Interaction Patterns for Communicating Processes

Ted Faison
Faison Computing Inc.
5 Santa Comba, Irvine, CA 92606-8896
ted.faison@computer.org

1 Abstract

Interaction dynamics describe the way entities relate and communicate with one another. Just as people interact in a variety of ways, communicating processes, such as software components, can also interact in many and highly complex ways. The software world is filled with examples of systems utilizing processes, objects, components, device drivers and entities by other names -- interconnected in different ways and with different purposes. But in the seeming chaos there is order. Interactions between communicating entities have many basic properties which can be isolated, described and used across many application domains and in many situations. This paper identifies some fundamental patterns that apply to communicating processes in general and software components in particular. The patterns can be used alone or in combinations, allowing more complex patterns to be built out of simpler ones. The success of components as a software technology is predicated on reuse and ease of reuse. In software component terms, ease of reuse is determined by how easily components can be interconnected, which depends solely on their interfaces and the interactions supported by those interfaces. By recognizing the existence of standard interaction patterns, and applying them to interfaces, components should be easier to interconnect, and therefore reuse.

2 Introduction

Nothing in the universe lives in isolation. Everything interacts with everything else. Fortunately, software engineers don't have to answer the philosophical questions of *why* things interact. Their greatest two concerns are how a software system interacts with its environment, and how the parts of the system interact with each other. Underlying these concerns is the recognition that interactions are very important. Interactions are so important that they can be considered semantic entities in their own right. Interactions are based on communication, and communication is in general a highly dynamic process that is difficult to partition and categorize.

Software systems are societies of communicating and cooperating processes. Process interactions are similar to conversations between people. The expression “Interaction Dynamics” is used in this paper to describe the ways software processes interact with each other over time, with particular interest in the following areas:

1. Roles: Which process is giving information to the other? The word *information* is used here to denote both commands and data. One process might assume the role of Commander or Data Provider with respect to the other.

2. Control: Which process is in charge of the interaction? Which one is responsible for initiating it? Can the interaction be aborted? If so, by which process? Is one of the processes responsible for monitoring the progress of the interaction? Which process decides when the interaction terminates?

3. Timing: When sending messages, can the sender wait forever for a response? Can the sender continue with other work while waiting for a response? Does the sender require a response within a certain time frame? Can the receiver accept other messages while processing a prior one?

4. Flow: Is information sent in a single exchange, or it is broken down into an iteration of smaller ones? If an iterative flow is used, how is the end of the iteration signaled?

Because this paper is primarily devoted to the discussion of patterns, a detailed description of all of the issues above cannot be presented here. Most of what follows will center on some of the fundamental patterns that apply to the 4 areas listed above. Each pattern is described briefly in a Context paragraph. Because interaction patterns are rather basic and not tied to a specific type of software system, the Contexts are necessarily somewhat abstract. To illustrate how the patterns capture real world situations, numerous examples are shown. Because interaction patterns often apply to generic communicating parties, and not just software components, in many cases the examples are also drawn from everyday life experiences.

3 Interaction Dynamics and the Patterns Movement

Much of the current work on software patterns centers on the structure, relationship and organization of objects. The emphasis is on characterizing what objects do and what structure they have, especially in relationship to one another. This inclination towards function and structure is possibly due to the influence of Alexander's work twenty years ago [Alexander77] [Alexander79], regarding the use of patterns in architecture.

But software systems are more than aggregations of objects with a purpose. They are more like a group of people that accomplish a task by conversing, or exchanging messages, with one another. Studying the structure of the conversations can be useful, because it reveals that there are patterns in the way people and software processes talk. There are implied assumptions, expectations, understandings and goals that drive entities to communicate. What is important is how the entities carry out a conversation or transaction: who starts it, who ends it, who gives information to the other, whether both parties can talk at the same time, what the goals are, etc. Whether a system uses a particular interaction to implement a Visitor pattern, an Adapter, or no pattern is a consideration that comes 'a posteriori' with regards to the interaction itself. Structural and

behavioral patterns are often layered on top of interaction patterns. Discovering and understanding the patterns of these dynamics may help simplify and standardize software component connections and interfaces.

A significant amount of research has been directed to the study of human interaction communication [Kendon90], Human-Computer Interface interactions [Pinhannez+97] and process-to-process communication [Liu+96]. Some patterns have started to emerge from this work, but most do not apply to software component or concurrent process interactions. Although process interfaces have been studied for some time [Andrews91] [Frølund+94] [Fowler97], they haven't been raised yet to a status level deserving patterns. Software component interactions can draw on patterns that apply to communication in general, including conversations between people and interactions between people and machines. These patterns can be applied to concurrent processes in general, transcending the software domain completely. Some patterns that relate to communicating concurrent processes have been described [Schmidt+95] [Schmidt+96] [Shaw96], but the emphasis again is on the relationship between the processes and not their interaction.

4 The Notation

Although many languages exist that deal with communicating processes, such as Lotos [van Eijk+89], CCS [Milner89], CSP [Hoare85] and SDL [Ellsberger+97], in this paper I'll use a simplified notation based on the Unified Modeling Language Interaction Diagram [Fowler+97]. What is omitted from the diagrams are object-lifetime designators, which are boxes drawn over the vertical object timelines, denoting when objects come into existence and when they die. In the diagrams, the vertical lines will indicate processes or software components. As usual, time increases in the downward direction. An addition to the standard notation is the || symbol, designating an asynchronous operation. The sender of a command, the receiver or both can be asynchronous. When the symbol is not used, an interaction is implicitly synchronous. Examples are shown Figure 1.

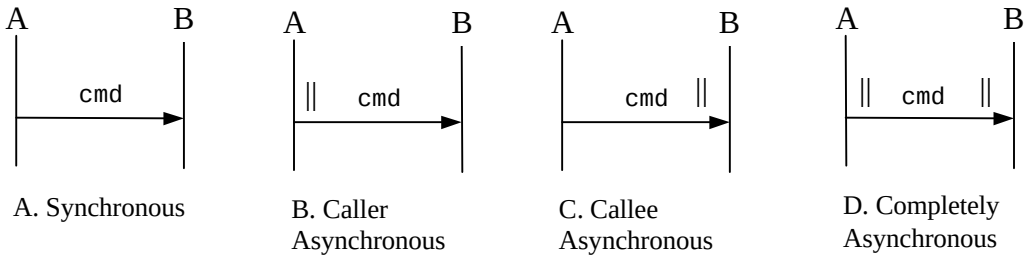


Figure 1 - Synchronous and asynchronous interaction diagrams.

Figure 1 shows a process A sending a command called `cmd` to a process B. Arrows denote the direction of control flow. Since control flow is implemented through function calls,

there is a caller and a callee for every interaction. In Figure 1, A is the caller and B the callee. The direction of data flow in the diagrams is not shown and can't be deduced. Data may flow in the same direction as commands, such as when passing data as arguments to a function. Data flow may also go in the opposite direction, such as when data is returned from a function.

Synchronous interactions make the caller block until the callee is finished executing the command. In some sense, the caller is at the mercy of the callee. If the callee dies and never returns, the caller will appear to have died as well. Also, while the callee is busy executing its command, the caller can do nothing except wait. Notwithstanding these limitations, synchronous connections are the most common type of basic interaction. The main reason is they are simple to implement, and synchronous systems are easier to build and test, since the interaction sequence is known a priori, rather than being variable at runtime.

Sometimes it isn't desirable or feasible for the caller to block until the callee is finished. If the callee is expected to carry out a lengthy operation, the caller may wish to carry on with its own internal processing in the meantime. In this case, the caller makes an asynchronous call to the callee [Satoh+95]. In most programming languages, this is done by creating a separate child thread on which to make the call. As soon as the call is made, the main thread can continue. An interaction may also be asynchronous on the callee side. The callee will then create a child thread to carry out the received command, and return control immediately to the caller while the command is being executed. By spawning child threads to carry out the work, the callee can receive multiple commands concurrently. Of course an interaction may be asynchronous on both the caller and callee sides. In general, asynchronous interactions utilize some form of synchronization, so the caller knows when the callee finished executing its command.

5 The Provider/Observer Role Pattern

Context	One process P_1 needs to know something about a process P_2, such as the value of a variable. P_1 is the Observer of P_2. Either P_1 or P_2 may initiate a status-reporting interaction that provides P_1 with the necessary information.
Forces	If processes need to communicate at all, it is to share information. One party must somehow be able to send the information to the other. Whether the communication paradigm uses messages, commands, callbacks, interrupts, shared memory, pipes, databases, etc., information goes from one party to another. Saying that one process is the talker and the other the listener identifies the direction of command – not information – flow. The talker may actually be requesting and obtaining information from the listener. Defining interactions in terms of Providers and Observers allows us to clearly identify the direction of information flow.

In many cases, the caller and callee play out specific roles in an interaction and many roles can be considered patterns. [Gamma+95] [Coad+95] [Coplien96]] and [Rising+98] describe numerous role, or behavioral, patterns. One of special interest is the one called the Observer pattern, in which a Subject can be connected to one or more Observers. When a change occurs in the state of the Subject, the latter notifies all the Observers.

The situation described by Gamma et al. is common, but their Subject/Observer pattern is a combination of both behaviors and interactions: the Subject takes the initiative of telling the Observers whenever a state change occurs. It is possible to generalize the Subject/Observer pattern by splitting it into smaller ones and studying them separately. The behavioral part describes a relationship in which one party contains information that is of interest to others. The interaction part describes how the parties talk to each other. I use the expression Provider/Observer to describe the relationship in this paper, because the word Subject gives a connotation of control, which the Provider doesn't always have.

There are two basic ways for the parties to interact, which I discuss below. Figure 2 shows the Provider/Observer behavioral role. A double headed arrow is used to connect them, because there is no implicit definition of interaction in the behavior: communication may go in either or both directions.

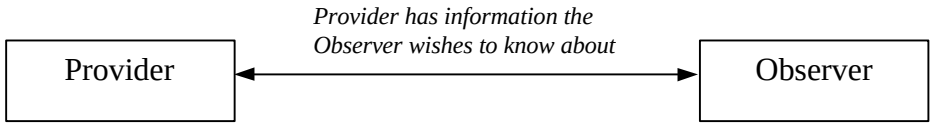


Figure 2 - The Provider/Observer behavior role.

The Provider is the component or process that can make information available to one or more other components, called Observers. Providers and Observers interact through Push or Pull models: the exchange of information between Provider and Observer may require signals to be exchanged in either direction, based on whether Push or Pull is used. How the Provider gets the data, and where it comes from, is irrelevant to the Observer. The data may be contained in an array held inside the Provider, or retrieved from a global database, the internet, or even another Provider. On the other hand, Observers make no commitment about what they do with the data they get from a Provider. They are simply interested in some aspect of that data. For example one Observer may want only to know if Provider's data has changed, without ever needing to access the data itself. Another Observer may need to display a subset of the data on the screen.

An important incarnation of the Provider/Observer pattern is when an Observer gets information from one Provider and makes it available to another Observer. In this case, the component is acting as an Observer to one component and a Provider to another, as shown in Figure 3.

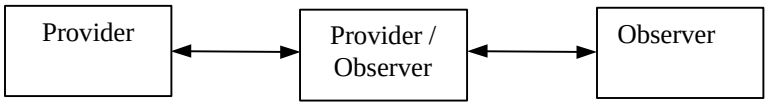


Figure 3 - Chaining Providers and Observers.

Each component acts as an Observer to the one on its left, and a Provider to the one on its right. If is possible for the Observer on the right of Figure 3 to be the same component as the Provider on the left side. In this case, the components are mutual Providers and Observers, as shown in Figure 4.

Figure 4 - Components that are mutual Providers/Observers.

In Figure 4, component A is the provider of some kind of data for B, so B is the Observer. But B is also the provider of some other kind of data for A, so A is the Observer. Mutual Provider/Observer behavioral patterns require attention in implementation to prevent livelock or deadly embrace runtime conditions.

There are a number of common uses of the Provider/Observer role pattern. Some of the styles used in software architecture can be considered applications of the pattern. Figure 3 is reminiscent of the Pipeline style [Shaw96], in which the interactions involve data flowing from process to process through system pipes, as shown in Figure 5.

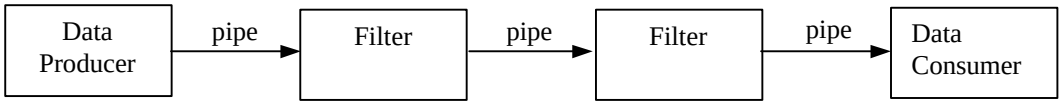


Figure 5 - The Pipeline architecture style uses the Provider/Observer behavioral pattern.

In the case of Pipe-and-Filter architecture, arrows indicate the flow of data through the pipes. The data flow direction also corresponds to the flow of control, that ripples from left to right.

Client/Server architectures also use the Provider/Observer pattern. Clients are generally the observers of some aspect of the data maintained and accessed through a Server, as shown in Figure 6.

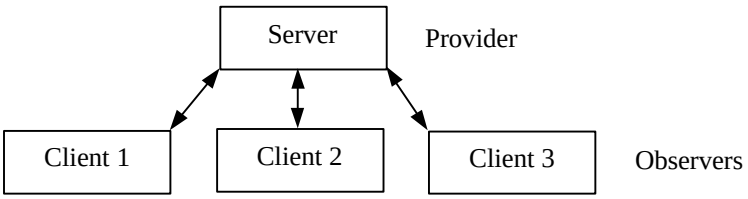


Figure 6 - The Client/Server architecture style uses the Provider/Observer role pattern.

Figure 6 shows the dominant roles for the Client/Server style, but clients are sometimes allowed to request changes to the server data, in which case the roles are inverted. Multiple clients can observe the data managed by a server. The pattern is not broken when multiple servers are added to the system. Observers may observe the data of any number of Providers. Clients don't communicate directly, and in fact generally are oblivious of each other.

Agent-based systems, communicating through a central Blackboard, represent another architectural style that is found in many problem-solving systems. The Blackboard contains information the agents are interested in. Current Agent-Based Systems utilize a variety of techniques for controlling the transfer of information from the BlackBoard to the Agents, and prioritizing the scheduling of Agents. From an Interaction Dynamics standpoint, the BlackBoard acts generally as a Provider, the Agents as Observers, as shown in Figure 7.

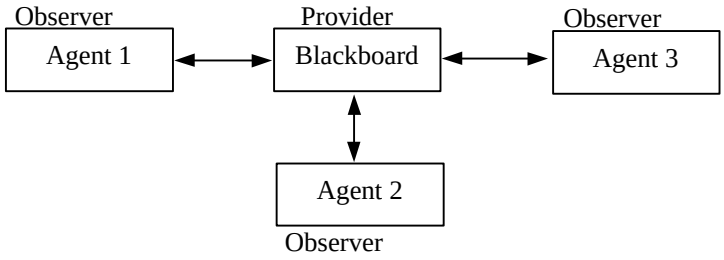


Figure 7 - The Blackboard architecture style uses the Provider/Observer role pattern.

Agents are concurrent processes with expertise in a certain domain. Agents use information posted on the Blackboard to make decisions, posting partial or complete results back to the Blackboard for other agents to see. While an agent is writing results back to the Blackboard, the roles are temporarily inverted: the Agent becomes the Provider, the Blackboard the Observer. The Agents and Blackboard participate in a collaborative Provider-Observer relationship to solve a problem.

Examples of the Provider/Observer role pattern can be found in all types of software designs and systems, including the following:

- The popular Model/View/Controller architecture: Views act as Observers, Models as Providers.
- The Microsoft ActiveX notification mechanism: ActiveX Containers act as Observers, ActiveX controls act as Providers. The roles can also be swapped.
- The callback architecture used by Motif to signal events to application programs: Motif acts as the Provider, applications as the Observers.
- The Java event/listener model: Event sources are the Providers, event listeners the Observers.
- The World Wide Web browser/server interaction: The server is the Provider, the remote web browsers the Observers.

6 Push and Pull Interaction Patterns

In the Provider/Observer role, it was stated that one process made data available to another. Nothing was said about how the data is made available. An important rule in good communication is establishing who controls or starts the conversation [Bijnens+94]. In Provider/Observer interactions, either the Provider or the Observer can initiate the interaction. Both situations represent fundamental interaction patterns, which I'll discuss separately.

The Pull Interaction Pattern

Context	A process P_1 needs to monitor the status of a process P_2. P_1 only needs to know the status of P_2 at specific times, and gets the status by issuing a request to P_2. P_2 does not provide any information unless requested by P_1. If the status of P_2 changes after a status-request, P_1 will not find out until it issues the next request.
Forces	If the Observer doesn't need status information except at specific times, it is time-consuming and wasteful for the Provider to notify the Observer every time a status change occurs. Many, possibly even all, of the notification messages may be ignored by the Observer. If only the Observer knows when status information is necessary, it is natural for the Observer to control when status information is exchanged

The Pull Interaction Pattern arises when the Observer process is also the controller of the interaction. When the Observer needs to know something about the Provider, it makes a request to the Provider. The Observer only gets data when it asks for it. Figure 8 depicts the Pull pattern for the Provider/Observer role.

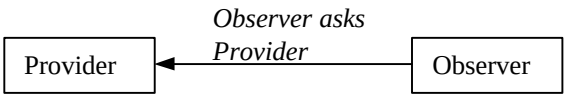


Figure 8- The Pull interaction pattern.

Pull interactions are very common for two reasons:

1. They are simple to implement, because both the Provider and Observer can be synchronous.
2. They apply to the common situation in which status changes in the Provider don't require the Observer to react immediately: the Observer will discover the change the next time it requests status.

There are numerous examples of the Pull pattern in everyday life and in computing.

Example 1

When you want a glass of water, you put a glass under the kitchen faucet and turn the water on to fill your glass. You are the Observer, the faucet is the Provider. The faucet will never fill your glass unless your turn the faucet on.

Example 2

Polling a keyboard device for typed keys: Consider a simple system in which a keyboard device contains an initially empty FIFO buffer that stores the keys typed by the user. A separate process issues READ commands to the keyboard device to get the keys typed. Each READ command returns 1 key. The reader process is the Observer, the keyboard the Provider. The reader will never get typed keys unless it explicitly asks the keyboard device.

The Round Robin Polling Pattern

Context	An Observer process needs to continuously monitor the status of a number of other Provider processes. Status changes can only be handled at a certain rate, decided by the Observer. The status requests can be fulfilled by the Providers *very quickly*, e.g. by returning the value of an internal state variable or performing a computation, in order to minimize the effect on the cycle time of the Round Robin loop.
Forces	If a single Observer is monitoring multiple Providers, it is possible for more than one Provider to have new status information at the same time. Allowing the Providers to send the information at will would create the requirement for the Observers to handle messages simultaneously from different Providers. This would require the Observer to use multiple threads or other computational resources, which could result in a significant amount of complexity. If the Observer controls the timing of the status exchanges, a specific order can be imposed over the Providers, and only one Provider will be allowed to talk at a given time.

Polling interactions apply to Provider/Observer Role patterns. The Observer is always in control of the interaction, repeatedly requesting information from the Provider. Polling is a Pull interaction pattern and is especially efficient when new data is frequently or always made available by the Provider. If this is not the case, many requests by the Observer will result in a *no-new-information* response, which usually equates to a wasted or useless transaction.

A pull interaction pattern can be used to poll multiple devices in sequence. After one device is polled, the next one is, and so on. After polling the last device, the first one is polled again and the entire process repeats. This interaction arrangement is known as *round-robin polling*, and is shown in Figure 9.

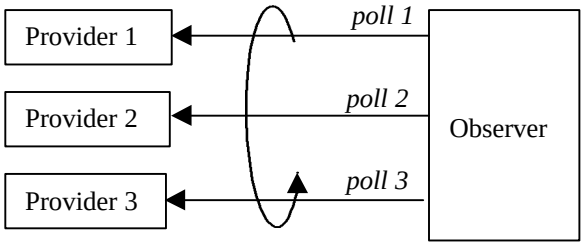


Figure 9 - Using the Pull Interaction Pattern in Round-Robin polling.

Round-robin polling applies to situations in which Providers are trustworthy and can guarantee that status requests will not fail. If a Provider hangs for any reason, such as crashing or waiting for results from another process, the entire Round Robin loop hangs.

One drawback of simple round-robin polling, in which the polling sequence is fixed, is the lack of a priority hierarchy. Consider a scenario in which multiple Providers have new data simultaneously. If one Provider has a greater chance of having new information, or if its information is more important that that of other Providers, the Observer might need to poll that Provider more frequently than the others, or poll the Providers in a certain order. Because the Observer is in control, it can decide when and how often to check Providers, and in what order. A complication may arise if the Providers have dynamically changing priorities or highly variable traffic patterns, in which cases Push interactions are often more suitable.

The Push Interaction Pattern

Context	An Observer process monitors the status of a Provider. When a change occurs, the Observer needs to be informed as soon as possible.
Forces	If changes in status are either expected to be infrequent, it is inefficient for the Observer to continually poll the Provider. Because there often is no way to know how often to check for status changes, Observers would have to poll the Provider at a rate faster than the highest expected status-change rate possible. Failure to do so would create the possibility of losing status-change messages. Because the Provider *knows* when status changes occur, it can notify the Observer directly, allowing the Observer to use its resources more effectively when status changes don't occur.

For high performance transfer of information between a Provider and Observer, the Push pattern can be used. In Push interactions, the Provider *tells* the Observer or Observers when new information is available, as shown in Figure 10.

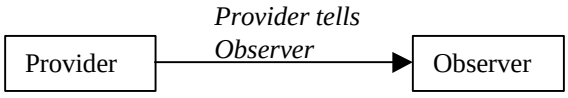


Figure 10 - The Push interaction pattern.

In Push interactions, no time is ever wasted on polling. Event-driven processing is an example of the Push pattern. Most GUI interfaces are event-driven: when the user takes some action, the system notifies the applications running, by sending events. Multiple events are sometimes multiplexed into a single notification, in which case a parameter indicates the precise event type. Because many software systems utilize rapid-fire parameter-based events, techniques have been devised to handle event demultiplexing efficiently [Schmidt94].

When the system sends an event , applications can choose whether to do something or not in response. In many cases, the system doesn't care whether applications take action or not. The applications are the Observers of the interaction, the GUI system is the Provider. Push interactions also support *broadcasting*, in which multiple Observers are efficiently

notified together [Aarsten+96]. The order in which Observers are notified in a broadcast is implementation dependent. As for the pull pattern, the Push interaction also has numerous example from everyday like and computing.

Example 3

The familiar Hollywood Principle "Don't call us, we'll call you": When a large number of observers are attached to a single provider, the provider's resources can be greatly taxed if each of the Observers constantly requests status. It is more efficient for the Provider to push status changes as a broadcast to the observers when changes occur.

Example 4

Consider the process of get up one morning at a given time, using a clock. One option is to stay awake, monitor (poll) the clock and get up on time. The problem with polling is it wastes resources when the desired data (the time to get-up) isn't available frequently (once every 24 hours). To use polling and get up on time means you must stay awake the entire night to monitor the time. The push interaction makes the Provider notify the Observer when data is available. An alarm clock is just the ticket. You tell it what data you are interested in, and get your full night's sleep. The alarm clock notifies you when it's time to get up.

Example 5

A system that must react to the depressing of a button: Because the button is only used occasionally, it isn't efficient for the system to constantly poll the button's status. Using a Push interaction, the button notifies the system when it is depressed.

Example 6

The Java event-listener model: *Listeners* are the observers of events that are issued by objects called *event sources*. When a listener is interested in some aspect of an event source, it registers itself as a listener of that object. From that point on, the event source notifies that event listener when the events of interest occur. Event sources can support multiple listeners, and listeners can unregister themselves at any time. The model is very efficient for dealing with high-bandwidth events. Consider a GUI element that is interested in tracking mouse motion, perhaps to allow itself to be dragged around on the screen using the mouse. In the traditional event-driven processing model, each time the mouse is moved, the system sends mouse motion events to the GUI element located under the mouse cursor. If this element is not interested in tracking mouse motion, the event dispatch is a complete waste. Using the event-listener model, the Java runtime system only dispatches mouse motion events if there are registered listeners for that type of event.

Example 7

The classic Model View Controller paradigm: More specifically, consider the way the Model interacts with the Viewer or Viewers. When the Controller makes changes to the Model, the latter *pushes* notifications to the Viewers. The Model acts as a Provider, the Viewers as Observers. The Model can support any number of Viewers. When changes occur in the model, a *change notification* is broadcast to all Viewers. In response, the latter usually refresh all or part of a window. Viewers have no knowledge of each other, and are controlled by the Model.

Using a push interaction patterns frees the Observer from devoting substantial resources to the monitoring of the Provider. While the Provider has no new data, the Observer uses its own resources doing other things. In Push interactions, the Provider controls when the Observer is notified. In general, the Observer has no control of when or how often, or the order in which mutually independent Providers notify it. It may be desirable for the Observer to not have control, perhaps because the dynamics of the interactions are unpredictable or constantly changing.

7 Opaque Interaction Patterns

I leave behind the discussion of Provider/Observer patterns to focus on the dynamics of command execution across component or process boundaries. In many systems based on communicating concurrent processes, operation is largely through the exchange of commands. With few exceptions [Lavender+96] [Aarsten+95], most design patterns abstract away the details of communication and concurrency, but these details are important in determining how command execution affects the processes involved. In this section I'll show the ways one process can invoke commands for execution in another.

Context	A process P_1 wishes to send a message to a process P_2. While the message is being processed, P_1 does not need to obtain progress feedback from P_2.
Forces	Many interactions produce an *atomic* outcome in the receiver, in the sense that the receiver returns a completion-status only when the message has been processed. The receiver may be incapable of reporting partial completion-status results, because it is too complicated to do, not warranted, not practical, or even impossible.

A situation in which commands are exchanged between processes implicitly defines the roles of Caller and Callee, but I won't consider these roles to have any special meaning: they are direct consequences of the way commands are invoked using function calls.

While the callee is executing the command given, in many cases the caller has no notion of how long the process will take, or how soon it will be finished. When the caller can't peer into the callee to see how much work is ahead, or to monitor execution progress, I will call the interaction *opaque*. It is reminiscent of the batch jobs from older times, in

which after submitting a job, the user would have no interaction with it, being notified upon job completion. Opaque interactions come in 2 varieties, based on whether they are synchronous or asynchronous.

Synchronous Opaque Interaction Patterns

Context	A process P_1 wishes to send a message to a process P_2 and P_1 must wait until P_2 has finished with the message before proceeding. P_2 is trusted to return control within a timeframe that is acceptable to P_1.
Forces	While waiting for a message to be processed by the callee, the caller in a synchronous interaction has few options, because the caller thread is blocked until the callee returns control. If the response from the callee is crucial in determining what the caller does next, the caller may have no other choice than to stop and wait for the callee to finish.

This is the simplest interaction to implement and indeed the most common type. The caller makes a call to a function contained in the callee. The caller blocks until the callee finishes. The callee returns control to the caller only when the command has been completely executed. If the command cannot be completed due to errors, the premature completion will still be regarded as a completion. When errors are expected, the interaction should arrange for a way to notify the caller, such as by returning an error status code. Raising signals or throwing exceptions are ways of signaling errors, but are system or language dependent. In the interest of reusability, it is often desirable to avoid the use of language dependencies between processes. Figure 11 depicts a synchronous Opaque interaction.

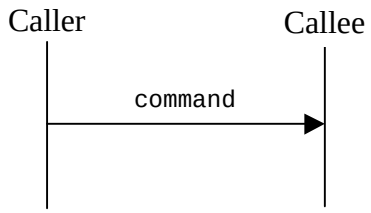


Figure 11- A synchronous opaque interaction.

The notion of trust is important in this type of interaction. If the callee fails or dies as a result of executing the command, control is never returned to the caller, and the whole system will appear to have failed or died. A system built exclusively using opaque synchronous interactions will only be as reliable as the least reliable component. Components calling other components synchronously place a great deal of faith in the callees. For this reason, synchronous opaque interactions are used most commonly with simple commands. One aphorism from the trenches: For *every synchronous interaction is a situation that will test the faith put in it*.

Example 8

Two persons are talking on the phone. One asks, "What time is it ?". The person asking the question expects the request to take a short amount of time to complete. Because the interaction is expected to be brief, he puts his other activities and thoughts on hold momentarily until the response is returned. The interlocutor knows the request will only take a few seconds, and also decides to put everything else on hold while looking at his watch. The interaction is therefore synchronous. Because the person asking the question can't see whether the other is looking at his watch, or whether he even has a watch, he can't determine whether the person looks at his watch, or how long it will take to read the time. The transaction is therefore Opaque.

Example 9

A navigation System in an avionics package uses a Global Positioning System (GPS) receiver to get the current position. The GPS receiver always maintains the current position internally, so requests can be honored immediately. The navigation component uses a synchronous opaque interaction to interrogate the receiver for the current position.

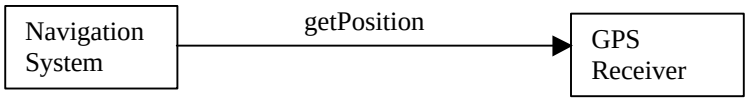


Figure 12 - Using a Synchronous Opaque Interaction Pattern in a Navigation system.

The Navigation system is designed to interact with the GPS receiver synchronously because the specs for the receiver indicate that data is generally returned immediately, allowing the Navigation System to decide when and how often to request new data. While the receiver is fetching the data to return, the Navigation System has no idea how much time the response will actually take, although it has expectations derived from the spec sheet. If the GPS receiver loses contact with a satellite and needs to resynchronize, there may be a delay before a response can be returned. For critical operations, often asynchronous interactions are desirable, because they allow the caller to recover from the callee's failure to respond.

Asynchronous Opaque Interaction Patterns

Context	A process P_1 wishes to send a message to a process P_2. While P_2 is handling the message, P_1 and/or P_2 need to be able to continue with other tasks.
Forces	When a message is expected to take a significant time to be processed, it may be inefficient to block the caller or callee. If the caller can continue with other tasks without waiting for a completion status for the message, it can make the call asynchronously, using a separate thread. When the caller needs to monitor the time-to-complete of the callee, it may be necessary to run a completion-timer on a separate thread. When the callee must be able to handle multiple simultaneous requests, such as in a client-server environment, devoting a separate thread for each call is a scalable and effective way to provide service.

As stated earlier, Opaque interactions can also be asynchronous. While asynchronous interactions can in general have an asynchronous nature on the caller, the callee, or both sides, Opaque asynchronous interactions are slightly more restricted, as shown in Figure 13.

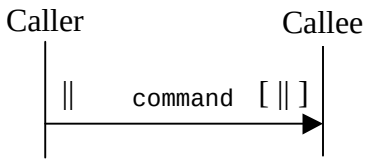


Figure 13 - An Opaque Asynchronous Interaction Pattern, returning no completion information.

While the caller must be asynchronous, the callee may or not be asynchronous. The square brackets on the callee side indicate that asynchronicity is optional. After invoking the command, the caller has no idea of when the command completes, or whether it completed with errors or not.

Often the caller is interested in something about the outcome of the executed command, i.e. when it finishes execution, what outcome the command had, or both. In these cases, the callee must use a separate communication channel to return some form of completion information to the caller as shown in Figure 14. The completion information is sometimes referred to as a *token* [Schmidt+96].

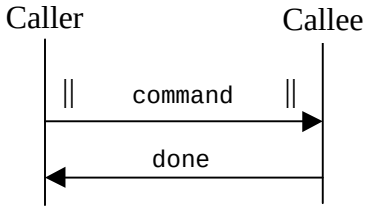


Figure 14 - The Asynchronous Opaque interaction pattern with completion notification.

Here, both the caller and callee must be asynchronous. The callee returns immediately to the caller after beginning execution of the command. A status code might be used here to indicate whether the command was accepted or not. Upon termination, the callee sends a `done` command, with optional completion status information, to the caller. The caller then knows when the operation completed, and optionally also how it completed.

For opaque asynchronous interactions, the caller has no way of knowing how long a command will take, or how far along the execution has progressed. If the callee provides no completion notification, the caller doesn't even know when the command has completed.

Example 10

A person wishes to rewind a video cassette. He pushes the Rewind button on the VCR. The person doesn't know how long the operation will take, but decides there will be enough time to get something from the refrigerator. While the Rewind command is in progress, the user is therefore busy doing something other than staring at the VCR. When a characteristic Click sound emanates from the VCR, the person knows the Rewind command has completed.

Example 11

A person throws a rock from a car passing over a bridge. The implied command is "Put rock into river below". The car passes the bridge before the rock reaches the river. The person will never know if or when the rock reaches the river. Although experience tells the user it should, and allows him to estimate when this will occur, based on the bridge height and other variables, there will never be certainty. Perhaps a tree branch intercepts the rock, or the rock was thrown too late and misses the water.

Example 12

A satellite system, during the course of deployment initialization, needs to start up several subsystems to become operational. Each subsystem requires time to start up, and the satellite can begin operation only after all subsystems have initialized. The Startup Controller issues asynchronous Start commands to all the subsystems, and resumes other

tasks, such as system monitoring, until the subsystems report they are all up. Figure 15 shows the system.

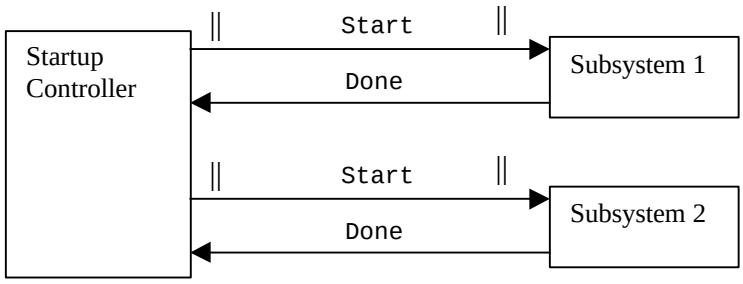


Figure 15 - Using Asynchronous Opaque interactions to deal with multiple processes concurrently.

Opaque asynchronous interactions are useful for time-constrained operations, when the caller wishes to monitor the amount of time the callee takes to complete a command. By making the call on a child thread, the caller can run a timer concurrently with the command execution, and timeout of the call if necessary. The caller may also wish to perform other processing while the call is in progress.

8 Monitorable Interaction Patterns

Context	A process P_1 wishes to send a message to a process P_2. The message is expected to either take a long time to process or to involve numerous intermediate steps. P_1 needs to be informed of execution progress by P_2.
Forces	When an operation can result in widely varying intermediate steps or processing times, the caller may wish to monitor the callee's progress. This allows the caller to alter or adjust its course of action, based on intermediate results. An alternative would be to have the caller be responsible for controlling the intermediate steps, perhaps through separate commands. This approach would essentially break the encapsulation of the operation by the callee, and result in otherwise unnecessary sharing of details by the caller and callee.

Opaque interactions are adequate for operations that have a predictable duration. But often the processing time of a command is not known in advance, or the speed of the system can vary dramatically at runtime. In these cases, the caller may need or wish to monitor the callee, to determine that progress of an executing command. Monitored interactions allow the caller to get status back from the callee during command execution. This type of interaction requires the caller to be asynchronous. The callee may or not have to be asynchronous, based on how the interaction is implemented, as described in the next two sections. Because monitorable interactions allow the caller to get status back during execution, it is an *interactive* interaction. Although the expression is somewhat

unfortunate, interactive patterns give a great deal of flexibility to a system. Callers can change their course of action based on slow progress, unexpected estimates or disappointing results from the callee.

Monitorable Interactions bear a resemblance to Asynchronous Opaque ones, but with an extra status reporting channel. Monitorable interactions are also used in different situations.

Pull-Monitorable Interaction Patterns

Context	A process P_1 wishes to send a message to a process P_2. P_1 needs execution feedback at specific times, designated by status requests from P_1. Feedback is not necessary except at those specific times. P_2 must be able to provide the feedback quickly and reliably.
Forces	When the caller doesn't care about or can't handle changes in status of the callee except at given moments, it is inefficient for the callee to issue unsolicited status-change notifications. By having the caller ask for status, it can control the frequency of status-requests during the various intermediate steps taken by the callee.

In this type of interaction, the caller *pulls* the progress status information from the callee. It does so by means of a `getStatus` type of command. The status may return any information useful for progress monitoring, such as the number of elements processed, the fraction of work accomplished, the estimated time to complete, or other. Figure 16 depicts the Pull-Monitorable interaction pattern.

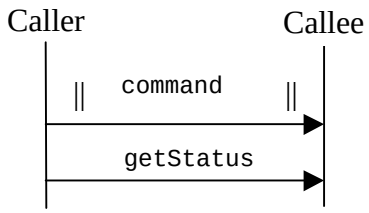


Figure 16 - The Pull-Monitorable interaction pattern.

Because the callee must be able to support calls to its `getStatus` input while it is carrying out the designated operation, it must support asynchronous command execution. The interaction pattern generally uses a synchronous channel for `getStatus` command, since status reporting is often simple for the callee, and responses are returned quickly.

Example 13

A person needs to catch a plane and is running late. She stops a taxi cab and says, "Take me to the airport". Along the way, traffic conditions change considerably. In fear of missing her flight, the passenger asks the driver "How much more time do you think the

ride will take?". Eventually she gets to the airport, she gets out, and her interaction with the taxi ceases.

Example 14

A person surfs the web with a web browser. He clicks the hyperlink on one HTML document to go to another document. Inside the browser, the user interface component (UICOM) issues a `LoadURL` command to the web navigation component (NAVCOM), as shown in Figure 17.

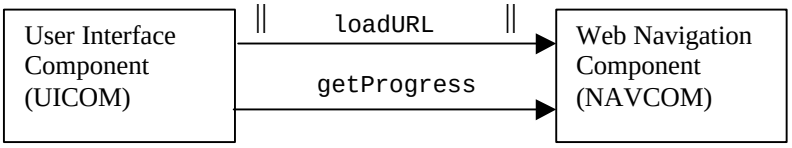


Figure 17 - Loading a web page using the Pull-Monitored interaction.

While NAVCOM loads the requested page, UICOM issues a series of `getProgress` queries to NAVCOM. UICOM uses the information to display a progress bar on the user's screen, to give an indication of how long the load process will take.

Push-Monitorable Interaction Patterns

Context	A process P_1 wishes to send a message to a process P_2. P_1 needs execution feedback as soon as P_2 has the information available. P_2 sends status reports back during execution as the status changes.
Forces	When intermediate status information is needed during the processing of a message, the caller may not know how often to request status, possibly because the message entails a widely varying number of steps or amount of processing time by the callee. It is wasteful for the caller to continuously poll the callee, especially if there is the possibility of not requesting status often enough and losing a critical status report. A safe approach is to put the burden of status reporting on the callee.

Rather than having the caller poll the callee for status information, the callee can also push the information to the caller. This might be useful when the callee has infrequent or rare changes in status, or when the caller can't afford to spend time polling the callee while execution is in progress. The caller is notified of changes in status only when changes occur. Because the callee pushes the status information to the caller, it may or may not be asynchronous. For example, if a command requires a series of internal steps to be carried out in sequence, the callee might issue status notifications each time a new step is completed. If the operation requires executing steps in a loop, notifications might be sent at the completion of each iteration. Asynchronous behavior might be required by the callee if the execution of the command required the invocation of a monolithic function that returned only when the entire command was executed. A separate process

would be necessary for the callee to send notifications back to the caller. Figure 18 depicts the Push-Monitorable Interaction pattern.

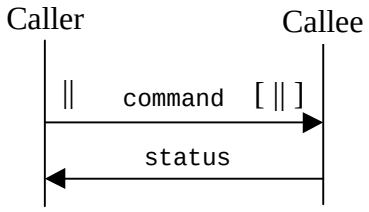


Figure 18 - The Push-Monitorable Interaction pattern.

A special case of the Push-Monitorable Interaction is when the callee reports status only once, upon command completion. This scenario is equivalent to the Asynchronous Opaque interaction described earlier. In Figure 18 the callee is not aware of what the caller does in response to the status notifications. The caller may even choose to ignore the notifications entirely, but the callee will never know.

Example 15

A person gets in an elevator and pushes the button for the desired floor. While traveling, the person checks some papers for an upcoming meeting. As the elevator travels, it notifies the passenger of progress using a numeric readout display in the elevator. The passenger has the option of watching or ignoring the floor notifications.

Example 16

A person uses batch files to copy groups of files from one directory to another. After entering the name of the batch file, the system displays the invocation of each `copy` command and its completion status. When the batch file is finished, a command-line prompt is displayed. The system provides interactive feedback to the user using the system display. The user may watch the screen to make sure things are progressing as planned. The computer system is not aware of what use the notification feedback information is used for, or whether it's used at all.

Example 17

A laser printer uses heat to fix toner ink to paper. Before loading paper into the paper path, the system controller (SYSCON) must turn on the heater in the FUSER unit and wait for a certain termperature to be reached, as shown in Figure 19. The FUSER reports its temperature through a sensor. When the correct temperature is reached, SYSCON starts the print process.

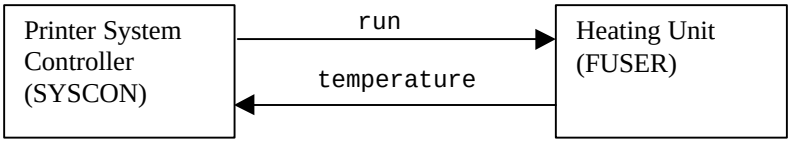


Figure 19 - Software components in a laser printer using Push-Monitorable interaction.

The FUSER sends the temperature notifications for each 1 degree change in temperature. It doesn't know what SYSCON will do with the information, and in fact has no knowledge of how SYSCON works.

9 Abortable Interaction Patterns

Context	A process P_1 wishes to send a message to a process P_2. The message is expected to take a *long time* to be processed. A *long time* is a duration that gives P_1 a chance to perform other tasks, such as checking for excessive time-to-complete conditions. While message processing is underway, either P_1 or P_2 may decide to abort it.
Forces	Many systems have the capability to adapt to unusual situations. A database server that doesn't respond may require a system to notify the user or to contact a different server. By designing an interaction as abortable, the system has a clear definition of which party (the caller, the callee, or both) has the power to abort it, and how the interaction is affected in the case of abortions.

During the execution of a command on behalf of a callee, it may be desirable or necessary to abort execution for a variety of reasons, such as excessive time-to-complete, errors detected in other parts of the system, or other. While the concept of abortable interactions is simple, aborting a command may require extensive processing to clean up partial results or other side effects of the partially completed command. This clean up work is not considered part of the Abortable Interaction pattern itself.

In order for interactions to be abortable, both the caller and callee must be asynchronous, because the caller must have the ability to send the `abort` command during the execution of a previously issued command, and the callee must be in the position to receive the `abort` command while executing a previously received command. Both Opaque and Monitorable interactions can be made Abortable, as shown in the following sections. Interactions can be aborted by the caller or callee. The hallmark of interactions that are caller-abortable is the existence of a dedicated communication channel to carry `abort` commands. Callee-abortable interactions may or may not have a dedicated communication channel back to the caller to signal aborts.

Abortable interactions might be considered composite ones, combining the features of an abstract Abortable interaction and an Asynchronous one. Because Abortable interactions by themselves are not meaningful – there must be something to abort – they are considered refinements of other interactions.

Abortable Async Opaque Interaction Patterns

Context	A process P_1 sends a message to a process P_2. While P_2 is busy, processing of the message may need to aborted at any time. The initiative for aborting may be either the caller's, the callee's or both.
Forces	When an opaque interaction is underway, the caller has no way on knowing how far along the callee is, or how soon the callee will finish. The caller can only monitor the amount of time the callee is taking to respond, or other side effects of the interaction. The caller may choose to abort the interaction if the callee doesn't finish within a given time. The callee may decide to abort if it detects conditions that make further processing futile. A separate abort channel supports both caller- and callee-abortable interactions.

Opaque asynchronous interactions have the drawback that the caller has no way to determine how long a command will take, or even *if* the command will succeed, except perhaps for previous experience with that command. Even so, a system whose computing resources vary at runtime in power and speed may be difficult to predict, in terms of how quickly a command will be executed. There are many situations in which a system may need to attempt a command. Should the command fail to complete within a certain time limit, the system may opt to abort the command and change course of action. In another scenario, a caller may initiate concurrent commands in multiple callees through Abortable Asynchronous Opaque interactions. Should one critical command return an error, all other outstanding commands need to be aborted. Figure 20 Abortable Asynchronous Opaque interaction.

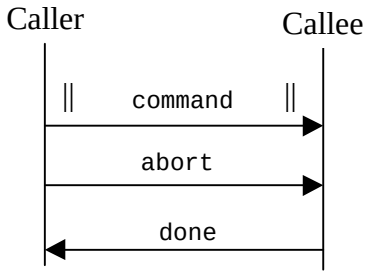


Figure 20 - The Caller-Abortable Asynchronous Opaque interaction pattern.

Abort commands only make sense if issued during the execution of a previously issued command. An Abort command may abort a single command or a whole series of

concurrent commands. The Abortable interaction pattern doesn't characterize how the `abort` command works inside the callee, what it affects or how long it takes. If a callee receives an `abort` command after a command has already been terminated, or before a command was received, it may choose to return an error code to the caller, or take some other action.

Example 18

A person knows that a ferry boat service exists to take passengers to an island off the coast. The person goes to the ferry's dock and waits. The person has no way of knowing whether the ferry will actually arrive, but has the ferry's schedule and a reasonable amount of trust in the schedule. If the ferry fails to arrive within a certain amount of time, the person decides not to wait any longer and changes plans. In this example, the person has some knowledge, i.e. that a ferry service exists, where it departs from and the departure schedule. The person also has certain expectations, i.e. the ferry will depart from a given location at a given time. Abortable commands are often, but not always, used when there is a plausible reason for aborting, not just a random flip of the coin. Plausibility requires knowledge of, experience with, or expectations for a given interaction scenario.

Example 19

A person desires to test a TCP/IP local area network connection between computers by using a `ping` command. The person types in the IP address of the computer and waits for the `ping` response. If the command doesn't complete within a certain amount of time, the person gives up. Because the command was opaque, the person will have no idea why the command was taking so long. Some part of the networking software may have failed, or perhaps there is a hardware failure. How long a person waits, i.e. how tight the time constraint will be before aborting, will depend on how long the person can or is willing to wait. An experienced user may recognize that it is futile to wait for more than one minute. An impatient person may give up after 5 seconds. The user may also abort simply because he changed his mind. The interaction pattern is not concerned with the reasons or the criteria for aborting.

Abortable Monitorable Interaction Patterns

Context	A process P_1 sends a message to a process P_2. While P_2 is busy, P_1 needs to obtain execution progress information from P_2. At any time during message execution, a mechanism is required to interrupt the transaction.
Forces	Monitorable interactions utilize status messages to keep the caller informed of progress the callee makes in processing a message. While it is possible to utilize the status messages to convey abortion commands, a separate abort channel is desirable because it makes a clear distinction between the ordinary and extraordinary processing. For example, if the callee sends an abort command to the callee, the command may need to be exeecuted in a thread that has higher priority than the original message. Using a separate abort message also makes it clear who has the power of abortion.

When an action is expected to take a long time, it might be desirable for the caller to be able to follow the execution progress of the callee. The Monitorable interaction pattern can be used for this purpose, and there are many cases in which monitored interactions may need to be abortable. Unexpected partial results, excessive time-to-complete estimates or disappointing performance may all be reasons. Both the caller and callee may need the option to abort commands in progress.

Abortable Pull-Monitorable Interaction Patterns

Context	A process P_1 sends a message to a process P_2. While P_2 is busy, P_1 makes execution progress status requests to P_2. At any time during message execution, P_1 or P_2 must be allowed to abort the message.
Forces	When the caller can only handle status-change messages at certain moments, it may be wasteful for the callee to issue status reports every time a change occurs. A barrage of status-change messages may tax the resources of the caller and callee unnecessarily. It may be more efficient to allow the caller to request status reports when it has a use for them. Using a separate abort channel is advantageous, especially when the callee has abortion control, because the callee can signal abortions immediately. Without a separate channel, the callee would only have two ways to signal abortions back to the caller: by returning a special code from the original message or from the next status-request message.

In this interaction, the caller periodically requests progress status from the callee. Figure 21 shows the configuration of an Caller-Abortable Pull-Monitorable interaction. At any time during command execution, the caller may issue an abort command.

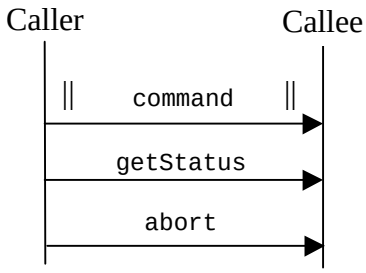


Figure 21 - The Caller-Abortable Pull-Monitorable interaction.

The other case is when the callee can abort commands. When it does so, it can either inform the caller with the next status message, or use a dedicated channel to push the abort notification to the caller. The first case can be considered a special case of the Pull-Monitorable interaction, the latter is shown in Figure 22.

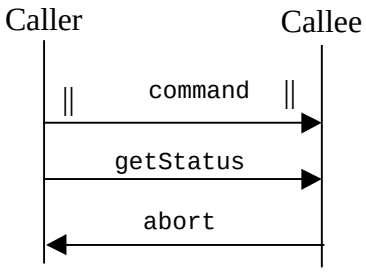


Figure 22 - The Callee-Abortable Pull-Monitorable interaction with dedicated Abort notification channel.

The are situations in which both the caller and the callee have the option to abort command execution, a case handled by combining the caller- and callee-abortable interaction patterns.

Example 20

An office worker wishes to have lunch. She goes to the company cafeteria, sees there is a short waiting line and gets in line. The line progresses along nicely until the drink dispenser machine breaks. The line stops moving. After waiting a few minutes, to see if the problem gets resolved, she decides to get lunch somewhere else.

In this situation, the woman could check and monitor the progress of the waiting line, and see that a problem developed before reaching the head of the line. Being in a position to monitor the situation, the woman was able to make an informed decision to stop waiting and choose a different course of action.

Example 21

A machine is designed to measure a person's blood pressure. It works by having patients put their left arm into a circular strap and pressing a Start button. The machine internally uses two software processes, as shown in Figure 23. The first, the system controller (SYSCON) monitors the Start button. When the button is pressed, it issues a `start` command to the strap control system (STRAPSYS), which begins pressurizing the strap to tighten the strap around the patient's arm to get a pressure reading. As pressurization is in progress, SYSCON makes frequent status requests to obtain the instantaneous pressure. The strap pressure needs to be increased until a systolic pressure can be read by STRAPSYS. If the pressure reaches a limit value before obtaining a systolic pressure, SYSCON concludes that the strap wasn't placed correctly around the arm and issues an Abort command.

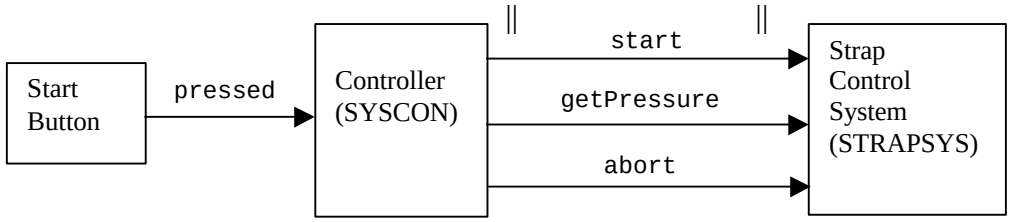


Figure 23 - A blood-pressure reading machine using a Caller-Abortable Pull-Monitorable interaction.

Abortable Push-Monitorable Interactions Patterns

Context	A process P_1 sends a message to a process P_2. While P_2 is busy, P_2 issues execution progress information to P_1. At any time during message execution, P_1 or P_2 must be allowed to abort the message.
Forces	It is advantageous for the callee to push the execution progress messages to the caller when status changes either occur at unpredictable times, or when they are critical. A separate abort channel isn't absolutely indispensable to support abortable interactions. The status-reporting messages pushed from the callee to the caller could be setup to use a pre-defined code to handle abortions.
	A separate abort channel allows a clean separation between status reporting and command termination, allowing both the caller and callee to immediately signal abortions, without complicating the status-reporting mechanism.

Push-monitorable interactions can also be abortable. Figure 24 shows the caller-abortable interaction. Callee-Abortable Push-Monitable interactions are also possible and similar to the Callee-Abortable Pull-Monitored case described in the last section.

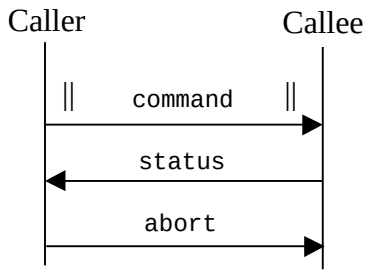


Figure 24 - The Caller-Abortable Push-Monitorable Interaction.

Example 22

An investor asks his stock broker to purchase some stock at $90 a share, with the intention of selling it at $100. The broker is instructed to call every day with the current stock price. The stock first increases to $92, then starts a slow decline. After 2 months, the investor decides to sell the stock.

The investor monitored the stock price through telephone calls from the broker. Because of unsatisfactory progress, the investor issued a sell command, aborting the investment.

Example 23

Printing a file from a computer. The user selects a file and invokes the print command. The system displays a message on the computer display while it is printing, indicating the percentage of work accomplished, or the number of the page being printed. After starting the print job, the system reports that the document has 1000 pages. The user realizes that he chose the wrong file and stops the command by clicking an Abort button.

The user started the interaction with a Print command. While the command executed, the system pushed status notifications to the caller, by displaying messages on the system display. The user asynchronously terminated the operation with an Abort button.

Example 24

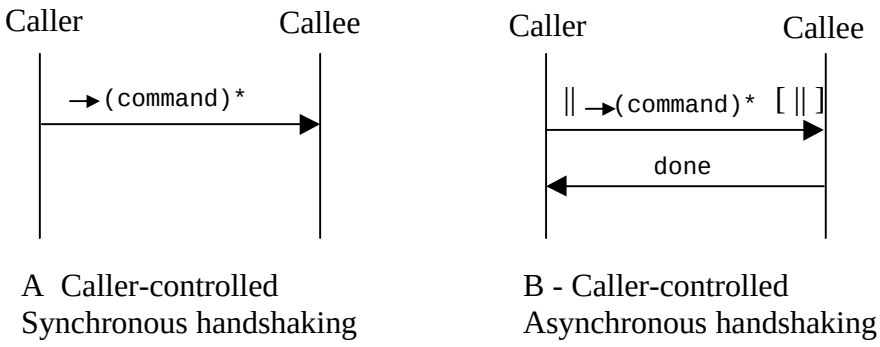
Loading an image with a web browser from a remote server over the internet. The browser displays a bar at the bottom of the screen, showing the progress achieved while loading images. If the bar advances quickly and then stops for a while, the user may decide to abort the loading of that image and do something else.

10 Handshaking Patterns

Context	A process P_1 wishes to transfer a large amount of information to a process P_2. The data can be broken down into a series of messages. P_1, P_2 or both may have control over when messages should stop being sent, depending on who has knowledge of when *enough is enough*.
Forces	Some interactions require significant amounts of data to be exchanged. Sometimes the caller gains access to this information one piece at a time, such as when receiving it from another party, and hands it off to a callee in pieces. Other times, a process needs to break a large amount of information into smaller chunks to accommodate characteristics of the environment, such as network packet-size limitations. Using multiple smaller interactions to accomplish a more complex operation requires the designation of which party controls when the overall operation can be considered finished. Sometimes the caller needs to keep sending data until the callee signals satisfaction. Sending more data would be futile. Other times, the caller knows how much information it needs to send in advance, and tells the sender when there is no data left.

When lengthy or complex operations can be carried out by the repeated application of simpler ones, the interaction consists of an iteration: the caller issues a command, the callee executes it. The caller continues to repeat the command until the overall operation is complete. I call this type of interaction a *handshaking pattern*, because of its similarity with hardware handshaking, used frequently in data communications, in which the callee tells the caller when it is ready for the next command or message.

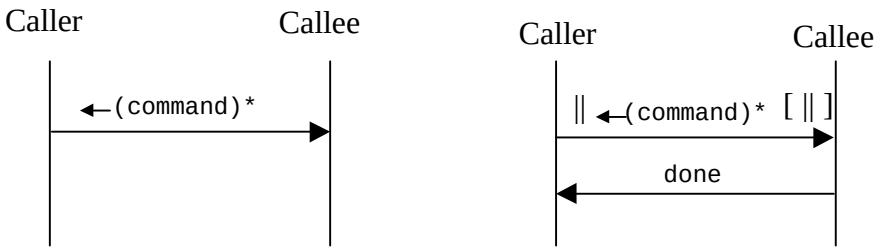
Handshaking can be used with synchronous and asynchronous commands. In both cases, a new command is issued only after the previous one has completed execution. Completion of the interaction can be driven by the caller, the callee, or both. In the first case, which I'll call *Caller-Controlled Handshaking*, the caller somehow determines that the operation is complete, and stops sending commands. In the second case, called *Callee-Controlled Handshaking*, the callee determines the operation is complete, and returns a signal to the caller to make it stop issuing commands. The situation in which both the caller and callee can stop the interaction is called *Dual-Controlled Handshake*. For synchronous commands, the signal may be a special return status code. For asynchronous commands, the signal may be a return status code, a parameter used in the `done` notification, or a dedicated `finished` channel. Figure 25 and Figure 26 show the two main configurations for the Handshaking interaction.



A Caller-controlled Synchronous handshaking

B - Caller-controlled Asynchronous handshaking

Figure 25 - The Caller-Controlled Handshaking Interaction patterns.



A Callee-controlled Synchronous handshaking

B - Callee-controlled Asynchronous handshaking

Figure 26 - The Callee-Controlled Handshaking Interaction patterns.

To indicate the iterative nature of the interaction, the command is enclosed in parentheses and followed by an asterisk. As indicated in the figures, the notation uses a small arrow in front of the command, to indicate who is the controller. The controller determines when the iteration terminates. An arrow from the caller to callee is used for Caller-Controlled Handshaking. The arrow points the other way for Callee-Controlled Handshaking, or points in both directions for Dual-Controlled Handshaking.

Example 25

A person is eating a plate of spaghetti for dinner. He decides to add some salt. He picks up the salt shaker and repeatedly shakes it over the pasta until the desired amount of salt has been added. In this example, the operation is add salt to spaghetti. The caller is the person, the callee is the salt shaker. The operation is a Caller-Controlled Handshake, because it is the person who decides when to stop adding salt.

Example 26

A system is designed to transmit satellite images over a network. The transmission channel uses packets to carry information, and each packet can hold 2048 bytes. Because the images are high resolution and are expected to be larger than 50MB in size, they are sent in blocks, so the sender breaks them into 2048-byte chunks. To signal the end of an image, a last packet is sent with 0 bytes. During the interaction, the receiver doesn't know how much data will arrive. Only the caller does, and it uses a special signal to tell the receiver when the interaction is complete.

11 Combining Patterns

Patterns can be built up into larger composite ones. Composite patterns have been presented at the Object Design level [Riehle97] [Vlissides98], and composites also apply to interactions. For example an asynchronous operation may be lengthy to abort, because the callee needs to clean up the side effects of partial execution. The user may need the option of monitoring the progress of abort commands. The configuration suggests the combination on the Abortable Asynchronous Opaque interaction for the initial command, and the Pull-Monitorable interaction for the Abort command, as shown in Figure 27.

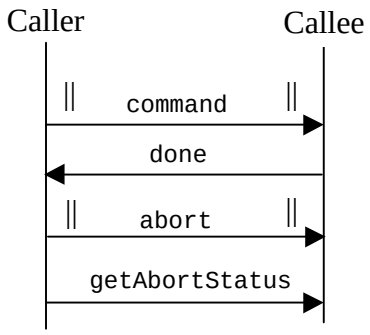


Figure 27 - Combining the Abortable Asynchronous Opaque and Pull-Monitorable interactions.

Example 27

A homeowner decides to have a swimming pool built into his back yard. He finds a contractor to do the job and tells him to start construction. During construction, a pre-existing gas line is discovered underground where the pool was going to be built. The homeowner decides to cancel the project, requiring the backyard to be restored to its original state. This entails repairing the damage to the landscaping, filling the partially dug pool hole, and removing construction equipment. The order to start construction was short, consisting in a phone call to the contractor. The process of aborting construction required considerable effort, which the homeowner wished to monitor.

Example 28

Consider a rocket on the launchpad in the final stages of pre-launch countdown. 5 minutes before launch, an auxiliary fuel tank is to be pressurized. During pressurization, the launch manager orders the launch aborted due to severe weather conditions. The fuel tank must be depressurized, which requires liquid hydrogen to be pumped back into a storage tank, a process lasting 15 minutes. The interaction between the fuel tank control system (CTLSYS) and the tank system (TANK) is shown in Figure 28. The initial `pressurize` command is asynchronous, since the fuel tank supervisor is free to do other things while pressurization is in progress. The abort command requires a lengthy operation that needs to be monitored.

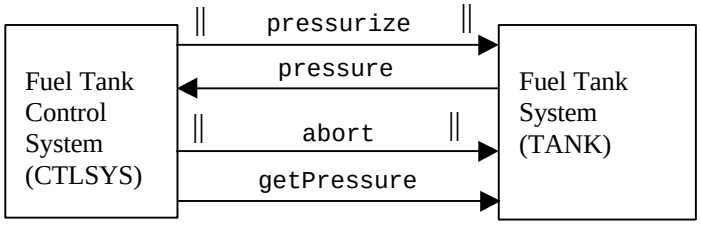


Figure 28 - Combining the Abortable Push-Monitorable Pull-Monitorable interactions.

12 Conclusion

If the answer to the software crisis is greater productivity, at least one of the questions is, "How can we create components to make them more easily reusable?". Interfaces and interactions are pivotal in reusability. Recognizing the importance of interaction dynamics is a first step to more reusable software. The patterns presented in this paper can be considered at the bottom of the interaction dynamics pattern hierarchy. In time, more will de discovered, including domain-specific ones. In our endeavors to produce *better-quicker-cheaper* software, interaction patterns should have an important role. In some sense, the maturity of the field of component-based development is based on how well we understand the interactions between communicating entities.

References

[Alexander77] C. Alexander. *A Pattern Language*. Oxford University Press, New York. 1977.

[Alexander79] C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[Aarsten+95] A. Aarsten, G. Elia and G. Menga. "*G++: A Pattern language for computer integrated manufacturing."* In J. Coplien and D. Schmidt (eds.), *Pattern Languages of Program Design.* Addison-Wesley, Reading, MA. 1995.

[Aarsten+96] A. Aarsten, G. Menga, L. Mosconi. "Object-Oriented Design Patterns in reactive Systems." in J Vlissides, J. Coplien, N. Kerth (eds.), *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA. 1996.

[Andrews91] G. Andrews. Paradigms for process interaction in distributed programs ACM Computing Surveys. 23(1) (March 1991), pp 49-90.

[Bijnens+94] S. Bijnens, W. Joosen, P. Verbaeten. Sender-Initated and Receiver-Initated Coordination in a Global Object Space. *ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*. pp 14-28.

[Coad+95] P. Coad, D. North, M. Mayfield. *Object Models: Strategies, Patterns and Applications*. Prentice Hall, Englewood Cliffs, NJ. 1995.

[Coplien96] J. Coplien. *Software Patterns: Management Briefs*. SIGS Books, New York, 1996.

[Ellsberger+97] J. Ellsberger, D. Hogrefe, A. Sarme. *SDL : Formal Object-Oriented Language for Communicating Systems*. Prentice Hall, Englewood Cliffs, NJ. 1997.

[Fowler 97] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, MA. 1997.

[Fowler+97] M. Fowler, K. Scott. *UML Distilled : Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, MA. 1997.

[Frølund+94] S. Frølund, G. Agha: Abstracting Interactions Based on Message Sets. *ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*. pp 107-124.

[Gamma+85] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Resuable Object-Oriented Software*. Addison-Wesley, Reading, MA. 1995.

[Kendon90] A. Kendon. *Conducting Interaction: Patterns of Behavior in Focused Encounters*. Cambridge University Press. New York. 1990.

[Lavender +96] G. Lavender, D. Schmidt. "Active Object: An Object Behavioral Pattern for Concurrent Programming", in J Vlissides, J. Coplien, N. Kerth (eds.), *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA. 1996.

[Liu+96] L. Liu , and R. Meersman. The building blocks for specifying communication behavior of complex objects: an activity-driven approach. *ACM Transactions on Database Systems.* 21, 2 (Jun. 1996), pp 157 - 207.

[Hoare85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ. 1985.

[Miler89] R. Milner. *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ. 1989.

[Pinhanez +97] C. S. Pinhanez , K. Mase , A. Bobick. "Interval scripts a design paradigm for story-based interactive systems". Conference proceedings from *CHI 97 - Human factors in computing systems,* pp 287 - 294. 1997.

[Riehle97] D. Riehle. “Composite Design Patterns”. OOPSLA’97 Conference Proceedings, published in *ACM SIGPLAN Notices* 32 (10) (October 1997) pp 218-228.

[Rising+98] L. Rising, (eds.) *The Patterns Handbook: Techniques, Strategies, and Applications*. Cambridge University Press, New York. 1998.

[Satoh+95] I. Satoh and M.Tokoro, ``Time and Asynchrony in Interactions among Distributed Real-Time Objects,'' in *Proceedings ECOOP'95,* pp. 331-350. 1995.

[Schmidt94] D. Schmidt. Reactor -- An Object Behavioral Pattern for Event Demultiplexing and Event Handler Dispatching. *Proceedings of the First Pattern Languages of Programs, 1994.*

[Schmidt+96] D. Schmidt, T. Harrison, I. Pyarali. Asynchronous Completion Token -- An Object Behavioral Pattern for Efficient Asynchronous Event Handling. *Proceedings of the 3rd annual Pattern Languages of Programs conference.* 1996.

[Schmidt+95] D. Schmidt, C. Cranor. Half-Sync/Half-Async -- An Architectural Pattern for Efficient and Well-structured Concurrent I/O. *Proceedings of the 2nd Pattern Languages of Programs conference*. 1995.

[Shaw96] M. Shaw. "Some Patterns for Software Architectures, Programming", in J Vlissides, J. Coplien, N. Kerth (eds.), *Pattern Languages of Program Design 2*. Addison-Wesley, Reading, MA. 1996.

[van Eijk+89] P. H. J. van Eijk, C. A. Vissers, M. Diaz (eds.) in *The formal description technique LOTOS.* Elsevier Science Publishers B.V., 1989.

[Vlissides98] J. Vlissides. Composite Design Patterns, *C++ Report*, 10 (6) (June 1998), pp.45-53.