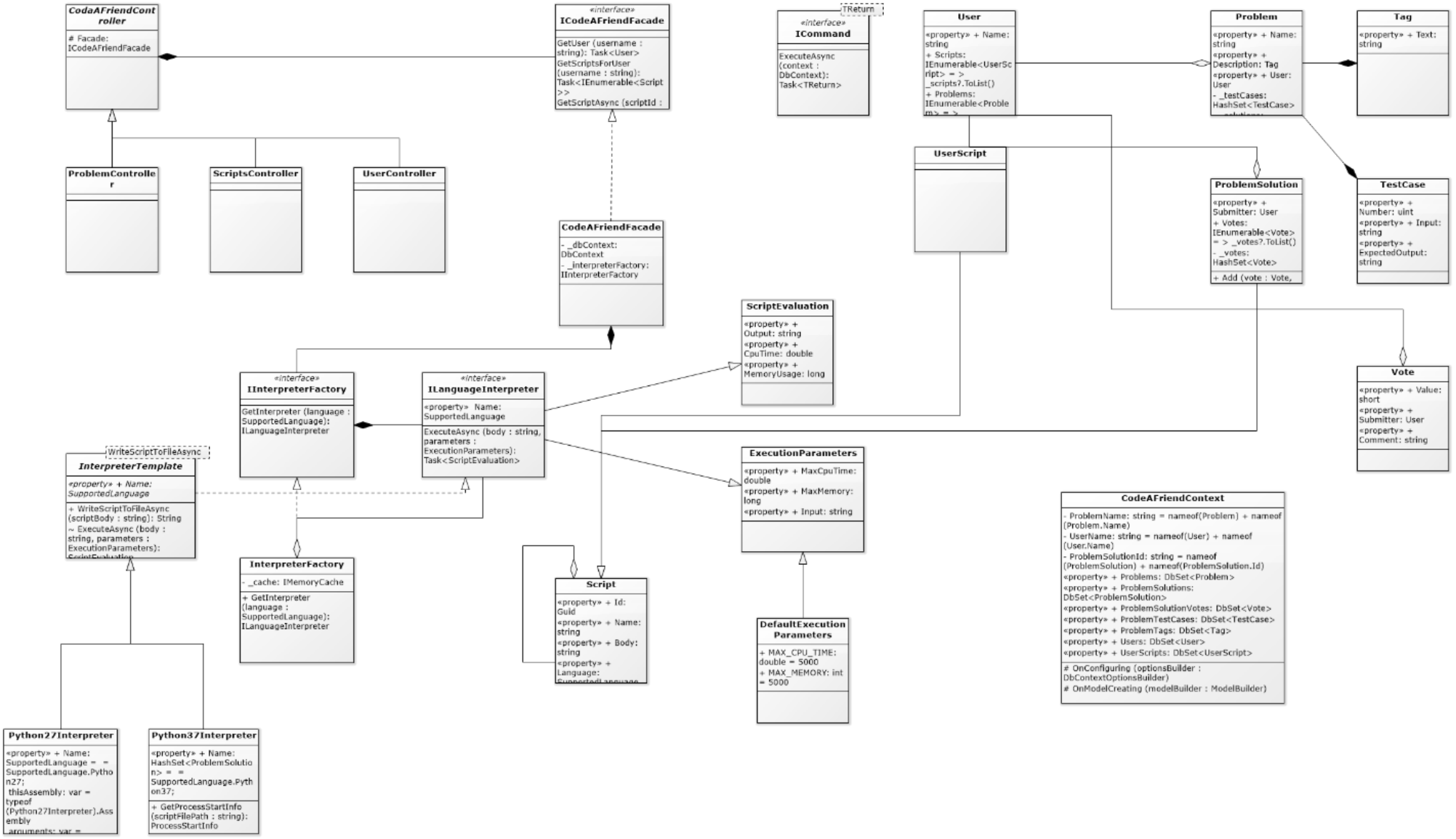
1. **Name**: Warren Ferrell
2. **Project Description**: A website for submitting and voting on code samples that solve user submitted problems. Will only support python at first.
3. **Implemented Features**:

Requirement Id	Requirement Name
UR-01	Script Editing
UR-02	Script Saving
UR-03	Script Execution
UR-04	Script Analysis
UR-05	Problem Editing
UR-06	Problem Saving

4. **Not Implemented Features**:

Requirement Id	Requirement Name
UR-07	Problem Testing
UR-08	Solution Submission
UR-09	Problem Solutions
UR-10	Solution Voting
UR-11	Problem Tagging
UR-12	Problem Searching
AR-01	Script restrictions
AR-02	Problem deletion

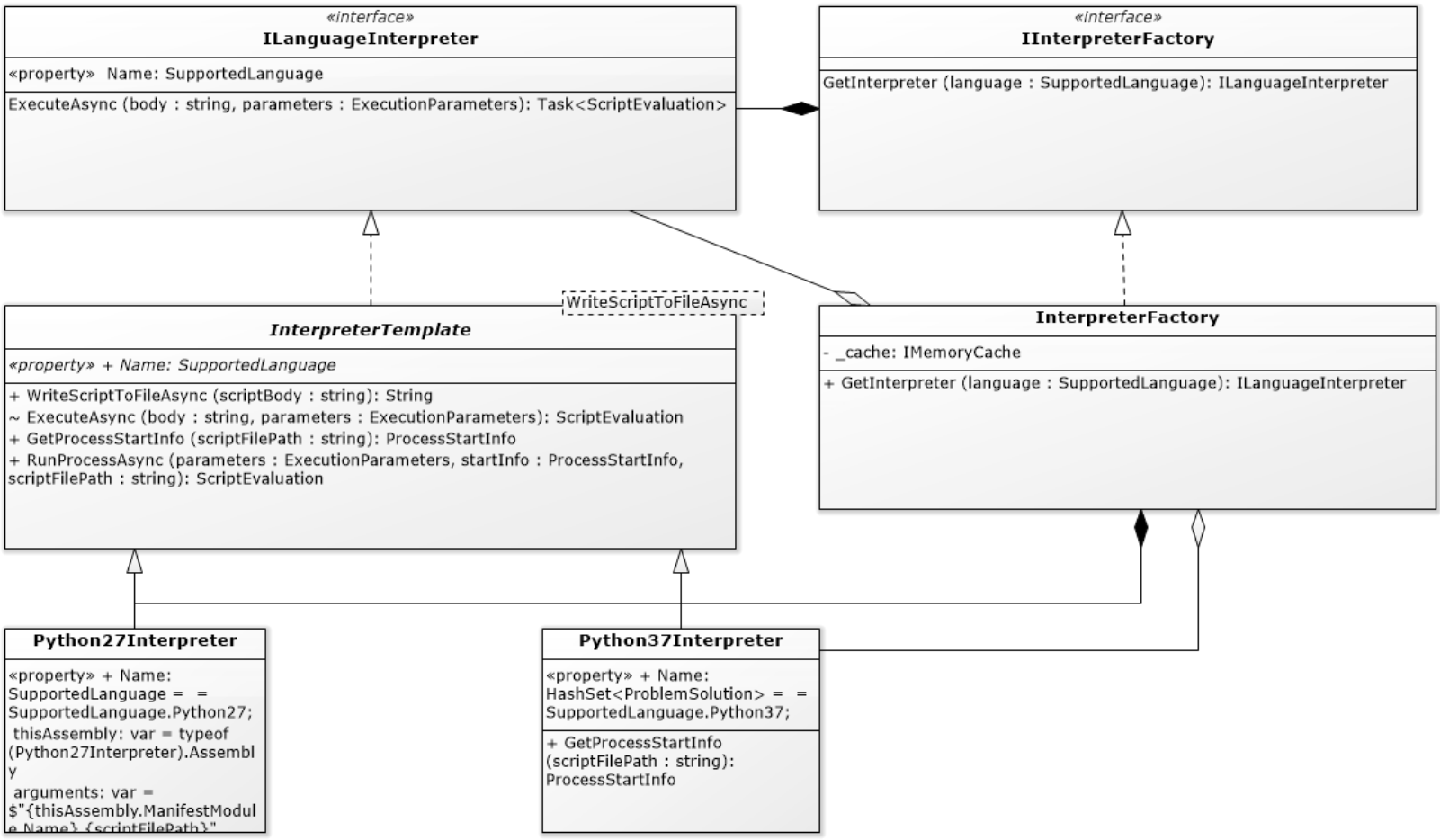
5. **Class Diagram:** I don't have access to VS Enterprise so I couldn't have the VS architectural model generated for me so I used Software Ideas Modeler to generate my class diagram from source code. Unfortunately, it did not do a very good job and tidying up the class diagram is a lengthy process. My class diagram did not change a great deal. The names of many classes changed and I added a few layers of abstraction but the overall design was very close to the design I originally generated. Having the class diagram and auto-generated code from my UML was very helpful as it gave me a lot of direction while working on the project. One of the main issues I had with it was that Umbrello's C# code generation did not make use auto-properties which is one of the most import features of C# features that make it easier to work with than java.



6. Design Patterns:

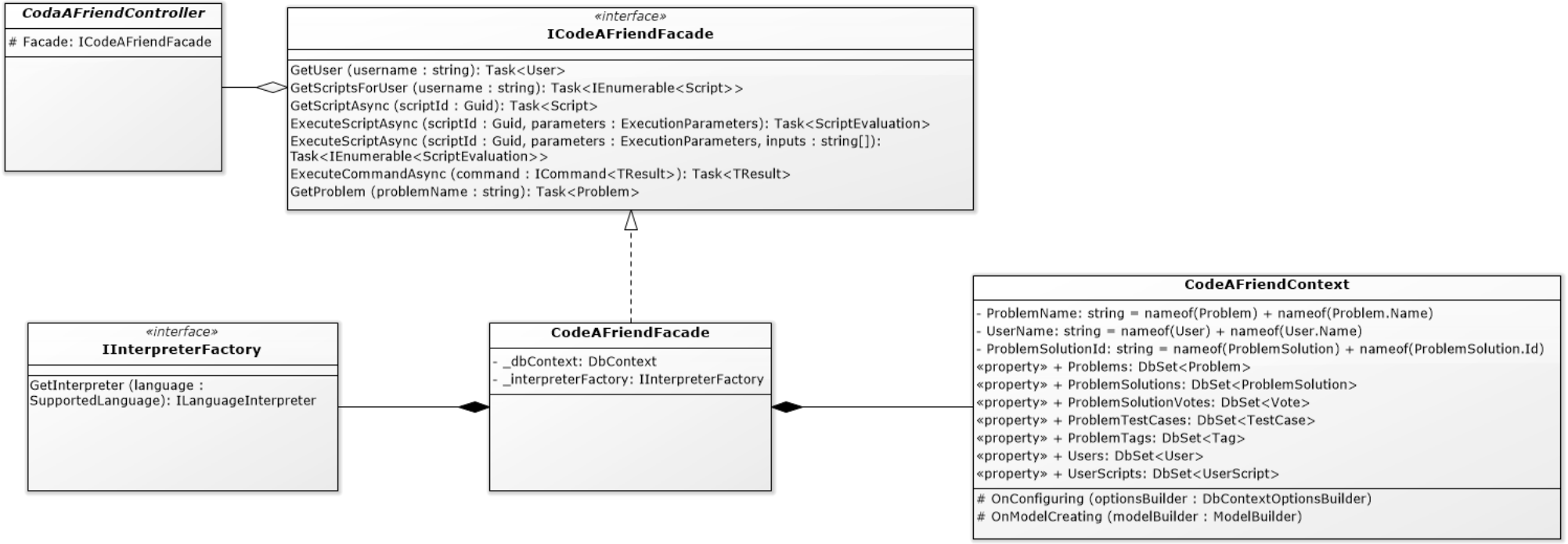
a&**b** : **Factory** & **Template**… & Strategy

I am showing these two DPs together because they're very closely integrated in my architecture. I chose the Template design pattern because some of the code around handling processes is quite complex, could vary from language to language, OS to OS, etc. The Language template implements ILanguageInterpreter (essentially strategy pattern as a different template could be defined for LanguageInterpreters) it declares three steps for interpreting a body of code: WriteScriptToFile, GetProcessStartInfo, and RunProcessAsync. Although I was not able to get Python27 interpreter working, its design was to run an interpreter inside another .NET core process, while the Python37 interpreter uses a natively installed python 37 distribuition. The InterpreterFactory grabs an instance of the correct Interpreter for the calling based on a passed in SupportedLanguage Enum. With the Python37Interpreter a new process is started by the interpreter so only one instance of the interpreter needs to exist at any one time, thus we cache that instance in an IMemoryCache. The Python27Interpreter is meant to reuse the same process over and over so we would run into issues if two threads tried to call it at the same time, to resolve this issue, it just creates a new Interpreter for every call.



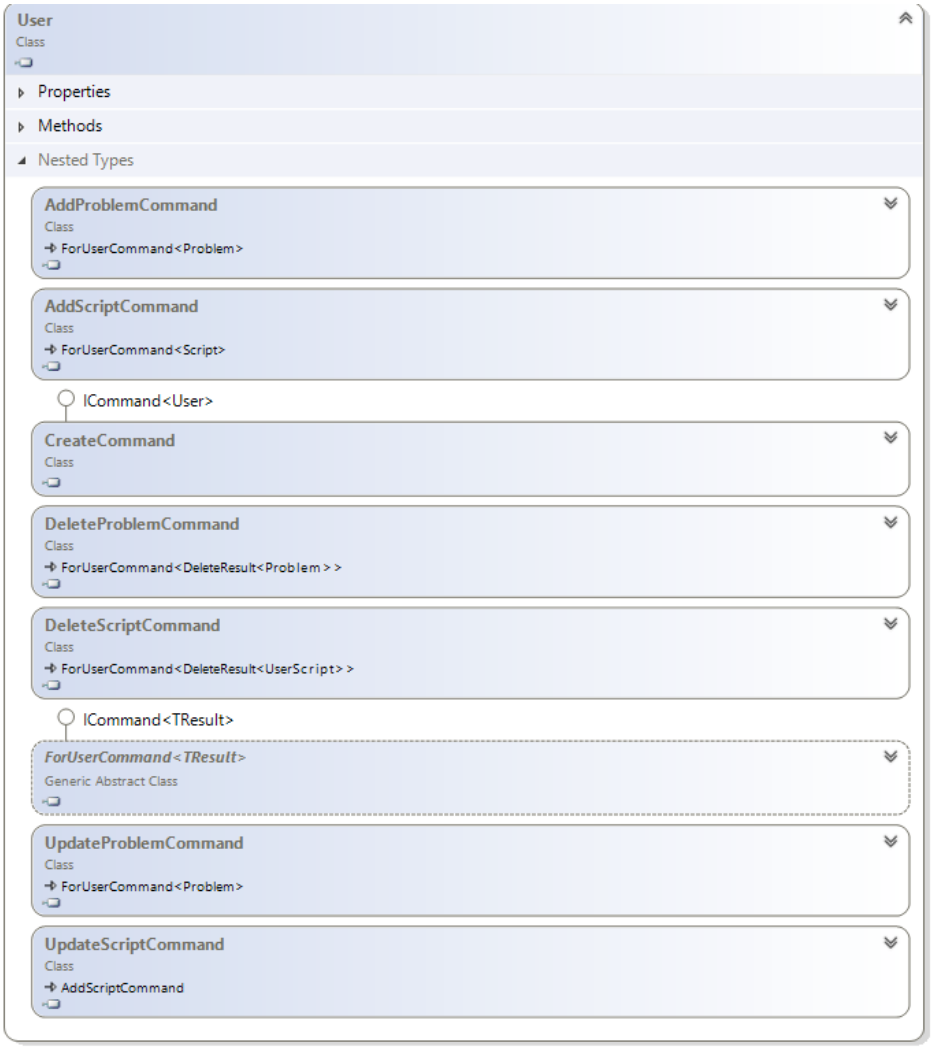
c: **Facade**

Controller classes very quickly and very easily get overloaded with code. My response to this is to keep every controller method to less than 5 lines. The Façade design pattern is one very good way of maintaining that separation of concerns. By only injecting one dependency into my controllers I know there is a unified interface for all business logic and controller method can just act as stubs to fill out api documentation. In the CodeAFriend project my Façade only combines two underlying services, the IInterpreterFactory and DbContext, it is reasonable to believe that in future development more services might be required and instead of adding each to a controller they can just be added to the Façade and be lazily initialized if not all users of the Façade need every service for their method calls.



d: **Command**

What properties of an entity are editable when, where, and by whom, in REST apis can very quickly become confusing. Command Query Responsibility Segregation (CQRS) is a DP that emulates the Command DP to help alleviate this confusion. The Command DP as described by the GoF states that each command should handle its own invocation via an Execute method. Another common variation is to have a CommandHandler that takes care of command invocations. CQRS does not specify which or either method to use – it only states that objects retrieved from queries should not be the same objects used for editing persisted data. I've informally experimented with the commandHandler method before so I took a self-invocation approach in CodeAFriend. I declared all my Commands as public child classes of the entity that would be submitting them. In code it looks fine because I separate them into partial classes and name them based on the sort of commands contained within. In a class diagram it looks awful and Software Ideas Modeler didn't even detect that these classes existed. Here is a class view diagram from Visual Studio showing all the Commands I implemented. They all take a DbContext which is supplied to them by the Façade ExecuteCommand method.



7. I have designed, created, and, implemented systems before. My last 5 months have been exactly about designing one that was originally in java and rearchitecting it in C#. In that process I had far more demands that needed to be filled in the moment, so I was unable to step back and design before sitting down to code. CodeAFriend has been an interesting experience for me to have that time beforehand to consider my decisions before making them. I think the most important thing I've learned from it though is that you can't plan without doing and you shouldn't do without some planning. I learned many lessons in my last 2 years of enterprise software engineering and I was able to apply a lot of them to CodeAFriend but with CodeAFriend I was able to make mistakes that wouldn't be considered acceptable risk in a live environment. The curse of agile is that it's all about short term results and it often pushes aside projects that have or could have a lot of value but would take considerable time to be made available. Small planned out side projects like this are a great way to get around that drive to not make mistakes. And mistakes are the best way to learn.