of the core, as well as its components.

## The necessity of a core

As mentioned in the introductory chapter, our architecture tries to remove the need to fit modules to predefined, restrictive interfaces. However, in a meaningful system, modules need to communicate. Not introducing a lowest common denominator at a certain level leads to some problems.

One big issue is the use of STL container types in interfaces. A minor difference in the STL implementations between two modules can break binary compatibility. Even if the same compiler is used and the runtime libraries only differ in a patch version, the problem is likely to occur.  
The only solution is to abandon STL containers and provide a custom implementation, which is guaranteed to be equal for some specific core version. Although binary compatibility cannot be guaranteed for different compiler versions (due to potentially different class layouts etc.), the interfaces won’t break if the runtime library version differs.

Further, the module interfaces should agree on a set of basic types like integral types, floating point types, strings and so on. To avoid having new type definitions and implementations for the same types in each module interface, those types need to be provided by a common entity shared with all modules: the core.

For asynchronous communication, developers may tend to create very specialized solutions for their specific modules. If many modules are developed independently this results in redundant and bloaty interfaces and is especially inconvenient for modules which need to interface with many others. Asynchronous communication is just another example of a recurring standard task; thus a general implementation should be provided by the core.

Memory management is problematic when creating a stable and efficient plugin architecture. Memory that is allocated by one module must not be freed by another module. Again, the differing runtime library versions cause problems here. If two modules link against the exact same runtime library, there is only one heap created for the whole process. However, in all other cases multiple heaps may be created, so trying to free memory on a heap other than the one the memory block was allocated on, results in undefined behaviour. Such an error is easily incorporated and often difficult to spot. For example when using STL containers with the default *std::allocator*.

On the other hand, if heap allocation is used carelessly more problems will arise. In the general, standard new/delete and malloc/free calls result in system calls to the OS. Those are slow, because on all modern platforms a context switch must be performed. Furthermore, from a purely conceptual viewpoint, the performance that an application can achieve is now strongly coupled to the systems memory management facilities. If the application is some sort of simulation with interactive frame rates (e.g. games) the additional overhead can have a measurable impact on the maximum achievable performance. In addition to that, careless handling of memory management can literally cripple the performance on memory constrained systems, as it creates memory fragmentation problems. In a typical OS memory manager, allocation takes longer when the memory is heavily fragmented. In the worst case allocations start to fail frequently, and the application becomes unstable and crashes. An application that messes with memory allocation also slows down the whole system it is running on.

To prevent described issues, memory management should be done entirely by the core. This way, with the right allocation strategies, memory allocation becomes less prone to fragmentation and OS specific limitations and therefore faster and more robust.

In general, by using a core, we can abstract away recurring, error prone tasks from the modules and make communication easier between them.

## Core responsibilities

In this section all the core responsibilities are briefly explained.

### Platform abstraction

The core should provide a platform independent interface to do common, usually platform specific, tasks such as:

* File I/O and file system navigation
* Networking (sockets!)
* Drawing-surface creation
* Shared library loading
* Access to OS-specific messaging system

To keep the core’s code itself platform independent, that basic functionality should be abstracted through a thin layer. We call that layer PAL or „Platform Abstraction Layer“.

Modules shouldn’t be able to access the PAL directly. Instead the core should provide an interface to access PAL functionality. This way, we keep the PAL „flat“, easy to maintain and implement for different platforms.

The PAL is also a good place to store platform specific configuration that must be known at compile time (such as cache line size etc.).

### Memory management

The memory management interface should consist of low-level allocation strategies and a high-level allocator concept which is compatible with custom container implementations.

The low-level part could be build upon a simple paging system, where a number of pages are allocated on application startup or single pages are allocated on demand. The allocation strategy implementations then acquire pages for themselves and hand over parts of these pages to the client on allocation.

A few examples of allocation strategies are:

Linear allocator

Very easy to implement and fast due to its simplicity. It is perfect for allocating memory for data that lives for exactly one iteration of an algorithm. The idea is that a large chunk of memory is managed. There’s a pointer to the beginning of the chunk and allocations move this pointer by the number of allocated bytes. Individual objects are never deallocated, instead, at some point the pointer is simply reset to the beginning of the chunk. All deallocations happen within a single pointer-write operation. Hence, there is virtually no overhead.

Stack allocator

Similar to the linear allocator, but individual objects must be deallocated in the reverse order they were allocated. Well suited for some algorithms.

Free lists

Very useful for arbitrary allocations and deallocations of small objects. Large chunks of memory are split into small blocks of the desired maximum object size, which form an in-memory linked list. A block is either occupied (allocated) or free. In the free case, a pointer at the beginning of the block points to the next free block. Allocations and deallocations push and pop free blocks onto that list. This is quite efficient for many allocations and deallocations of similarly sized objects, because no system call is neccessary and allocation and deallocation are O(1). The idea can be extended to a general-purpose allocator scheme, where there are free lists for different size classes. Based on the size the best fitting size class is chosen. This approach trades memory footprint for speed.

Many others…

The cost of frequently allocating small objects can be greatly reduced.

If the client wants to allocate a chunk of memory that is larger than the page size, either adjacent pages could be combined, or a fallback could be made to standard new and delete. The fallback option is actually not that bad, because large allocations don’t happen very often and secondly create less fragmentation than small ones.

The low-level allocators must also respect alignment requirements specified by the clients, this is particularly important in strongly parallelized contexts where stricter alignment rules than the default must be used to prevent false sharing and similar problems.

On top of such a basic set of low-level allocators, a high-level allocator concept similar to the standard library allocators should be implemented. These allocators then should be used with the custom container classes. If they are fully compatible with the STL containers, even STL containers could be used efficiently with the system. (Don’t use them in interfaces, though!)

### Multi-threading support

Multi-core CPUs are present in almost every device nowadays. To achieve the best performance for an application, that source of compute power must be leveraged. A problem is though, that multi-threaded programming is hard, time consuming and very difficult to test, therefore designing thread-safe applications and algorithms that can use the hardware to full capacity is often neglegted.

To make multi-threaded programming more convenient, the core should provide the basic building blocks as well as ready-to-use thread-safe datastructures.

The first idea that comes to mind for parallelizing the work is to assign each major subsystem to a dedicated thread. That can work if the subsystems do a similar amount of work. If one system does very little work and idles most of the time, either a whole core wastes CPU time while idling or the thread suspends itself until the next frame. The first option is obviously bad, the second is not optimal as well. For putting a thread to sleep or waking it up again, system calls must be performed. That adds latency and overall hurts the performance. Scalability is also a problem, because the number of parallel executable threads on a system is finite, whereas the number of subsystems is arbitrary. When there are many more threads than logical cores, the systems scheduler becomes the bottleneck, and thus the maximum achievable performance is strongly coupled to the underlying platform.

A thread pool is a way better option. Here, a set of worker threads is kept running all the time.  
The client enqueues tasks, workers pull them from the queue and execute them. To reduce contention on the global work queue, each worker has a local work queue. If a worker tries to get a new task, first the local queue is checked. If the local queue is empty, the worker tries to pull a task from the global queue. When the global queue is empty too, the worker tries to steal a task from the back of a local queue of one of the other workers. This concept is called work stealing. The advantage over the thread-per-subsystem approach is, that the work is automatically balanced, and as long there is enough work to do, the CPU can be completely utilized.

Blocking or waiting calls are a problem, because they stall the worker. For such asynchronous workloads the core should provide a future implementation and the neccessary primitives for asynchronous programming.

At the lowest level access to synchronization primitives and actual threads should be granted, in the case a module developer needs that amount of control for a specific task.

To sum up the multithreading features, the core should provide the following:

* Creating/Destroying threads
* Synchronization primitives (mutexes, atomics, fences…)
* Ready-to-use, thread-safe data structures
* Asynchronous programming (futures!)
* Thread pool

### Core type library

As mentioned in the introduction, the core implements a basic set of containers and types to create a lowest common denominator for the communication between modules.

There should be thread-safe and non-thread-safe versions of containers, thread safety should be deducable from the respective class names. Making everything thread safe would be unneccessary and hurt the performance in situations where no thread safety is needed, because solutions that prevent data races always create some overhead.

Beside the containers, all primitive types should be defined by the core. Especially fixed width integers, floats, doubles etc.

Complex standard types like the string type should also be implemented in the core.

In addidtion to that, some special types are introduced which are useful for communication. Two examples are the any type and the function type. The any type uses type erasure to hide the type of the contained data. The function type is a general-purpose function wrapper. Other than the STL version, this type doesn’t differentiate between member functions and free functions, so all callable entities with the same signature can be stored in a single container of the same type.

### Debugging/Monitoring

For exception handling, the modules may use the standard exception language feature of C++,  
but care must be taken that no exceptions are thrown across module boundaries. That means that a module must at least catch all exceptions just underneath the interface layer.

The core therefore implements central error management which includes logging and controlled shutdown when critical errors occur and an error management helper class that can be used by the modules to handle exceptions at interface boundaries. The modules can register callbacks to implement specific behaviour for certain exceptions. All exceptions are then internally passed to the central error management, so the core can react to critical errors and do some logging.

The helper class further can be used to query the state of a module: if it’s errornous or not, get the last exception etc.

In addition to error management, the core should implement some machinery that can gather data and generate some statistics for monitoring and debugging.

### Scheduling

Since the project is primarily a toolkit for soft realtime applications like simulation and games, we need a simulation loop.

One could let the modules implement their own simulation loops, but that would mean that there is no easy way to synchronize between the modules. Synchronization at some point is important because many modules produce different output data for each simulation frame. Some modules may depend on data that another module produces for that frame.

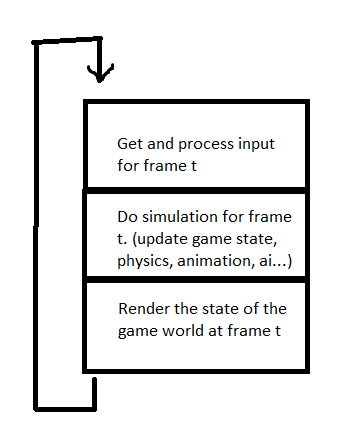
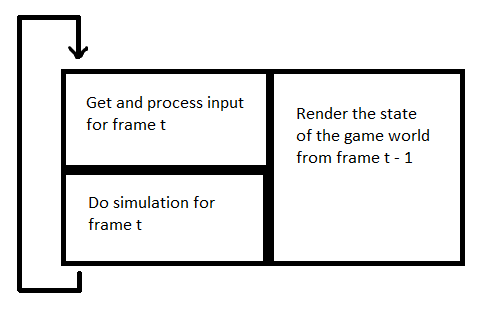


Figure 2: An optimized game loop

Figure 1: Standard game loop

Figure 1 shows the standard version of a game loop. It is designed without multicore CPUs in mind.

Much higher frame rates can be achieved through parallelizing parts of that loop, and letting some subsystems work with data that is one frame old, as shown in figure 2. The functional blocks themselves can then again be parallelized internally to utilize all available processing power.

Because there is no single best design of a simulation loop, we can’t enforce one. Instead the core provides a facility named scheduler. A block, like those in figure 1 and 2, is called a main task. Main tasks are the entry point of the per-frame program of a module. To get a main task executed, a module creates a subscription via the scheduler instance exposed by the core.  
This subscription can be parameterized:

* Should the main task be executed once per frame or in a fixed interval?
* What dependencies does it have?
* Can it be executed from the thread pool or does it have to run on the main thread? (Some libraries like OpenGL do have that restriction)
* …

At the end the scheduler should build a tree internally, that models the main task’s data dependencies, and thus, figure out automatically which tasks can be executed in parallel. Another option would be a hand-crafted tree of main tasks, read from a config file, but that approach can get difficult to implement if modules are added and removed at runtime.

However, the best design for the scheduler itself must be elaborated in the future.

### Console

A system composed of many modules has complex state and many options at runtime. To manipulate that state a console is the tool of choice. Console input is read and parsed in a seperate thread and transformed into reified command calls[[1]](#footnote-1). Those calls are queued up and then, at some point, executed with the parameters from the command line, if a command with that name exists. Modules can add their own commands by simply registering a name and a callback.

### Messaging

Asynchronous communication via messages helps decoupling modules and can increase throughput. The core should implement a convenient messaging system. The system consists of messages, an endpoint class and an endpoint registry.

Messages have a type and some payload.

Endpoints can be created by the modules. An endpoint has a queue for incoming messages and one for outgoing messages. If a module wants to send messages, it pushes them onto the outgoing-message queue and at some point in time tells the endpoint to send the enqueued messages. If a module wants to receive messages, it simply pops available messages from the incoming queue. A module can subscribe to another endpoint by connecting the own endpoint to it, after that the messages that are send from the connected endpoint are pushed onto the own incoming message queue. If an endpoint gets destroyed, all subscriptions are automatically cancelled.

To find endpoints of other modules, the endpoint registry keeps a list of named endpoints that were registered by the modules. The names of the endpoints that a module publishes should be listed in the modules documentation.

Regarding message types, the registry could provide a <type name, type id> dictionary to remove the need of string comparisions for every incoming message.

### Configuration

The core and every module will have some configuration values that have to be stored. To prevent the modules from storing custom configuration files all over the place, the core provides a centralized configuration management, which stores all the data in a single configuration file.

Standard file formats like XML and JSON could be used for that task but XML is, although human readable in theory, inconvenient to maintain and has an overall bloaty appearence. JSON is much more streamlined but the heavy use of special characters and braces is a source of error in large configuration files that makes it inconvenient to use, too. Other common formats like INI are not suitable for the single-config-file approach because they lack a hierarchical structure which is important to organize many values in a large file.

Therefore, we drafted a very simple file format. Listing 1 shows a snippet of such a config file.  
The hierarchical structure is achieved through python-like indentation and is part of the syntax.  
The modules can access the values by passing the dot-qualified name of the desired value.  
For example, to access the value of „fval1“, the qualified value name „section1.subsection1.fval1“ is passed. The most basic implementation should support boolean, integral, floating-point and string values. The parser should deduce the type automatically based on the text on the right hand side of the equal sign.

Values can be read, written, added and deleted programmatically and the config manager should be able to generate config files from the respective current set of values.

Some general-purpose code editors, like Visual Studio Code in this case, offer a collapse feature for indented blocks of text, wich makes editing this file format very convenient.

Listing 1: Configuration file format

# this is a comment

# configuration values for section 1

section1

# a boolean value

bval = true

# a string value

sval = "sadf"

# a subsection

subsection1

# a floating point value

fval1 = 1.5

fval2 = 1.e-9

# another subsection

subsection2

subsubsection1

# an integral value

ival = 42

section2

some\_important\_setting = false

global\_value = "this value belongs to the global scope"

## Core operation

The core has no integrated simulation loop. To drive a realtime simulation like a game, the application implements such a loop which constantly calls a special core function named *tick().*This essentially mimics the clock that drives the whole system.

The core should do the following actions during a *tick*:

1. Handle pending exceptions
2. Execute pending console command calls
3. Trigger the scheduler
4. Send messages that are enqueued in the outgoing message queue of the core’s endpoint
5. Dispatch and process messages that are enqueued in the incoming message queue

The tick function could also provide several callbacks like pre- and post tick.  
Eventually the simulation loop must know if it should proceed with calling tick or terminate the application. That could be realized through a boolean return value or a passed-by-reference parameter.

## Core linking

The core will be deployed as a shared library that is linked *implicitely* to the modules at application startup. A full-blown, pure virtual interface for the core would introduce too much overhead, because some of the core’s components are called very frequently. The thread pool demonstrates that, as it is designed to execute thousands of tasks per tick. However, possible problems with binary compatibility are a major drawback which need to be solved, yet.

## Application Base

A class *Application* defines a base class for concrete applications. It combines injector and core and runs the main loop that drives the core. In the initialization phase, a core instance is created and then used to create an injector instance. This instance is used to load the modules specified in the dependency graph config file. Next the main loop is entered which continuously calls the tick() function of the core until the core signals the application to stop.

Beside that, a dedicated console thread is launched which waits for input, and forwards that to the core’s console instance, that in turn parses it and queues up valid command calls to be executed on the next core tick.

# Project Structure proposal

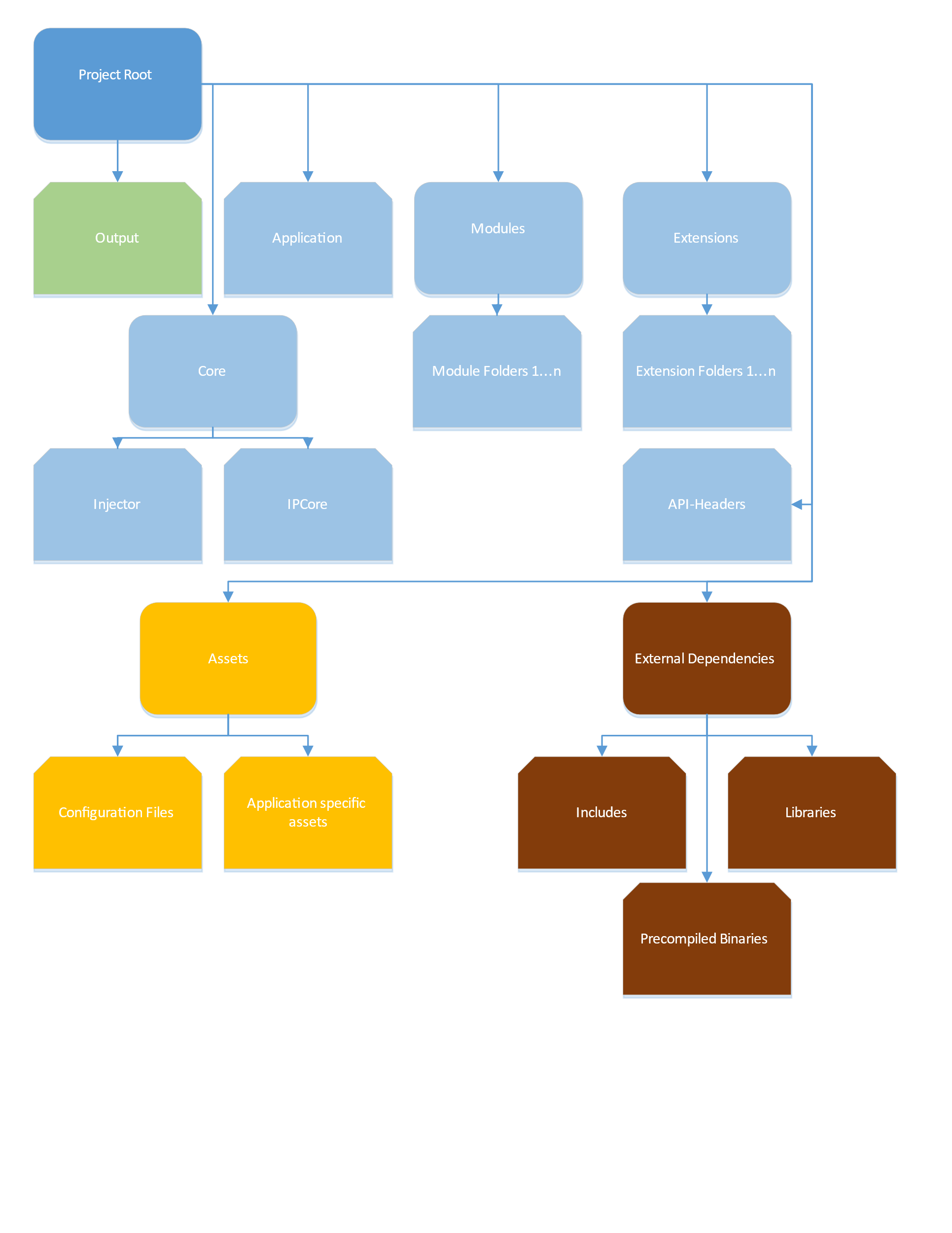
To combine all aspects discussed throughout this document, we propose the project structure illustrated in figure 3. The boxes represent directories, as they should be laid out in the actual project.  


Figure 3: Project structure

1. The term *reified command call* means a command call that is transformed into an actual object, so it can be stored. This allows for collecting command calls and passing them around for delayed execution. [↑](#footnote-ref-1)