# Current Structure

## Description

All code is hosted in a single DLL. Headers in a pch are precompiled allowing for reduced build times. Internally, modern C++ features such as function overloading and passing by reference are exploited frequently. Classes are usually instantiated as value types on the stack, eliminating the need for pointers and simplifying memory management. A single front facing C-Style Interface will be employed to manage front facing features and dependency checking.

## Technical Debt

* There are a massive amount of unused legacy functions and failed experiments in the code
  + Plenty of old database code
  + Visibility graph has gone through many iterations
* Reorganize file structure for readability and maintainability
  + The graph generator, Visibility Graph, and View Analysis are in the same CPP file, which may impact readability
  + HF::Graph, HF::Node, and HF::Edge could be moved to a seperate file, since they’re not exactly exclusive to the graph generator anymore
* Standardizing types throughout the project will drastically improve maintainability
  + Any time where points from the graph or POIs are used, only HF::Nodes should be accepted. This will drastically reduce the complexity of inserting them into the DB, as there will not need to be any extra work to figure out their IDs later when inserting results into the DB
* Decouple algorithms from the Database
  + Algorithms should not read or write to the database. That should be the responsibility of the object that owns them
  + Either the Coordinator or the C Interface should perform these actions
* C-Style Interface with language specific wrappers
  + Create the C-Interface at the highest level and give access to entire codebase, this way internal structure has no impact on public interface.
  + Use Pinvoke/CTypes to develop a wrapper in each language to shield end users from implementation details
  + Existing C# proof of concept provides a working demonstrat

### [Current] File.png

* Separate code and structures by file
* 4 Constructs of code
  + Class - A class holding objects with functions, etc.
  + Static - A static class with functions but no state
  + Utility - A collection of functions or libraries that are used throughout the program
  + Interface - Where external code links into private code
* The python interface uses C++ MVSC style linkage, and value types, therefore it uses the standard arrow

### [Current] Dependency.png

* Illustrate dependencies between classes
* Every edge implies a connection between all higher level branches

### [Current] Object Usage.png

* Green edges imply that a class uses types from another class to ensure consistency or reduce redundant code
* Common is not displayed to prevent visual clutter, however it exports no unique classes, only functionality.
* Version 1
  + Green hollow arrows  imply object usage, but not necessarily dependency. For example, the Visibility Graph accepts a Graph from the Graph Generator as input for convenience, but also accepts a list of points

# Proposed Framework 1: Independent DLLs

## Details

### Description

Code is split into independent feature complete DLLs for every class each following the *PIMPL*, Pointer to IMPLementation, idiom (https://herbsutter.com/gotw/\_100/). Users only explicitly load the DLLs they need, and are provided with a fully featured interface in each one. This will allow for each component to be compiled separately and swapped out as needed.

### Concerns

* Every class is in it’s own DLL, with its own binary interface intended to be used independently
  + Design Overhead
    - Changes to *public*  function signatures or class members will break ABI, forcing a recompile on all dependent DLLs.
    - New classes will require the creation of new projects with new interfaces
    - Function overloading can’t be used between DLLs due to the need for a C style interface
    - Each language specific wrapper will need to link to several DLLS
  + Multiple Copies of code
    - Libraries that are header only will be recompiled and stored within each DLL. This is required for the usage of templates https://en.cppreference.com/w/cpp/language/templates
  + Every DLL must check dependencies
    - Dependency checking must be built into every DLL’s interface since they’re all intended to be usable as an entry point.
* Complications from using PIMPL on every class
  + Implement Factory Design or require the use of raw pointers <https://herbsutter.com/2013/05/30/gotw-90-solution-factories/>
  + All classes must be allocated on the heap instead of the stack, which incurs some additional overhead for each allocation https://www.sciencedirect.com/topics/computer-science/heap-allocation

## Diagrams

### [Independent] Conceptual Dependency

* Displays conceptual dependencies between DLLS in a simple format.
* The Database, Common, and the OBJ loader have been merged since most DLLs will need a database connection, a way to load geometry, and the use of common functions
* Much of common would remain in its public implementation as it makes use of templates and can’t be represented in cpp files.

### [Independent] ABI Use

* Displays Actual dependencies between DLLS
* Double arrowheads, , indicate a C Style call through an Application Binary Interface returning a pointer to an object on the given dll’s heap. This is required to separate the loaded DLL’s private and public implementation.
* Most connections to Common/DB are to store/retrieve data, as every DLL needs to be able to both read and write from the DB

# Proposed Framework 2: Plugin DLLs

## Details

### Description

A more targeted approach to making our codebase modular. Instead of breaking all functionality into individual DLLs that are able to operate independently , we keep the program as a single codebase with a variety of different *procedures* that make calls out to external DLLs Plugins will have several different types depending on what functions we decide to break out.

Our analysis methods for example can be broken down into a basic set of inputs and outputs which all plugin dlls must have, then the developer is able to compile their own DLL without ever compiling any of our core library. This allows us to break out the parts that would benefit from being modular while keeping our core codebase and interface simple.

### Concerns

* Could be used to execute malicious code

## Diagrams

[Plugin] Group Concept

* Illustrates the core concept of DLLs
* Each plugin type would have a standard input/output specification which all plugins would abide by.
* Some plugins would act as alternatives, others would be extra steps in the process
* Core DLL handles which plugins to load, and provides information to each individual plugin as needed