# Architecture

The solution is separated into a seven projects:

* **Sekhmet**  
  General purpose support library not written for this project. Used both at runtime and in the tool-chain used during compilation.
* **GPPG**Support library for the generated parser – written by the developers of the Garden Point Parser Generator. Used at runtime.
* **FamPoly.Utils**Tool created as part of a different project[[1]](#footnote-2). The purpose of the tool is to make a few changes to the code generated by the parser and lexer generator tools to make them fit nicer with the rest of the codebase.
* **VMILLib**A standalone library dealing with the source language. Originally it read and wrote both a textual and binary representation of said language, but as the project progressed the binary format was abandoned.
* **VM**The main virtual machine binary. This is built as a library that can be included in other applications.
* **VM.Debugging**Library implementing most of the interface for a debugger. The debug interpreter and execution stack is part of the main VM library but the Windows Communication Foundation interfaces and classes are implemented here. This library is not completed but part of it is used in the Swapper test mode.
* **VMShell**The current console based driver for our VM. Simple console based program that simply invokes the virtual machine based on the parameters given.

While each of these projects contains code that might be of interest, we will focus on the **VM** library.

The static class *VirtualMachine* (**VirtualMachine.cs**) implements the interface programs wishing to use the VM. This interface is simply the ability to execute a given file in a synchronous and asynchronous manner (no callback is currently specified for the asynchronous case) as well as the ability to attach and detach a debugger. When requesting the execution of a file the VM is automatically initialized, resulting in the creation of a memory manager and the loading our base library, before execution is begun. Out base library is embedded in the binary but is written in the VM’s source language with references to external methods[[2]](#footnote-3).

Execution itself is performed by creating a new instance of *InterpreterThread* (**InterpreterThread.cs**) specifying an object instance and a message handler within defined by the class of that object. Each *InterpreterThread* instance represents a distinct thread and so creation of a new thread is done in the same manner though initiated from within the VM.

The *InterpreterThread* then creates a new interpreter using the currently specified *IInterpreterFactory* (**IInterpreter.cs**). This factory specifies methods for creating new *IInterpreter* (**IInterpreter.cs**) and *IExecutionStack* (**IExecutionStack.cs**) instances. The interpreter itself has a single entry point: **Run**. To ensure that the interpreter can be swapped at more or less any point we have an invariant specifying that only the *InterpreterThread* instance of a given interpreter is allowed to call this entry point. So to ensure that external method calls can call back into the VM a stack of messages stored in the thread is employed. All code has access to the *InterpreterThread* instance it is executing in and can add a new message to its stack – implemented as .NET delegates – allowing it to call either other external method for the interpreter.

A property in the interpreter ensures that it can be made to return when the currently executing instruction is executing. Note that due to the message stack any external call that the interpreter may initiate is actually relegated to be performed by the *InterpreterThread*. An interpreter running our own language can thus be paused almost instantly and any running .NET code will pause when returning from the most recently made call – if a thread is blocked it is simply specified that it should pause.

1. Part of Kim Birkelund’s PhD project. [↑](#footnote-ref-2)
2. The syntax for these external references is part of the language and can be used by anyone. [↑](#footnote-ref-3)