BrianHetrick.com
Application Note AN-7.1

Interface Technology-Independent Applications

Table of Contents

Goals and Introduction

The user interface is the most volatile aspect of applications. Numerical processing has existed since the 1940s; general data processing and computer networks since the 1950s; word processors and computer-mediated human-to-human interaction since the 1960s; and user-programmed applications such as spreadsheets since the 1970s. None of what computers actually *do* today is new, although some formerly intractable problems (such as route planning) have become tractable as multi-terabyte databases and thousand-MIPS multicore processors enable what are essentially brute-force solutions. What *is* new is the user interface. Originally plug boards, then coding forms and punched cards, then character-cell terminals, then graphical displays, and now web browsers and voice-responsive GUI mobile phones, the user interface technology in use has changed as improvements in technology have made more fluid and approachable interfaces possible.

The lifetime of the typical application is 10 years; then it is discarded. Typically applications are discarded because they cannot be economically adapted to newer user interface technologies. It is not that there is no longer a need for the application's function, or that the underlying technology has become unsupportable. You can run 1960s FORTRAN and COBOL mainframe data center applications on the Android or iOS telephone in your pocket, should you want to: the compilers are available, and a typical mobile phone's processing power, memory, storage, and communications capacity exceeds that of the entire 1960s datacenter. Rather applications are discarded as the need for a new user interface technology overwhelms the value of an in-place solution that cannot use that technology. We still need accounts receivable processing—but not ac-

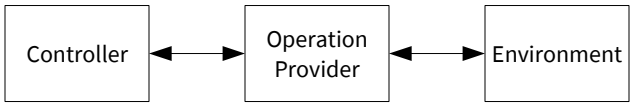

Figure 1. Overall Application Architecture.

counts receivable processing that requires control cards and operators reconciling totals between batch runs of different applications implementing different phases of the process, and that produces thousand-page reports on fanfold green bar paper.

This Application Note addresses the software architectures needed to isolate the application from the user interface technology in use. This architecture permits replacement of not only the particular user interface technology, but the entire style of user interface, be it scriptable (batch), desktop GUI, web, mobile GUI, voice-responsive, or (in the future) anticipatory or thought-responsive. The architectural concepts are of course independent of implementation language and framework; this Application Note uses the .NET framework and C# language for concreteness when discussing implementation techniques and for the example application.

Application Structure

To identify an architecture that will permit application longevity, it is first necessary to determine what an application is. So what is an application?

Basic Structure

An application can be defined as a mechanism to manipulate its environment, typically by applying a set of operations to durable state. The durable state might be a database, a set of files, a cloud repository of some sort, a local or remote user interface, a process being controlled, the velocity of a vehicle, sensor readings, the operations commanded to another application, some combination of these, or other environmental state. The operations upon the durable state might be adding, removing, changing, or observing. The use of the application is then:

- directing the operations to be performed, and
- characterizing the environmental state elements upon which the operations are to be performed.

In response to this use, the application manipulates its environmental durable state in accordance with the operations specified.

Is this a rich enough model of an application? Social media fits this model. Goal-oriented artificial intelligence fits this model. Autonomous vehicle navigation fits this model. Industrial process control fits this model. Numerical simulation fits this model. Web browsers and file managers fit this model. Word processors and spreadsheets fit this model. Voice-controlled home control systems fit this model. Even accounts receivable processing fits this model. Without state (data) from the environment, computation is useless. Without durable effect on the environment, computation is irrelevant. With these things, an application can do whatever is implementable. So yes, this is a rich enough model of an application: an application is an operation provider.

This, the overall architecture of an application is as shown in Figure 1. There is an external controller, which specifies the operation, characterizes the data on which the operation is to be performed, handles any clarification inquiries from the current operation, and receives information on the status of the current operation (typically problems encountered, progress, and either successful or unsuccessful completion). There is an environment, containing durable state to be manipulated in some fashion. In between these is the application: an operation provider, or an engine for environmental manipulation.

Desktop Application Structure

What we call the "user interface" of a desktop application is a type of controller. It allows a human user to specify an operation to be performed on the environment and to observe the change in the environment. This arrangement is shown in Figure 2.

The desktop specialization of the architecture splits the user interface into two pieces: the GUI implementation and the GUI controller. The GUI implementation is technology-dependent: it may be WinForms, WPF, Gtk#, or some other GUI framework or arrangement. The GUI controller, however, need not be dependent on the particular technology in use.

The GUI controller instead deals with a conceptual model of the user interface. It provides properties which serve to inform items on the user display and

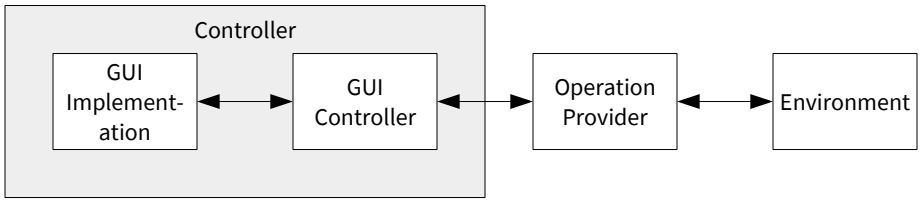

Figure 2. Typical Desktop Application Architecture

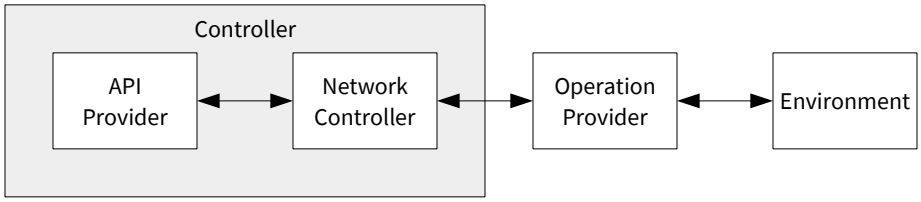

Figure 3. Typical Service/Network Application Architecture

accepts events published by the GUI as commanded actions to perform. User-supplied data may be transferred from the UI to the GUI controller by either properties or through event arguments. The GUI controller provides abstract information: "this situation is relevant" rather than "make this text visible and set the background red." It is the GUI implementation that deals with icons, text, visibility, and color. The GUI controller is also typically natural language-independent: a list of available alternatives would be represented an a list of enumeration members rather than a list of strings. The GUI implementation in turn uses the appropriate technologies and user language and culture information to implement the conceptual model of the UI state presented by the GUI controller.

The arrangement shown in Figure 2 differs from the currently popular model-view-view model (MVVM) architecture by separating the view model into two pieces, the GUI controller and the operation provider. In MVVM, the view model typically directly implements operations on the model (the environment) as commanded by the view (the GUI implementation). While this permits some independence from the GUI implementation, it ties the application (the operation provider) to the environment of synchronously responding to requests from a local UI. Also in MVVM, it is typically convenient to let dependencies on the UI technology in use infect the GUI controller: the awareness of WPF dependency properties, or Unity activation of a UI, are typical examples.

The role of the GUI controller is to permit the underlying user interface technology to be replaced without impacting the implementation of the operation provider —or, alternately, it is the extraction of the operation provider from the GUI controller that permits this.

Both the GUI controller and the operation provider are typically implemented as plain objects. They expose properties and events, and accept events, with no particular dependencies on frameworks. As will be seen in subsequent sections, however, the operation provider typically provides asynchronous operations and the GUI controller typically uses dependency injection to locate the GUI implementation. Therefore both the GUI implementation and GUI controller are event-driven state machines running on the UI thread.

Service Application Structure

Applications do not need to have a "user interface:" services, for example, typically have an API (an Application Program Interface) rather than a UI (User Interface). Despite this, they consist of a controller (commonly providing the API), the operation engine, and the environment upon which the operation engine operates. This arrangement is shown in Figure 3.

Mobile Application Structure

Mobile applications typically have a separate UI application that runs on the mobile device, which controls and observes a server application through its API; typically an HTTP request-response web API is used, although other arrangements can also be used. Other arrangements include the CORBA and SOAP RPC mechanisms, which are currently somewhat popular, and private (dedicated) protocol request-response mechanisms have historically been popular for such things as name resolution, file transfer, and email. This arrangement is shown in Figure 4.

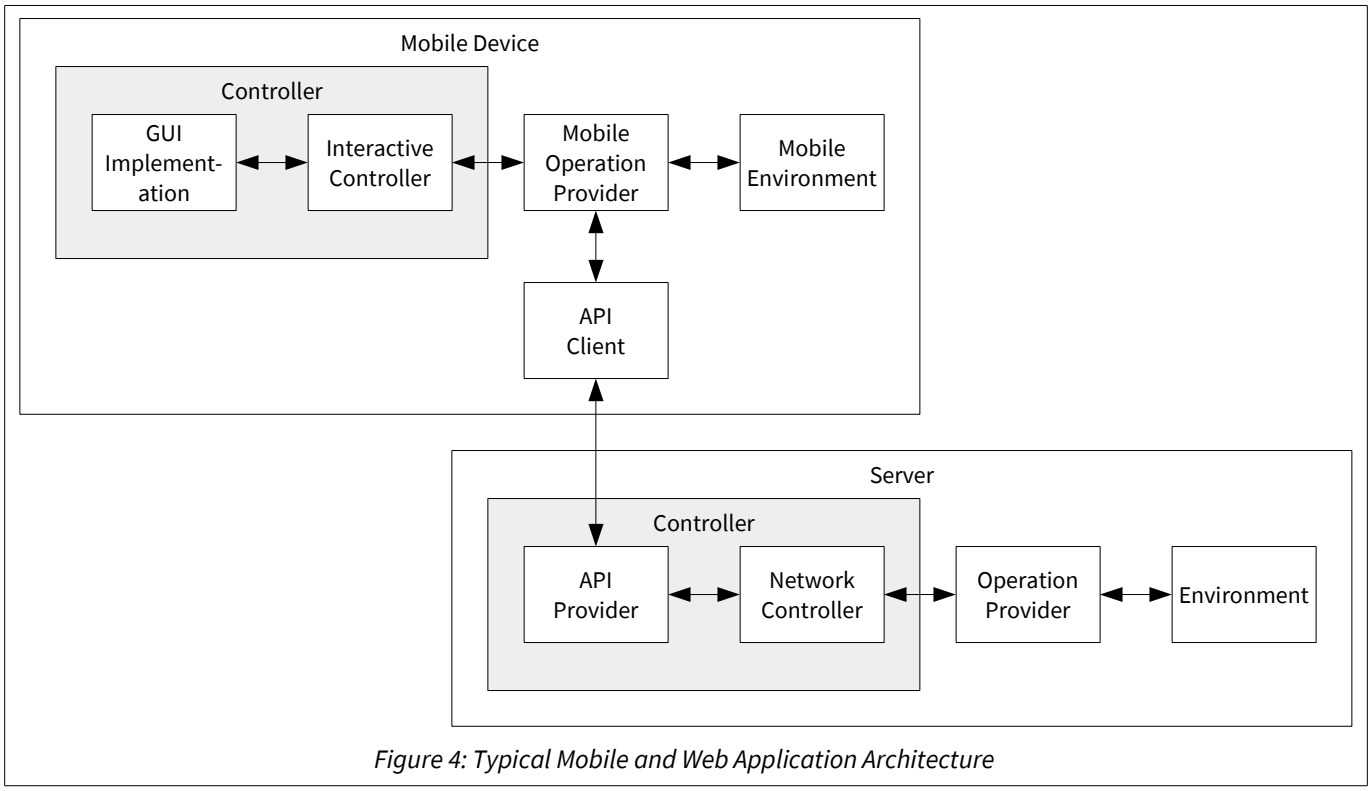

Figure 4: Typical Mobile and Web Application Architecture

Rather than considering the assemblage as a single application, it is typically fruitful to consider a mobile application as a typical GUI application communicating with a service application. Each of the GUI application (the "mobile app") and the service application (the "cloud back-end") can be designed and implemented separately.

Web Application Structure

Current web applications are typically identical in overall architecture to mobile applications, although the "mobile device" is a (possibly desktop) web browser.

Previous web application arrangements, as exemplified by JSP, ASP, the Microsoft Silverlight framework, and others, implemented the mobile device GUI controller and mobile operation as part of the server, comingled with the API Provider. Access to the mobile environment and sometimes the mobile GUI itself was provided by special-purpose controls downloaded to the device web browser. These previous arrangements were essentially attempting to use a web browser as a remote display for a monolithic desktop application. This approach has been abandoned as overly cumbersome to the developer and overly burdensome on the server.

The typical differences between a mobile application and a web application are the implementation language for the mobile/web portion (a compiled language for mobile, JavaScript for web) and the packaging of data transfer between the mobile/web portion and the server portion (typically XML for mobile, JSON for web). Neither of these differences have architectural implications; thus Figure 4 serves for both mobile and web applications.

Structural Separation of Concerns

Current practice embeds UI technology into what this architecture calls the UI controller, and often embeds substantial parts of the operation executor into the same mass of code. Such intertwining of concerns makes the application exquisitely sensitive to its environment. This sensitivity commonly results in both operational problems and extensive difficulty in migration to a new UI paradigm.

Typically application migration is bedeviled by the expectation or requirement that the style of application become the characteristic style of the migration target. It is commonly assumed, at least at the CxO level, that, for example, "migration" of a mainframe green screen environment application to a Windows workstation environment will result in a client-server application

with the clients having native Windows look and feel. Sometimes this expectation is even verbalized. The business goals of such a desired deliverable would not be met by simply rehosting the application to a Windows server and replacing block-mode terminals with Windows terminal emulators. The expense and risk of the "migration" and "rehosting" efforts also differ dramatically, which may contribute to confusion. Although the expense of the migration deliverable is typically stubborn, risk can be minimized by refactoring the mainframe application into the appropriate target interface technology-independent architecture *in situ*, on the mainframe.

Application architecture changes through refactoring are similar to preparatory refactorings for other technology substitutions. Preliminary abstraction and refactoring of implementation are also aids in replacing, for example, navigational databases with relational databases, or a monolithic application with an eventing application or a messaging system, or a proprietary data interchange mechanism with XML over HTTP or other standard mechanism.

Controller/Operation Provider Isolation

As mentioned in the previous section, the distinguishing characteristic of the interface technology-independent architecture is the separation of the controller and the operation provider. This separation permits the controller to be replaced wholesale: the same function can be realized as a scriptable (batch) application, a desktop application with any GUI technology, a mobile application, and a web application without change to the "heart" of the application. Presumably the "heart" of the application, the operations the application performs on its environment, constitutes a majority of the application investment.

Common desktop GUI frameworks are so similar in approach that replacing the UI technology can leave the GUI controller substantially or completely intact. This in turn can be a key enabler of platform portability. Considering how to implement all of a scriptable UI, all of a WinForms, WCF, and Gtk# GUI, and a mobile UI leads one to the correct allocation of functions among the various pieces. It is typically not necessary to actually implement all of these; to partition functionality correctly and implement the currently desired components and interfaces is sufficient.

As the application is essentially an operation engine, it is worthwhile to consider the various aspects of an operation. This then provides guidance as to what is required of an operation engine. The first guidance comes from the basic operation life cycle, considered in the section following.

Operation Basic Life Cycle

In considering the separation of the controller from the operation provider, it is useful to focus on the lifecycle of a typical operation. In general terms, the lifecycle is this:

- First, the need for the operation is recognized. This typically occurs at the user level, but for automatically controlled operations may occur at the goal-seeking or homeostasis level. A control system might recognize a discrepancy between the current environmental state and the desired environmental state, and seek to remove this discrepancy through an action.
- Then, the operation to be performed is planned and described. When there is a human user, the description is typically made by manipulating screen and keyboard, which the GUI implementation translates into text and command events and the GUI controller into an operation description. An automatically controlled operation might directly prepare an operation description.
- Then, the operation description is presented to the operation performer. The operation performer operates on its environment in accordance with the operation description. It might create a database record, change a voltage, send a network message, interrogate a setting, construct and run a query, or a variety of other activities. Typically the operation performer's action will cause further actions in accordance with the design of the system: a database system will update or interrogate the database, a relay closes and changes the power state of some component, a network server receives and acts upon a message. System design boils down to the careful construction and arrangement of components so that the emergent behavior displayed by their combination is useful and desired.

This three-step process is the simplest alternative. There are, however, three commonly encountered additions and variations: operation refinement, operation cancellation, and operation progress and completion reporting. A final variation, applicable only in limited circumstances, is operation undo/redo. Each of these

additions and variations is treated in a section following.

Operation Refinement

Operation refinement can be used when the environment does not support the operation exactly as specified. Changing the name of a file, for example, might fail if a file exists with what is specified as the new name of the file, or if the file to be renamed is marked as read-only. Deleting a database row might fail if the row does not exist or if the row is referenced by other rows using a foreign key. One approach to operation refinement is to simply fail the operation with an indication as to why. This lets the user or controller interpret the indication, and specify other operations to replace or to enable the failed operation.

However, it is sometimes convenient for refinement to occur "on the fly." For example, an operation to delete database rows might present the table names and number of rows in each of the rows to be affected, and seek confirmation of the specified data destruction. A file rename operation might want to confirm, rather than simply deny, the renaming of a read-only file, or to confirm the removal of an existing file having the new name.

Such refinement implies the existence of a secondary, bi-directional communication channel between the controller and the operation performer. This secondary channel permits an anomalous, but recoverable, situation to be reported, and the controller to intervene to rectify the anomaly or to confirm the operation as specified.

Operation Cancellation

Operation cancellation may be required for lengthy or autonomous operations. Autonomous operations are typically encountered in distributed systems and in local asynchronous operations. The user or control system may realize completing the specified operation would be too costly, or destroy valuable assets, or is no longer needed. In such a case, canceling the operation may be appropriate. One approach to operation cancellation is simply not to provide it: small or easily reversible operations, such as inserting a character into a document, typically cannot be canceled. Instead of canceling an operation, the environmental state change can be reversed by applying an inverse operation. Environmental state changes that cannot be reversed (such as writing a disk partition table, or deleting a file from the trash or recycle bin) typically involve an operation refinement confirmation operation.

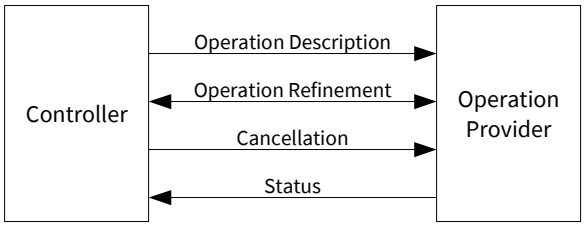

Figure 5. Controller/Operation Provider Communication Channels

Operation cancellation implies the existence of a secondary communications channel between the controller and the operation performer. This secondary communications channel permits the cancellation command to be delivered to the operation performer. For self-directed operation performers, such as Unix daemons or Windows services, this secondary channel may in fact be the only communications channel available: a homeostasis application does not need operation specifications supplied, a file processing service might observe an "input hopper" directory rather than accepting explicit requests, and so on.

Status Reporting

Finally, it is typically required that the operation performer confirm that the operation has taken place, or indicate that the operation failed. Closely related to this requirement is progress reporting: the operation reporting achievement of intermediate milestones along the path to operation completion.

There are therefore four distinct communications channels between the controller and the operation provider, as shown in Figure 5: the channel for the operation description, an optional channel for the operation refinement, a channel for operation cancellation, and a channel for status and completion information.

Operation Undo/Redo

A final consideration in application structuring is operation reversal, or "undo." As will be discussed below, undo functionality is possible only in quite restrictive circumstances.

When applicable, undo functionality is typically provided by constructing a list of the operations performed, and for each operation associating a constructed reversal operation. Undo is then implemented as applying the reversal operation at the tail of the list,

and discarding the operation and reverse operation pair.

Undo and redo (the ability to undo an undo) is a minor variation where the operation and reverse operation pair is removed from the tail of the undo list and moved to the head of a redo list; the redo operation performs the operation at the head of the redo list and moves the operation and reverse operation from the head of the redo list and appends it to the tail of the undo list. Both the undo list and redo list act as stacks, and may be implemented as such rather than as lists. The redo list is discarded when an operation other than undo or redo is performed.

Undo and redo require the environmental durable state to be stable: there cannot be other activities manipulating the state. If the environmental durable state is not stable, there can be no expectation that an inverse operation restores the original state, nor an expectation that reapplying a removed operation restores the state resulting from of the original operation application. For this reason, undo is applicable only to effectively synchronous operations on private data: asynchronous operations (unless serialized) independently manipulate the same environmental state, and the environmental state being accessible by other mechanisms permits independent changes invalidating the saved operations and inverse operations.

When the environmental durable state is not stable, it may be possible to use transactions for a limited undo capability.

Enabling operation reversal requires either the controller or the operation provider to maintain the undo-redo information. Either of these allows operations already applied to be removed, and operations already removed to be reapplied. Where this list is maintained is a matter of choice and application design. If maintained in the controller (the typical arrangement), undo and redo become meta-operations rather than operations: there is no explicit undo operation implemented by the operation performer, rather the controller provides an undo effect by applying the inverse operation. In either case, however, the operation provider is responsible for constructing an inverse operation for each commanded operation. This inverse operation would then either be saved alongside the operation by the operation provider, or be provided to the controller along with the completed status of a commanded operation.

Application Interface Styles

There are currently three overall styles of application interfaces: scriptable, interactive, and network. These are discussed in the following sections.

Scriptable Interfaces

Typically, scriptable interfaces are command line interfaces (CLIs) or more special purpose interfaces such as PowerShell cmdlets or systemd unit files. Scriptable interfaces permit an application to be used as a building block in a larger automated process. They may be considered as a style of interface that decouples in time the specification of an operation from the performance of the operation. The script specifies the operation to be performed, along with environmental data to be supplied later; the execution of the script supplies the missing environmental state (typically file contents) and performs the operation.

Three currently common scriptable interface styles are the Unix- and Linux-style command line, the VMS- and Windows-style command line, and the Windows PowerShell cmdlet. These alternatives have substantial stylistic differences among them, and the overall paradigm of the PowerShell cmdlet is markedly different than that of the command line. For this reason, the command line and PowerShell approaches are discussed separately in the following sections.

Scriptable interfaces have the property that the operation is completely specified before the application is launched. The application therefore does not need to either accept or deny the specification of additional operations while one operation is executing.

As a whole, scriptable interfaces are typically not interactive and operation refinement is not available: instead, a “do it regardless” indicator may be provided. The lack of operation refinement, combined with the lack of need for responsiveness on any particular time scale to the operation request, may be considered the defining characteristic of scriptable interfaces.

Command Line Interfaces

A command line is text specified to a command interpreter which results in the invocation of a program. The program is passed the contents of the command line, which typically includes indications as to the operations the program is to perform and a description of the location or other characteristics of the data upon which the operation is to be performed.

In the VMS- and Windows-style command line, the command consists of a command verb (identifying the overall function), command qualifiers (giving details of the overall function), command parameters (typically identifying the data files to be operated upon), and parameter qualifiers (giving details of operation on a per-parameter basis). Stylistically, command parameters are file names or lists of file names, and qualifiers are identified by a leading slash character (which cannot appear in a file name). Qualifier names are typically multiple character strings, which can be optionally shortened to a non-ambiguous leading substring. Qualifiers may have values, and these values are quite rich: they may be atoms, sub-qualifiers, lists of sub-qualifiers, and so forth. Parameters are ordered, optional parameters can be omitted, qualifiers are unordered, and all qualifiers are optional.

A simple VMS-style command might be:

```
$ PRINT FOO.TXT
```

VMS command verbs are case-insensitive and file names are typically case-insensitive. Windows command verbs are case-insensitive and file names are typically case-preserving. The file system in use may change the case rules of file names in either operating system.

A complex VMS-style command might be:

```
$ PRINT /PRINTER=LPA0: /STOCK=85X11 -
$_ FOO.TXT+[-.TAILS]BAR.TXT /PAGE= -
$_ (HEADER=STANDARD, BODY=LINES=55)
```

(This is intended not to show an actual VMS command, but to exemplify the command style.)

In the Unix- and Linux-style command line, the command consists of a command verb (giving the operation to be performed), command qualifiers (typically called “flags”) some of which may take values, and command parameters (typically names of files to be acted upon). Stylistically, qualifiers are introduced with a hyphen character, qualifier names are typically a single printable character, qualifiers without values may be “bundled” into a single string, and qualifier values are given as a separate token following the qualifier name. A double hyphen may be used where a qualifier is expected to terminate the qualifiers so file names starting with a hyphen can appear on the command line without ambiguity. Extensions to this traditional practice have arisen: for example, multi-character qualifier names introduced by a doubled hyphen, or negated qualifiers introduced by a plus sign rather than a hyphen (minus sign).

A simple Unix-style command might be:

```
$ lpr foo.txt
```

Unix commands and file names are case-sensitive. The Unix command verb is traditionally the name of the program to be invoked.

A complex Unix-style command might be:

```
$ lpr -p lp0: -f 85x11 -H -l 55 \
> foo.txt ../tails/bar.txt
```

(As with the VMS example, this is intended not to show an actual Unix or Linux command, but to exemplify the command style.)

For both VMS- and Unix-style command lines, the effect of the command is that a program is started and the text of the command line is presented to it in some fashion. In VMS, the command line is presented in a series of method invocations and callbacks to the command line interpreter, or optionally as a single string. In Windows, the command line text is presented as a single string. In Unix and Linux, the command line is broken into a token stream, where each token is a quoted string or a maximal sequence of non-white-space characters from the command line. In all cases, however, the uninterpreted command line is either available or reconstructable (although with loss of white space spelling in the Unix and Linux case).

Thus, the role of the controller in a command line interface is to

- parse the command line, where this might be aided by components of the operating system;
- construct the operation descriptions from the contents of the command line, typically one for each file or concatenation of files specified by the command line; and
- present the operation descriptions to the operation provider.

With suitable abstraction, the first of these consists of invoking the proper command line parser engine and passing it the proper command line syntax description. Thus the CLI controller is basically independent of the syntax of the command line implemented.

PowerShell Interfaces

The functionality in which Windows PowerShell wraps an application is essentially one of an operation controller, rather than a shell. The Unix shell, for example, implements a pipeline of files; in contrast, PowerShell implements a pipeline of objects. The PowerShell cmdlet is presented with the following sequence:

- an initialization event,
- a (possibly empty) series of objects, and
- a termination event.

The sequence of objects emitted by the cmdlet, if any, are then passed to the next cmdlet in the pipeline.

The nature of this relationship between the PowerShell environment and the application is made clear by the fact that the cmdlet is not a program. Instead, it is a .NET class which is dynamically loaded by PowerShell and invoked through a standardized interface.

The objects passed to the cmdlet may be objects to be directly acted upon, or may be (for example) names of files containing data to be acted upon. These objects might be directly specified in the command line or might be provided by a prior cmdlet in the object pipeline.

The controller for a PowerShell application is therefore best thought of as a secondary operation provider. This secondary operation provider in turn invokes the application operation provider, typically once for each object in the series of objects presented by PowerShell.

Interactive Interfaces

Interactive interfaces, in contrast to scriptable interfaces, are immediate: typically the application operation starts immediately after the specification of the operation and the environmental state to be affected. Graphical User Interfaces (GUIs) are currently the most common example of interactive interfaces.

Use of the term "GUI" as a synonym for an interactive interface may reflect the currently most common technology, but is misleading. The overall functions of an interactive interface are to provide information to the user and accept direction from the user. This can be done with a wide variety of current technology: teletype, Braille writer, character cell video terminal and keyboard, fixed graphical display with keyboard and mouse, touch-sensitive mobile graphical display, or voice annunciator and voice recognition are all currently in use. Crude thought recognition is currently commercially available; future advances will doubtless include both refined thought recognition and thought reply. A simple "voice in the head" (trivial with physical contact with the scalp, and reputedly current, although not commercial, technology without) may suffice for those with linguistic cognition, but serving the roughly 60% of humanity that has visual, auditory, kinesthetic, or other non-linguistic primary cognitive styles will be an interesting challenge. There is current research in transcranial (non-invasive) brain stimulation; possibly stimulation of specific sensory cortex areas may be a future user interface technology.

Particular examples of interactive interfaces include character cell addressable displays with keyboard and optional mouse, and graphical displays with keyboard and optional mouse. These interfaces use a two-dimensional visual display surface to simultaneously present various possible aspects of the operation and various options for characterizing the environment components to be affected, and controls by which the user can select these aspects and options. Each two-dimensional visual UI framework typically imposes a characteristic appearances and methods of interaction with the user. However, available frameworks are generally similar enough that a single, property abstracted, application-specific GUI controller can be used with any GUI framework, although the GUI implementation remains an application- and framework-specific adapter between the GUI controller and the framework.

Interactive interfaces also include one-dimensional "twenty questions" style interfaces. This type of interface is typically used when only a one-dimensional interface medium, such as a teletype, limited-area braille display, small LCD display, screen reader, or voice command and response, is available. Such one-dimensional interfaces are notably frustrating to the user.

There has been some investigation of exploiting human binaural hearing, auditory depth processing, and ability to extract a single voice out of a multitude of simultaneous voices to produce two- or even three-dimensional soundscape interfaces. Such interfaces would be appealing to the blind and users with more urgent visual tasks than interfacing with the application. Curiously, interfaces of this type do not appear to be commercially offered, despite their obvious appeal, utility, ease of implementation, and low cost. Multi-dimensional auditory interfaces appear limited to fighter

aircraft, where they are typically used for non-verbal auditory weapon and flight status indicators.

Tactile two-dimensional interfaces, such as stimulating the surface of the tongue or of the back, have been investigated as vision replacements for blind users; again, these are not readily commercially available.

Three-dimensional visual interfaces ("virtual reality") exploiting human binocular vision and visual depth processing are under current development using commercially available hardware.

Interactive interfaces must deal with an interactive user. The user acts asynchronously with respect to the application controller. The user might manipulate the user interface while the controller is busy controlling a previously specified operation and reflecting the operation status in the UI. This can confuse the user and possibly change the operation description during the course of the operation, a situation that is typically undesirable. The UI controller must therefore either maintain several operations status contexts, one for each active operation, or disable user interaction with the UI while an operation is performed. Typically this choice is reflected in the style of the UI itself: a "single document interface" (SDI) will typically disable user interaction as a whole during operations, while a "multiple document interface" (MDI) will typically disable user interaction with only the portions of the user interface representing the operations in progress. The UI must, of course, continue responding to non-operational events (window minimization or positioning, or system power notifications, for example) during the operation.

Rather than the technology used for interaction, the primary characteristic of interactive interfaces is that they are interactive: a volitional entity is available for synchronously interacting with the operation. Currently the "volitional entity" (user) is limited to a human being; however, cephalopod researchers are currently extending, and intelligent powered pet prostheses will shortly greatly extend, this scope.

The unconditional availability of operation refinement, and the need for responsiveness on a human time scale (three- and four-digit milliseconds, rather than single digit milliseconds or hours), may be considered the defining characteristic of interactive interfaces.

Network Interfaces

Network interfaces come in a multitude of styles: messaging, request-response, remote method invocation (RMI), and remote procedure call (RPC) are some. These may be partitioned into families based on whether operation refinement is available:

- Operation refinement not available: messaging, request-response, RMI (some)
- Operation refinement available: RMI (some), RPC.

Typically the architectural difference between a request-response protocol (such as REST) and a full RPC protocol (such an SOAP) has to do with the availability of functional refinement. In an RPC protocol, the operation provider can do callbacks to the client, publish events which are delivered to the client, throw exceptions that are caught by the client, and so forth. In a request-response protocol, the operation provider can only provide a response and a success or failure code.

Of course, mere inappropriateness does not significantly hinder technology adoption. The robot network ROS is a publish-subscribe model implemented with point-to-point connections between each publisher and each subscriber of that publisher. Current web applications implement a continuous connection atop REST (a stateless request-response protocol) atop HTTP (a stateless request-response protocol) atop TCP (a stateful connection protocol) atop IP (a stateless datagram protocol). One might imagine that different technologies will arise soon.

It is certainly possible to abstract the details of the network interface away to some extent, and to offer any or all of a series of network interfaces. This is commonly done by database servers, for example, which may offer all of TCP, named pipe or Unix-domain socket, and local shared memory interfaces simultaneously. In such a case, the network controller can instance multiple network implementations simultaneously, just as a GUI controller can instance any of a number of GUI implementations.

A common characteristic of network controllers is that multiple operation requests may occur simultaneously. This characteristic can be dealt with by serializing requests—typically this serialization can be imposed at the network level—but a more common approach is to multi-thread the network controller. The necessity of dealing with multiple simultaneous operation requests is a primary difference between scriptable and interac-

tive controllers on the one hand, and network controllers on the other.

This difference, incidentally, does not extend to the operation provider itself. The operation provider typically must offer asynchronous operations, to avoid stalling the UI thread; and typically must offer multiple simultanous operations, to increase performance when operations are to be performed on multiple elements of the environmental state. This is discussed in a later section of this Application Note.

The need for responsiveness on a machine time scale (milliseconds to seconds) may be considered the defining characteristic of network interfaces.

Multiple Interface Styles

It is sometimes convenient for an application to provide multiple interface styles. The particular case of both a scriptable interface and an interactive interface to the same application is common.

Typically, the most appropriate approach to meeting this need is to construct two executable programs, one implementing each desired interface style. Thus an application might contain an executable that interprets a command line and an executable that offers a GUI. The common "application," the operation performer, would be accessed by both executable programs, typically using a dynamically loadable library approach. Static linking can of course also be used, but this leaves open the possibility of a mismatch among the operation performers bound to the multiple interface providers, a situation which can cause confusion.

It may be desirable to provide multiple interface styles with a single executable. A typical approach to this situation is for the application entry point to perform an initial test to see whether a command line is present. If a command line is present, the scriptable interface is used; if not, the interactive interface is used. If a choice of interactive interfaces is to be offered, configuration information can be used with dependency injection to select the interactive interface to use. Similarly, either configuration information or a preliminary command line scan could select the command line style.

In the case of a PowerShell cmdlet, the program entry-point is not used; instead the PowerShell interface is activated. This unambiguously identifies the interface style to be used.

So an application could offer in a single binary executable all of a Unix-style command line, a VMS-style command line, a PowerShell cmdlet, a WinForms GUI, and a Gtk# GUI. Typically less aggressive interface multiplicity is used: a single GUI technology and a single scriptable interface is commonly regarded as sufficient for a given multi-interface application.

Such generality can come at a cost. On Windows, for example, applications must declare themselves as either console applications or windowed applications. If an application declares itself as a windowed application, it cannot attach to the console (if any) that started it, and so cannot act as a traditional CLI application. A multi-interface style application must therefore declare itself a console application. It is then started attached to the console (if any) that started the application, or a console is created for it if there was no console that started the application. Thus on Windows, a single binary providing both CLI and GUI interfaces flashes a console ("DOS box") upon being started from a windowed application (as by double-clicking the file name in a file explorer). The console is created and appears as the application starts, and disappears and is destroyed shortly thereafter as the application detects the lack of command line parameters and detaches from its console. The creation and destruction of the temporary console can be avoided if distinct executable programs are used for the various interface styles provided.

The question as to whether network interfaces can, or should, be added to an executable program providing a scriptable or an interactive interface is more nuanced. Typically network applications idle, but do not exit, when not active. Scriptable interface applications typically exit when the commanded operations have been performed. Interactive applications typically exit when the user dismisses them. GUI applications might be minimized or their interfaces hidden when the user is not actively interacting with them, but the lack of an exit command would be counter-cultural.

Certainly an application offering an interactive interface can also implement a network interface: as an example, many peer-to-peer file transfer programs do exactly this. However, an implication of this arrangement is that the network availability of the application is dependent on the interactive interface being available; alternately, if the application continues execution in a detached mode, re-establishing a dismissed interactive interface can be problematic. This restriction on the availability of the network interface or of the interactive interface might or might not impede the purpose of the application.

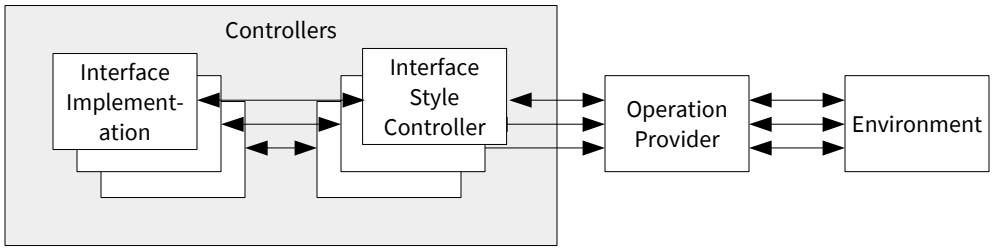

Figure 6. Overall Application Architecture, All Interface Styles

When the network interface of an application must be durable, a typical approach is to use an evanescent (exiting) scriptable or interactive interface application to communicate with a durable (non-exiting) network interface application as operations provided by the network interface application are needed.

Summary

After considering all of scriptable interfaces, interactive interfaces, and network interfaces, it becomes apparent that the overall structure of all of these interfaces consists of two parts: a style-dependent, application-dependent, but interface technology-independent controller which interacts with one or more application-dependent, interface technology-dependent implementation. This arrangement is shown in the overall application architecture diagram of Figure 6.

A multiplicity of style-dependent controllers is optional, but valuable when an application must be both scriptable and interactive and a multiple executable approach cannot be used. The multiplicity of application-dependent, technology-dependent, interface implementations is typically optional, although user acceptability considerations may make such multiplicity desirable. A multiplicity of interface implementation instances is typically required in network interfaces, and a multiplicity of network technology-dependent interfaces may be valuable.

Operation Provider Styles

The separation of the operation controller and the operation provider imposes a number of constraints on, and requires a number of characteristics of, the operation controller. These constraints and characteristics, discussed in previous sections of this Application Note, can be summarized as follows:

- The overall function of the operation provider is to perform a fully-specified operation on fully-specified components of the current environment.
- The operation provider typically must not block the operation controller thread.
- The operation provider is typically to provide multiple simultaneous operations.
- Individual operations can be canceled.
- The operation provider reports operation progress, and operation completion status, to the operation controller.
- The operation provider may need to seek further information about a specific operation.

There are several ways to implement an operation performer which meet all these requirements. Approaches typically fall into one of three categories: constructing an instance of an operation provider for each operation and invoking it asynchronously, constructing an operation provider that provides asynchronous operations, and using an application server. These alternatives are discussed in the sections following.

Instance Per Operation Approach

In the instance per operation approach, the controller has access to a number of operation provider classes. Each operation provider class provides a single type of operation. When an operation is to be performed, the controller constructs an instance of the appropriate class, fills in properties specifying the details of the operation to be performed and the selection criteria for the environmental data to be affected, provide a currently unique object identifying the operation, and (typically) places the operation provider instance on a queue.

Typically, a dedicated set of threads within the application, known as operation execution threads, remove operation provider instances from the queue and execute them. When an operation completes, the operation provider instances is discarded and the operation execution thread returns to accepting operation provider instances from the queue.

Alternatively, each operation provider can be started immediately. This approach may provide an advantage of a millisecond or so when demand for the application's operations is light, but typically results in substantially worse performance and overall throughput when demand is heavy.

The operation provider instances themselves communicate with the controller by publishing asynchronous events. These events identify the operation with which they are associated using the controller-provided object identifying the operation, and provide operation progress and completion information. The controller uses this information to update the appropriate interface.

Asynchronous events publish on one thread and are subscribed to on another thread. Typically event publication is implemented as a method invocation; asynchronous event publication instead uses an inter-synchronization domain mechanism that is provided by the target synchronization domain. UI frameworks typically turn an event publication into an event placed on the UI event queue, which is later dispatched to the appropriate method as the event queue is processed. This asynchronous dispatch of events adds certain complications to the use of events for operation refinement. Solutions to these complications are covered in a later section of this Application Note.

Asynchronous Operation Provider Approach

In the asynchronous operation provider approach, the asynchrony mechanisms are provided by the operation provider rather than by the controller.

Typically the asynchronous operation provider constructs a per-operation context, which is then passed as a parameter to an operation provider-maintained thread for execution. In the asynchronous operation patters, the requester-supplied object identifying the operation is called the "user object." Status reporting and completion notification occur via asynchronous event publication; the event arguments include the user object to associate the event with the operation.

The entire asynchrony mechanism can be provided by an abstract class from which individual operation providers can be derived.

Application Server Approach

An approach conceptually related to the instance per operation approach is typically provided by application servers such as IIS. In this approach, the application server instances a rudimentary controller for each request. The controller in turn instances a single operation provider and synchronously executes it.

The application server approach is typically easier to implement than the other approaches, in that a great deal of the controller is implemented by the application server. However, performance is substantially harder to manage: the application server is typically dispatching a great many applications (both web programs and web services), typically with associated environment creation and destruction. The application server approach can be regarded as a capital equipment management tool, in that it ensures "the server" is kept busy. However, this conflicts, to some degree, with predictability and manageability of performance at the individual application level. Finally, the environment provided by the application server is typically already partitioned into threads and possible address spaces by operation request, both permitting and encouraging synchronous, rather than asynchronous, operation providers.

The use of an application server also introduces a dependency on the application server. Conceiving of an application as available only through the application server can invite blurring the line between the controller and the execution provider. Application servers are not unique in this of course: this blurring is common whenever only a single interface type is considered. However, the use of an application server provides such ubiquitous presence of the application that the blurring can be regarded as acceptable and become structural: and then instead of having to abandon the application when card readers and thousand-page reports go out of style, one has to abandon the application when application servers go out of style.

Summary

In the instance per operation style, typically the thread creating the executor instance and the thread starting (and so presumably controlling) the operation are the same. In the asynchronous operation provider case, the thread starting the operation only needs access to the operation provider instance, and so need not be the thread that actually created the operation provider. This leaves open the choice of the synchronization domain used for asynchronous events: are asynchronous events to be published on a thread compatible with that of the operation provider creator, or the operation requester? Typically the latter is chosen for parallelism with the instance per operation style.

In the application server style, much of the thread and state management of the controller is supplied by the application server. This lessens development expense of the application, but can increase the variability of performance. It also serves as an attractive nuisance, in that it encourages laxity in the controller/operation provider differentiation and in the operation provider implementation.

The choice among the styles is to some degree a matter of taste. When a small number of operations are to be provided, typically the asynchronous operation provider approach is regarded as cleaner and somewhat less prone to error. When a large number of operations are to be provided, placing all operation providers in a single class (to obtain a single queue of pending operations) becomes unwieldy. Balancing resource consumption among the different operations leads to the instance per operation design, with the controller class maintaining the queue of pending operations and the set of execution threads. When the application is one of a suite, which are to be managed as a whole, the introduction of an application server can provide both a management tool and a development cost savings, although with some caveats and risks.

Implementation Techniques

This section deals with common implementation techniques used to implement applications in accordance with the interface technology-independent architecture.

Dependency Injection

An interface style controller will typically be written without knowledge of the interface implementation to be used. This means the interface style controller must discover the interface implementation to use at run time. This information can be provided through configuration information or through packaging. In the former case, the interface style controller can load the assembly containing the interface implementation and use reflection to instance the interface implementation or implementations to use. In the latter case, it is typical to use a dependency injection framework to satisfy requests for an implementation of the interface describing the interface implementation.

The Managed Extensibility Framework (MEF), now part of the .NET framework in the `System.ComponentModel.Composition` namespace, makes this relatively convenient. The MEF can be instructed to form a catalog of the types available in all the dynamic link libraries in a directory. Types intended to fulfill dependencies are marked with the `ExportAttribute` attribute naming the `interface` type they implement. The controller can then use constructor parameter or property injection by simply requiring the appropriate implementation in constructors marked with the `ImportingConstructorAttribute` attribute and properties marked with the `ImportAttribute` attribute, or by using the `GetExport<T>` method of the composition container.

The MEF leverages the packaging of the application: if, for example, a WinForms GUI dynamic link library is present alongside the interactive controller, MEF will supply the linkage between the `interface` requested and the implementing class.

MEF is not unique in supporting this technique. Other dependency frameworks exist for .NET, and they too can be used in much this fashion.

GUI Framework Application Structure Evasion

GUI frameworks typically support the development of applications using the framework, rather than support the addition of GUIs to an existing application. Part of this support is providing an overall structure to the application. Using this structure, unfortunately, ties the application to the GUI framework rather thoroughly.

It is therefore necessary to evade the application structure imposed by the GUI framework. This evasion is done in the interactive interface implementation, although the evasion is enabled by the interactive interface controller—which is typically the "program" part of the assemblage of code that is the application.

WinForms, for example, has the convention that the main program of an application consists of the following:

```
static void Main ()
{
    Application.EnableVisualStyles ();
    Application.
        SetCompatibleTextRenderingDefault
            (false);
    Application.Run (new Form1 ());
}
```

where `Form1` is the name of the main or startup form. `Application.Run` starts the WinForms event loop. WinForms also typically shows a new form in response to the `Show` or `ShowModal` methods on the form itself;

```
using (ApplicationCatalog catalog = new ApplicationCatalog ())
using (_container = new CompositionContainer (catalog))
{
    try
    {
        IApplicationLifetimeManager lifetimeManager =
            GetInstance<IApplicationLifetimeManager> ();
        lifetimeManager.StartApplication ();

        using (IMainViewModel mainModel = GetInstance<IMainViewModel> ())
        {
            mainModel.Start ();
        }

        lifetimeManager.StopApplication ();
    }
    catch (Exception exc)
    {
        _logger.Fatal ("Unhandled exception.", exc);
    }
}
```

Figure 7. Typical Interactive Interface Controller Main Program

these method calls are typically performed by other forms.

WPF has the convention that the main program of an application consists of XAML containing:

```
<Application
  x:Class="Bar.App"
  StartupUri="MainWindow.xaml">
</Application>
```

where `Bar` is the name of the application and `MainWindow` is the name of the main or startup window.

In applications using the interface technology-independent architecture, or sometimes even the simple MVVM architecture, interface controllers (view models) are typically invoked rather than interface implementations (views). This then requires the application to subvert the form- or window-centric structure assumed by the GUI framework. The main program then becomes something like the program shown in Figure 7. The GUI framework-dependent application lifetime manager configures the windowing system; the technology-independent view model starts the GUI framework-dependent view; the main view starts the windowing system; the view models start other view models which in turn start their views; the other views show themselves; and, at application exit, the application lifetime manager shuts down the windowing system, if needed.

Synchronous Event Publication

Event publication is a typical strategy for decoupling the producer of a notification from the consumer of the notification. The producer exposes the subscription mechanism for the event; the consumer registers interest in the event; and upon the event occurring, the producer invokes the registered consumers and passes them a description of the event.

Event publication is a convenient way to share user actions between an interactive interface implementation and an interactive interface controller. A variation, asynchronous event publication, is a convenient way to share progress and completion information between an operation provider and an interface controller. Asynchronous event publication is discussed in a later section of this Application Note.

The C# language implements events through a construct called delegates. This construct permits a number of targets to be invoked sequentially by what appears to be a single method invocation. Unfortunately, the defined semantics of an event are not implemented by a simple delegate invocation: in particular, a delegate target throwing an exception prevents the invoca-

tion of subsequent delegate targets. Also there is question as to whether an exception thrown by an event subscriber should be propagated to the event publisher: what is the event publisher supposed to do with the exception?

Typically event publication therefore determines the target list of the event subscription delegate, then iterates through invoking the targets while catching and suppressing exceptions thrown by the targets.

When this event publication activity occurs synchronously with the event publication demand, the event publication is synchronous. This is the typical case. One consequent of synchronous event publication is that all subscribers have been invoked by the time control returns to the event publisher. This then makes the event arguments, an object passed to the event subscribers is turn, available as a communication channel between the subscribers and the publisher. This mechanism can then be used, for example, to warn of an impending state transition and let the subscribers cancel the transition.

Property Binding

A technique closely related to, and dependent upon, event publication is property binding. In property binding, the value of a property in one object is bound to a property of a different object: changes to the source property value result in an assignment to the target property. These bindings may be one-way (source to target) or two-way (either to the other). This copying is typically implemented by a separate binding object. The binding object can be provided by the GUI framework in use, or can be constructed rather simply in other contexts. Bindings may also include value formatting and type conversion functionality; if unavailable in the binding object itself, these can be provided by intermediate properties on one of the two objects participating in the binding, or by binding to each side of a separate conversion object.

The binding object subscribes to property change notifications from the source object, and when the appropriate property changes, retrieves the source property value and assigns it to the target property value.

Property bindings are a convenient way to share data between an interface implementation and an interface controller.

There are three property change notification mechanisms used in .NET. The .NET 1.0 mechanism is to provide a property changed event for each property: so a `Name` property would have a corresponding `NameChanged` event which would be published when the `Name` property changes. The `EventHandlerList` class is responsive to the resulting need to define a large number of events, most with no subscribers, without maintaining a large number of (empty) delegates.

This cumbersome mechanism was deprecated in .NET 2.0, which replaced the multitude of property-specific events with a single `PropertyChanged` event. The event arguments to the `PropertyChanged` event contain the name of the property for which the value changed.

WPF introduced a different eventing mechanism associated with dependency properties, which replaced per-instance properties with per-class properties; this allowed one type to provide extension properties to other types. Dependency properties and their corresponding routed events and class static handlers are typically unused outside of WPF.

For general use, the .NET 2.0 arrangement is recommended. WinForms controls typically must also implement the .NET 1.0 arrangement, and WPF controls typically must also implement the WPF arrangement.

The .NET framework provides the `INotifyPropertyChanging` and `INotifyPropertyChanged` interfaces, which define pre-change and post-change property change events. Neither of these provides for transition cancellation. The semantics of the Java `VetoableChangeListener` interface, which allows for constrained properties—where observers can cancel a proposed property change—must be constructed explicitly if desired. In particular, throwing an exception when presented with a property changing event does not register an objection to the property change: it merely indicates an error in the subscriber, an error the publisher can and typically must ignore.

Producer-Consumer Queueing

Synchronization among threads is a difficult proposition in general. The general theory has been elucidated in many places, with CAR Hoare's *Communicating Sequential Processes* and Edsger Dijkstra's *Cooperating Sequential Processes* being generative in the field.

Fortunately, a number of common approaches have historically been used to encapsulate the general complexity of thread synchronization. One of these, the

asynchronous operation provider, is assumed in the overall interface technology-independent architecture described previously in this Application Note. Another encapsulation is the producer-consumer queue, discussed in this section.

In a producer-consumer queue, one or more producers produce objects which are consumed by one or more consumers. When used with controllers and operation providers, the objects typically represent operations to be performed. The queue serves as the single point of contact between the producers and the consumers: producers produce objects as they wish, and consumers consume objects as they can.

The producer-consumer queue is, in essence, a simple queue with access points guarded by a mutex: at any time, the queue may be unaccessed, accessed for queueing an object, or accessed for dequeueing an object, exclusively. There are lock-free designs of the producer-consumer queue, but this simple mutex description will suffice. Consumers are typically suspended when there are no objects for them to consume.

There are two possible complications with the producer-consumer queue. The first is that the queue may have a maximum size; the second is that production may have stopped. In the first case, a maximum queue size, producers must be either suspended or notified that an attempt to queue an item has failed. In the second case, consumers must be notified that their attempt to dequeue an item has failed. The manner in which these notifications are arranged is typically the major design difference among producer-consumer queue designs.

The .NET 4.5 framework includes a BlockingCollection type that implements producer-consumer data patterns. The pattern is also easy to implement in previous .NET versions: typically a mutex (lock) controls access to a queue and an event. The event is set when there are elements in the queue and reset when not. The typical enqueue pattern is to obtain the mutex, enqueue an object, and set the event. The typical dequeue pattern is to wain on the event, obtain the mutex, and either clear the event (if the queue is empty) or dequeue an object (if the queue is not empty). The condition of the queue being permanently empty is typically handled by a queue producer count; the count being zero requires the event to be set so consumers can abort their operations.

Asynchronous Event Publication

The asynchronous operation provider must inform the interface controller of operation progress and operation completion. As the interface controller is on an independent thread, and this thread may be a UI thread using event dispatching for asynchrony management, a technique called asynchronous event publication must be used.

Asynchronous event publication is handled in the .NET framework with the help of the `AsyncOperation` class, of which instances are produced by the `AsyncOperationManager` class. The `AsyncOperationManager` class is a static holder for a GUI framework-specific `SynchronizationContext` instance, with a default (free-threading) synchronization context when a UI framework is not in use.

The method of use of the `AsyncOperation` object is to use the `Post` method to schedule the event publication, rather than to directly publish the event. The Post method either directly calls the supplied event publication delegate (in the free-threading situation) or places the event publication delegate on the UI event queue (in the UI situation). The upshot is that the event publication takes place on the UI thread, despite having been scheduled by a non-UI thread.

The construction of the `AsyncOperation` instance must take place at one of two times. An operation provider-wide `AsyncOperation` instance can be constructed along with the operation provider itself; this implies that the operation provider must be constructed on the UI thread. Alternatively, the `AsyncOperation` instance can be associated with an individual operation, and constructed when the operation is requested; this implies that the operation must be requested by the UI thread, which is the typical case.

If `AsyncOperation` instances are created on a per-operation basis, receivers of asynchronous events must be prepared to receive events on the "wrong" thread. This can be handled even without synchronization with the UI thread by checking the "user state" object identifying the operation. If the "user state" object is one identifying an operation requested by the object, the thread is correct; if not, the event should be ignored, and no synchronization is needed.

Queued Event Publication

Synchronous event publication invokes event subscribers as events are published. One implication of

this is that in event-heavy designs, numerous events may be in-publication at one time: whenever an event subscriber publishes an event, publication of that event occurs immediately, during the publication of the original event. This has two effects: receipt of the original event is delayed while the subsidiary events are published and received, and the run-time stack must hold the stack frames of all simultaneously active event subscribers (and multiple instances of the event publication helper).

It may be desirable to finish publication of one event before starting the publication of events resulting from the original event publication. This is termed queueing the event publications.

Queued event publication can be handled entirely within the event publication helper class. The event publication helper, instead of simply expanding the delegate and iterating through the invocation list, first located a thread-dependent queue of pending event publications. If the queue indicates event publication is currently occurring on the thread, the helper queues clones of the event description objects and returns. If the queue indicates event publication is not currently occurring on the thread, the helper sets the indicator and starts publishing the event in hand. When the current event is published, the helper removes the first event on the queue and publishes it. When the queue empties, the helper resets the indicator and returns.

The effect of event queuing is two-fold: it delivers events sequentially, in the order in which they were published, and it trades heap storage (the clones of the event description objects) for stack storage (method stack frames). Whether the storage trade-off is beneficial depends on the shape of the eventing tree: deep, narrow trees typically require less storage when queued, while short, broad trees typically require less storage when not queued.

The ordering of event delivery with queued events differs from that of synchronous events. If one object subscribes to several events in an event chain, then with synchronous events the timing of event delivery depends upon the receiving object's position in the various event subscriber lists. The subscriber may receive delivery of events out of order. With queued events, each subscriber receives events in the order in which they were published. While this is typically desirable, it is an observable behavior change from synchronous event publication.

Queued events are also asynchronously delivered with respect to the publisher: upon return from the event publication helper, there is no guarantee that the event has been delivered to subscribers, only the guarantee that the event will be delivered to subscribers. Thus it cannot be used for eventing patterns where the subscribers provide information to the publisher. This limitation can be worked around to some extent by each event having an associated delivery event, by which the publisher is notified that event delivery has occurred. The delivery event pattern typically depends on the event publisher already using an event-driven architecture to maintain a state machine.

Example

The .ZIP file containing this Application Note also contains a Visual Studio solution for an application exemplifying the interface technology-independent architecture. The application is a simple file splitting program: a specified list of files is split into 1 MB segments, each named with the original file name with a two-digit segment number appended. All generated segments are placed in a designated directory.

The solution is split into six directories, five of which each build one assembly; three of the assemblies correspond to the three non-environment components of the architecture. The overall implementation uses log4net as its logger and MEF as its dependency injection framework.

The Common project contains a general utility library. This library contains interfaces for views and view models (interactive interface implementations and interactive interface managers), abstract classes for view models and asynchronous operation providers, helper classes for both synchronous and queued event publication, and so forth.

The Common.WinForms project contains an implementation of an application lifetime manager for a WinForms interface implementation. The lifetime manager is never explicitly referenced: it is loaded through MEF.

The Interactive Controller project contains an interactive controller for the application. This includes a main program and a main view model; the program locates the view model through MEF. The Interactive Controller has no dependencies on any windowing system, although the executable it produces it must be marked as a windowing program rather than a console pro-

gram. The main program has no dependencies on the main view model: the main program uses MEF to locate both the application lifetime manager (which is arranged to be WinForms compatible) and the main interface controller (which in turn uses MEF to find the interface implementation, which is arranged to use WinForms).

The WinForms Interface project contains the WinForms implementation of a GUI for the application. Although the Interactive Controller contains directory functions, directory structure and directory contents (to select both the files to be split and the directory into which segments are to be placed) are produced directly by the WinForms interactive interface implementation. This is possible because the interface implementation and interface controller execute in the same environment. With small modifications, the Interactive Controller can be used also with a remote (mobile) interface, where these directory functions would find use.

The WinForms interface also declares a dependency on the Common.Winforms assembly, although it in fact is not so dependent. This declaration is to assure the WinForms appropriate common library extensions are available at run time.

Linkage between the interactive interface implementation and interactive interface controller is through PropertyChanged notifications (both ways), a SplitRequested event (implementation to controller), a SplitProgressChanged event (controller to implementation), and a SplitCompleted event (controller to implementation). The properties and events are defined in the interfaces ISplitView and ISplitViewModel, which are defined in the Interactive Controller project. Note that the interactive interface controller does not depend on the interactive interface implementation; instead the implementation depends upon the controller.

Linkage between the interactive interface controller and the operation provider is through direct calls into the operation provider to request an operation, and asynchronous events from the operation provider to the interface controller. This is the well-known asynchronous operation pattern.

After building the solution, the bin folder of the WinForms Interface project contains the appropriate executable program and dynamic link libraries—this despite the executable program being produced by another project. While somewhat curious, this is how the dependencies work out for applications structured with this architecture.

The packaging into assemblies used by the example project is for illustrative purposes, and would typically not be used for a deliverable project. A separate solution (or project), using links to all source files, would produce a single executable with all components.

Further Information

This Application Note and a Visual Studio solution containing all source code for the sample application are contained in a .ZIP file at: http://brianhetrick.com/an/AN-7.1.ZIP .

The Visual Studio 2013 Community Edition software used to produce the sample application is available for no charge at https://www.visualstudio.com/en-us/news/releasenotes/vs2013-community-vs .

The LibreOffice office suite used to create this Application Note itself is available at http://www.libreoffice.-org.

This Application Note uses the Book Antiqua font for body text and the Source Sans Pro fonts for headings, headers, and footers. The Book Antiqua font can be licensed from Monotype at http://catalog.monotype.-com/family/monotype/book-antiqua and is included in several of Microsoft's products. The Source Sans Pro font is available at https://adobe-fonts.github.io/-source-sans-pro/.

Disclaimer

The author believes all content of this Application Note and any software and design it describes is a straightforward application of existing well-known concepts and techniques and is evident to anyone versed in the state of the art. Despite this, the author makes no claim or warranty of any kind that any item developed based on this Application Note, the software it describes, the designs it describes, or any portion of any or all, will not infringe any copyright, patent, trade secret or other intellectual property right of any person or entity in any country.

Copyright and Grant of License