Abstract Session

An Object Structural Pattern

Nat Pryce
Department of Computing,
Imperial College,
London, UK.

np2@doc.ic.ac.uk

Abstract

Object-oriented frameworks are structured in terms of client/server relationships between objects; an object's services are invoked by client objects through the operations of its interface. A common design requirement is for a server object to maintain state for each client that it is serving. Typically this is implemented by returning handles or untyped pointers to the client that are used to identify the per-client data structure holding its state. The lack of strong typing can lead to obscure errors that complicate debugging and maintenance. This paper presents the *Abstract Session* pattern which allows objects to maintain per-client state with full type safety and no loss of efficiency.

Intent

Allow a server object with many client objects to maintain per-client state and maintain type-safety.

Also Known As

Service Access Point (SAP), Context, Service Handler.

Motivation

Object-oriented frameworks are structured in terms of client/server[1] relationships between objects; an object's services are invoked by client objects through the operations of its interface. Typically, object interactions are defined in terms of abstract interfaces, which increases the reusability and extensibility of the framework because the set of client and server types that can be used together is not bounded and can easily be extended by framework users.

A common design requirement is for a server object to maintain state for each client that it is serving. When the interactions between the server and client objects are defined in terms of abstract interfaces the per-client state cannot be stored in the client objects themselves because different server implementations will have to store different information. This problem is usually solved by using the *Session* pattern [1] which imposes a three-phase protocol upon the interactions between the client and the server:

1. Clients make an initial "request" call to the server when they begin to use its services. The server responds by allocating a per-client data structure for the new client and returns an identifier of that data-structure to the client.

[1] In the context of this pattern, the term "server" is used to denote an object which provides a service to "client" objects within the framework and in no way implies any notion of distribution between clients and servers.

2. The client passes the identifier it received as an argument to subsequent operations of the server. The server uses that identifier to find the per-client information it holds about the caller.
3. When the client has finished using the server, it makes a final "release" call to the server, passing in the identifier that the server uses to find and deallocate the appropriate per-client data-structure.

Implementing the *Session* pattern involves resolving the following forces:

- **Per-Client State**: The server object must be able to store state about each client that is making use of its services.
- **Efficiency**: Invocation of service operations should involve minimum run-time overhead.
- **Safety**: The implementation should catch the use of incorrect session identifiers. Ideally, such errors should be caught at compile time.

Implementations of the *Session* pattern typically use one of two methods to map from identifiers to the data structures used internally to store per-client state.

- **Untyped Pointers**: Identifiers are defined to be untyped pointers that the server object initialises to point to an implementation-specific data structure and casts to the appropriate type on each invocation. For example, the Win32 API [2] uses untyped pointers to identify windows and other system resources allocated for an application.
- **Handles**: Identifiers are defined to be values of simple types, such as integers, or opaque values. The server uses a private associative container to map between identifiers and implementation-specific data structures and must perform a look-up each time a client makes a request. For example, the UNIX file API [3] uses integer handles to identify open files.

The choice of whether to use untyped pointers or handles is one of safety versus run-time efficiency. Untyped pointers are efficient but passing an incorrect pointer to a server could cause the program to corrupt memory or crash. Using handles as indices to an associative container allows incorrect identifiers to be detected at runtime but at the expense of performing a look-up on each invocation, which can be costly if many invocations are made during a time-critical part of a program. Both approaches are unsafe; a client can pass an incorrect identifier to the server without the error being caught at compile time.

Example Problem: Network Communication

Consider a communication protocol service that provides reliable transmission of data streams, such as is provided by the TCP/IP or Novell SPX protocols. The protocol manager itself is implemented as an object that provides services for connecting to remote endpoints and accepting incoming connections. Client objects wanting to connect to a remote endpoint must request a connection from the protocol object before being able to transmit and receive data. The protocol object must maintain state information, such as unacknowledged packets and flow-control information, for each connection and so clients must have some way to identify the connection that they are using to transmit or receive data. When the client has finished transmitting data, it must close the session to release any resources held for the session by the protocol manager.

The following code demonstrates a possible error when using untyped handles. The code uses the UNIX Sockets API [14] to open a network connection to a remote endpoint and transmit integer values. The code is incorrect; the first argument to the write function in line 14 should be the socket identifier `sock` but the loop counter is erroneously used instead. Because UNIX uses integer handles to identify sockets, the bug cannot be caught by the compiler and only manifests itself as errors at runtime.

```
1    // Select a protocol and get a socket - i.e. request a session from
```

```
2    // the selected protocol
3
4    int sock = socket( SelectProtocol(), SOCK_STREAM, 0 );
5
6    // Connect to a remote endpoint
7
8    connect( sock, ... );
9
10   // Send 10 integers to the remote endpoint
11
12   for( int i = 0; i < 10; i++ ) {
13        int number = GetANumberFromSomewhere(i);
14        write( i, &number, sizeof(number) );
15   }
16
17   // Release the protocol session
18
19   close(sock);
```

Solution

The *Abstract Session* pattern provides a way for an object to store per-client state without sacrificing type-safety or efficiency. A protocol service object, rather than providing a client with a handle to be passed as an argument to the operations of its abstract interface instead creates an intermediate "session" object and returns a pointer to the session object back to the client. The session object encapsulates the state information for the client which owns the session and is only exposed to the client as an abstract interface through which the client can access the protocol's functionality with full type-safety. When the client invokes operations of the session, the session co-operates with the service object to complete the operation. When the client has finished using the protocol, it "releases" the session, after which any pointers o the session object are invalid.

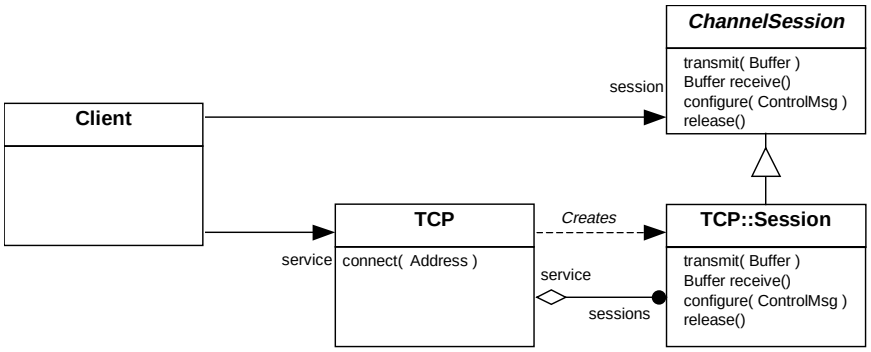

Figure 1. Service and Session classes for the TCP protocol and an example Client class

The class diagram in Figure 1 shows how the *Abstract Session* pattern can be used in an implementation of the TCP protocol. The TCP class represents the TCP protocol service and provides an operation, *connect*, with which client objects can request a connection to a remote endpoint The TCP service then creates a new TCP::Session object, initiates the handshaking protocol with the remote endpoint and returns a pointer to the ChannelSession interface to the caller. The client object can call operations of the ChannelSession interface to transmit data over the connection.

As shown in Figure 2, the TCP object keeps track of the sessions it has created. It demultiplexes each packet received from the IP layer by passing it to the session identified in the packet's header. The session performs protocol processing and then queues the data for reading by the client object that owns it.

When the client object wants to close the connection, it calls the *release* operation of its session, after which all pointers to the session are invalid.

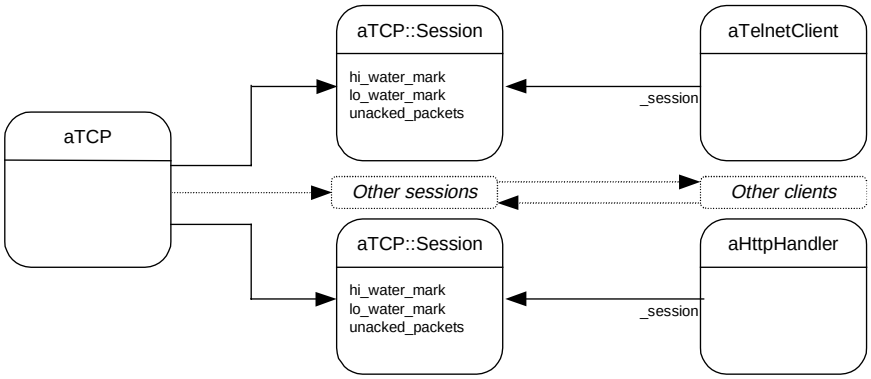

Figure 2. Dynamic Structure of Objects Implementing the TCP Protocol

The *Abstract Session* pattern successfully resolves the forces outlined above:

- **Per-Client State**: The server object stores state about each client in the session object associated with that client.
- **Efficiency**: Calling operations of a session object imposes no performance penalty compared to passing untyped pointers to operations of the service object and is more efficient than mapping handles to data structures using some hidden table.
- **Safety**: The client object invokes operations on the abstract interface of the session object and so never needs to manipulate untyped handles. The session object knows the full type of the service object that created it, and vice versa, so the interactions between them are completely type safe. When the client invokes an operation on the session, the session can call private operations of the service object passing the manager a fully-typed pointer to itself. The manager does not have to perform unsafe casts to access the state it stores within the session objects.

Applicability

Use the *Abstract Session* pattern when:

- Interactions between server objects and client objects are defined in terms of abstract interfaces, and...
- Server objects must maintain state for each client object which makes use of their services

Structure and Participants

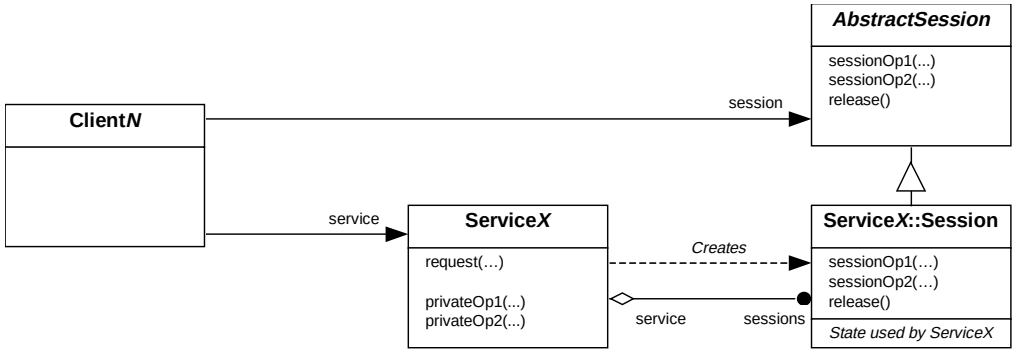

Figure 3. Class Structure of the *Abstract Session* Pattern

- **Service*X*** (TCP::Service)
  - Classes of server objects that create session objects (of type ServiceX::Session) for clients which are bound to them.

- **Abstract Session** (ChannelSession)
  - The interface which through which clients bound to a server object make use of the service provided by that object.
- **Service*X*::Session** (TCP::Session)
  - The session classes used by the Service*X* classes to store information about clients which are bound to them. Clients invoke the AbstractSession interfaces of these objects to interact with the server objects to which they are bound.
- **Client*N*** (HttpHandler, TelnetClient)
  - Objects that are making use of Service*X* through the AbstractSession interface.

Collaborations

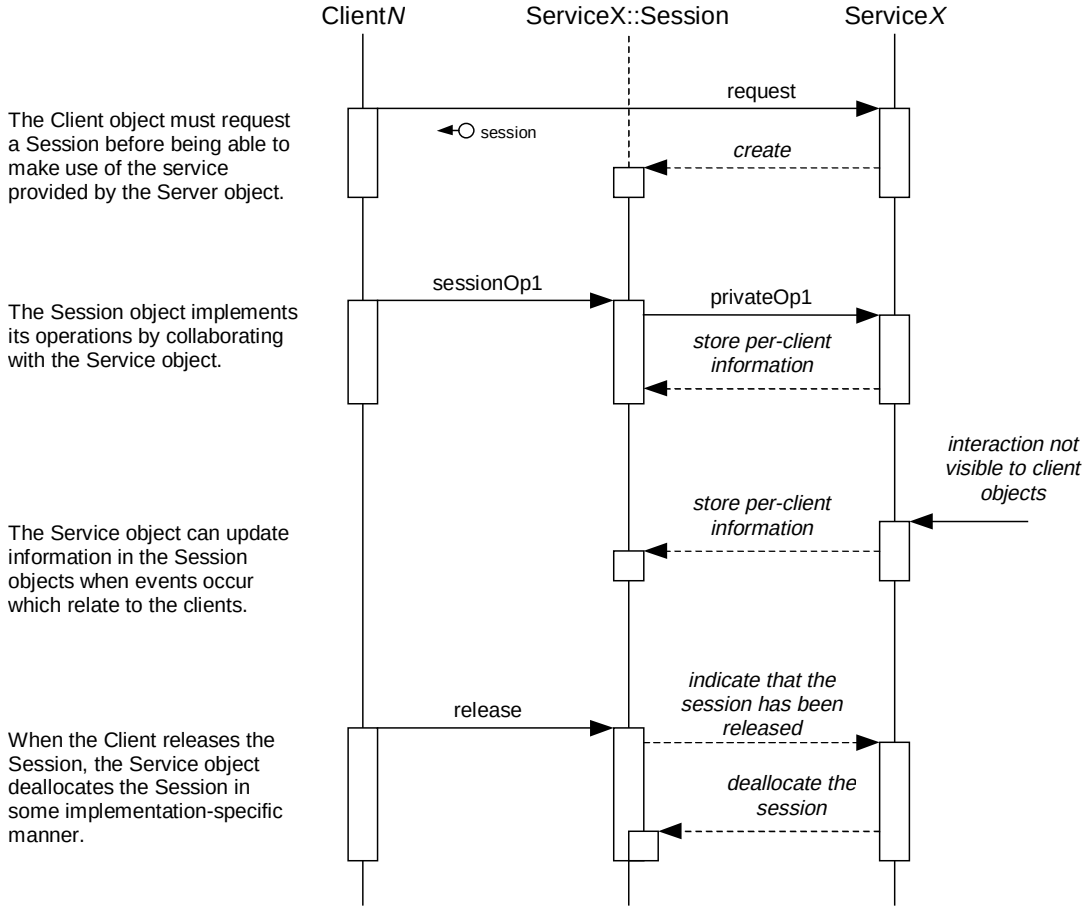

Figure 4. Object Interactions in the *Abstract Session* Pattern

Figure 4 illustrates the collaborations between objects in the *Abstract Session* pattern.

- A client object that wants to make use of a server object requests a session from the server. The server object creates a session object, which conforms to the AbstractSession interface, initialises the session with information about the client and returns a pointer to the session back to the client.
- The client makes use of the service provided by the server object by invoking the operations of its session object's AbstractSession interface. The session object co-operates with the server object in order to complete the invocation.

- The server object uses information stored in the session object to process requests from its clients and can update the per-client information when events occur behind the scenes that relate to the client.
- When a client object has finished using the service provided by the server object, it calls the release() operation of its session. Forcing clients to release sessions by calling the release() operation allows a service to hide the way it allocates session objects from its clients. A service could allocate sessions on the heap, in which case the release() operation would free the session object, or a service could have a fixed number of sessions in an array, in which case the release() operation would update a record in the service object of which sessions were unused and could be handed out to new clients.

Consequences

The *Abstract Session* pattern has the following advantages:

- *Type-Safety*. Interactions between clients and sessions and between sessions and servers are all completely type-safe. This reduces the likelihood of obscure errors, making the code easier to debug and maintain.
- *Performance*. The interactions between clients and sessions are as fast as or faster than those using unsafe methods such as untyped pointers or handles.
- *Flexibility.* The use of abstract interfaces and encapsulation of per-client state within each server class reduces coupling between client classes and server classes. A client can make use of any server that creates sessions that implement the AbstractSession interface.
- *Extensibility.* The pattern makes it easy to add server classes to the system; such extensions do not require that existing client and server classes are changed.

The *Abstract Session* pattern has the following disadvantages:

- *Dangling Pointers*. It is possible that a session might be referenced and used after the it has been released.
- *Distribution*. It is difficult to pass a session object from the server to the client if the client and server exist in different address spaces. When creating a session, the server must send information to the client to allow the client to create a *Proxy* [11] session in its local address space. Distributed object brokers, such as CORBA [5] or DCOM [6] can be used to implement this functionality.
- *Multiple Languages*. It is difficult to call *Abstract Sessions* from another language, especially from languages which are not object oriented. This can be solved by writing an *Adaptor* [11] layer that hides session objects behind a set of procedures callable from the other languages and uses one of the unsafe implementations of the *Session* pattern to identify session objects.

Implementation

The following implementation issues are worth noting:

1. *Use of the heap.* Allocating and deallocating memory from the heap is an expensive operation. Forcing clients to discard sessions via the release() method in the AbstractSession interface, rather than an explicit deallocation of the session object, gives server objects more flexibility in the allocation of session objects. For instance, if the server allocated the session from the heap, the release() operation would delete the session object. However, a server might preallocate sessions in a cache; when a client invoked the release() operation of a session it need only mark that the session is unused rather than perform a heap deallocation. If a server object does not need to store any information about its clients, it can implement the AbstractSession interface itself, perhaps using private inheritance; in this case, the release() operation would do nothing.

2. *C++ Smart pointers.* A C++ implementation of this pattern can automate the management of session lifetimes through the use of smart pointer classes. A smart-pointer object would store a pointer to a session interface and release the interface in its destructor. A smart pointer on the stack would automatically release the session when it went out of scope. A smart pointer that was held as a member of an object would automatically release the session when the object was destroyed.
3. *Object Finalisation*. In a language with automatic garbage collection and object finalisation, the act of releasing a session can be made synonymous with releasing the last reference to the session object. The functionality of releasing a session can be performed by the finalisation method of the session object and so will be called automatically by the garbage collector.
4. *Java Outer/Inner Classes.* A Java implementation of this pattern would be simplified by the use of inner classes. The ConcreteService class would be implemented as an outer class and the ConcreteSession classes would be defined as inner classes. The compiler implements inner classes by using a hidden pointer from each object of an inner class to the object of the outer class that created it and generates code to delegate methods calls and references to instance variable from the inner class to the outer class. This would reduce the work required to implement the pattern and reduce the possibility of programming errors.
5. *Defining server interfaces.* If clients are always bound to servers by some third party that knows the complete type of all servers it is using, server objects do not need to conform to some abstract interface. However, in some systems clients will bind themselves to servers by finding a suitable server object in something like a trader or namespace and then request a session from it; in this case servers will need to conform to some abstract interface that can be used polymorphically.

Sample Code : Communication Protocols

The following source code expands on the networking example described above in which the program opens a network connection to a remote endpoint and transmits ten integers. The example is taken from the *Regis* protocol framework, an object-oriented framework that allows programmers to construct communication protocol software by composing simple components to form protocols with rich functionality tightly tailored to the needs of the application.

The framework defines a number of abstract interfaces and base classes that are used by protocol implementors, as well as useful abstract data types. For the purposes of this example, the relevant interfaces are ChannelUpcall, through which data flows up the stack from network devices to the application's threads, and ChannelSession, through which data flows down the stack from the application to the network devices. Raw data is managed by UBuf objects when travelling up the stack and DBuf objects when travelling down the stack. These classes perform performing memory management, segmentation and reassembly of blocks of raw data. Unlike the example code above, an individual protocol can provide multiple service interfaces. For example, TCP provides an interface named "connect" of type ConnectService through which objects can connect to remote endpoints and an interface named "listen" of type ListenService through which objects can listen for incoming connection requests. Service interfaces are abstract, allowing objects to dynamically select the communication protocol to be used at run-time and acquire the appropriate interface from a factory. For this example, we will consider only the ConnectService interface.

```
/*  Interface for data flowing up protocol stacks
 */
class ChannelUpcall
{
public:
    virtual void deliver( UBuf &buf ) = 0;
    virtual void notify( const ControlMessage &m ) = 0;
};
```

```
/*  Interface for data flowing down protocol stacks
 */
class ChannelSession
{
public:
    virtual bool transmit( DBuf &buf ) = 0;
    virtual bool configure( ControlMessage &m ) = 0;
    virtual bool query( ControlMessage &m ) const = 0;
    virtual void release() = 0;
};

/*  Service interface for connecting to remote endpoints
 */
class ConnectService
{
public:
    virtual ChannelSession *request( ChannelUpcall&,
                                     const Reference& ) = 0;
};
```

Unlike the Sockets API, protocols in the *Regis* framework are upcall-driven: system threads carry data up the stack, invoking the operations of the protocol objects directly. Application threads communicate through "interaction abstractions" that synchronise the system and application threads and perform presentation-layer marshalling. Interaction abstractions include queued message ports, event dissemination and distributed object invocation. The framework provides several base classes from which interaction abstractions are derived. The ClientEndpoint class is the base class of objects that can be connected to a remote endpoint and communicate with that endpoints over a channel through a ChannelSession interface. The bind method is used to hand ownership of a session that has been acquired by some third party over to the endpoint object. The interaction abstraction derived from ClientEndpoint uses the session to transmit data and releases the session when it is destroyed.

```
/*  Base-class of user-defined interaction abstractions
 */
class ClientEndpoint : public ChannelUpcall
{
public:
    /*  Upcall interface: implement in derived classes
     */
    virtual void deliver( UBuf & ) {};
    virtual void notify( const ControlMessage& ) {};

    /*  Binding interface: attach the session object of the channel
     *  used by this endpoint
     */
    bool bound() const;
    void bind( ChannelSession *transport );

protected:
    ClientEndpoint();

    /*  Allow derived classes access to the private _transport session
     */
    bool transmit( DBuf& );
    bool configure( ControlMessage& );
    bool query( ControlMessage& ) const;
    void releaseTransport();

    ~ClientEndpoint();

private:
    ChannelSession *_transport;
};
```

The programmer selects an appropriate interaction abstraction that meets the needs of their application, or defines one by deriving from the ClientEndpoint class. Here we assume the former: they are using the class PortClient<T> with which a thread can transmit values to a remote object of type Port<T> over the channel to which the PortClient is bound. We will not describe the implementation in any detail since it involves marshalling and the use of thread synchronisation libraries that are not germane to the issue.

```
template <class T>
class PortClient : public ClientEndpoint
{
public:
     bool out( T &message ) {
          DBuf buf;
          ... marshal message into the buffer...
          return transmit(buf);
     }
};
```

The main application code then looks as follows:

```
1.   PortClient<int> send;
2.   Reference remote_server = GetReferenceFromSomewhere();
3.
4.   // Select a protocol and get the interface of its connect service
5.
6.   ConnectService &connect = SelectProtocol();
7.
8.   // Acquire a ChannelSession connecting the PortClient to a
9.   // remote Port and pass ownership of it to the PortClient.
10.
11.  ChannelSession *session = connect.request( sender,  remote_server );
12.  send.bind(session);
13.
14.  // Send 10 integers
15.
16.  for( int i = 0; i < 10; i++ ) {
17.      int number = GetANumberFromSomewhere(i);
18.      send.out(number);
19.  }
20.
21.  // Ending the program will cause the PortClient object to
22.  // be destroyed, which will automatically release the protocol
23.  // session.
24.
25.  cerr << "Transmitted 10 integers" << endl;
```

Lines 11 to 12 show the use of the *Abstract Session* pattern. The program selects a protocol service and requests that the service connects the PortClient object to a remote Port. The connect service returns a ChannelSession for the connection that the program hands over to the PortClient. When the program calls the send operation of the PortClient, in line 18, the PortClient object marshals the integer into a buffer and transmits it over the ChannelSession. When the PortClient goes out of scope at the end of the program, its destructor releases the session, closing the connection.

Known Uses

The *Abstract Session* pattern is widely used in the implementation of object-oriented communication protocol software. The x-kernel framework [7] & the ACE communications toolkit [8] both use this pattern.

Microsoft's Object Linking and Embedding (OLE) framework [9] uses the *Abstract Session* pattern for managing the size and location of embedded objects. An object containing embedded objects, such as a word-processing document, is known as a "container" and stores pointers to the IOleObject interfaces of its embedded objects, through which it can, among other things, query the

required size of the embedded object and set the size and position of the object. When a new IOleObject is embedded in the container, the container creates a session object, known as a "client-site", in which it stores information about the actual position and size of the embedded object. The client-site object implements the IOleClientSite interface that the container passes to the to the embedded object and through which the embedded object can request to be resized. When an embedded object makes a resize request through its IOleClientSite interface, the container updates the size and position of all its embedded objects based upon the information stored in the client-site session objects.

The Java Abstract Windowing Toolkit (AWT) [10] uses the *Abstract Session* pattern in several places. An example is the Graphics interface, which provides a common interface for drawing graphics on a variety of devices, such as windows, bitmaps and printers. An object that wants to draw onto a device requests asks the device to create a graphics context object and receives a reference to the object's Graphics interface. The graphics context stores the current drawing state, such as the current font, background and foreground colours, and other state required to render drawing operations onto the associated device.

Related Patterns

The *Abstract Session* pattern is one way of implementing the *Session* pattern [1].

It is also related to the *Facade*, *Factory Method* and *Mediator* patterns from the GOF book [11]:

- The *Facade* pattern uses a single intermediate object to hide the complexities of a framework of co-operating objects from the users of that framework. In contrast, the *Session* pattern uses multiple intermediate objects to decouple objects that provide a service from the objects that make use of that service and to provide type-safe interaction between objects that only interact through abstract interfaces.
- The server object uses the *Factory Method* to create sessions for a client. This ensures that sessions can be initialised correctly by the server object.
- A *Mediator* object controls the interaction of multiple co-operating objects. The server object of the *Abstract Session* pattern can be viewed as a form of *Mediator* controlling the interaction of all of its clients. The session objects can be viewed as simple *Mediators* controlling the interaction of the server and a single client.

The *Acceptor* and *Connector* patterns [12] are both examples of higher-level patterns that make use of the *Abstract Session* pattern.

The *Abstract Session* pattern can be used to implement an *Adaptor* [11] around objects or non-OO libraries which use an unsafe implementation of the *Session* pattern.

Acknowledgements

The author acknowledges many stimulating discussions with members of the Department of Computing, Imperial College during the crystallisation of this pattern and the writing of this document, in particular Steve Crane, Naranker Dulay and Hal Fossa. The author would also like to thank Doug Schmidt, who shepherded this paper through its submission to EuroPLoP'97 and suggested many improvements. Last, but not least, many thanks go to those who gave comments on the paper at EuroPLoP'97 and all who made the workshop such an enjoyable and rewarding experience.

We acknowledge the financial support of British Telecommunications plc through the Management of Multiservice Networks project.

References

1. D. Lea. *Sessions*. Presented at ECOOP'95, 1995. Available on the WWW at the URL http://gee.cs.oswego.edu/dl/pats/session.htm

2. C. Petzold. *Programming Windows 95*. Microsoft Press, 1995.
3. D. Lewine, *POSIX Programmer's Guide: Writing Portable Unix Applications*, O'Reilly & Associates, Inc., 1991.
4. W.R. Stevens. *Unix Network Programming*. Prentice Hall, 1990.
5. The Object Management Group, *The Common Object Request Broker: Architecture and Specification, Version 2.0.* The Object Management Group, OMG Headquarters, 492 Old Connecticut Path, Framington, MA 01701, USA. July 1995.
6. D. Rogerson. *Inside COM – Microsoft's Component Object Model*. Microsoft Press, 1997.
7. N. C. Hutchinson and L. L. Peterson. *The x-Kernel: An architecture for implementing network protocols*. IEEE Transactions on Software Engineering, 17(1):64#76, Jan. 1991.
8. D.C. Schmidt. *ACE: an Object-Oriented Framework for Developing Distributed Applications.* In *Proceedings of the 6th USENIX C++ Technical Conference*, Cambridge, MA, USENIX Association, April 1994.
9. Microsoft Press. *Object Linking and Embedding Version 2 (OLE2) Programmer's Reference. Volumes 1 and 2*. Redmond, WA, 1993.
10. J. Gosling, F. Yellin, and The Java Team. *The Java Application Programming Interface Volume 2: Window Toolkit and Applets*. Addison-Wesley. 1996.
11. E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley. 1994.
12. D.C. Schmidt, *Acceptor and Connector: Design Patterns for Actively and Passively Initializing Network Services*. In *Pattern Languages of Program Design*, Reading, MA, Addison-Wesley, 1997.