Hello and thank you for looking at my winmodbus repo!

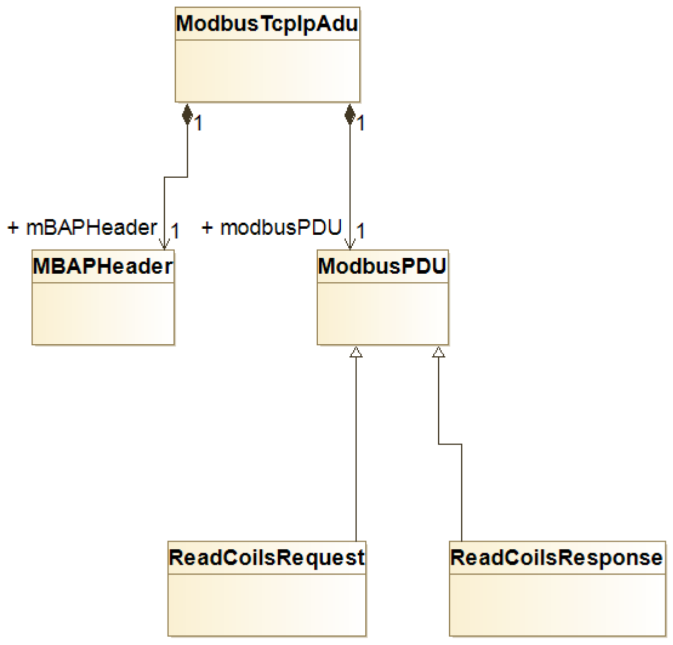
This collection of code was born out of an effort to explore test driven development C++ using Visual Studio and github while also studying the MODBUS over TCP/IP protocol. It helps to have something to write code about!

I would like state a few disclaimers on this code in its present state.

1. This is a version 0.0.1 level of completeness. It has unit tests that are passing, but I'm not yet even completely sold on the design concept. It is, however, at a point where I would bring it to colleagues for discussion (a day or two of work max). It is good to fail early!
2. The main goal of this design so far has been to extract simple, testable classes. I feel like there is definitely some OO elegance to evolve out of this. But I'm also not forcing it to happen. I'm trying my best to follow a TDD approach and just pass the test cases.
3. Some of these classes live at the boundary between the octet stream and protocol definition and C++ objects. Because of this there's a lot of C in there! I'm definitely open to coding alternatives, but what I did manage to do is fully encapsulate it so that these C code bits don't need to escape into the larger application. Again, in fully testable objects!
4. There are known inconsistencies between my low level treatment of the MBAP header and the PDU data. These would need to get rectified.
5. Last but not least, I'm not fully happy with the class naming yet, but it's early!

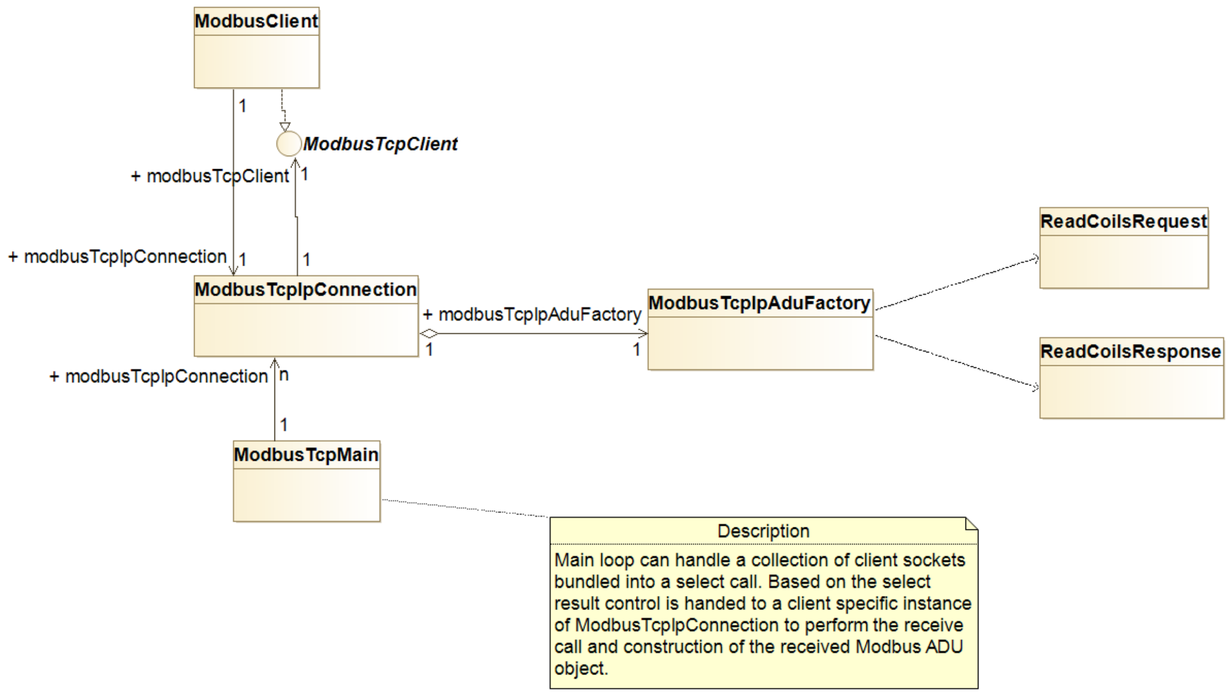
Anyway, on to the design. I started with the PDU classes, again just as a vehicle for playing with the IDE.

After a few test iterations I pretty quickly arrived at the ADU breakout below. One of the things that was already in the back of my mind was that there would also ultimately be an RS-485 version of the ADU with different addressing. I wanted that to be separated from a reusable PDU definition.



PDU specific definitions are all contained within the ModbusPDU subclasses.

Next I wanted to put it all in a greater context. My current thinking of what this looks like can be seen below.

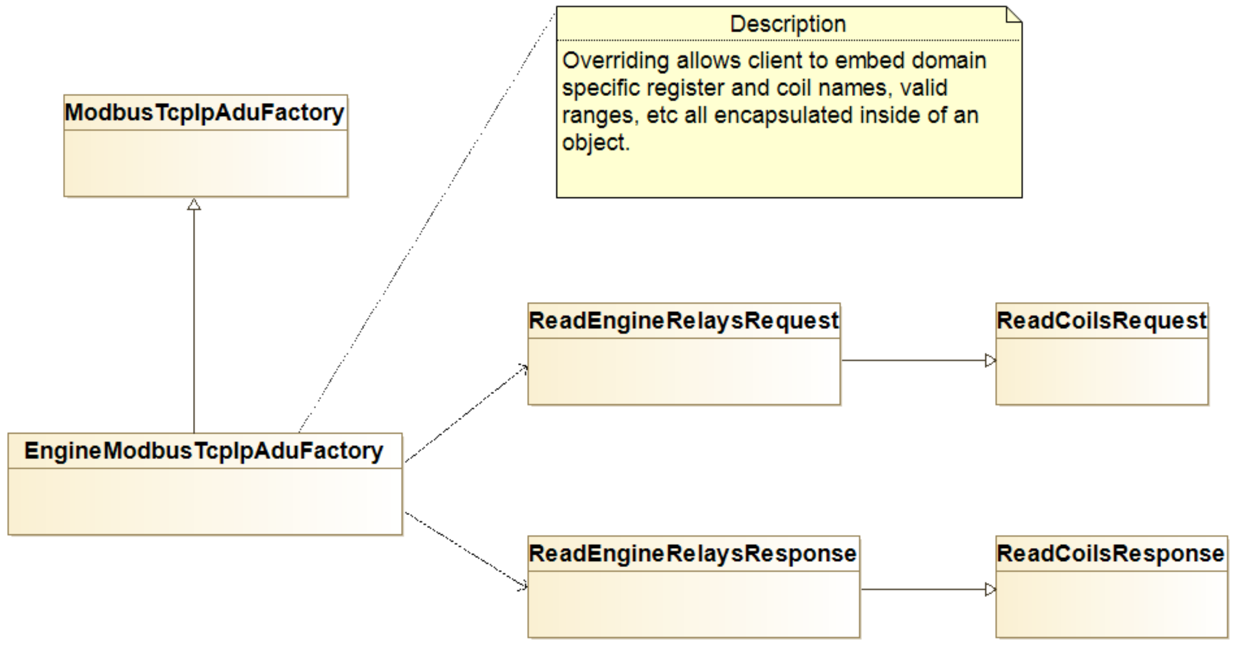

ModbusClient
ModbusTcpClient
+ modbusTcpClient
+ modbusTcpIpConnection
ModbusTcpIpConnection
+ modbusTcpIpAduFactory
ModbusTcpIpAduFactory
ReadCoilsRequest
ReadCoilsResponse
+ modbusTcpIpConnection
ModbusTcpMain
1
1
1
1
1
1
n
1
Description
Main loop can handle a collection of client sockets bundled into a select call. Based on the select result control is handed to a client specific instance of ModbusTcpIpConnection to perform the receive call and construction of the received Modbus ADU object.

Each TCP/IP MODBUS server would get its own ModbusTcpIpConnection object which contains the socket definition. This would get plugged into a ModbusTcpMain object which contains the blocking receive thread and the *select()* call. Based on the results of *select()* the appropriate ModbusTcpIpConnection instance is invoked. This ModbusTcpIpConnection class is also where all of the connection maintenance state machines and logic found in the MODBUS application notes would get realized.

The incoming octet stream is pulled out of the socket by the connection object and passed along to the ModbusTcpIpAduFactory for ADU creation. After the incoming ADU is ultimately constructed it is passed up to the client which is realizing the ModbusTcpClient interface.

Lastly I had some thoughts on possible future extensions. Let’s say an object is managing some relay statuses provided via MODBUS. It would be utilizing the ReadCoilsRequest and ReadCoilsResponse PDUs. These baseclasses contain the details of how to read and write information out of the PDU structures but they don’t have any specific application context. So one *possibility* would be to extend the protocol PDU object with an application specific one. If one were to do that then the ADU factory would also need to be subclassed and the “build ADU” method becomes virtual and we use polymorphism to get these enhanced application aware PDU objects.

An argument this approach would be that once we get up to the application we may want our data to be organized in more conventional C++ containers. In that case maybe it makes more sense to have the application extract the relevant data from the PDU object and store it separately. If the application data is going to stick around in this object for a while then it makes the memory management aspects more predictable as the PDU should probably be released predictably after the client is finished reading it.


Description
Overriding allows client to embed domain specific register and coil names, valid ranges, etc all encapsulated inside of an object.
ModbusTcpIpAduFactory
EngineModbusTcpIpAduFactory
ReadEngineRelaysRequest
ReadCoilsRequest
ReadEngineRelaysResponse
ReadCoilsResponse

Thanks for reading through this! I hope it serves as an example of how I work, my capabilities and my communication skills!

Enjoy!