A File Transfer Problem

Objective Engineering, Inc.

699 Windsong Trail
Austin, Texas 78746
512-328-9658
FAX: 512-328-9661
ooinfo@oeng.com
http://www.oeng.com

Part I: The Problem

You must support client applications that will release software electronically. One part of that support is a mechanism that allows those applications to transfer software release files to remote sites. Therefore, you must provide software that allows a client application to transfer a specified file to a specified destination. (The client indicates the destination by providing a site name.)

Assume that your site already has low-level file transfer protocol classes, each instance of which interacts with one remote site. (These classes exist, so you need not design and implement them.) Two different protocol classes exist. One, a direct send protocol object, has the following (synchronous) public operation:

`send(f: File)`, transfers the file and returns a status of `fileTransferred` or `transferFailed`.

The other is a partial send protocol. It can transfer files of up to 100KB in size. Longer files must be transferred in pieces. To facilitate this, however, the partial send protocol has the notion of a connection; all fragments sent between the opening and closing of a connection are assumed to be parts of the same file. A partial send protocol object has three public (synchronous) methods:

`openConnection(filename)`, opens a connection to the remote site so that a file of the stated name can be transferred, returns a status of `connectionOpen` or `connectionDown`;

`sendFilePart(file)`, sends a file (or portion of a file) of up to 100KB in size, returning a status of `transferSuccess` or `transferFailure`; and

`closeConnection( )`, informing the remote site that all parts of this file have now been sent (and allowing that site to reconstruct the complete file), returning a status of `connectionClosed` or `connectionDown`.

The protocol you use to send a file must match the protocol the client site is using to receive the file. A client site may use different transfer protocols at different times, however, and so to select the correct type of protocol, you must query the remote site to determine what type of protocol it is currently using. You must then create a protocol object of the appropriate type. Assume that the constructors for the Direct Send Protocol and Partial Send Protocol classes take the remote site's address (such as an IP address) as an argument.

When transferring a file, each protocol object will place the transferred file in a special directory at the remote site. You can assume that software at that site will handle the receipt of files in that directory.

Because new protocols may by introduced in the future, you should make your design as tolerant of new protocols as possible.

Your file transfer package should handle transfers at any of three priorities: background, normal, and emergency. Files sent with emergency priority should be sent before files with normal priority, files with normal priority before files with background priority. Once a file is handed to a protocol object, however, its transfer cannot be interrupted.

You should allow transfers with retry counts. In the case of transmission errors, a transfer with a non-zero retry count will be retried that number of times. The default retry count should be zero.

Your package should also support transfers at (or about) time t. If a client asks to transfer a file at midnight, for example, that request should be postponed until that time.

A client must be able to query the status of a transfer. Possible status values include pending, in progress, completed successfully, and completed unsuccessfully. In addition, a client must be able to cancel a pending transfer. (An attempt to cancel a transfer that is no longer pending, however, should be ignored.)

You can support broadcasts to multiple sites, but you need not do so. (This feature is on someone’s wish list.) If you permit this, you must determine how you will handle the status of the transfer (because the file may have been transferred to some sites but not to others) and the notification (such as whether a single notification or multiple notifications are sent).

Part II: A Solution

The document contains a description of a file transfer facility. This facility permits client programs to transfer files to remote sites that employ different transfer protocols. It allows a client to specify a priority, a retry count, and a time at which to start the transfer. In addition, a client can cancel a pending transfer request, and can query the status of a transfer request.

The solution presented here includes a model of functional requirements and a design. The functional requirements are defined in terms of a use case diagram and textual descriptions of each use case. The requirements model also includes an activity diagram for the most complicated use case, the transfer of a file.

The design is cast as a class diagram and a set of interaction diagrams.

A Requirements Model

A client program can make three types of requests:

a) It can request that a file be transferred;

b) It can query the status of a transfer; and

c) It can cancel a pending transfer.

Several variations of the first use case exist based on the arguments supplied by the client. For example, a client can specify a retry count, a particular time at which to send a file, etc. That single use case represents the entire transfer process, and so it includes several possible outcomes (the file is transferred successfully, the transfer fails, and the transfer is canceled).

A client program and a remote site are the two types of actors in this system. Figure 1 contains an initial use case diagram for this problem.

The textual descriptions of these use cases are:

Request Transfer. The client program requests that a specific file be transferred to a specific remote site. The client may include a priority, a retry count, and a subsequent time at which to transfer the file. The file is transferred to the remote site.

Query Transfer Status. The client program requests the current status of a specific transfer. The client is returned an indication of whether the transfer is pending, in progress, or has completed successfully or unsuccessfully.

Cancel Transfer. The client program asks that a specific transfer be canceled. If the transfer is pending, it must be canceled.

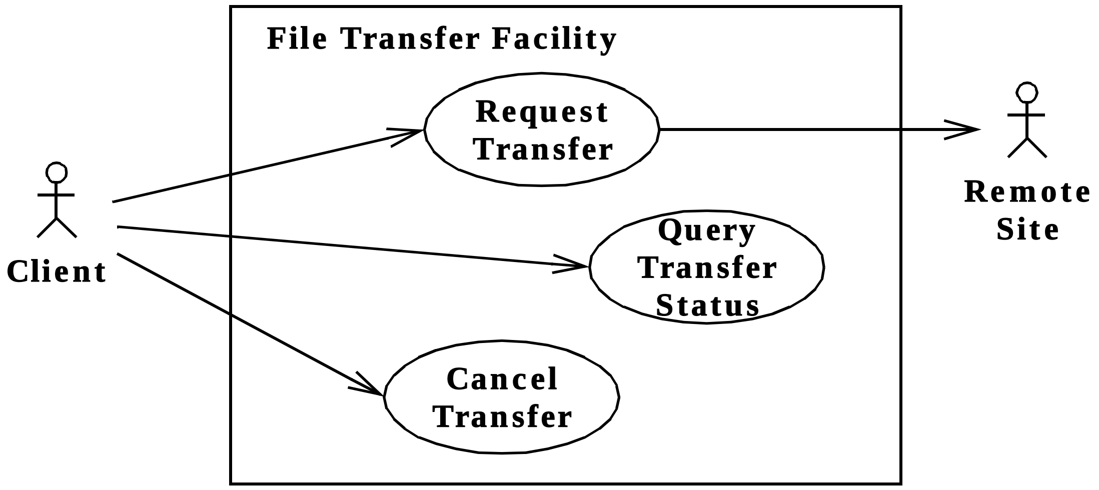

Figure 1: A use case diagram for file transfer.

To clarify the details of the Request Transfer use case, you can model that function with an activity diagram. Figure 2 contains such a diagram. As the figure indicates, the initial activity in the use case is the creation of the request itself. In particular, your transfer facility must create an internal representation of the client's request.

If the file is to be transferred now, the request is immediately queued by the Queue Request activity. Conversely, if the file should be transferred later, the transfer is delayed. The Delay Request activity is simply a "waiting" activity. While located in this activity, the request will age until either its transfer time is reached, in which case it is queued by the Queue Request activity, or until it is canceled (by an application of the Cancel Transfer use case).

When the request reaches the head of the (logical) transfer queue, it is transferred by the Transfer File activity. This activity includes the selection of the protocol and the actual transfer itself. Observe that both the Delay Request and Queue Request activities will complete in the "request is canceled" state when the Cancel Request use case is invoked during those activities.

The Design

First, consider the interface to be provided to client programs. What type of interface should you provide? Assume that the client is to use direct remote method invocations rather than an application protocol, such as the HyperText Transfer Protocol (HTTP). (A later portion of this document will revisit that assumption.) You must therefore design a direct interface through which a client can initiate, query, and cancel transfers.

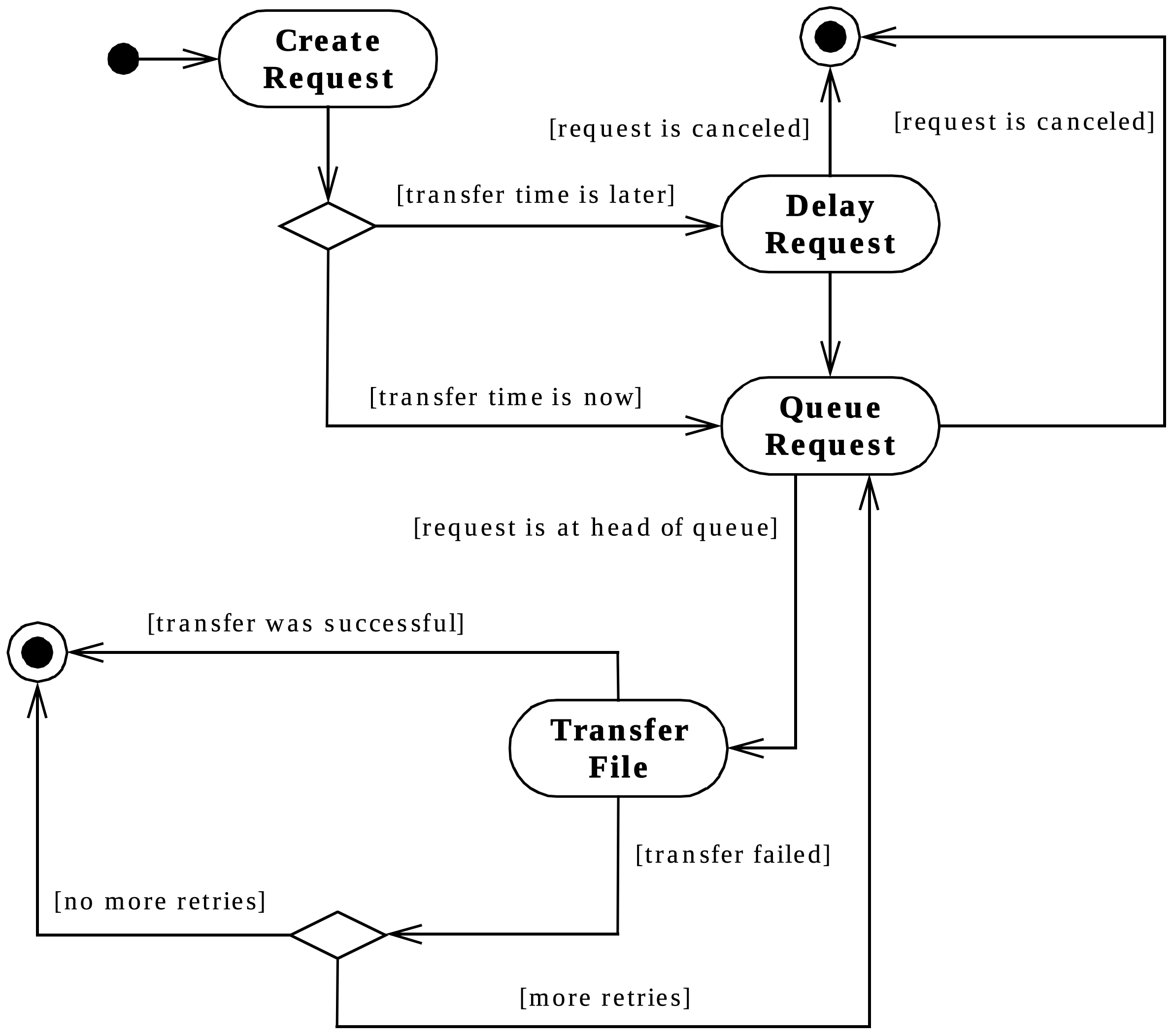

Figure 2: An activity diagram for the Request Transfer use case.

One possible interface is a *facade* [GHJ&V, pp.185-193] that offers all the capabilities required by a client. The File Transfer Facility facade class provides methods to request a transfer, check on the status of a transfer, and cancel a transfer. When a client program issues a transfer request, the facade must return an identifier for that request. The client can subsequently use that identifier to check and cancel the request. A File Transfer Facility facade class is shown in Figure 3.

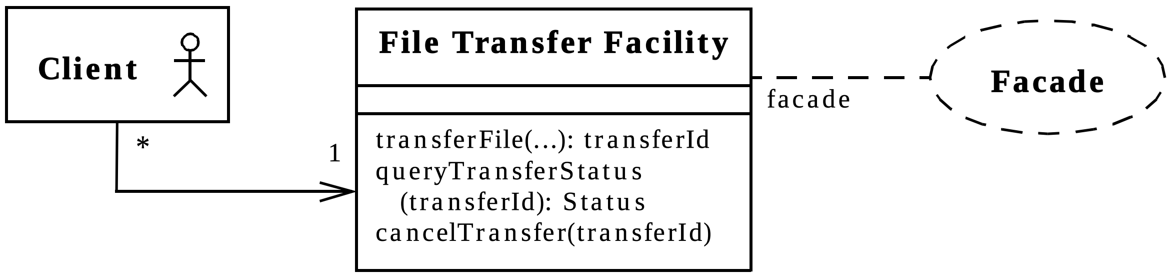

Figure 3: A File Transfer Facility facade class.

The `transferFile` method in the File Transfer Facility class must be overloaded to include variants for all combinations of parameters a client may provide. While these variants are absent from Figure 3, the client must be able to specify any combination of a time at which to send the file, a priority, and a retry count.

A minor disadvantage of the facade class is the need to create and maintain an identifier for each transfer. Because a client may query the status of a request at any time, furthermore, the facade must maintain persistent information about the success or failure of a completed transfer.

As an alterative to the File Transfer Facility facade, you could provide the client with a File Transfer Request class. To initiate a transfer request, a client creates a File Transfer Request instance and invokes its `execute` method. A File Transfer Request object also has methods to query its status and to cancel it. This class is depicted in Figure 4.

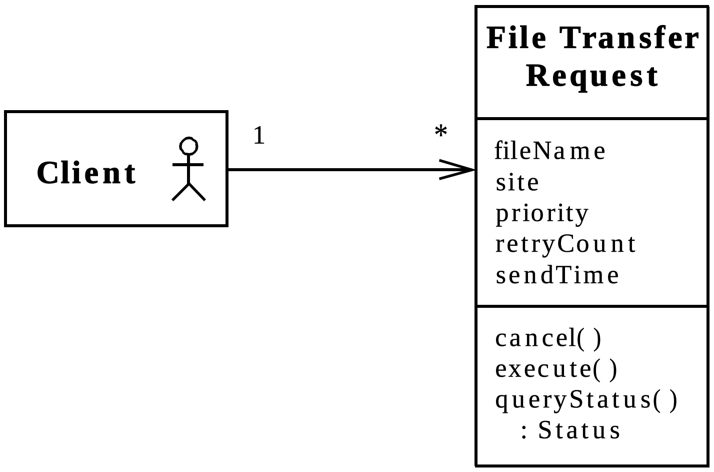

Figure 4: The File Transfer Request class.

The File Transfer Request class is Figure 4 is an application of the Command design pattern [GHJ&V, pp. 233-242]. In this pattern, a request is treated as an object that encapsulates the state and behavior required to carry out the request. As a result, the invoker of the request need not know anything about how the request is effected. In this case, however, the request is a long-lived action (something similar to a transaction) in that the overall execution of the request (i.e., the interval from the initial request to the actual transfer of a file) may span several hours.

Aside from the actual interface provided to client programs, the designs with a facade and with a File Transfer Request class are very similar. In each case, a separate thread or process must carry out the transfer. (Otherwise, the client will be blocked until the request completes.) In a design based on the File Transfer Request class, a Request instance must somehow initiate such a process or thread.

Figure 5 shows one way in which that initiation can be handled. The figure is an elaboration of Figure 4 that includes a File Transfer Execution class. An instance of this class will run as a separate process or thread and will carry out the actual transfer. The double-headed association indicates the need for an Execution to communicate status information back to its Request. For example, when a transfer has completed, the Execution for that transfer must inform its File Transfer Request of the final resolution of the transfer. The actual communication mechanism between the two must be tolerant of the disappearance of the object on either side. (That is, a File Transfer Request should not fail when issuing a request to a File Transfer Execution that has disappeared, and vice versa.)

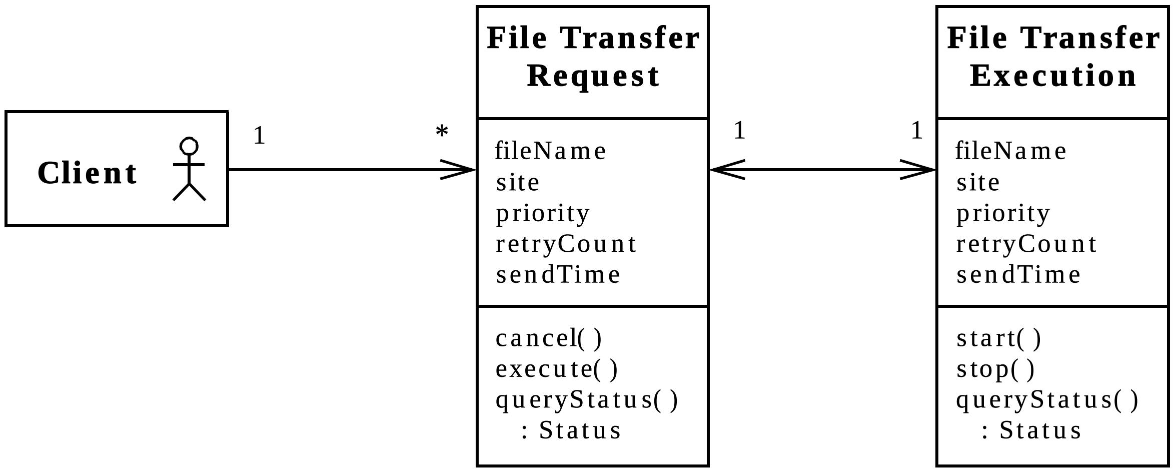

Figure 5: The File Transfer Request class.

The disadvantage of the solution in Figure 5 is that each transfer requires a separate thread or process. For the most part, however, each such process or thread is simply awaiting its turn to be transferred. As an alternative, you could have the File Transfer Request ask a central agent to post the transfer request. That agent would provide methods to post, cancel, and query the status of a request. This is exactly the functionality provided by the File Transfer Facility facade in Figure 3!

The design described from this point forward employs the facade class in Figure 3, as well as the following classes identified using abstraction:

File Transfer Request. Unlike the class of the same name in Figures 4 and 5, an instance of this class is created by the File Transfer Facility facade class when a client asks to transfer a file. The instance holds the relevant information about the request.

Transfer Queue. This is a priority-ordered queue that holds the File Transfer Request instances. Entries are added to the Transfer Queue by the File Transfer Facility facade.

File Transfer Agent. An instance of this class takes File Transfer Requests from the front of the Transfer Queue and executes those transfers. The instance runs as a separate process. If you desire to execute multiple transfers concurrently, this instance could include several threads, each of which is executing one transfer.

Protocol Factory. The File Transfer Agent uses an instance of this class to create the appropriate Protocol object for a specified remote site.

When asked to transfer a file, the File Transfer Facility must create a File Transfer Request instance that describes the transfer request, after which it must place that Request object on the Transfer Queue. The Transfer Queue orders its entries by priority. It might be implemented as a single Queue or a triad of internal Queues, one for each priority level. (Alternatively, you could have a Queue for each remote site, but this permits unusual transfer sequences in situations where one site has only low-priority or normal-priority transfers in its Queue whereas another site has several requests queued at emergency priority.)

Figure 6 depicts the File Transfer Request class and its relationship with the Request Queue class. The File Transfer Facility maintains a list of File Transfer Request objects keyed by a transfer identifier. (This identifier is returned to the client program when the Request is created.) The File Transfer Facility then uses the Queue's `addEntry` method to add the Request to the Queue. If the client subsequently cancels the transfer, the facade will invoke the Queue's `removeEntry` method to (attempt to) remove that Request from the Queue.

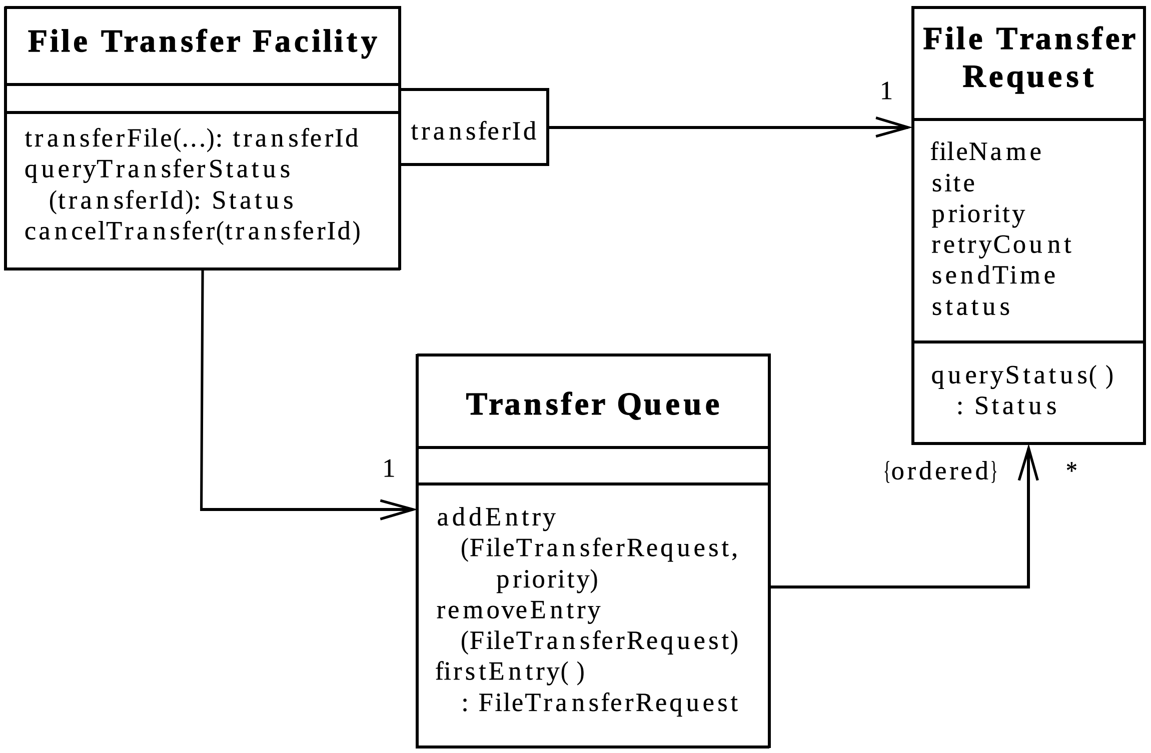
File Transfer Facility
transferFile(...): transferId
queryTransferStatus (transferId): Status
cancelTransfer(transferId)
transferId
1
File Transfer Request
fileName
site
priority
retryCount
sendTime
status
queryStatus( ) : Status
{ordered}
*
Transfer Queue
1
addEntry (FileTransferRequest, priority)
removeEntry (FileTransferRequest)
firstEntry( ) : FileTransferRequest

Figure 6: The File Transfer Request and Request Queue classes.

Note that when it creates a File Transfer Request with a later transfer time, the File Transfer Facility does not immediately add that Request to the Queue. Rather, it starts a timer that will expire at the desired transfer time. When the timer expires, the facade (or a helper object) adds the Request to the Queue. The mechanism to achieve this timing is not depicted in Figure 6.

Figure 7 contains a collaboration diagram for a scenario in which a client issues a request to transfer file named `foo` to a site called "JoesBar." The File Transfer Facility creates a File Transfer Request instance with transfer identifier 03175, then places that Request object on the Request Queue (with normal priority).

How does the actual transfer described by a File Transfer Request instance occur? An instance of the File Transfer Agent class is responsible for enacting the transfer. When that instance is ready to conduct the next transfer (perhaps when the previous transfer has completed), it removes the File Transfer Request that occupies the head of the Request Queue (by invoking the Queue's `firstEntry` method), then carries out that request. Figure 8 includes the File Transfer Agent class. As noted above, an Agent runs as a separate process (as indicated by the bold box and «`process`» stereotype in the figure) and could have multiple threads if you wish to permit multiple concurrent transfers.

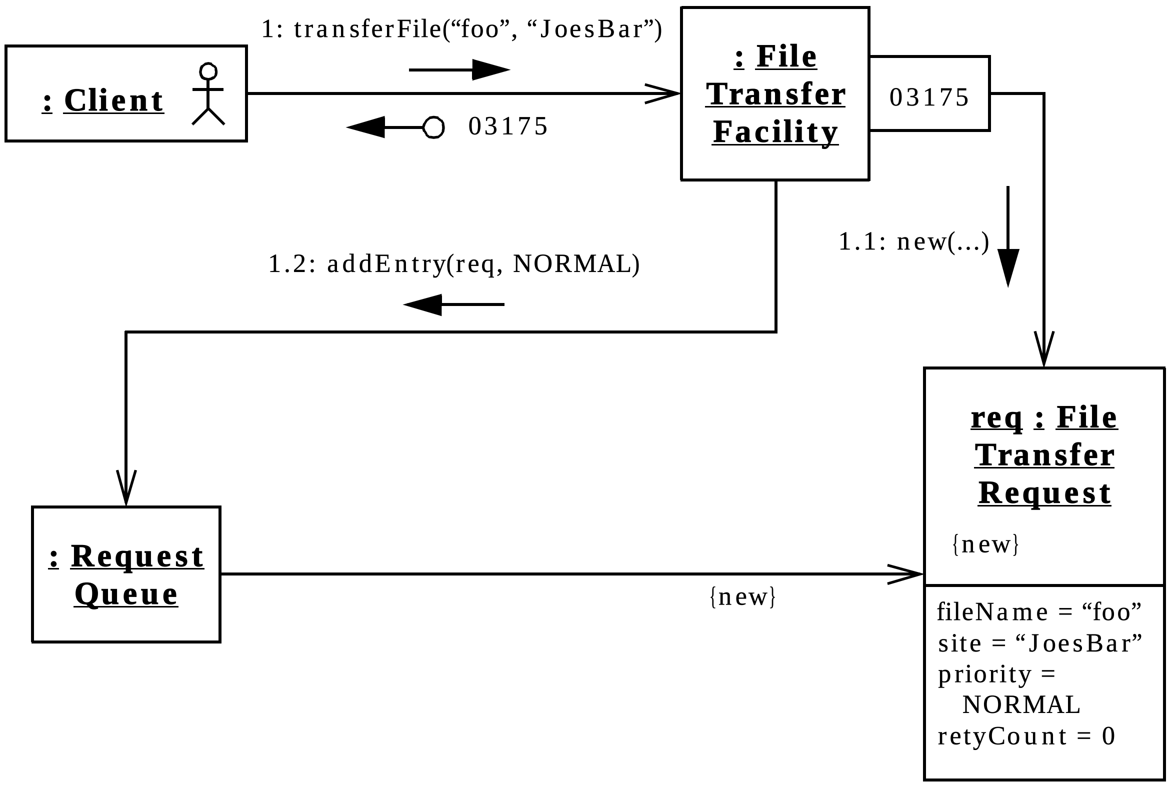

Figure 7: A collaboration diagram depicting the creation of a Request.

The File Transfer Agent will use a Protocol Factory object to obtain the appropriate Protocol object for a Transfer Request. Figure 9 contains a class diagram for the design to this point. Still pending in this discussion is the definition of the Protocol Factory and the Protocol classes.

Recall that the two existing Protocol classes have different interfaces. The Direct Send Protocol class has a single `sendFile` method, whereas the Partial Send Protocol class defines three methods required to send a single file. To provide a uniform interface for both classes (or, put another way, to isolate the differences in these interfaces), you can apply the Adapter design pattern [GHJ&V, pp. 139-150].

An *object adapter* is placed atop each specific Protocol instance. That adapter, an instance of the Partial Send Adapter or Direct Send Adapter class, offers a uniform interface, a `transferFile` method that takes a file as an argument and returns the status of the transfer. The Partial Send Adapter and Direct Send Adapter classes define the translation from that interface to the methods

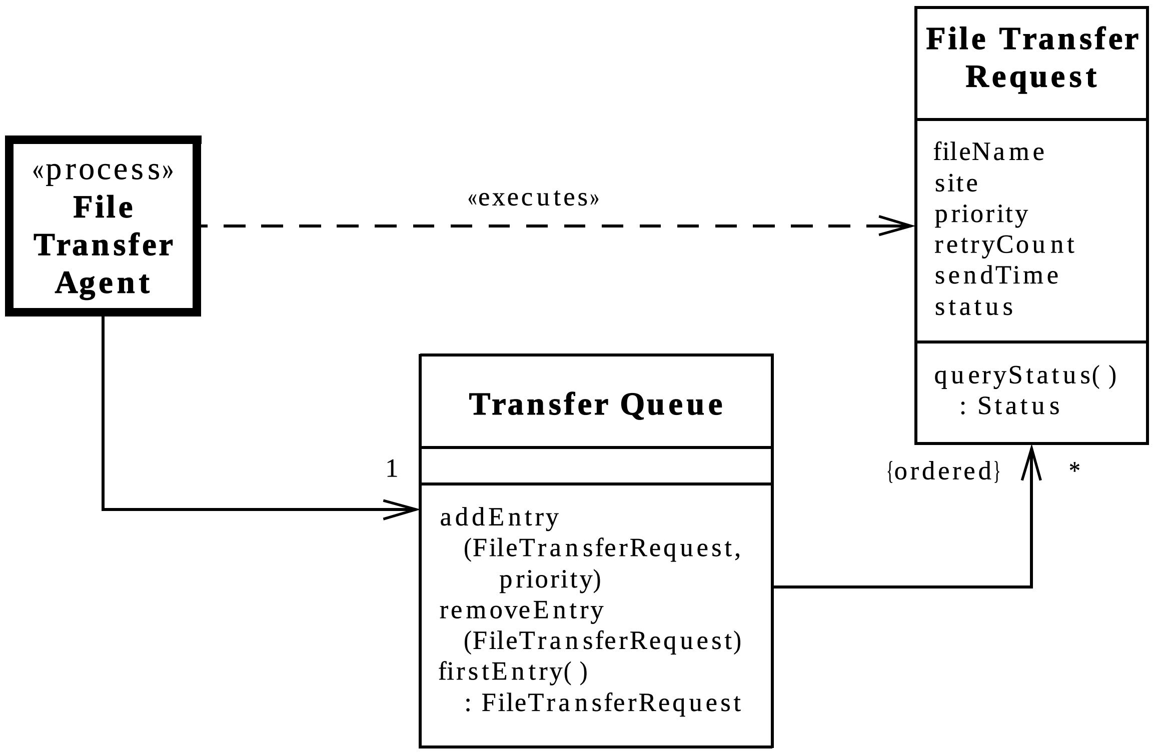

Figure 8: The File Transfer Agent class.

provided by Partial Send Protocol and Direct Send Protocol, respectively. For example, when you invoke `transferFile` in a Direct Send Adapter, it will call `sendFile` in its underlying Direct Send Protocol instance.

Both adapter classes are derived from a common File Transfer Protocol interface class that defines the common method interface, `transferFile`. The adapter classes and the interface class are illustrated in Figure 10. Recall that an interface class (that is, a class with the «`interface`» stereotype) contains only public, abstract methods. The class name (File Transfer Protocol) and its method name (`transferFile`) are not italicized in Figure 10 because, by virtue of being an interface class, the class and its methods must be abstract.

The Protocol Factory class defines a single method, `makeProtocol`, that takes a site name as an argument and returns a File Transfer Protocol reference. Internally, a Protocol Factory instance must query the remote site with the specified site name to determine which protocol is currently in use at that site. It must then create the appropriate Protocol instance and its corresponding Adapter instance.

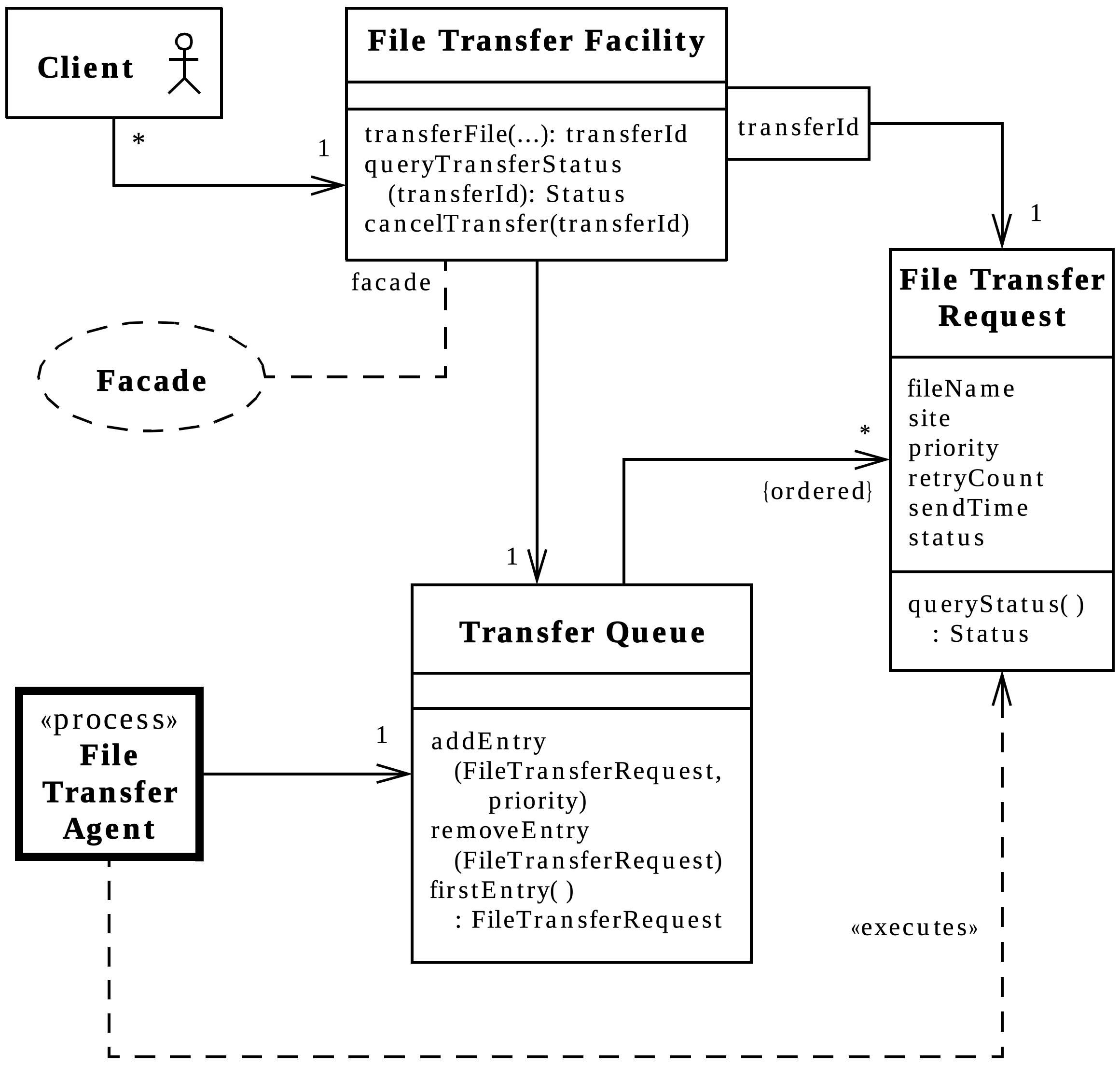

Figure 9: The evolving design.

> **Note:** This figure and others in this solution assume that a Java-style or Smalltalk-style superclass reference is returned. For C++, you would return a base class pointer.

Figure 11 depicts the Protocol Factory class. The single Protocol Factory instance maintains a local Remote Site Proxy object for each remote site. Each such Proxy object serves as a surrogate for an object on the remote site, encapsulating the mechanism required to communicate with that remote object. The Protocol Factory selects a particular Remote Site Proxy instance using the

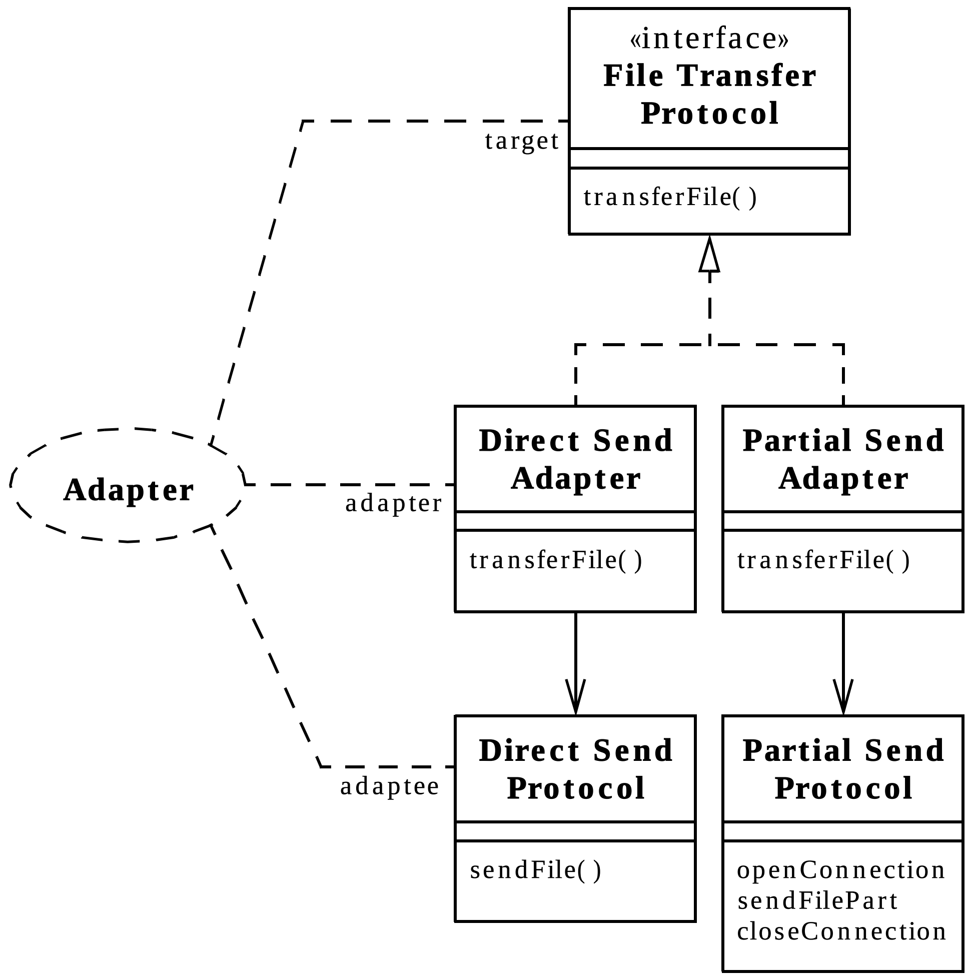

Figure 10: The adapter classes.

site name as a key. It then asks that Proxy for the remote site's protocol (and, by some magic, the Proxy connects to its remote object to determine a reply). If the remote object implements the same interface as does its local proxy, then the use of a Remote Site Proxy is an application of the Proxy design pattern [GHJ&V, pp. 207-217].

Although not included in Figure 11, the Protocol Factory class has a dependency on the two Adapter classes as well as the two Protocol classes. A Factory must create an instance of both an Adapter and a Protocol.

The class diagram for the complete design is the combination of Figures 9 and 11. Figure 12 contains a collaboration diagram for the following scenario:

1. The Transfer Agent removes a File Transfer Request object from the head Request Queue.

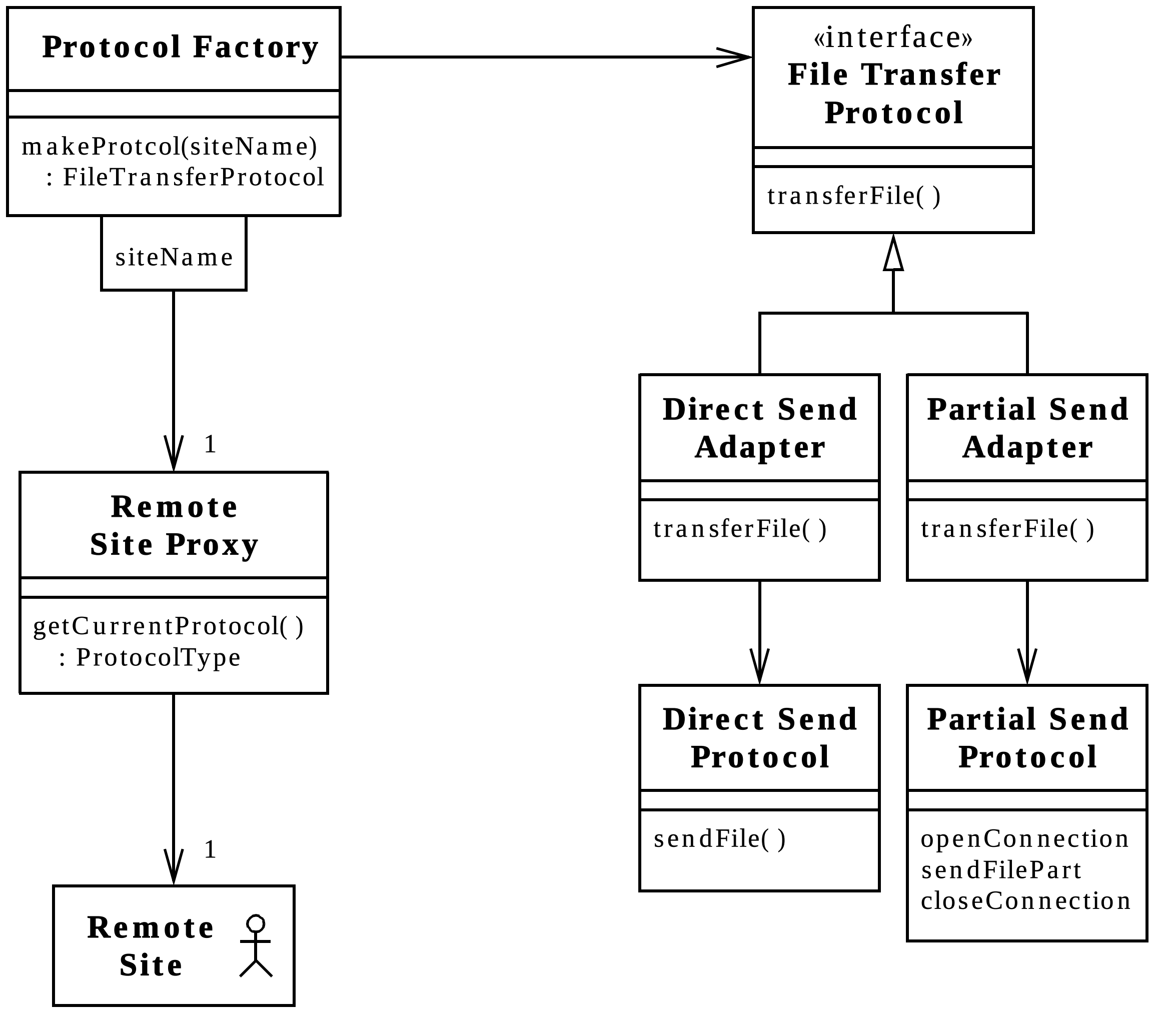

Figure 11: The Protocol Factory class.

2. The Transfer Agent obtains the remote site name from the Request, then asks the Protocol Factory to create a Protocol object for that site.
3. The Protocol Factory queries the remote site to determine the protocol that site is currently using, then creates the appropriate Protocol and Adapter instances. It returns an Adapter reference to the Transfer Agent.
4. The Transfer Agent asks the Adapter to transfer the file.

The collaboration diagram in Figure 12 does not model the Remote Site Proxy's communication with the remote site.

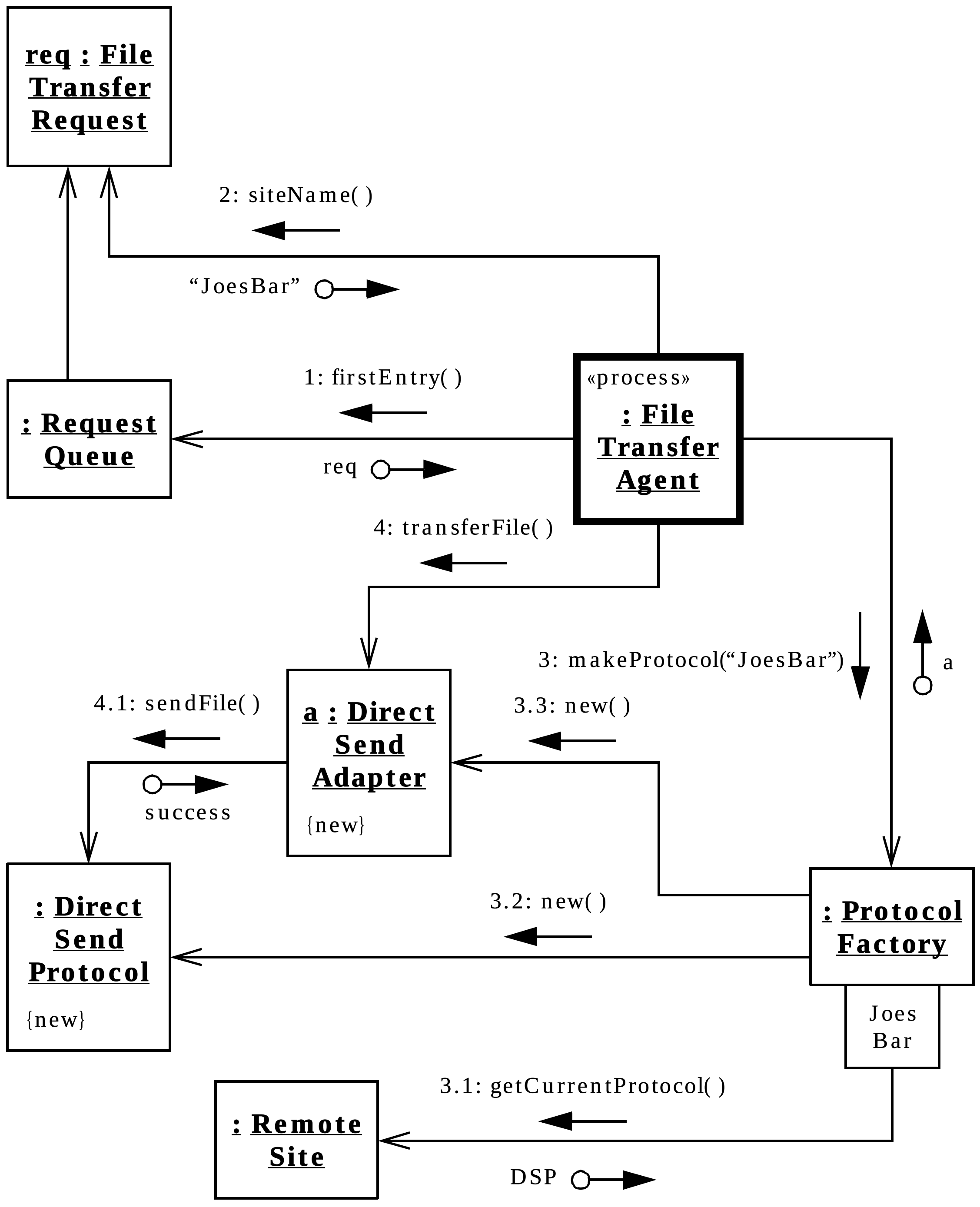

Figure 12: A collaboration diagram depicting a file transfer.

Recall that the bold border around the File Transfer Agent instance in the figure indicates that the instance is an active object. An active object has its own thread of control in the form of a task, process, or thread. In this case, the File Transfer Agent instance (like its class in previous figures) is stereotyped to indicate it is a process.

The Transfer Agent handles any errors that occur when sending a file. For example, if the transfer of a file fails, the Transfer Agent will continue to attempt to send the file until the File Transfer Request's retry count is exhausted. Because such a failure may be due to a problem at the receiving site, the Transfer Agent probably should place the Transfer Request back onto the Transfer Queue. (If the remote site is down, resending the file immediately will produce an identical, unsuccessful result.) Likewise, if the Protocol Factory is unable to contact the remote site to determine which protocol to employ, the Transfer Agent should re-queue the Transfer Request.

Variations

Recall the assumption that clients use remote method invocations to call methods on application objects (in this case, on a facade instance). Suppose, however, that clients are browsers using the HyperText Transfer Protocol (HTTP) to communicate with your file transfer facility. How would that affect the design?

In an HTTP-based solution, a client interface (such as a browser) interacts with an application by issuing an HTTP request. A web server on the application machine receives the request and dispatches it to a server-side entity, such as a Java servlet or a CGI script, responsible for processing such requests. A typical design has one web page for each use case initiated by an actor (as well as a "home page" presented when the actor first initiates the application). Each web page in turn might be associated with one servlet or CGI script that handles requests for that page.

To support browser-based clients of the file transfer facility, you replace (or complement) the (RMI-oriented) File Transfer Facility facade with a servlet-based or CGI-based approach. For a servlet-based approach, for example, you design a web page for each use case (as well as a "home page" that initially displays), and you introduce a servlet to process each web page. Figure 13 depicts a client's interaction with those servlets. The "stacking" of the servlet class and its generic name are intended to indicate that multiple transfer servlets, the specifics of which do not appear in the diagram, will be a part of this design.

Consider Figure 6 again. The class diagram in this figure depicts the File Transfer Facility facade's interactions with other transfer classes. Figure 14 illustrates the analogous interactions of the Transfer File Servlet class with the

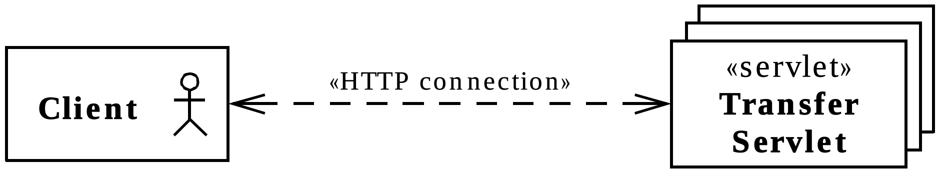

Figure 13: A part of a class diagram for HTTP-based interactions.

transfer classes. An instance of this servlet will handle applications of the Transfer File use case. When handling a transfer request, the servlet must create a File Transfer Request instance, and it must add that instance to the Transfer Queue. Put another way, the servlet must carry out the same steps for this use case as did the facade in the RMI-based design.

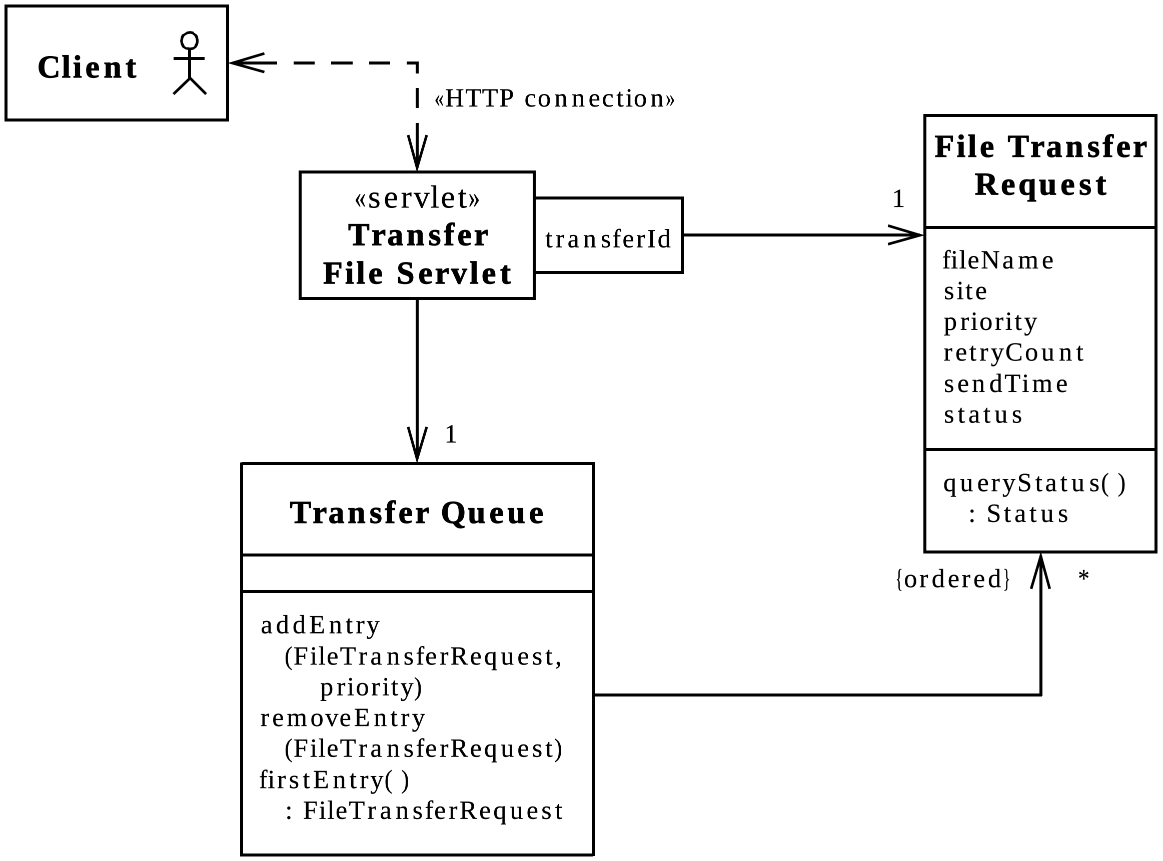

Figure 14: The Transfer File Servlet class.

A servlet-based design also includes a servlet for each of the home page, the Cancel Transfer use case, and the Query Transfer Status use case. Each servlet would mimic the steps of the facade when carrying out that use case. Note that the servlets might in fact be implemented as JavaServer Pages. Additionally, you might use one or more JSPs to present the results of each servlet request. A discussion of the detailed design of servlets and JSPs for this application is beyond the scope of this paper.

References

[GHJ&V] E. Gamma, R, Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Trademarks