OpenPASS Naming Conventions

General

OpenPASS is based on modern C++ (currently C++17). For coding guidelines, please refer to ISO C++ Core Guidelines

Concise Summarized Naming Conventions Example

```
/* foo_bar.cpp */                                  // File: lower_snake_case (to be discussed)
class FooBar                                       // Class: UpperCamelCase
{
private:
    static constexpr int MAGIC_NUMBER {-999};      // Constants: UPPER_CASE
    int myMember;                                  // Members: lowerCamelCase
    FooBar();                                      // Ctor: UpperCamelCase

    InputPorts inputPorts;                         // Inputs of the class if used as model
    OutputPorts outputPorts;                       // Outputs of the class if used as model

public:
    void Bar();                                    // Methods: UpperCamelCase
    void BarBar(bool flag, int counter);           // Arguments: lowerCamelCase
    void YaaBar(); /* Yaa = Yet Another Abbreviation */ // Abbreviations: UpperCamelCase
}
```

File Names

- File names should be in lower snakecase
- Current State: Mixed (UpperCamelCase, Snake_UpperCamelCase, ...)

Header Files

- Use `*.h` as file extension

Source Files

- Use `*.cpp` as file extension

Libraries

General

- When multiple libraries may be used to achieve or handle a particular functionality, they should be organized by "scope" to maximize clarity
  - Scope refers to components that share a tier in the agent structure
  - See **Topics For Discussion** at bottom of document

Interfaces

- Interfaces should be named descriptively according to the functionality they outline with an UpperCamelCase name
  - e.g. The interface for the world would be `class WorldInterface`
- We excessivly use gmock, so it would be great to privide a fake for each interface
  - e.g. `class FakeWorld : public WorldInterface {...};`

Classes

- Classes should be named descriptively according to the functionality they implement with an UpperCamelCase name
- If the Class implements an Interface, the Class name should generally be the same as the Interface name, with the `Interface` portion removed
  - e.g. The implementation of the AgentBlueprint would be: `class AgentBlueprint : public AgentBlueprintInterface`
- (See Topics for Discussion) When the Class implementing an Interface is to be exported as a part of a Library, the Class name should replace `Interface` with `Implementation`
  - e.g. The implementation of the world would be: `class WorldImplementation : public WorldInterface`

Member Variables

- Member variables should be descriptively named in lowerCamelCase
  - e.g. The member variable containing the AgentNetwork for the World module should be named `agentNetwork`

Input / Output Signal Naming

The template recommends to use the abstraction

- `std::map<int, ComponentPort *> outputPorts;`
  `bool success = outputPorts.at(localLinkId)->SetSignalValue(data);`
- `std::map<int, ComponentPort *> inputPorts;`
  `bool success = inputPorts.at(localLinkId)->GetSignalValue(data);`

Current state (without abstraction):

- Some models use `in_` and `out_` as prefix for signals.
- Some use no specialization

Methods

- Methods should be descriptively named in UpperCamelCase
  - e.g. The World Method for retrieving the time of day should be named `GetTimeOfDay()`

Additional Rules

- Use UpperCamelCase for abbreviations used in files/classes/methods/variables
  - This does not apply if the abbreviation is the entire name or the beginning of the name - in such a case the name is written with the rules for the appropriate type
    - `ID` → `id`
    - `AgentID` → `AgentId`
    - `ADASDriver` → `AdasDriver`
- Use UPPER_CASE for all constants
- Global variables, if necessary, should be constants, if possible; otherwise, they should be named in lowerCamelCase
- Enums should be preferably defined as enum class; as such, enum names should be in UpperCamelCase
- Decorate container by type aliases and use UpperCamelCase:
  - e.g. `using FooParts = std::vector<FooPart>;`
- Use `//` for comments

Avoid

- Do **not** use Hungarian notation for variables names (`iCounter` → `counter`)
- Do **not** specify the type of the underlying implementation (`partMap` → `parts`)
- Do **not** use magic numbers in the code; explicitly define constants instead
- Do **not** use `/* */` for comments

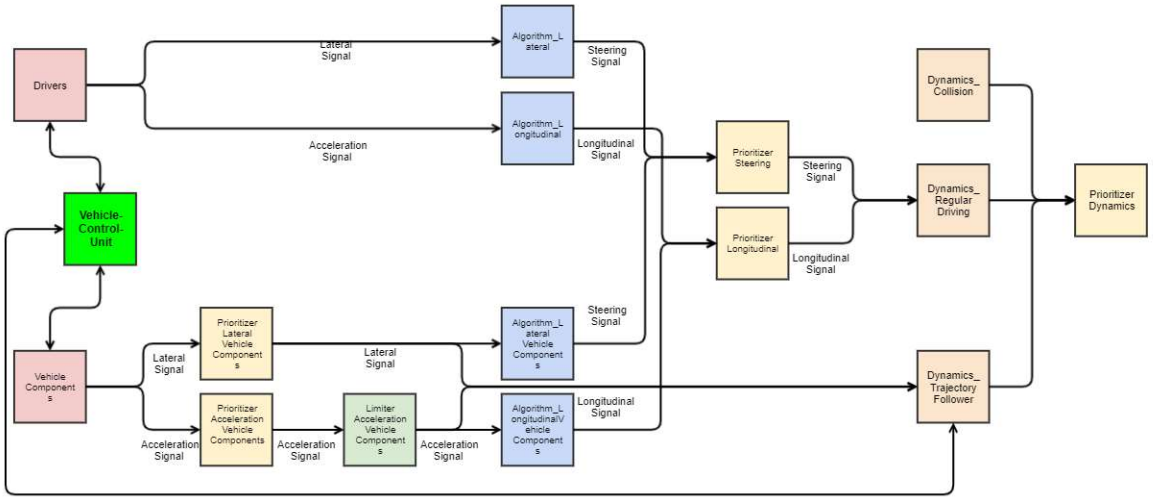
Discussion

Project file / paths / component names

Current situation:

- No common naming
- Algorithm/Sensor/Dynamics... used as prefix for module names, such as `Algorithm_Foo` or `AlgorithmBar`

Consider grouping components based on scope:



Scenario specific case:

- In folders:
  - "Components/Drivers"
  - "Components/VehicleComponents"
  - "Components/DynamicsPreProcessors" (C) CK
  - "Components/Dynamics"
- Alternatively, utilize the above names as prefixes for the module names
  - e.g. "Driver_AgentFollowing"

In every case, the content should be wrapped in a Namespace, such as

- ::openPASS::Algorithm::Collision
- ::openPASS::Driver::Afdm

Questions

- How should PCM be handled in the future - will this be integrated into existing component structure?
- If they remain separated, the same guidelines should apply to the PCM folder as well

The Export Case

Files that contain the exported library function are current just named after the class. They could be highlighted as "Export".

- e.g. "EventDetectorExport" contains all exported functions of the library.
- Currently all library classes have the "Implementation" suffix, e.g. "WorldImplementation". The `Implementation` -suffix does not provide any additional information, but it was used to avert name duplication with the framework modules.