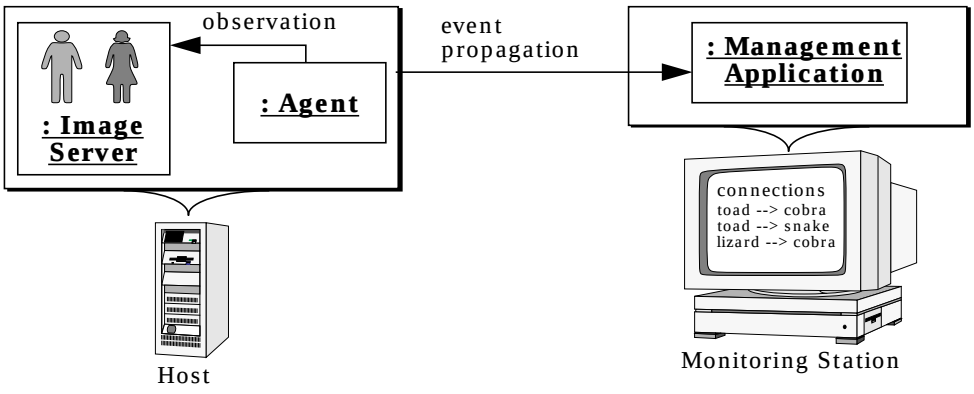
Asynchronous Completion Token

The *Asynchronous Completion Token* design pattern efficiently dispatches processing actions within a client in response to the completion of asynchronous operations invoked by the client.

Also Known As Active Demultiplexing [PRS+99]

Example Consider a distributed Electronic Medical Imaging System (EMIS) [PHS96] consisting of multiple components, such as *image servers*, which store and retrieve medical images [JWS98].[1] The performance and reliability of an EMIS is crucial to physicians, medical staff, and patients using the system. Therefore, it is important to monitor the state of EMIS components carefully. Specialized agent components address this need by propagating events from other EMIS components, such as image servers, back to management applications. Administrators can use these management applications to monitor, visualize, and control [PSK+97] the EMIS's overall status and performance.



For example, a management application can request that an agent notify it every time an image server on a host accepts a new network connection. Typically, such a request is issued by invoking an

1. Other components in a distributed EMIS include modalities, clinical and diagnostic workstations that perform imaging processing and display, hierarchical storage management (HSM) systems, and patient record databases [BBC94].

asynchronous operation on one or more agents in the system. When an agent on a particular host detects a new connection to the image server, it sends a *completion event* to the management application so that it can depict the new connection graphically on its monitoring console display.

However, a management application may have requested many different agents to send it notifications asynchronously for many types of events, each of which may be processed differently in the management application. For each completion event, therefore, the management application must determine the appropriate action(s), such as updating an administrators display or logging the event to a database.

One way a management application could match up asynchronous operation requests with their subsequent completion event responses would be to spawn a separate thread for each event registration operation it invoked on an agent. Each operation would block synchronously, waiting for its agent's responses. The action and state information required to process agent responses could be stored implicitly in the context of each thread's run-time stack.

Unfortunately, this synchronous multi-threaded design incurs several drawbacks. For example, it may lead to poor performance due to context switching, synchronization, and data movement overhead.[2] As a result, developing management applications using separate threads to wait for each operation completion response can yield inefficient and overly complex solutions. Yet, the management application must associate service responses with client operation requests efficiently and scalably to ensure adequate quality of service for all its agents.

Context An event-driven system where clients invoke operations asynchronously on services and subsequently process the responses.

Problem When a client invokes an operation request asynchronously on one or more services, each service indicates its completion by sending a response back to the client. For each such completion response, the client must perform the appropriate action(s) to process the results of

2. The Example section of the Reactor pattern (97) describes the drawbacks of synchronous multi-threading in more detail.

the asynchronous operation. Thus, we must define a demultiplexing mechanism to associate service responses with the actions to be performed by the client. To address this problem we must resolve the following three *forces*:

- When a service response arrives back at a client, it should spend as little time as possible determining the action(s) and state needed to process the completion of the associated asynchronous operation. In particular, searching a large table to associate a response with its request can be unacceptable if the performance of the client degrades significantly.
- It is hard for a service to know what information its clients need to process operation completions because it does not necessarily know the context in which clients invoked asynchronous operations. Therefore, it should be the responsibility of the client, not the service, to determine what actions and state to associate with completion responses.
- There should be as little communication overhead as possible between client and service to determine which action to perform in the client when the asynchronous operation completes. This is particularly important for clients that communicate with services over bandwidth-limited communication links, such as modem connections or wireless networks.

Solution Associate application-specific actions and state with responses that indicate the completion of asynchronous operations. For every asynchronous operation that a *client* invokes on a *service*, create an *asynchronous completion token* (ACT) that uniquely identifies the actions and state necessary to process the operation's completion, and pass the ACT along with the operation to the service. When the service replies to the client, its response must include the ACT that was sent originally. The client can use the ACT to identify the state needed to process the completion actions associated with the asynchronous operation.

Structure The following participants form the structure of the Asynchronous Completion Token pattern:

A *service* provides some type of functionality.

A *client* invokes operations on a service asynchronously.

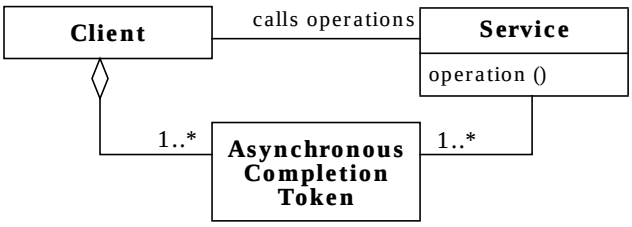
An *asynchronous completion token* (ACT) is a value that identifies the actions and state necessary for a client to process the completion of an asynchronous operation invoked on a service. The client passes the ACT to the service when it invokes an operation; the service returns the ACT to the client when the asynchronous operation completes. Services can hold a collection of ACTs to handle multiple client requests simultaneously. For a client, an ACT can be an index into a table or a direct pointer to memory. To the service, however, the ACT is simply an opaque value that it returns unchanged to the client.

Class Client	***Collaborator***
Responsibility	• Service
• Invokes operations on a service asynchronously	• ACT

Class Service	***Collaborator***
Responsibility	
• Provides application functionality	

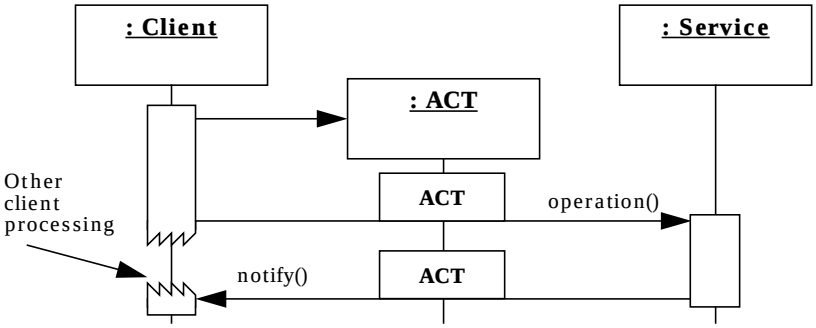
Class ACT	***Collaborator***
Responsibility	
• Identifies actions and state necessary for a client to process the completion of an asynchronous operation invoked on a service	

The following UML class diagram illustrates the participants of the Asynchronous Completion Token pattern and the relationships between these participants:



Dynamics The interactions in the Asynchronous Completion Token pattern are as follows:

- Before invoking an asynchronous operation on a service, the client creates the ACT associated with the operation.
- When invoking an operation on the service, the client passes the ACT to the service.
- Clients can continue executing while the service performs the operation asynchronously.
- When the asynchronous operation completes, the service sends a response to the client that contains the original ACT. The client uses the ACT to regain any necessary state and apply the application-specific completion actions for that operation.



Implementation There are four steps involved in implementing the ACT pattern.

1 *Define the ACT representation*. An ACT representation must be meaningful for the client and opaque to the service. The following are three common ACT representations:

- *Pointer-based ACTs*. ACTs are often represented as pointers to programming language constructs, such as pointers to abstract base class objects in C++ or references to abstract base class objects in Java. When a client initiates an operation on a service, it creates the ACT and casts it to a `void` pointer. The pointer is then passed to the service along with the asynchronous operation call. Pointer-based ACTs are primarily useful for passing ACTs among clients and services running on homogeneous platforms. Thus, it is necessary to use portability features, such as typedefs or

macros, to ensure a uniform representation of pointers throughout heterogeneous distributed systems.

- *Object reference-based ACTs*. When implementing distributed object computing middleware using CORBA, developers can represent ACTs as object references. Upon receiving an object reference-based ACT back from a service, a client can use the `CORBA _narrow()` operation to downcast the ACT to a type that is meaningful to it. CORBA object references provide a standard, portable, and interoperable means to communicate ACTs between clients and services. Naturally, object references may be inappropriate if CORBA is not used as the middleware platform.
- *Index-based ACTs*. Rather than using pointers or object references to represent ACTs, it is possible to implement them as indices into a table accessible by the client. Any state and actions can be associated with the appropriate index into the table. When a response arrives from the service, the client simply uses the ACT to access the corresponding entry in the table. This technique is particularly appropriate for languages, such as FORTRAN, that do not support pointers. Index-based ACTs are also useful for associating ACTs with persistent entities, such as database identifiers or offsets into memory-mapped files.

2 *Determine how to pass the ACT from the client to the service*. This step is typically straightforward. Clients can pass ACTs as explicit or implicit parameters in asynchronous operation requests. Explicit parameters are defined in the signature of the asynchronous operations. Implicit parameters are typically stored in a context or environment that's passed transparently to the service.[3] In the *Example Resolved* section we show an example of how ACTs can be explicit parameters to service methods.

3 *Determine a mechanism for holding the ACT at the service*. Once the ACT is received by a service, it must hold on to the ACT while performing the designated client operation. If a service executes synchronously, the ACT can simply reside in the run-time stack while

3. The CORBA service context field [OMG98d] is an example of an implicit parameter.

the service processes the operation. Thus, if a service is running in a different thread or process than its client, its operations can execute *synchronously*, while still providing an asynchronous programming model to the client [SV98].

Services that process client operations asynchronously, however, may need to handle multiple requests simultaneously. In this case, the service must maintain a collection of ACTs in a data structure that resides outside the scope of the run-time stack. The Manager pattern [PLoPD3] can be used to implement the collection of ACTs.

4 *Determine a demultiplexing mechanism to dispatch the ACT back to the client.* In some applications, when a client initiates an asynchronous operation request on a service, the ACT is returned just once, usually with the response. In other applications, however, ACTs from a single request may be returned multiple times. For instance, in our EMIS example, the client can use the same ACT to demultiplex and dispatch many agent responses that are associated with the same registration, such as a 'connection established' notification request.

In both the single and multiple response models, the following are common demultiplexing strategies used to dispatch ACTs back to a client when an asynchronous operation completes:

- *Callbacks*. In this strategy, a client specifies a function or object/method that can be dispatched by a service, or a local service proxy if the service is remote, when an operation completes [Ber95]. The ACT value itself can be returned as a parameter to the callback function or object/method.

  Callbacks can be delivered to a client *synchronously* or *asynchronously*. In the synchronous approach, the client application typically waits in an event loop or reactor (97). When the response returns from the service it is dispatched to the appropriate callback. In the asynchronous approach the callback is invoked via a signal handler [POSIX95]. Thus, clients need not explicitly wait for notifications by blocking in an event loop. The *Example Resolved* section shows an example of the synchronous callback approach using objects.

- *Queued completion notifications*. In this strategy, the client 'pulls' the ACT from a completion queue at its discretion.[4] ACT notifications are placed in a completion queue by a service or a local ser-

vice proxy. Windows NT overlapped I/O and I/O completion ports [Sol98] use this approach, as described in the *Known Uses* section.

Regardless of how the ACTs are returned to the clients, once they are returned, the job of the service is complete. Clients are responsible for processing the responses from completed operations and freeing any resources associated with the ACT.

Example Resolved

Consider a scenario where an EMIS administrator uses a management application to monitor and log all connections made to a particular image server. The management application must first invoke an asynchronous operation request that registers with the image server's agent to notify it when connections are established. Subsequently, when connections are established, *completion events* are sent by the agent and the management application must efficiently log the data and update its GUI accordingly.

The Asynchronous Completion Token (ACT) pattern supports the use case described above by allowing the management application to pass an opaque value as an ACT when it registers with the agent. For instance, the management application can pass a reference to a state object as the ACT. The state object itself contains references to a logging object and the appropriate GUI window that will be updated when connection notification responses arrive from the agent. The management application can use the state object referenced by the ACT to update the correct user interface and record the event with the appropriate logging object.

The C++ code below illustrates the use of ACTs for a management application class that handles asynchronous EMIS events received from agents. We first define an `ACT` to be a generic C++ pointer, as follows:

```
typedef void *ACT;
```

Next, we define an `EMIS_Event_Handler` class that defines the state and actions necessary to process an `ACT`.

```
class EMIS_Event_Handler : public Receiver {
     // The Receiver base class defines the pure
     // virtual <recv_event> method.
public:
     // References to State will be passed
     // as ACTs via asynchronous operations.
```

4. In contrast, callbacks 'push' the ACT back to a client.

```
        class State {
        public:
            State (Window *w, Logger *l) :
                window_ (w), logger_ (l) {}

            Window *window_; // Used to display state.
            Logger *logger_; // Used to log state.
            // ...
        };

        // Called back by EMIS agents when events occur.
        virtual void recv_event(const Event& event, ACT act){
            // Turn the ACT into the needed state.
            State *state = reinterpret_cast <State *> (act);

            // Update a graphical window and log the event.
            state->window_->update (event);
            state->logger_->record (event);
            // ...
        }
    };
```

The following code defines the Agent interface that can be invoked by clients to register for EMIS completion events.

```
    class Agent {
        // The types of events that applications
        // can register for.
        enum Event_Type {
            NEW_CONNECTIONS,
            IMAGE_TRANSFERS
            // ...
        };
        // Register for <receiver> to get called
        // back when the <type> of EMIS events occur.
        // The <act> is passed back to <receiver>.
        void register (Receiver *receiver,
                       Event_Type type,
                       ACT act);

        // ...
    };
```

The next code fragment shows how a management application invokes operation requests on an Agent and subsequently processes Agent notification responses. The application starts by creating its resources for logging and display. It then creates State objects that identify the completion of connection and image transfer events. The State objects contain references to Window and Logger objects. The address of the State objects are used as the ACTs.

```
int main (void) {
     // Create application resources for
     // logging and display. Some events will
     // be logged to a database, while others
     // will be written to a console.
     Logger database_logger (DATABASE);
     Logger console_logger (CONSOLE);

     // Different graphical displays may need
     // to be updated depending on the event type.
     // For instance, the topology window showing
     // an iconic view of the system needs to be
     // updated when new connection events arrive.
     Window main_window (200, 200);
     Window topology_window (100, 20);

     // Create an ACT that will be returned when
     // connection events occur.
     EMIS_Event_Handler::State connection_act
          (&topology_window; &database_logger);

     // Create an ACT that will be returned when
     // image transfer events occur.
     EMIS_Event_Handler::State image_transfer_act
          (&main_window, &console_logger);

     // Object that will handle all incoming
     // EMIS events.
     EMIS_Event_Handler agent_handler;

     // Binding to a remote Agent that
     // will call back the EMIS_Event_Handler
     // when EMIS events occur.
     Agent agent = ... // Bind to an Agent proxy.
```

Next, the application registers the EMIS_Event_Handler instance with the Agent for each type of event. Once these registrations are complete, the application enters its event loop, where all GUI and network processing is driven by callbacks.

```
     // Register with Agent to receive
     // notifications of EMIS connection events.
     agent.register (&agent_handler,
                     Agent::NEW_CONNECTIONS,
                     reinterpret_cast <ACT>
                             &connection_act);
```

```
        // Register with Agent to receive
        // notifications of EMIS image transfer events.
        agent.register (&agent_handler,
                        Agent::IMAGE_TRANSFERS,
                        reinterpret_cast <ACT>
                           &image_transfer_act);

        run_event_loop ();
    }
```

When an event is generated by an EMIS component, the Agent sends the event to the agent_handler, where it is dispatched to the management applications via the recv_event() callback method. The management application then uses the ACT returned by the agent to access the State object associated with the operation completion in order to decide the appropriate action to process each event completion. Image transfer events are displayed on a GUI window and logged to the console, whereas new connection events are displayed on a system topology window and logged to a database.

Variations *Synchronous ACTs*. ACTs can also be used for operations that result in synchronous callbacks. In this case, the ACT is not really an *asynchronous* completion token, but a *synchronous* one. Using ACTs for synchronous callback operations provides a well-structured means of passing state related to an operation through to a service. In addition, this approach decouples concurrency policies so that the code that receives an ACT can be used for either synchronous or asynchronous operations.

Chain of service ACTs. A chain of services can occur when intermediate services also play the role of clients that initiate asynchronous operations on other services in order to process the original client's operation.

➥ For instance, consider a management application that invokes operation requests on an agent, which in turn invokes other requests on a timer mechanism. In this scenario, the management application client uses a chain of services. All intermediate services in the chain—except the two ends—are both clients and services because they receive and initiate asynchronous operations. ❑

A chain of services must decide which service ultimately responds to the client. Moreover, if each service in a chain uses the ACT pattern

several issues related to passing, storing, and returning ACTs must be considered:

- If an intermediate service does not associate any completion processing with the asynchronous operation(s) it initiates, it can simply pass along the original ACT it received from its previous client.
- When completion processing must be associated with an asynchronous operation and an intermediate service can be sure that its clients' ACT values are unique, the service can use client ACT values to index into a data structure that maps each ACT to completion processing actions and state.
- If an intermediate service cannot assume uniqueness of client ACTs, the original ACT cannot be reused to reference intermediate completion actions and state. In this case, an intermediate service must create a new ACT and maintain a table that stores these ACTs so they can be mapped back to their original ACTs when the chain 'unwinds'.
- If no service in the chain created new ACTs, then the last service in the chain can notify the client. This design can optimize the processing because, in this case, 'unwinding' the chain of services is unnecessary.

Non-opaque ACTs. In some implementations of the Asynchronous Completion Token pattern, services do not treat the ACT as purely opaque values. For instance, Win32 `OVERLAPPED` structures are non-opaque ACTs because certain fields can be modified by the kernel. One solution to this problem is to pass subclasses of the `OVERLAPPED` structure that contain additional state.

Known Uses Operating system **asynchronous I/O mechanisms**. The Asynchronous Completion Token pattern is used by most operating systems that support asynchronous I/O. The techniques used by Windows NT and POSIX are outlined below.

- **Windows NT**. ACTs are used in conjunction with handles, Overlapped I/O, Win32 I/O completion ports on Windows NT [Sol98]. When Win32 handles[5] are created, they can be associated with completion ports using the `CreateIoCompletionPort()` system call. Completion ports provide a location for kernel-level services to

queue completion notification responses, which are subsequently dequeued and processed by clients that invoked the operations originally. For instance, when clients initiate asynchronous reads and writes via `ReadFile()` and `WriteFile()`, they specify `OVERLAPPED` structure ACTs that will be queued at a completion port when the operations complete. Clients use the `GetQueuedCompletionStatus()` system call to dequeue completion notifications, which contain the original `OVERLAPPED` structure as an ACT.

- **POSIX**. The POSIX Asynchronous I/O API [POSIX95] uses ACTs for its asynchronous I/O read and write operations. In the POSIX API, results can be dequeued through the `aio_wait()` and `aio_suspend()` interfaces, respectively. In addition, clients can specify that completion notifications for asynchronous I/O operations be returned via UNIX signals.

CORBA demultiplexing. The TAO CORBA Object Request Broker [POSA3] uses the Asynchronous Completion Token pattern to demultiplex various types of requests and responses efficiently, scalably, and predictably on both the client and server, as described below.

- On a multiple-threaded client, for example, TAO uses ACTs to associate responses from a server with the appropriate client thread that invoked the request over a single multiplexed TCP/IP connection to the server process. Each TAO client request carries a unique opaque sequence number (the ACT), which is represented as a 32-bit integer. When an operation is invoked, the client-side TAO ORB assigns its sequence number to be an index into an internal connection table managed using the Leader/Followers pattern (299). Each table entry keeps track of a client thread that is waiting for a response from its server over the multiplexed connection. When the server replies, it returns the sequence number ACT sent by the client. TAO's client-side ORB uses the ACT to index into its connection table to determine which client thread to awaken and pass the reply.
- On the server, TAO uses the Asynchronous Completion Token pattern to provide a low-overhead demultiplexing throughout the various layers of features in an Object Adapter [POSA3]. For

5. For Win32 overlapped I/O, handles are used to identify network connection endpoints or open files. Win32 handles are similar to UNIX file descriptors.

instance, when a server creates an object reference, TAO Object Adapter stores special object ID and POA ID values in its object key, which is ultimately passed to clients as an ACT contained in an object reference. When the client passes back the object key with its request, TAO's Object Adapter extracts the special values from the ACT and uses them to index directly into tables it manages. This so-called 'active demultiplexing' scheme [PRS+99] ensures constant-time *O*(1) lookup regardless of the number of objects in a POA or the number of nested POAs in an Object Adapter.

EMIS network management. The example described in this paper is derived from a distributed Electronic Medical Imaging System (EMIS) developed at Washington University for Project Spectrum [BBC94]. A network management application monitors the performance and status of multiple components in an EMIS. Agents provide the asynchronous service of notifying the Management Application of EMIS events, such as *connection events* and *image transfer events*. Agents use the Asynchronous Completion Token pattern so that the management application can efficiently associate state with the arrival of events from agents that correspond to earlier asynchronous registration operations.

FedEx inventory tracking. One of the most intriguing examples of the Asynchronous Completion Token pattern is implemented by the inventory tracking mechanism used by Federal Express postal services. A FedEx Airbill contains a section labeled: 'Your Internal Billing Reference Information (Optional: First 24 characters will appear on invoice).' The sender of a package uses this field as an ACT. This ACT is returned by FedEx (the service) to you (the client) with the invoice that notifies the sender that the transaction has completed. FedEx deliberately defines this field very loosely: it is a maximum of 24 characters, which are otherwise 'untyped.' Therefore, senders can use the field in a variety of ways. For instance, a sender can populate this field with the index of a record for an internal database or with a name of a file containing a 'to-do list' to be performed after the acknowledgment of the FedEx package delivery has been received.

Consequences There are several **benefits** to using the Asynchronous Completion Token (ACT) pattern:

Simplifies client data structures. Clients need not maintain complex data structures to associate service responses with completion actions. The ACT returned by the service contains all the information needed to demultiplex to the appropriate client completion action.

Efficient state acquisition. ACTs are time efficient because they need not require complex parsing of data returned with the service response. All relevant information necessary to associated the response with the original request can be stored either in the ACT or in an object pointed to by the ACT. Alternatively, ACTs can be used as indices or pointers to operation state for highly efficient access, thereby eliminating costly table searches.

Space efficiency. ACTs need not consume much space, yet can still provide applications with sufficient information to associate large amounts of state to process asynchronous operation completion actions. For example, in C and C++, ACTs that are four byte `void` pointers can reference arbitrarily large objects.

Flexibility. User-defined ACTs are not forced to inherit from an interface in order to use the service's ACTs. This allows applications to pass as ACT objects for which changing the type is undesirable or even impossible. The generic nature of ACTs can be used to associate an object of any type with an asynchronous operation. For instance, when ACTs are implemented as CORBA object references, they can be narrowed to the appropriate concrete interface.

Does not dictate concurrency policies. Long duration operations can be executed asynchronously because operation state can be recovered from an ACT efficiently. Thus, clients can be single-threaded or multi-threaded, depending on application requirements. In contrast, a service that does not provide ACTs may force delay-sensitive clients to perform operations synchronously within threads to handle operation completions properly.

There are several **liabilities** to avoid when using the Asynchronous Completion Token pattern.

Memory leaks. Memory leaks can result if clients use ACTs as pointers to dynamically allocated memory and services fail to return the ACTs, if the service crashes, for instance. Clients wary of this possibility should maintain separate ACT repositories or tables that can be used for explicit garbage collection if services fail.

Application re-mapping. If ACTs are used as direct pointers to memory, errors can occur if part of the application is re-mapped in virtual memory. This situation can occur in persistent applications that may be restarted after crashes, as well as for objects allocated out of a memory-mapped address space. To protect against these errors, indices to a repository can be used as ACTs. The extra level of indirection provided by index-based ACTs protects against re-mappings, because indices can remain valid across re-mappings, whereas pointers to direct memory may not.

Authentication. When an ACT is returned to a client upon completion of an asynchronous event, the client may need to authenticate the ACT before using it. This is necessary if the server cannot be trusted to have treated the ACT opaquely and may have changed the value of the ACT.

See Also The Asynchronous Completion Token and Memento [GHJV95] patterns are similar with respect to the participants. In the Memento pattern, originators give mementos to caretakers who treat the Memento as 'opaque' objects. In the ACT pattern, clients give ACTs to services that treat the ACTs as 'opaque' objects. However, these patterns differ in motivation and applicability. The Memento pattern takes 'snapshots' of object states, whereas the ACT pattern associates state with the completion of asynchronous operations. Another difference is in the dynamics. In the ACT pattern, the client—which corresponds to the originator in Memento—*creates* the ACT proactively and passes it to the service. In Memento, the caretaker, that is the client in terms of Asynchronous Completion Token, *requests* the creation of a memento from an originator, which is reactive.

Credits Thanks to Paul McKenney and Richard Toren for their insightful comments and contributions.