Mango.io

This is intended to serve as a design overview for the Mango.io package. Understanding the concepts described herein should provide one with enough information to extend the IO system, or leverage it in ways that might not seem completely obvious at first glance.

The basic premise behind this package is as follows:

- The central concept is that of a buffer. The buffer acts as a queue (line) where items are removed from the front and new items are added to the back. Buffers are modeled by mango.io.model.IBuffer, and a concrete implementation is provided via mango.io.Buffer.

- Buffers can be written to directly, but a Reader and/or Writer are typically used to read & write formatted data. These readers & writers are bound to a specific buffer: often the same buffer. It's also perfectly legitimate to bind multiple writers to the same buffer; they will all behave serially as one would expect. The same applies to multiple readers on the same buffer. Readers and writers support two styles of IO: put/get, and the C++ iostream style << and >> operators. All such operations can be chained.

- Any class can be made compatible with the reader/writer framework by implementing the IReadable and/or IWritable interfaces. Each of these specifies just a single method.

- Buffers may also be tokenized. This is handy when one is dealing with text input, and/or the content suits a more fluid format than most typical readers support. Tokens are mapped directly onto buffer content (sliced; not allocated), so there is only minor overhead in using them. Tokens can be read and written by readers and writers also, using a more relaxed set of rules than those applied to formatted IO.

- Buffers are sometimes memory-only, in which case there is nothing left to do when a reader (or tokenizer) hits an end-of-buffer condition. Other buffers are themselves bound to a Conduit. When this is the case, a reader will eventually cause the buffer to reload via its associated conduit. Previous buffer content will thus be lost. The same concept is applied to writers, whereby they flush the content of a full buffer to a bound conduit before continuing.

- Conduits provide virtualized access to external content, and represent things like files or Internet connections. They are just a different kind of stream. Conduits are modeled by mango.io.model.IConduit, and implemented via classes FileConduit and SocketConduit. Additional kinds of conduit are easy to construct: one either subclasses mango.io.Conduit, or implements mango.io.model.IConduit. A conduit

reads and writes from/to a buffer in big chunks (typically the entire buffer). Note that buffers happily handle cases where the output cannot keep up with the input. In such cases, unconsumed data remains in the buffer (moved to the head) and less is read from the producer to compensate for the slowdown.

- Because all IO is buffered, you may find that a 'flush' operation is necessary at times. For example, when writing to a random-access file one would flush the output buffer before moving the file-pointer to a different location. You may have this operation performed automatically by using a buffer derivative (FlushBuffer) or, leverage the CompositeWriter class instead. The latter is the recommended approach.

- Additional file-system support is provided through two classes: FileSystem supports operations such as getting and setting the current directory, and FileProxy exposes facilities to manipulate both files and directories.

- Console IO is implemented via Stdio, using FileConduit and appropriate readers and writers. Console IO can be redirected in the traditional (shell) manner.

- The package is intended to be portable between linux and Win32 platforms.

Socket functionality is built upon the gracious provision of socket.d, written by Christopher Miller.