Data Flow Oriented Software Design in a FACE Architecture

US Army Aviation FACE™ TIM Paper by:

David Heath
Georgia Tech Research Institute

Trevor Stittleburg
Georgia Tech Research Institute

November, 2015

Table of Contents

Executive Summary

A primary means to creating portable, reusable software is the separation of business logic from data movement (I/O and otherwise). In this document, some object-oriented abstractions are discussed which were used as primitives in the development of components in a FACE system. These abstractions remove data movement concerns from business logic concerns, and can be utilized within FACE components and within the implementation of the TS and I/O interfaces. Their applicability in each of these cases is discussed.

Definitions of Terms

- **FACE Standard**: FACE Technical Standard, Edition 2.1
- **FACE Transport Interfaces**: FACE I/O API and FACE TS API
- **IOSS**: FACE I/O Services Segment
- **PSSS**: Platform Specific Services Segment
- **PSS**: Platform Specific Service
- **MIL-STD-1553**: A standard for a communications bus protocol and accompanying hardware

Motivation and Background

GTRI's work has included development of FACE software in both contexts. In this paper, some design concepts which have found to be useful in FACE development are presented. These design primitives aide in isolating common data movement source code from business logic, which makes integration a simpler process and further extends the effectiveness of the FACE architecture.

A primary driver for this work was the development of a generic I/O service library. GTRI developed a distributed I/O service (IOS) to provide a capability for communicating on a MIL-STD-1553 bus. The I/O service was to be used to test an Army-developed Platform Specific Service (PSS) to control radios. In order to properly test the distributed IOS, the IOSS was deployed into the context of a FACE system. In doing so, the component was designed such that driver specific code was isolated from the rest of the distribution tasks of the I/O Services Segment. In addition, 'stand-in' application components were integrated which exercised the architecture and sent control commands to the PSS. As part of the effort, GTRI searched for design abstractions to help represent FACE components more easily and factor out the common logic that occurs often in FACE integration; namely, that of moving and processing data. The benefits of this effort extended beyond representing mock components into the construction of more complex components.

Data Centric Design

The Transport Services Segment (TSS) enforces a data-centric interface definition for software components which discourages tight coupling of components. The FACE Technical Standard, Edition 2.1 does not standardize message sets used across portable components, but those messages must be documented in the data model. The result is that tight coupling of components via functional interfaces is removed and coupling via message (data) interfaces is made transparent.

The Transport Services (TS) interface is very natural for a 'data flow-oriented' component design. In this way, the TS interface encourages the breaking of complex components into simpler building blocks, especially where those building blocks can be defined as black boxes that handle input data flows and produce output data flows.

The FACE Standard encourages a separation of the concerns of system I/O and business logic. The key concern which the FACE Standard introduces for a developer building a component is the interaction of the component with the TS or I/O interfaces. A developer who is new to the FACE architecture might find these interfaces clumsy or restrictive. Any time the component needs to perform I/O it likely needs to go through these interfaces (depending on segment in which it resides). While these interfaces may be restrictive, they actually enforce better software development practices on the developer. Thus, the developer must separate the concerns of I/O from any calculation that their component performs. These interfaces provide the opportunity to "factor out" the core logic of data movement inside the TS and I/O interfaces.

GTRI developed C++ abstractions simplify the implementation and integration of data flow oriented components and the FACE TS and I/O APIs.

Channel Abstraction

The core concept of this approach is a data movement abstraction named C*hannel*. The Channel is used to represent anything that can accept data for writing and/or read data from another source. A channel can be used to abstract I/O operations or any other data flow oriented operation.

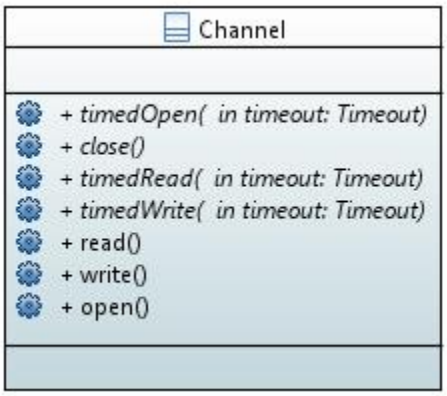

Figure 1. Definition of a Channel.

The Channel abstraction is used to encapsulate reusable TS and I/O interface logic, and can also be used to encapsulate data flow oriented program logic. The source code specific to a particular type of data movement (such as sockets or POSIX message queues) can be separated from the distribution logic inside the TSS and the IOSS. An example of an implementation of a Channel would be a POSIX Message Queue Channel. Note that instead of using a default timeout parameter of zero, methods with timeouts are explicitly defined. A user of a Channel does not need to know what type of underlying transport mechanism is being used to move the data.

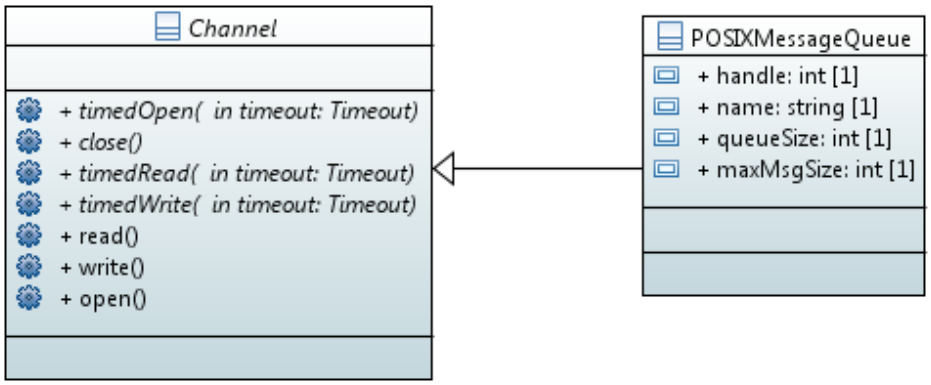

Figure 2. A POSIX Message Queue.

In addition to data movement, a common occurrence GTRI has found in its work is simple, repeated modification of data such as attaching or detaching headers. These operations are often one-to-one with a

Channel operation (reading or writing data). A natural progression of the Channel abstraction is a Refinement, which is an abstraction which uses a Channel and modifies data before it is written or after it is read. An example of a Refinement would be *PrependIOHeader* which prepends an I/O service header to a data buffer.

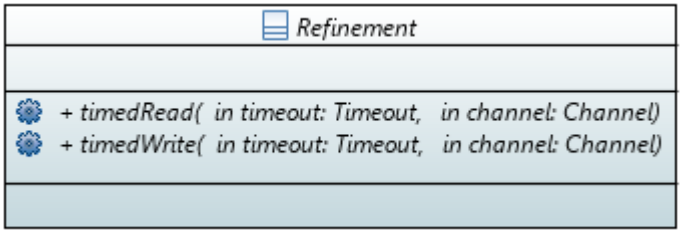

Figure 3. Definition of Refinement

The usefulness of these abstractions become clear when the configurability of the FACE TS and I/O interfaces is considered.

Generic Structured Data for Configuration

GTRI developed a generic configuration resource data structure which different software elements can be configured against. The critical decision for configurability was to develop a uniform data element which represents structured data. This data element can represent exactly one of the following:

- A Boolean value
- An integral number
- A floating point number
- A character string
- An ordered array of other data elements
- A character string labeled set of other data elements

With this in place, different elements can be configured against a data element. Consider the previously discussed POSIX Message Queue channel. The procedure for configuring this Channel involves looking at a configuration data element. It expects the element to be a labeled set. That set should contain an integral element labeled “size”, an integral element labeled “max-length”, and a character string element labeled “name”. The procedure will fetch each of these values and construct a POSIX Message Queue Channel with that information. If any of the configuration data is not structured as expected then the procedure will yield an error message indicating why the configuration failed.

To make this strategy useful, a method of getting configuration data into the system was needed. GTRI developed a utility which uses parsed XML data to structure this configuration data. Therefore, software elements can be configured through XML files. Since the configuration structure is independent of XML, procedures written against this structure will remain useful when the source of configuration data changes. For example, a configuration service could send this structured data directly to a component, rather than sending text-based XML data which is significantly larger and requires parsing.

Thus far, the primary use for this configuration strategy has been for configuring different Channels and refinements, as will be shown in the following section.

I/O Services Segment Design

I/O Services Segment is designed to be portable from one project to another with minimal changes. With the definition of the previous Channel abstractions complete and a strategy for configuring software elements in place, creating a generic I/O Services Segment implementation is relatively straight-forward.

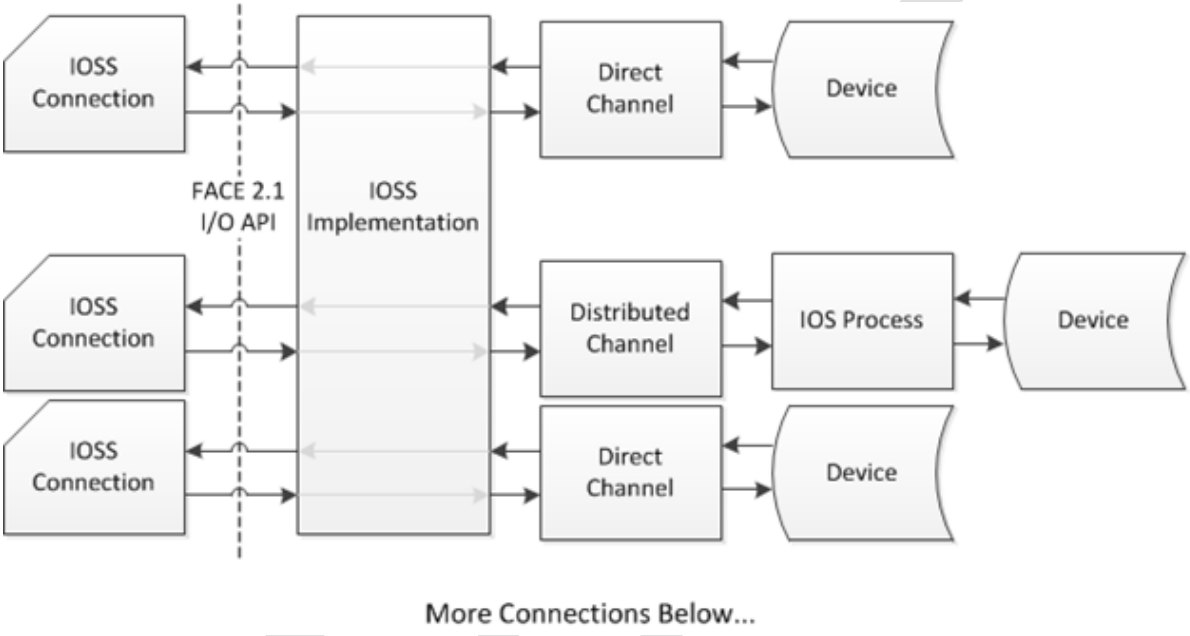

Figure 4. IOSS Implementation using Channels.

The IOSS implementation chosen can be thought of as the manager and interface to a collection of Channels.

As the user calls FACE specified I/O functions, this manager calls corresponding functions on underlying Channels. The structure of this implementation is shown in the figure below.

The IOSS implementation is greatly simplified by this strategy. The only real concerns of the IOSS are to:

- Delegate function calls to the correct Channel.
- Indicate any problems that occur with the correct FACE error code.

The figure above also distinguishes two different possible connection types that can be made: direct and distributed. This distinction is described in some detail in the FACE Standard. In a direct connection, the system is connected directly to the target device through the Channel. In a distributed connection, the system must communicate to the device indirectly through what is labeled as an IOS Process. The IOS Process can control access to the device, even if multiple programs are attempting to use it. In both cases, Channels are used to implement the connection.

The I/O distribution requirements in one project will differ from the I/O distribution requirements in another. Therefore, in order to finish the implementation of the IOSS, the Channels which are managed must be configurable. To provide this configurability, the IOSS provides the ability to add, remove, and change the Channels which correspond to named interfaces. This information is specified in a corresponding XML configuration resource. The following figure is an example configuration resource.

```
<ios type=”array”>
 <connection>
  <connection-name type=”string”>connection1</connection-name>
  <ios-type type=”string”>direct</ios-type>
   <channel>
    <udp>
     <src-ip type=”string”>0.0.0.0</src-ip>
     <src-port type=”int”>1234</src-port>
     <dst-ip type=”string”>192.168.1.1</dst-ip>
     <dst-port type=”int”>5678</dst-port>
     <direction type=”string”>inbound</direction>
    </udp>
   </channel>
  </connection>

<connection>
 <connection-name type=”string”>connection2</connection-name>
  <ios-type type=”string”>distributed</ios-type>
   <channel>
   <posix-mq>
    <size type=”int”>10</size>
    <max-length type=”int”>150</max-length>
    <name type=”string”>/connection2</name>
   </posix-mq >
  </channel>
 </connection>
</ios>
```

Figure 5. Example Configuration Service

When the IOSS is initialized with reference to the configuration file above, two connections will be instantiated. The first is a direct connection to some device. The IOSS communicates with the device via a UDP connection. The second connection is a distributed connection. This means that the established Channel communicates with an IOS Process. In this case, communication takes place over a POSIX Message Queue. Then, the IOS Process is responsible for contacting the device based on the command received from the IOSS.

In most cases, the IOSS should not require recompilation from project to project. Instead, different connections can be chosen through configuration information.

It should be noted that it is not possible to account for every potential Channel types in configuration information. For example, if a new, specific ARINC 429 Channel is required, it cannot be configured without changing the IOSS implementation code. Instead, only common connections (such as UDP, POSIX Message Queues, etc.) are configurable. Still, this still allows configurability in most cases. When a new type of communications bus is required, developers can implement it as a distributed process (compiled separately

from the IOSS). Then, the IOSS can communicate with this new process over one of the common connection types. IOSS recompilation should only be required in the case where a new type of communications bus is required and the connection must be direct.

Transport Services Segment Design

In the case of the Transport Services Segment, a configuration capability can be provided which allows the system resource to be configured for a particular data flow. Using the Channel abstract class allows the data movement concerns of a particular system resource to be separated from the distribution logic that is specific to TS components. The same Channel implementation is used in the I/O Services Segment. As in IOSS, the TSS implementation configures the Channel objects at initialization, and from then on calls on them to distribute data.

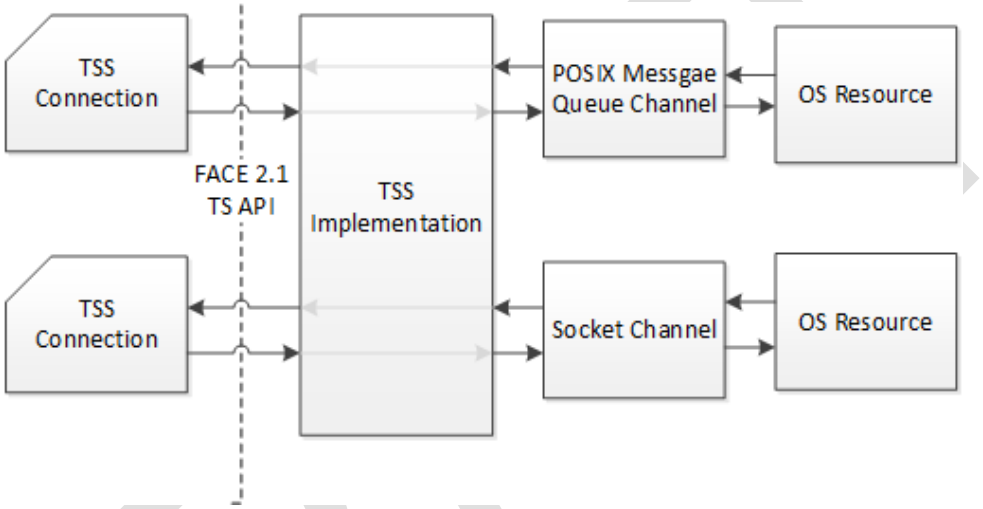

Figure 6. Channels Internal to the TSS.

The Channel abstraction makes it possible to separate the handling of sending and/or receiving data from the implementation of the TS API itself (configuration and distribution), and reuse data movement code between the IOSS and TSS.

Component Design

The use of the Channel concept extends beyond the implementation of the FACE TS and I/O interfaces. The implementation of FACE components is naturally data-centric, especially for portable components as communication between portable components is through the TS interface. This type of design leads to a recurrence of a "big loop" in which data is received from external sources (via the TS or I/O Services interfaces), processed, and then sent to its destination (again, via the TS or I/O Services interfaces). This pattern was also abstracted to isolate business logic from data movement in the context of a FACE component. To simplify this type of component, several additional classes and abstractions are defined.

First, the TS and I/O standard interfaces are accessed by a component using a TSS Access Channel and an IOSS Access Channel, accordingly. The TSS Access Channel simply wraps the TS interface into a simplified Channel interface. This also makes it easier to use the TS interface when combining different types of Channels together. Channels are not limited to I/O operations.

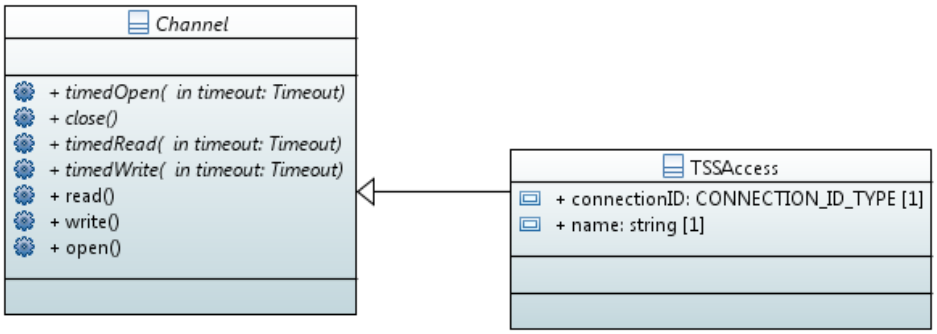

Figure 7. TSS Access Channel

The IOSS Access Channel makes use of refinements to provide the more complex logic necessary for use of the FACE 2.1 I/O API. The Channel which is refined simply wraps the I/O API into a simplified Channel interface. However, this is not sufficient for accessing the I/O API since message payloads require specific headers as specified by the FACE Standard. Therefore, this underlying Channel is refined by two Prepend Header Refinements. The first prepends a bus specific message header to the beginning of the data buffer. The second prepends the I/O Service Message Payload Header to this modified message. The result is that when the user sends a message over an IOSS Access Channel, the FACE specified headers are added to the payload in the correct order before passing the buffer to the IOSS.

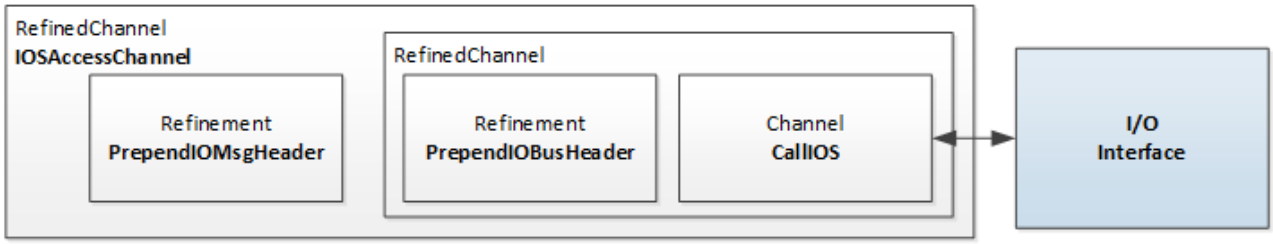

Figure 8. Definition of IOS Access Channel

More Data Flow Primitives

For data flow oriented components, the processing of each piece of data can be isolated from receiving inputs and sending outputs using Channels and some additional abstractions.

First, a Generator isolates the action of retrieving inputs for a logical component. An ErrorStrategy is defined as a method of specifying a handler for errors that occur and can be thought of as a callback function which is used when any error occurs. Finally, abstractions termed ChannelSinks and ChannelSources are used to handle data streams.

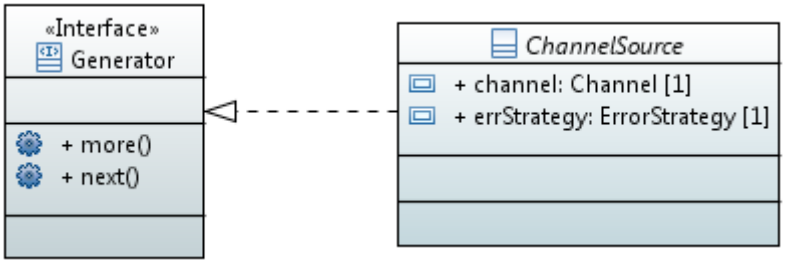

Figure 9. Definitions of Generator and ChannelSource

A ChannelSink is not a class but rather a method which is intended to run as an independent thread, directing a generator's output into a Channel. In C++, its signature appears as follows:

```
void sink(Generator gen, Channel outputChannel,
          const ErrorStrategy& errStrategy);
```

To make these abstractions useful, there needs to be a capability for Generators and Sources/Sinks analogous to the Refinement for Channels described previously. The Transformer abstraction provides this capability. A Transformer emulates a Generator but applies some processing to some other input Generator's data stream and returns other output type which is the result of this processing. In other words, it 'transforms' the input Generator's data stream into a new output data stream.

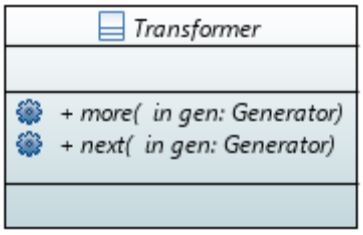

Figure 10. Definition of a Transformer

An example of how these design primitives come together to form a portable component is as follows. Consider a theoretical Flight Manager application which receives waypoints at a fixed interval from some user input source, checks those waypoints against some constraints, and then sends them to a destination. The design of such a component might be laid out as follows.

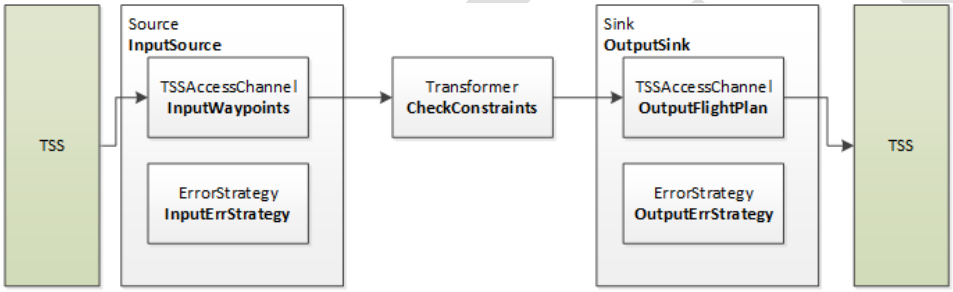

Figure 11. Example Application: A Flight Manager Component

Note the following properties of this design:

- The core logic of the component is isolated from input and output concerns
- Additional processing steps could be chained with the current CheckConstraints step with little effort
- An error strategy must be specified by the implementer
- The source and destination properties TSSAccessChannel can be controlled by configuration data

Conclusions

The intent of this paper was to present some useful design primitives for use in implementing FACE software. These design primitives discussed in this paper have been key in rapid development in GTRI of FACE software. They are presented here as reference material for others looking to implement portable FACE software beyond simply the use of the FACE TS and I/O interfaces. The development of the interfaces discussed above was an iterative process which took time and energy, and it is the hope of the authors that others may look to this work as reference when developing their own FACE systems.

References

The Open Group, Technical Standard for Future Airborne Capability Environment (FACE), Edition 2.1, 2014.

About the Author(s)

David Heath and Trevor Stittleburg are Research Engineers at Georgia Tech Research Institute specializing in FACE software development. David Heath has been on the FACE team since 2014 and Trevor Stittleburg has been with the program since 2012.

About The Open Group FACE™ Consortium

The Open Group Future Airborne Capability Environment (FACE™) Consortium, was formed in 2010 as a government and industry partnership to define an open avionics environment for all military airborne platform types. Today, it is an aviation-focused professional group made up of industry suppliers, customers, academia, and users. The FACE Consortium provides a vendor-neutral forum for industry and government to work together to develop and consolidate the open standards, best practices, guidance documents, and business strategy necessary for acquisition of affordable software systems that promote innovation and rapid integration of portable capabilities across global defense programs.

Further information on FACE Consortium is available at www.opengroup.org/face.

About The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through IT standards. With more than 500 member organizations, The Open Group has a diverse membership that spans all sectors of the IT community – customers, systems and solutions suppliers, tool vendors, integrators, and consultants, as well as academics and researchers – to:

- Capture, understand, and address current and emerging requirements, and establish policies and share best practices
- Facilitate interoperability, develop consensus, and evolve and integrate specifications and open source technologies
- Offer a comprehensive set of services to enhance the operational efficiency of consortia
- Operate the industry's premier certification service

Further information on The Open Group is available at www.opengroup.org.