Streams: A Pattern for "Pull-Driven" Processing*

Stephen H. Edwards

Department of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210
Tel: (614) 447–9803
E-mail: edwards@cis.ohio-state.edu

May 15, 1994

Pattern Name: Streams

Problem

This pattern allows the software designer to concentrate on the data flow of a complex piece of software without concern for the techniques individual components will use to distribute the computational burden. One can uniformly compose and reason about abstractions or "objectified" program fragments obeying this pattern, while they may each use very different control strategies and representations internally.

Context

The pattern applies to most imperative languages, including object-oriented and object-based languages. Streams are most effective when the architecture of a software subsystem is best captured by highlighting the data flow within it. Thus, streams work naturally with "pipe and filter" style conceptual models of program operation.

Streams may be more difficult to use within software subsystems that have one or more of the following properties [AAG93, AG92]:

- The data flow within the subsystem is highly dynamic and constantly changing.
- The subsystem is highly interactive, reactive, or event-driven, like a GUI.
- The processing elements ("filters") under consideration are highly parameterized, and these parameter settings change over time.

Forces

The following key issues must be considered when addressing the problem of how to structure software so that composability is maintained while decisions about computational approaches and strategies are deferred:

- What communications mechanisms will be used between components in the subsystem?
  - How will the flow of information between software components be represented in the architecture?
  - How will control flow between the software components be represented in the architecture?

*This work is supported in part by the National Science Foundation, CCR-9111892 and CCR-9311702; by the Advanced Research Projects Agency under ARPA contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714; and by the Graduate School of the Ohio State University.

- Different components within the subsystem may choose very different processing strategies, such as precomputation, partial precomputation and buffering, demand-driven computation, etc. If different components choose different strategies, how will those choices affect component interfaces? Data flow among the components? Control flow?
- How will composability of independent or semi-independent components be achieved? How will the designer be able to reason about compositions?

Solution

Model each component in the subsystem as a *stream* of data objects. An informal description of a stream is just "an ordered sequence of objects, all of the same type, and with a distinguished point representing the current position within the sequence." More formally, we can mathematically model a stream of objects of type `Item` as an ordered pair, where each component is a string of `Item`s. If we call these two string components the '`past`' and '`future`' of a stream `s`, the entire content of a stream `s` is '`s.past * s.future`'[1], and the division between these two substrings represents a cursor pointing to the current position within the stream. This model can then be extended to encompass the notion of "open" versus "closed" streams as shown below, by permitting a stream to have a $\perp$ value that denotes the equivalence class of all "closed" streams.

It is too restrictive, however, to require every class or type obeying the stream pattern to have this abstract state model. All that is really necessary is that a programmer using an object from such a class be able to reason about it *as if* it had the above structure, without getting into any trouble.

To this end, streams will be described here as a "common interface model" with the following definition:

```
common model Stream_Model

    schema context

        parametric context
            type Item

        interpretation context
            math function
                INITIAL_STREAM : math[concrete[Stream]]

    concept schema
        interface

            type family Stream is modeled by
                bottom union (
                    past   : string of math[Item] ,
                    future : string of math[Item]
                )
                exemplar s
                initialization
                    ensures   concrete[s] = INITIAL_STREAM


            operation Read_Item (
                    alters   s : Stream
                    produces x : Item
                )
                requires  s /= bottom  and
```

[1] Here, '`*`' is used as a shorthand for the concatenation operation on mathematical strings.

```
                s.future /= empty_string
        ensures   s /= bottom            and
                  s.past = #s.past * <x>  and
                  <x> * s.future = #s.future

    operation Test_If_Open (
            preserves s : Stream
        ) : boolean
        ensures   Test_If_Open  iff  s /= bottom

    operation Test_If_EOS (
            preserves s : Stream
        ) : boolean
        requires  s /= bottom
        ensures   Test_If_EOS  iff  s.future = empty_string

end Input_Stream_Model
```

The `Input_Stream_Model` CIM defines the pattern for input streams. Output streams are the same, with the `Read_Item` operation replaced by the following `Write_Item` operation:

```
    operation Write_Item (
            alters   s : Stream
            consumes x : Item
        )
        requires  s /= bottom
        ensures   s /= bottom               and
                  s.past   = #s.past * <x>  and
                  s.future = #s.future
```

As one would expect, bidirectional streams simultaneously provide all the features of both input and output streams.

An Example

Tokenizing character data is a simple example of where one might use a stream. In C++:

```
class Token {
   . . . .
};

class TokenStream {
public:
    TokenStream();
    ~TokenStream();

    // The basic operations for all input streams:
    Token   Read_Item ();    // Requires: stream is open, and there's more
    Boolean Test_If_Open (); // Legal any time
    Boolean Test_If_EOS ();  // Requires: stream is open

    // Additional operations for this type of stream:
    void Open (CharStream& seed);
                             // Consumes the seed, returning an
                             // initial value--the seed should NOT be
```

```
                              // shared
    void Close ();            // Requires: stream is open
};
```

Given some `Token` class, a `TokenStream` is an (initially empty) sequence of such `Token`s. The sequence may be "filled" by calling the `Open` method, which takes a `CharStream` (a stream of characters) as input. This character input stream is "consumed," or used up to create the new value of the TokenStream. Then `Read_Item()` and the various tests can be called to retrieve individual `Token`s, one at a time.

Note that one could actually define this class as an abstract base class, and create several alternative implementations. In addition to having different internal representations, these alternatives might differ in how they actually perform the tokenizing. The actual work could be distributed among the `Open`, `Read_Item`, and `Test_If_EOS` in many possible ways. The actual strategy used, however, is completely hidden from the user and does not affect the class interface.

Force Resolution

The pattern balances the various tradeoffs involved by promoting the data flow among software parts to be the primary architectural feature. This makes the resulting architecture transparent to choices about control flow and computational strategies. It also provides a simple model of composition which supports the informal (and even formal) reasoning efforts of designers and implementers.

Specifically, the forces described above are resolved as follows:

- Communications mechanisms: streams become the "channels" through which software components communicate, and are thus the sole communication mechanism.
  - Data flow is explicitly represented in the architecture, by showing the "wiring pattern" of the streams involved.
  - Control flow is not represented at the architectural level at all, and is instead an internal feature of each distinct stream abstraction.
- Because control flow is hidden within the stream abstractions, each distinct type of stream can choose an arbitrary processing strategy for any of the method(s) it supports. All such decisions are completely transparent to the architecture, and thus have no affect on the interfaces between components or the overall flow of data.
- Streams allow for a simple model of composition. The `Open` operation for one stream can take as parameters one or more "input streams", and the `Open` operations then provide the infrastructure for connecting up networks representing entire software subsystems.

  If streams are used as the sole communication mechanism, then one can view `Open` operations as "transformation" steps which convert streams of input data items into a stream of output data items. As long as the nature of this transformation step is unambiguously described, the process of reasoning about complex pipelines of streams is straightforward.

Design Rationale

The idea of a *stream* of homogeneous objects has been in use as a practical programming abstraction for a very long time and is well-known to LISP and functional programmers. A text book description of streams in this context appears in [AS85]. The streams described in [AS85] are composite data structures that are designed to help organize and structure computations on serial collections of data, "in a way that corresponds in spirit to an electrical engineer's concept of a signal-processing system" [AS85, p. 242]. Conceptually, a stream is simply a sequence of data objects. It differs from a regular list in the way it is represented, and in the fact that the sequence might be arbitrarily long. This stream concept is used in many functional programming languages for structuring computations that can be thought of in terms of a data flow model, where data objects flow through computational "stages" or filters.

The longevity and versatility of streams as programming objects leads to the question of whether there is a higher-level abstract *pattern* here, and if so, what problems this pattern addresses and in what contexts.

The problem that streams best address can be uncovered by further examining the "signal-processing" analogy made by Abelson and Sussman. Streams allow one to structure computations on (usually long) sequences of data in an intuitive way using simple data flow ideas, without regard for the details of how the computation will actually be carried out at run-time. Conceptually, a stream denotes an *entire* sequence of data objects, and operations on streams take as input an entire sequence and produce an entire sequence. Because of the operations available on streams, however, one cannot *access* the sequence arbitrarily—usually only the "head" element in the stream can be accessed. This access restriction means that a stream can simultaneously be viewed without conflict as either a channel for transmitting objects one by one, or as the entire sequence of objects that will (eventually) be transmitted across that channel.

One of the most interesting by-products of this duality is that any number of techniques can be used to represent a stream or to implement a stream operation, including any combination of:

- Directly representing stream elements;
- Precomputing and buffering groups of stream elements; or
- Computing elements on demand (delayed evaluation).

Regardless of the implementation complexities involved in achieving effective performance, however, streams with any or all of these representational strategies can be intermixed and composed because of the simple conceptual model that they all obey. Stream abstractions derive much of their power from this combination of interoperability and a uniform external treatment of representations that employ radically different control flow paradigms internally.

Examples of streams abound, even in imperative programming environments. The simplest example is treating any character-oriented device (files, the console, network message connections, etc.) as a "stream" of characters. In addition, streams lend themselves naturally to "pipeline"-style programming, where the main internal components of a program are written as stream filter/transformer operations, and then connected up to form a complete end-to-end filter in a manner very similar to Unix-style command line piping. With this programming style, however, one is not limited to simple character streams—program components can be connected via streams transmitting very sophisticated objects.

Another interesting use of stream-based programming is the construction of stream pipelines where each stage relies exclusively on delayed evaluation to produce results. This style of programming mixes well with pipeline-style software architectures. It is often called "pull-driven" computation, since one requests the next output data object from the last stage of the pipeline, which then "pulls" on its supplier, and so on down the line. Each pipeline stage only computes enough to supply its client, which is similar to the "just in time" supply policy in logistics. Pull-driven computation contrasts with "push-driven" computation, where one "pushes" input into the first stage of the pipeline, which performs its task and pushes its result to the next stage, and so on until the desired answer is produced by the final stage.

A pull-driven pipeline architecture can be used to describe many transformation-based programs, including a compiler. The process of scanning can be rephrased as the process of transforming a stream of characters into a stream of tokens, producing each new token on demand. Similarly, the process of parsing can be viewed as transforming a stream of tokens into a stream of abstract syntax trees, representing the sequence of compilation units, and so on. Many additional examples from the functional programming world are provided in [AS85].

Finally, is the concept of a "stream" really a pattern, or merely a hierarchy of related programming abstractions one would expect to find in a class library? Stream class hierarchies can be found in OOP libraries for Smalltalk, C++, and other languages, but these hierarchies really only capture an organized collection of implementations of particular ADTs derived from a particular stream interface. The notion of "behaving" like a stream is not tied to a particular inheritance hierarchy, however, nor to an object's *entire* interface. As long as some facet of an object, including some subset of its operations and behavior, can be coherently interpreted as "stream-like", there is the potential to use stream-oriented composition techniques when combining it with other objects. For example, iterator objects often behave like streams under controlled circumstances, as do many standard data structures that allow sequential access to their elements, like queues, stacks, or lists. Further, this "pattern" might be discovered at any time, including long after the object definition has been written and its position in a given class library determined. As a result, it is useful to describe this recurring commonality as a pattern, rather than simply considering it a "stock" interface in a library of low-level parts.

References

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. In *Proceedings of the ACM SIGSOFT'93 Symposium on Foundations of Software Engineering*, December 1993.

[AG92] Robert Allen and David Garlan. Towards formalized software architectures. Technical Report CMU-CS-92-163, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1992.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*, chapter 3.4: Streams, pages 242–292. MIT Press, Cambridge, MA, 1985.