AUSOM Application Design Framework

Overview, Current Implementations and Future Directions

Michael Herman
Microsoft Canada Co.
mherman@microsoft.com
August 6, 1998

AUSOM is an acronym for A User State of Mind -- the name of a framework or architecture for designing software applications that are easier to design, implement, test, document and support. In addition, an application developed using the AUSOM framework is more capable of being: incrementally enhanced, progressively installed and updated, dynamically configured and is capable of being implemented in many execution environments. This paper describes the Core Framework, the status of its current runtime implementations and its additional features and benefits.

What is AUSOM?

The AUSOM Application Design Framework is a new way to design client-side

applications. The original implementation of the framework is based on a few basic concepts: user scenarios and detailed task analysis, visual design using state-transition diagrams, and implementation using traditional Windows message handlers.

The original motivation for the framework grew out of the need to implement a highly modeless user interface that was comprised of commands or tasks that were very modal (e.g. allowing the user to change how a polygon was being viewed while the user was still sketching the boundary of the polygon).

"User State of Mind"

The key concept is the User State of Mind. To explain this, assume the user has launched an application and is part way through completing a task s/he has started. In the user's mind, s/he has some internal sense of where s/he is with respect to the task they are currently trying to accomplish (current state), how they arrived where they are (history) and what are the most likely actions they might take to move forward in completing their task (set of possible user actions). This context, in total, is referred to as the User State of Mind (USOM). It is comprised of the current state of the task, a history of other tasks they may also be in progress as well as the list of actions the user can take to complete the current task (or begin a new task). When a user selects and performs a user action, s/he will transition from the current state to another (or possibly the same) USOM. In the AUSOM framework, the modeling of the possible USOMs and the transitions that lead from one state to another is captured in a state-transition diagram.

Appendix A documents a complete example based on the Draw Line tool in Microsoft Word. The corresponding USOMs are identified and a final state transition diagram is produced. Finally, the example is extended to show how a Move Point task can be added to the existing Draw Line model.

Relationship to N-Tier Client/Server and MVC Application Architectures

Figures 1 and 2 help position where the AUSOM framework fits into two popular application architectures: N-Tier Client/Server and MVC (Model-View-Controller) application architectures.

Figure 1(a) illustrates a typical 3-tier application architecture comprised of User Services, Business Services and Data Services.

To understand where the AUSOM framework fits into this architecture, the User Services tier is split into 3 layers: User Presentation, User Event Handling and User Functional Logic (as depicted in Figure 1(b). The User Presentation layer is responsible for presenting the typical graphical component of the user interface to the user. Subsequent actions taken by the user (such as moving the mouse over the user interface, clicking on a control or by saying something (if the application is speech enabled) are interpreted by the User Event Handling layer. User Functional Logic is the 3rd layer within the User Services tier. It contains the basic logic to prepare for and make calls into the Business Services tier. AUSOM is a framework for designing and implementing the bottom 2 layers: User Event Handling and User Functional Logic. In the section Additional Features and Benefits, this notion will be extended to arbitrary distributed N-tier application architectures.

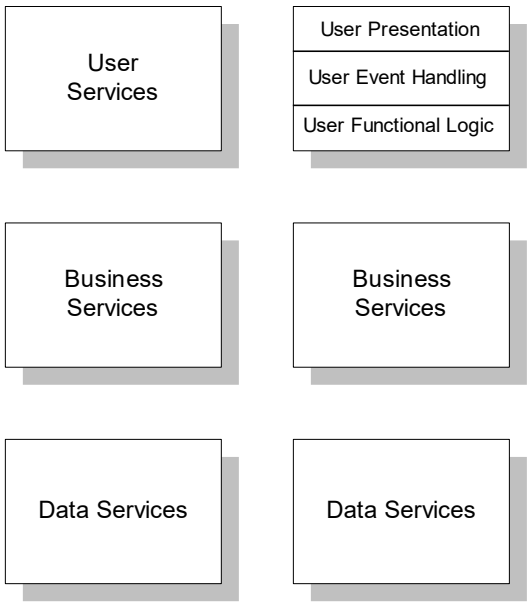


Figure 1. N-Tier Applications Architecture: (a) monolithic User Services tier; (b) partitioned User Services tier

Figure 2 is a diagram of the classic MVC (Model-View-Controller) application architecture that was popularized by the Smalltalk application development environment.

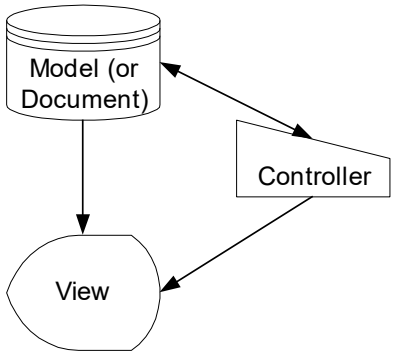


Figure 2. Model-View-Controller (MVC) Application Architecture

In the MVC architecture, AUSOM is a framework for designing and implementing the Controller component -- the part of an MVC application that is responsible for processing user input and applying changes to the underlying Model (or Document) and View(s). The Model stores the representations of the objects being manipulated or displayed. The View component is responsible for creating the one or more user views of the objects in the Model. Changes in the Model automatically cause the appropriate Views to update. User actions can affect either the objects in the Model or how the Model appears in a View (e.g. manipulating a scroll bar is a user action that affects a View).

AUSOM Core Framework

There are many facets to the AUSOM framework. At a high-level, there are two categories: the AUSOM Core Framework and Additional Features. As the names suggest, the Core Framework corresponds specifically to what has been implemented and proven to date. However, one of the most intriguing aspects of the framework is the additional application development and deployment features that can be enabled in AUSOM-designed applications. Figure 3 highlights the different categories of new features and benefits. These are discussed later in the section entitled Additional Features and Benefits.

Original Requirements

As mentioned at the beginning of this paper, the original motivation for the framework grew out of the need to implement a highly modeless user interface that was comprised of commands or tasks that were essentially very modal. More specifically, the original framework was used to develop a 3D graphics drawing application. For that application, AUSOM needed to support the following basic requirements:

- SDI (single document interface) application where the client area was used as a large 3D drawing canvas.
- Support for commands that were for the most part highly-modal commands (e.g. sketching the boundary of a polygon).
- In spite of the highly-modal command structure, provide the user with the appearance of the application being highly modeless.
- Implement Undo and Abort everywhere in every command.
- Runtime environment agnostic: able to run on Windows, Macintosh and UNIX workstations without a lot of effort.

Although the original thought process that led to the concepts behind the AUSOM framework have been lost (i.e. forgotten), the solution was to model each individual command as sequence of states -- specifically User States of Mind -- and to use manually created state-transition diagrams as the modeling tool. Further analysis indicated that UI activation/deactivation and the required modeless behavior were a key issues that remained to be solved.

The modeless behavior was achieved by allowing any "reasonable" command to be invoked and started in the middle of any other command. Previous commands and their application state could be pushed onto a stack and resumed when the user had completed the more recently started commands.

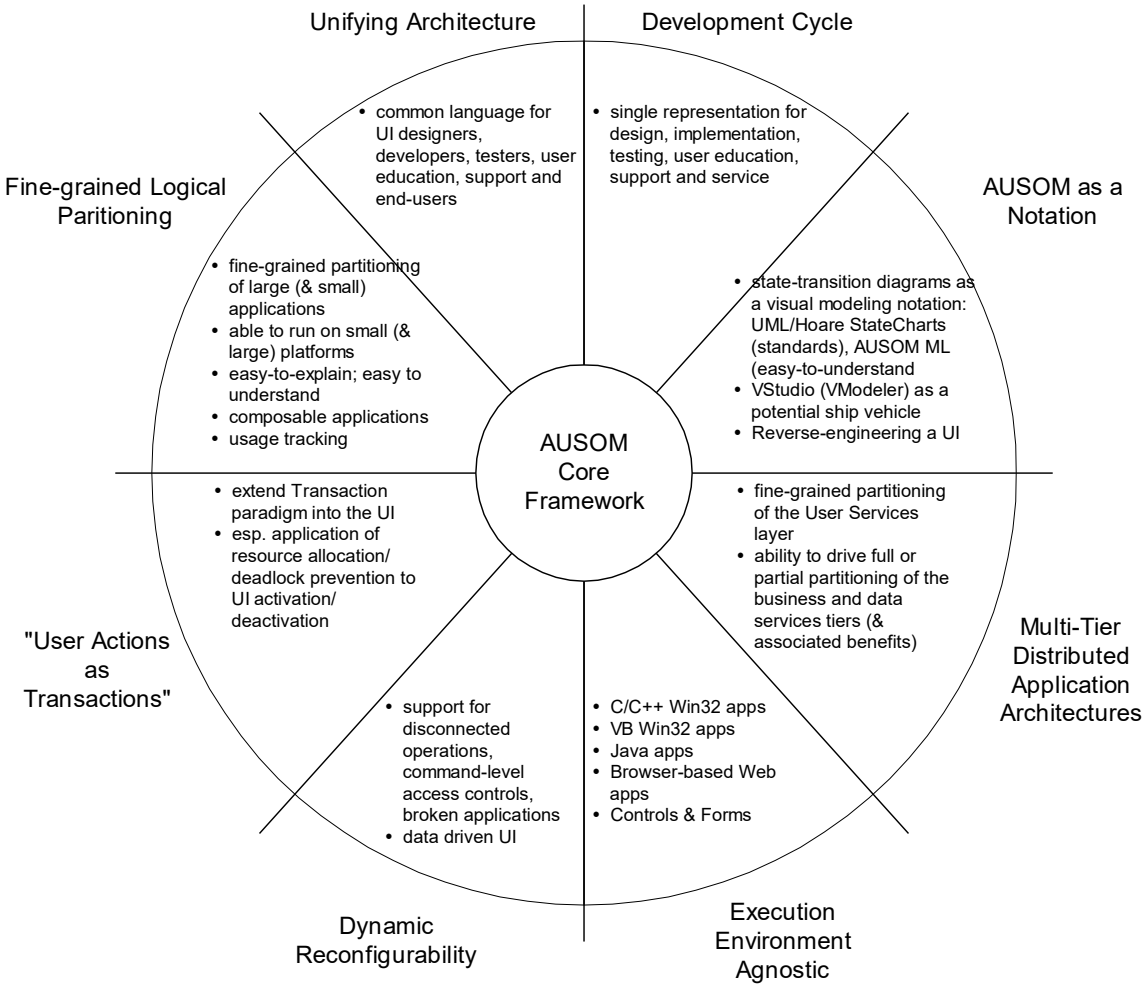


Figure 3. Building on the AUSOM Core Framework

Defining what a "reasonable" command (i.e. which commands could be activated at any point in the user's interaction with the application) was made possible by careful analysis of the application's state-transition diagram (and to some extent by hit-and-miss testing and

experimentation). A more reliable solution to this problem is proposed in the Additional Features section.

The Core Framework

The Core Framework was developed around the key concepts of States, Stategroups and User Actions. The concept of a State in the framework is analogous to a USOM (User State of Mind). A user (and the application) would be in one state at a time (the currently active state). Each User Action can be interpreted by the currently active State. Physically, User Actions correspond to some sort of gesture that the user would make (e.g. moving a mouse, clicking a mouse button, saying something, moving a joystick or foot pedal). Each command in an application is represented as a network of one or more states.

As States were used to design commands (or tasks), it became clear that a second mechanism, Stategroups, were needed for the following reasons: (i) as a natural grouping mechanism for the network of States that make up a command, (ii) to factor out and name common subnetworks of States that could be re-used across commands (stategroups) and (iii) to factor out the processing of events that was common to all States within a Stategroup. An example of the latter would be an Abort command that invoked the same functional logic regardless of which state the application (and the user) were in when the action was triggered.

With respect to UI activation/deactivation, each stategroup in the original framework defined a list of "unreasonable" commands that need to be deactivated when this command (stategroup) was started. The list of unreasonable commands would be or'ed with the list of unreasonable commands for the currently executing (but stacked set of commands). This or'ed list of unreasonable commands was then used to control activation and deactivation of the menu bar and tool bar UI elements.

After the execution of a stategroup (command), typically the Model (or Document) would have to be updated as a result of an object being changed, added or deleted. Frequently, the change would be built up through a series of user actions (and corresponding state transitions within the stategroup).

What happens when a user wants to undo or abort? With an AUSOM-designed application, the answer can be clearly and visually understood. Undo typically means to return the previous state in the stategroup -- undoing the corresponding subset of changes that have been built up but have not been formally committed to the Model/Document. Abort is interpreted as undoing the entire stategroup causing (i) the stategroup to be popped from the stack of currently executing commands and (ii) the entire change to be thrown away. Note the strong similarity between the semantics associated with the execution of a stategroup and the concept of a database transaction.

The tracking of the user's movement from one State (or USOM) to another based upon User Actions being interpreted by the current State and the fine-grained modeling and implementation of this movement using state-transition networks are the fundamental concepts behind the AUSOM Application Design Framework.

Development Cycle Benefits

In addition to producing software that is easier to explain and easier to understand, the AUSOM framework has several in-process benefits that are realized during the development and support of an application. The underlying theme is the benefits derived from the direct one-to-one correspondence between the AUSOM-based design and source code level implementation.

Design. Design and Implementation are the two key development activities that the AUSOM framework focuses on. The ability to take use cases and scenarios and map them directly into an AUSOM state-transition diagram that represents how an user is expected to move through a task is the first key benefit. User tasks map to Stategroups and a step within a task maps to a State (USOM).

Implementation. The detailed implementation of an AUSOM application varies

depending on the target execution environment. To date, most of the experience comes from developing Win16 and Win32 C language applications. In this environment, a State is implemented as a plain old Windows message handling function. The key implementation benefits arise from the one-to-one mapping between the AUSOM state-transition diagram and the implemented code: one Win32 C message handling function is used to implement each State with all intra and inter-State branches being described by the state-transition diagram. The benefits include: (i) easy communication of the intended design to the developer and (ii) the ability to more accurately manage the development project given the linear relationship between the number of States and the amount of time and effort required to implement them.

Testing. Because of the fine-grained enumeration of the task functionality in an application, it is also possible to more accurately manage the testing process. Measuring and verifying test coverage becomes a straight-forward process given the ability to enumerate the States and each State's testing progress. The fine-grained design and implementation also help to insure that the amount of functional logic in each individual State is small and as a result, usually not very complex. Black box testing essentially becomes white box testing, again, due to the one-to-one mapping of States to source code that is inherit in AUSOM applications.

User Education. The key user education benefit is the way the fine-grained nature of an AUSOM application enables very detailed context-specific assistance to be provided to the user. For example, consider the status bar that is usually found at the bottom of an application window. The status bar is most often used for just what it's name implies -- displaying the status of the last completed action. Because an AUSOM application has a very detailed understanding of where a user is an application (i.e. a particular State within a Stategroup), it is possible to use the status bar to implement real-time or "in-line" suggestion-based, on-line help where the status bar is used to suggest to the user the one or two most likely actions they could want to take to complete the task at hand. This feature can also be thought of as a task-oriented "user guidance" system which can reduce or eliminate the need for expensive upfront training (esp. for categories of users such as stock market traders who don't have the time nor desire to "learn" an application). The opportunity also exists to effectively leverage speech output in the above scenario.

Customer Support and Product Service. When a user calls Microsoft product support and reports "I was at this point in [an AUSOM] program and I tried to do X and Y happened instead", a support engineer familiar with the application could immediately begin to isolate the affected stategroup and state where the user encountered the problem. Even if the support engineer was not very familiar with the application's design, it is easy to imagine a design-driven tool that acts like a map leading the engineer to the affected stategroup. With respect to product service, given the same type of problem report, an AUSOM application enables a developer to become very productive when it comes to understanding, isolating and correcting the underlying fault. In many cases, a developer will be able to mentally forecast what the code for the affected state and stategroup actually looks like before they get to the real code.

Project Management. As mentioned earlier, the fine-grained enumeration of the new or extended functionality to be implemented in an AUSOM development project enables program managers and development managers to more accurately plan and track software projects.

Project Communication. Once a development team starts to use the AUSOM framework to develop an application, they quickly adopt an effective new language based on the names chosen for the stategroups and states used to design the product. The benefit begins during design and continues throughout the implementation, testing, user education, support and service activities.

AUSOM Runtime Environment

The implementation of an AUSOM Runtime environment does not require a lot of effort. The Win32/.C version of the basic runtime was

written using 220 lines of C source and 40 lines of header file (including comments and whitespace).

The key components of the basic runtime environment are: (i) the Dispatcher for routing events to the currently active State, (ii) an design template for implementing the States which in turn process the events that are expected in the corresponding USOM, (iii) a design template for the Application handler, (iv) an implementation of the System event handler and (v) a Status Bar object to manage the display of suggestion-based help. The specific implementations of these runtime components vary with the target execution environment. While the focus will be on the current Win32/C implementation, Win32/C++, HTML/Scripting and Win32/Visual Basic implementation issues will also be discussed.

The Dispatcher is primarily responsible for routing events (or in the case of a Win32/C environment, Windows messages) to the currently active state. Its secondary responsibility is to maintain a push-down stack of the currently active state (and stategroup). The AUSOM State Stack object supports the following methods: Initialize, Reinitialize, Start, CallCommand, SetNextState and Return. The semantics associated with these methods is straightforward. Initialize and Reinitialize (re)create the stack and initialize it to the default Application:Home State. Start is called to initialize or start-up the current stategroup. CallCommand and Return have semantics that are identical to subroutine calls and returns in most programming languages. CallCommand pushes its Stategroup argument onto the stack and then "starts it" causing the stategroup's default start state to become the currently active state. Return pops the current state (and stategroup) and then "resumes" the previous state which is now the top stack element. SetNextState is used within a Stategroup to transition from one state to the next (i.e. replace the top of stack state with SetNextState's next State argument).

A design template is needed to implement the event handler that implements each State. In a Win32/C environment, AUSOM uses a traditional Win32/C message handler function which consists almost entirely of a switch statement. The switch statement is used to invoke the functional logic that should be called for each event (and corresponding user action) that is expected and accepted by the State. Each of these message handlers or States can be thought of as a "Windows microprogram" that is capable of processing the (usually) small number of events that are expected by a given State (again due to the fine-grained nature of an AUSOM design).

Noting that for some events in a stategroup, the functional logic to be executed may be identical for all states in the stategroup, it is also useful to have an additional "Stategroup" handler in each stategroup that contains the functional logic for events that are always processed the same way regardless of the state within the stategroup. At a more detailed level, it turns out that this additional Stategroup handler is always required for starting and resuming a stategroup. The Stategroup handler contains the logic for the Start internal AUSOM event that is used to initialize a Stategroup. This logic, for example, can trigger the appropriate UI activation/deactivation to be performed.

Similarly, there are events that sometimes only make sense at an "application level" (e.g. the Exit item on an application's File menu). The AUSOM framework uses the concept of an Application handler to support this. By extension, there are events that can only be interpreted by the windowing system; hence, the addition of a Window System handler.

All AUSOM Win32/C event handlers have the same design template (a Win32 message handler function) regardless of whether the function is a State, Stategroup, Application or Window System handler. The default case of each handler's switch statement simply calls the state handler for the next level up in the hierarchy (see Figure 4). Those readers familiar with the Win32 DefWindowProc() function will understand exactly how this works.

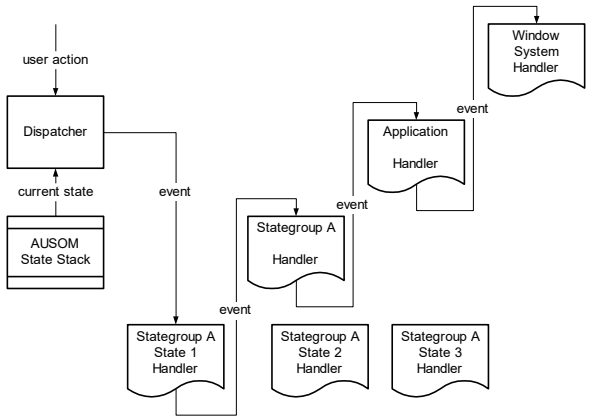


Figure 4. AUSOM Event Handler Hierarchy

The Dispatcher includes a state transition logger that is useful for validating application/state test coverage as well as for user-task profiling.

Code Size and Performance. A medium-size commercial application may have several hundred up to a thousand states. While the code overhead is limited to the one C function and switch statement per State and no comparable size measurements has been made, it is believed that an AUSOM application's executable .exe size is slightly larger than a similar non-AUSOM application. With respect to execution-time performance, the overhead is a minimum of one indirect function call (through the State Stack pointer). This is deemed to be acceptable given that the AUSOM framework is only processing user actions at the rate a human can generate them. Overall, it is believed that the code size and performance are nearly optimal compared to other imaginable ways of delivering the same set of AUSOM features and benefits.

Note: Attempts to physically optimize the source code produced using the AUSOM framework should be taken with great care. The benefits a software organization realizes from an support, service and future enhancement perspective are significantly reduced when the one-to-one correspondence between user features, the design and the physical source code are lost -- realizing this type of optimization benefit is better left to the C compiler.

Implementation Status

As indicated above, a working Win32/C implementation of the AUSOM runtime environment has been written.

In addition, a prototype Win32/C++ implementation has been developed that uses similar C++ classes to define the State, Stategroup, Application or Window System handlers. Each class has a method defined for each event that it accepts. The Window System, Application and Stategroup handlers are implemented as virtual classes and are never instantiated as objects within the actual application. Instead only single instances of each of the State handler classes are created and the behavior for any events not handled by the currently active state is inherited from the Stategroup, Application or Window System virtual super classes. This approach does suffer from some of the code size problems that MFC chose to avoid -- similar techniques could be used here to avoid this problem.[1] An additional note: an entire prototype Win32/C++ application was generated from a tabular (Microsoft Access) representation of the state transition diagram. In this example, the table stores the functional logic (C/C++ code) in a four-column table that is indexed by Stategroup ID, State ID and Event ID combination. The C++ code generation and positioning of the AUSOM State classes relative to one and other was handled automatically.

The analysis and design work required to turn the MFC (Microsoft Foundation Class) C++ library into an AUSOM framework has been completed. The solution here is to take the existing MessageMap class in the MFC library and redesign it so that it is possible to define multiple AUSOM message maps everywhere you can define a single message map today. Each AUSOM message map would correspond to a State handler. Additional methods would need to be defined in the MessageMap class to support the Dispatcher methods: CallCommand, Return and SetNextState.

[1] Although the MFC (Microsoft Foundation Class) C++ library benefited greatly in terms of code size by avoiding the use of virtual classes to implement MessageMaps, for example.

The Win32/Visual Basic environment is one of the least leveragable execution environments from an AUSOM perspective because all event processing is handling internally by the VB runtime and only exposed as a collection of separate "built-in" methods that are intended to be implemented by the application developer and called directly by the VB runtime. An AUSOM runtime implementation requires that the entire set of event handlers be switchable as a group or class when a state transition occurs. With the Win32/Visual Basic environment, you end up with a nasty scenario where all of the "built-in" method handlers have to be coded to call the central AUSOM Dispatcher.

The latest application execution environment to arrive is the HTML/Scripting environment used for conventional browser-based applications. Although only some preliminary design and prototyping of an AUSOM HTML/Scripting environment has been completed, the AUSOM approach has a key advantage over conventional HTML/Scripting approaches: the ability to cleanly separate the client-side application logic and scripting code from the (D)HTML used for the web pages visual presentation. Hidden frames are used to store the AUSOM runtime environment (Dispatcher code and the State Stack) and the Application-level handler. These two pages of script code only need to be downloaded once per invocation of the application. The Stategroup handler code for a particular command would be downloaded once per command invocation. This page of script code also contains the script implementations of each of the States within the stategroup. The only visible frame would be the Client Area that would be used to display the application's user interface. The user interface could be either downloaded as a (D)HTML page or created on the fly by functional logic within the current Stategroup (see Figure 5). Alternatively, the AUSOM runtime (plus any of the Application, Stategroup or State handlers) could be implemented as ActiveX components or Java applets.

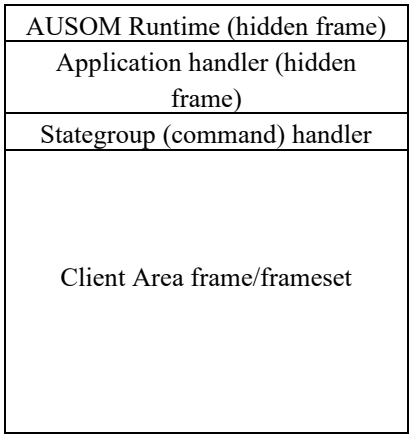


Figure 5. AUSOM HTML/Scripting Runtime Architecture

Lessons Learned

Managing the Design. As mentioned earlier, a medium size commercial application can contain several hundred up to a thousand states. Initial attempts to manually manage this type of design proved to be a great challenge -- even when the state-transition diagram was being maintained on a large 8'x8' surface. A visual modeling tool for designing and maintaining the state-transition diagram representation of an application is now seen as a pre-requisite. Products like Rational Rose and I-Logix Rhapsody come close to meeting most of the needs of an AUSOM designer.

Design Optimization. Once an initial design has been laid out, it is easy to begin to recognize repeating patterns or subnetworks of states that can be re-used in the design of the application. Having identified these, it is usually possible to factor them out as independent stategroups that can be called from higher-level states.

UI Activation/Deactivation. Similar to the challenges involved with manually managing an AUSOM state-transition diagram, manually determining which UI elements on menu and tool bars should be activated or deactivated for a given stack of states has been somewhat of a trial-and-error process. As mentioned above, each stategroup in the original framework simply had a list of "unreasonable" commands

associated with it. The list of unreasonable commands would be or'ed with the list of unreasonable commands for the currently executing but stacked set of stategroups. This list of unreasonable commands was used to control activation and deactivation of the menu bar and tool bar UI elements. In the next section, an interesting solution for the UI activation/deactivation problem is proposed based on solutions used to schedule data processing jobs to run in a mainframe batch processing environment.

Ease of producing alternative versions of a product. Once an AUSOM design has been created for an application, it is now very easy to produce alternative versions of that product. For example, "light" versions can be created where useful subset of functionality is exposed to the user or is exposed with a simpler user interface or with more elaborate suggestion-based help. Creating "demo" versions of a product where certain operations such as Copy and Save are disabled is another example. These types of alternative versions are created by simply re-wiring or short-circuiting some of the state transitions so that they either don't appear in the UI or appear disabled. An AUSOM application is also one way of creating "scaleable user interfaces where a user can be slowly exposed to more power, but can start their experience in a simple, non-threatening mode."[2]

Additional Features and Benefits

Up to this point, the whitepaper has focused on the author's actual experience implementing and using the AUSOM Application Design Framework. In this section, a number of additional features and benefits will be described that build on top of the Core Framework. These are highlighted in Figure 3. While the feasibility of each of the features mentioned below hasn't been tested in an actual implementation, it is believed that they are implementable and a benefit to organizations that build applications with the AUSOM framework.

[2] Jim Allchin, "Complexity", Microsoft Internal Report, Sept. 4, 1997.

Starting at the top of Figure 3 and moving clockwise, the development cycle benefits have already been described in the section on the Core Framework.

AUSOM as a Notation

From the perspective of writing applications that are easy to explain and easy to understand, the ability to express the detailed design and its one-to-one correspondence with the underlying source code implementation remain a key benefit of the AUSOM framework. This further enhanced when a diagramming notation (and modeling tool) is available to visualize the design. In AUSOM, three design notations and three modeling/diagramming tools have been used. The first attempt at using the AUSOM framework for a serious development project used a large 8'x8' sheet of Styrofoam with push pins, labels and elastics being used to represent the states and the transitions between states. The second tool to be used was Visio. Visio is a great drawing tool but it is sometimes difficult to use for creating connected network diagrams. It is also fully programmable using VBA (Visual Basic for Applications) making it possible to implement some basic code generation. However, the integrity of the diagrams (from a connectedness point-of-view) has ruled out any further investigation in the area. However, from a notation perspective, Visio was used to prototype a new visual notation which is referred to the AUSOM Modeling Language (AUSOM ML) (see Figure 6).

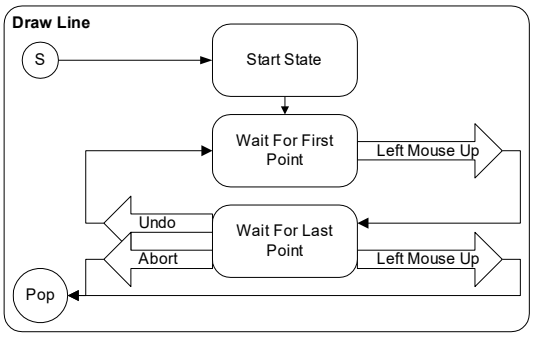


Figure 6. AUSOM ML Example

Rational Rose has proved to be the best tool the author has used to date. Rose supports the UML Statechart notation for designing state-transition diagrams and includes VBA support as well. Some very early prototyping has been completed where an AUSOM state-transition diagram has been drawn using Rose and a VBA program was used to extract the Stategroups and States into Excel spreadsheet. An AUSOM example from Rose appears in Figure 7.

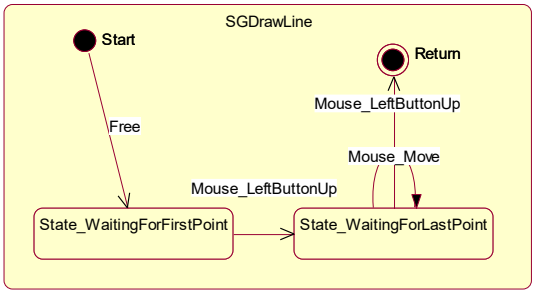


Figure 7. Rational Rose UML Statechart Example

Lastly, using AUSOM as a notation for describing how a user moves through an application makes it a good tool for reverse engineering or otherwise documenting an application's user interface.

Distributed N-Tier Systems Architecture

Figure 8 enumerates most of the many distributed N-tier systems configurations are used to implement a client/server solution. Recalling that the AUSOM framework is a way to design and implement the User Event Handling and User Functional Logic layers within the User Services tier, the framework can be used to design and implement a broad range of client/server and desktop applications. Of particular note is the (potential) ability to design applications that have a Distributed User Services tier with parts of the User Event Handling and User Functional Logic layers running on both the client and/or the server (refer to the Web N-Tier example in Figure 8).

In addition, given the fine-grained physical partitioning of an AUSOM-designed User Services tier, it is also possible to begin thinking about how the executable code for stategroups within an AUSOM application can be incrementally downloaded, on an on-demand basis, to the client PC. This is discussed further in the section on Related Topics.

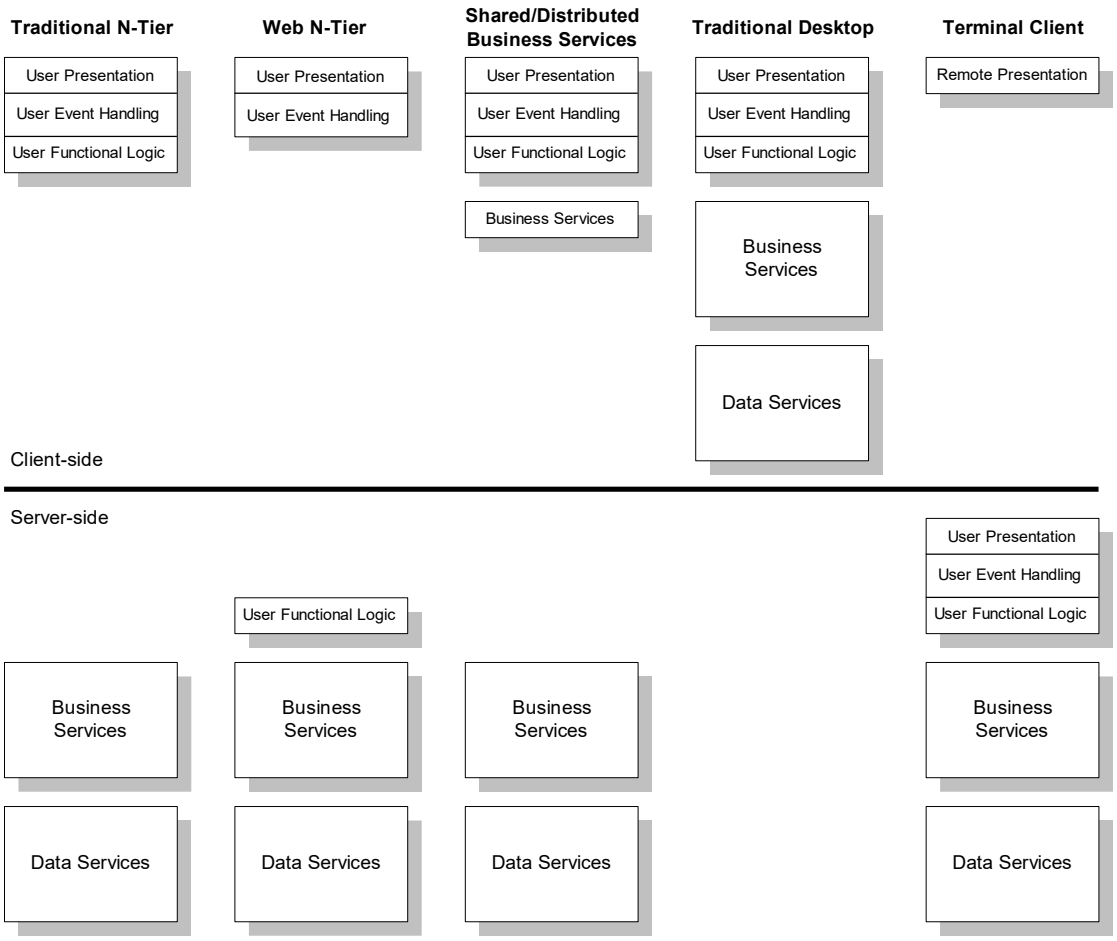


Figure 8. Distributed N-Tier Application Architectures

Execution Environment Agnostic

Previous examples have already described how the framework can be used to design and implement Win32/C, Win32/C++, HTML/Scripting and HTML/Visual Basic applications. Java applications also fit into this space.

Dynamic Reconfigurability

In the AUSOM framework today, transitions from one state to another are totally driven by user actions. This state-transition model can be extended by adding guard conditions to the stategroup's start state UI activation/deactivation model. This would allow an application's behavior to vary based on other environment factors; for example, whether a user or application is running in a disconnected network environment or whether s/he has sufficient stategroup-level authority to use a particular command or begin a particular task. This feature would also enable the ability to develop scalable user interfaces (mentioned earlier in this paper).

"User Actions as Transactions"

Microsoft has made huge advances by making high-end transaction processing capabilities available on commodity hardware. Now, in a manner similar to the way the Windows operating system has been extended to

the very low-end of the spectrum, Stategroups provide a similar opportunity to extend our transaction processing story to the client and the user interface itself. The CallCommand, Return, and Escape/Abort AUSOM Dispatcher methods become synonymous with the Prepare, Commit and Abort transaction processing primitives.

Advanced UI Activation/Deactivation

One way to potentially automate this process is to consider why some commands have to be deactivated in the first place. The author's hypothesis is that this is almost solely as result of resource contention. That is, while one command (stategroup) is running, it is changing, adding, deleting or exclusively reading from a resource in a way that requires any other command that also needs that resource to be disabled. An analogy can be drawn between the scheduling of data processing jobs in a mainframe batch processing environment and determining whether a particular AUSOM stategroup can be enabled or not based on the resources is needs to access or modify. Algorithms for computing this type of resource allocation/deadlock prevention are well known and could be applied to solve the UI activation/deactivation problem and potentially enable more modeless command invocation within an application. The idea of treating user actions or stategroups as transactions is also relevant here.

Fine-grained Logical Partitioning of Applications

Perhaps, this aspect of AUSOM applications has already been overly emphasized. However, there are a few more benefits to be realized from this characteristic of AUSOM-designed applications. First, when used in conjunction with the ability for AUSOM applications to be incrementally downloaded, on an on-demand basis, it may be possible to design "large" applications (e.g. Microsoft Word or PowerPoint) in way that allows them to run on small platforms such as palm-sized PCs and/or set-top cable TV boxes.

Versioning, Binding, Installation, Upgrading and Dynamic Downloading

Java and TCO (Total Cost of Ownership) have brought a lot of focus to the topics of versioning, binding, installation, upgrading and dynamic downloading. AUSOM applications with their many states and stategroups may be seen as way for making a bad situation even worse. The author doesn't agree.

Above all, the AUSOM framework is focussed on making it easier to design, implement, test, document, support and service great applications. It is agnostic with respect to when and how an application is bound, versioned, installed, upgraded or downloaded.

However, it is the very nature of the AUSOM-designed applications that makes them great applications to be incrementally delivered to a desktop. The smallest physical packaging that makes any real sense is at the Stategroup level. Depending on the execution environment, a stategroup can be packaged as a simple DLL, a COM component, HTML script page or a scriptlet (among others). Stategroups can also be aggregated into larger (and fewer) packages if the requirement presents itself.

Effective versioning support in the operating system is the highest priority requirement. Very rapid client-side binding of Win32 executable code would be the next enabling priority.

Related Topics

The AUSOM application design framework has a broad range of applications and benefits. Some of these are detailed below while others such as SDI vs. MDI applications, forms, controls and data-driven UI haven't been given as much consideration as much consideration as others and still require more thought.

COM+

The AUSOM framework supports the COM+ goals of being object-oriented, transaction-based, supporting 3-tier development, language agnostic and providing a way to develop applications that are easy to

model, develop and debug.[3] COM and COM+ haven't been mentioned very much in this whitepaper. This is largely because AUSOM is at the stage of being a "framework" as opposed to a specific architecture (i.e. a framework with a specific set of policies attached to it). At a basic level, COM is a way to package Stategroups as components. The number of methods or interfaces exposed by a Stategroup or State is very small and they are generic to all instances of a Stategroup and State (owing to the Win32/C legacy). This makes COM an ideal environment for packaging AUSOM applications. The binding and versioning issues remain.

MegaServer

The MegaServer vision is to produce a seamless blending of online services and software applications.[4] AUSOM is potentially one way to deliver on these requirements.

Schemas Everywhere

AUSOM stategroups (and their representations as state-transition networks) make them a candidate for defining common schemas for UI interaction and command interaction with the standard data schema frameworks that are starting to evolve. The AUSOM framework could leverage or extend the work being done by the Schema-by-Example project. More specifically, the framework could leverage named schema templates if they were cataloged as by product of the Schema-by-Example processes.

XML and XSL

Earlier sections have described how existing AUSOM applications (i.e. the state-transition diagram representations of the functionality they support) can be easily extended through the addition of new stategroups (i.e. subnetworks of additional states). This approach will ultimately be limited by the extensibility of the underlying Model. An XML-based Model or Document architecture may be a good basis of an incrementally extensible Model and in turn, a way to continue to extend the functionality delivered to the user through an incrementally extensible AUSOM-designed Controller.

Data-driven AUSOM Applications

Jose Blakeley suggested the framework might be applied be applied to Explorer-type applications where the type of the selected object (e.g. file vs. folder or database vs. table vs. field) could be used to drive the application.

MDI Applications, Forms and Controls

MDI Applications, Forms and Controls have the basic common requirement for the AUSOM framework to support nested and/or multiple independent instances of the AUSOM runtime stack. MDI (multiple document interface) applications differ from SDI (single document interface) applications in their ability to produce multiple simultaneous views of the underlying document model. While it is generally the case that only one view at a time has an active user event handling layer, it is very possible that, with the availability of multiple input devices and media (e.g. speech, vision, handwriting , hand, foot and other body gestures), application designers will want the ability to simultaneously interact with multiple views. The AUSOM runtime architecture would need to be extended to support this.

Composable Applications

Ability to build custom applications based on collections of stategroups. Some groups of user may need to launch a single application that contains a somewhat arbitrary combination to tasks or command included in the application. This is somewhat analogous to the incremental extensibility, scalable UI and composable application examples described earlier.

[3] Bill Gates, MGS 98 Closing Keynote, July 30, 1998.
[4] Ibid.

Workflow

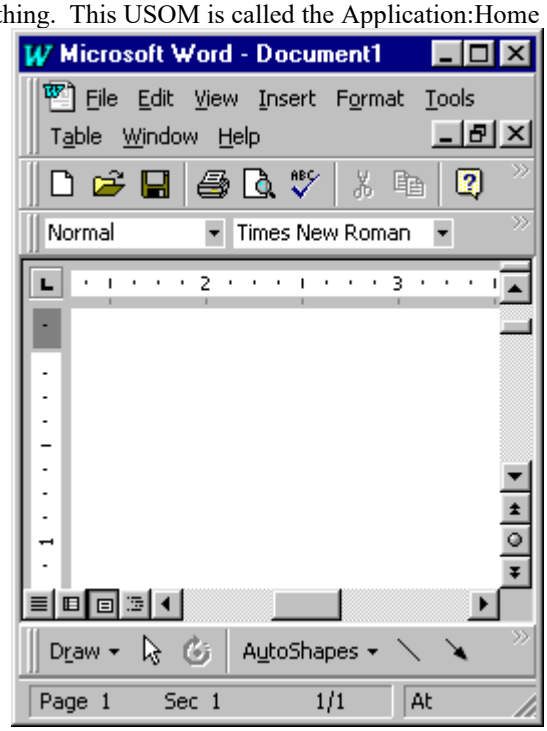
Similar to the way the AUSOM framework may enable close integration of transaction processing concept and an application's user interface, business process workflow can be more tightly and easily integrated with the application's UI.

Conclusion

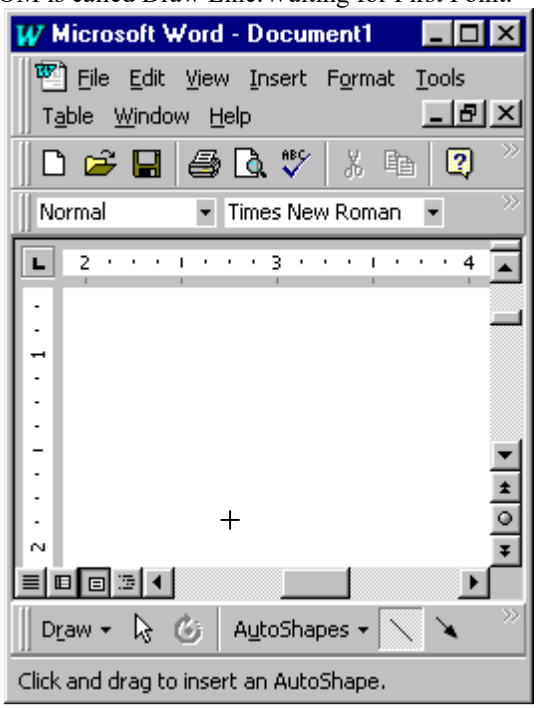
The AUSOM Application Design Framework represents a new way to produce applications that are easy to explain and understand. This paper has documented the AUSOM application design framework's core features as well as helped identify a large number of additional directions and benefits. It is AUSOM's vision to become a dominant new framework for designing and implementing broad classes of applications.

Appendix A - Microsoft Word "Draw Line" Scenario

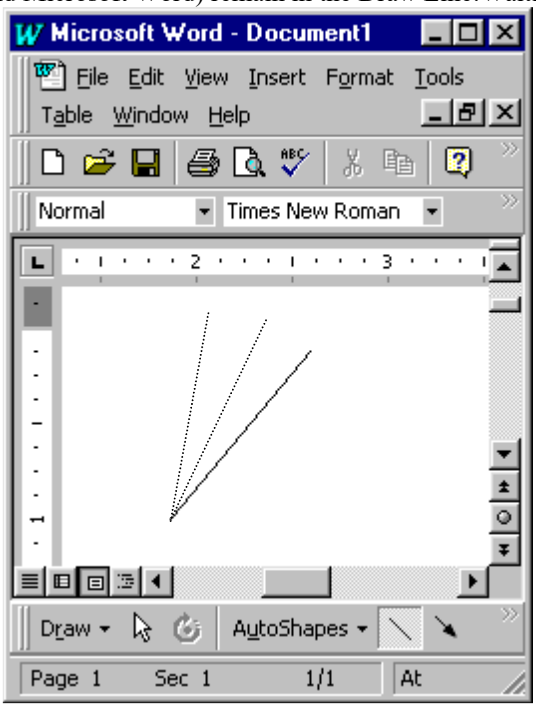
1. User launches Microsoft Word. By default, a new document Document1 is created. Word is waiting for the user to do something. This USOM is called the Application:Home State.



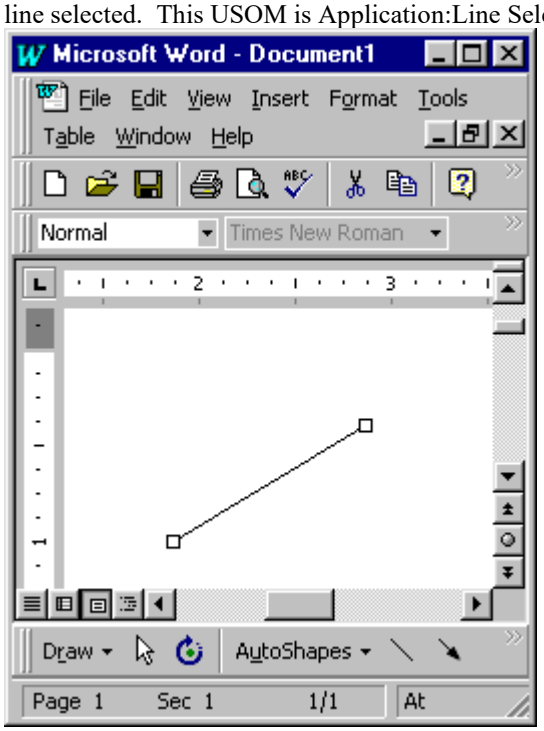
2. User selects the Draw Line tool from the Drawing tool bar located below the client area. A cross-hair cursor appears on the document. Microsoft Word is waiting for the user to select the first point of the line to be drawn. This USOM is called Draw Line:Waiting for First Point.



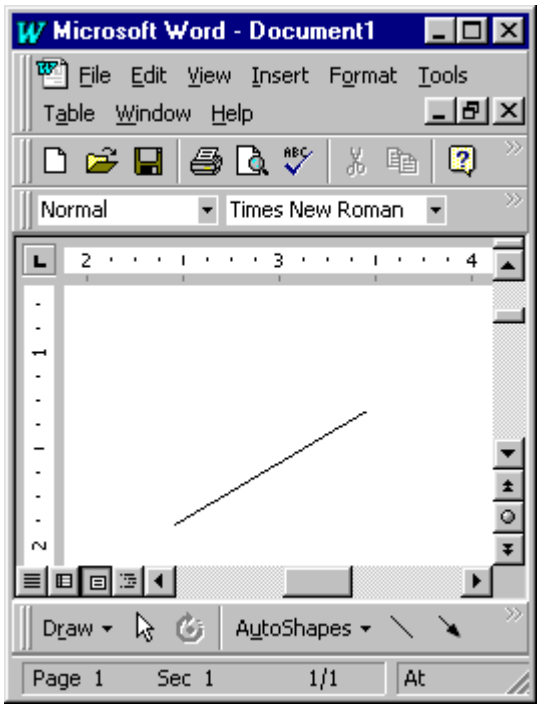
3. User selects the first point of the line by clicking down on the left mouse button. Microsoft Word remembers this first point and begins drawing a rubberband line from the saved first point to the current mouse position. This USOM is called Draw Line:Waiting for Last Point. In this state, each mouse move event causes the rubberband line to be erased and redrawn to the new mouse position. The user (and Microsoft Word) remain in the Draw Line:Waiting for Last Point USOM.



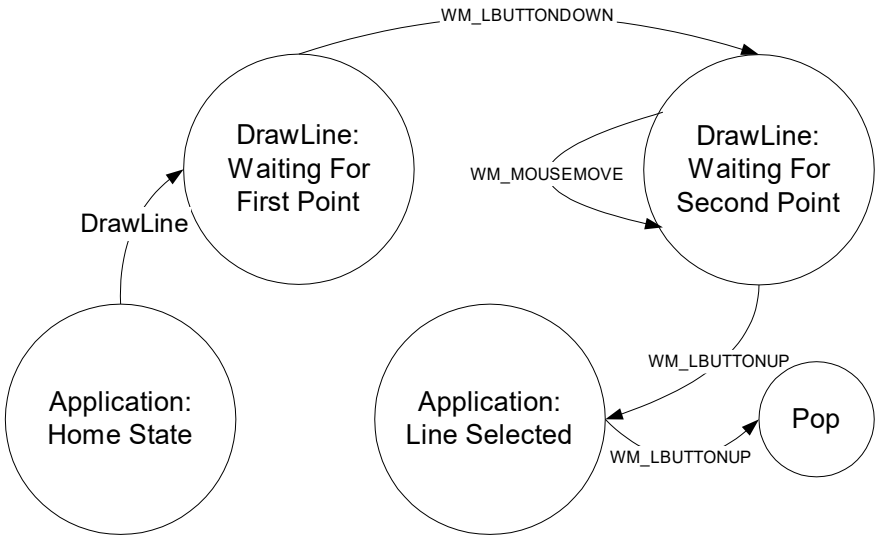
4. User releases the mouse button and Microsoft Word draws the final line from the first point to the last point; leaving the line selected. This USOM is Application:Line Selected.



5. User clicks on the background of the document and the line is no longer selected. The user has returned to the Application:Home State.



AUSOM State-Transition Diagram for the "Draw Line" Scenario



AUSOM "Draw Line" Scenario extended to include "Move Point" Functionality