SOFTWARE DESIGN DOCUMENT (SDD)

Delta3D - After Action Review (AAR)

Volume 2 - Task Tracking

Version 1.4

May 2, 2006

Authored by:
BMH Operation of
Alion Science and Technologies
5365 Robin Hood Road, Suite 100
Norfolk, VA 23513-2416
Telephone: (757) 857-5670
Fax: (757) 857-6781

For more information on Delta3D, go to the website at www.delta3D.org or contact Curtiss Murphy at cmmurphy@alionscience.com, (757)-857-5670, ext 308.

Table of Contents

Table of Figures

1 Overview

After Action Review (AAR) is a verification and validation system, heavily utilized by simulation and training applications. It plays a vital role in determining the success or failure of a simulation. AAR supports multiple levels of mission debriefing and analysis, thereby providing the necessary evaluations of a particular training event.

Delta3D is an open source 3D game engine designed to help commercial, academic, and governmental organizations create training and education applications. Therefore, it is critical for the Delta3D game engine to support, at its core, an After Action Review system. This implies a necessity for creating a generic software infrastructure that can be used for gaming as well as training and simulation applications.

This Software Design Document (SRD) discusses the objective and task tracking portions of the After Action Review system for the Delta3D game engine. The heart of this design is the concept of tasks. Tasks represent anything the user or trainee of the system can accomplish; they describe the what and the how. The task architecture is defined by two main concepts: game events and tasks.

1.1 Game Events

In order to support tasks, we need a new type of notification within Delta3D called a ‘Game Event’. Game events are simple notifications about things that have happened in the system. Some examples include “Hallway door was closed”, “Light switch toggled”, and “Player Entered Room 1”.

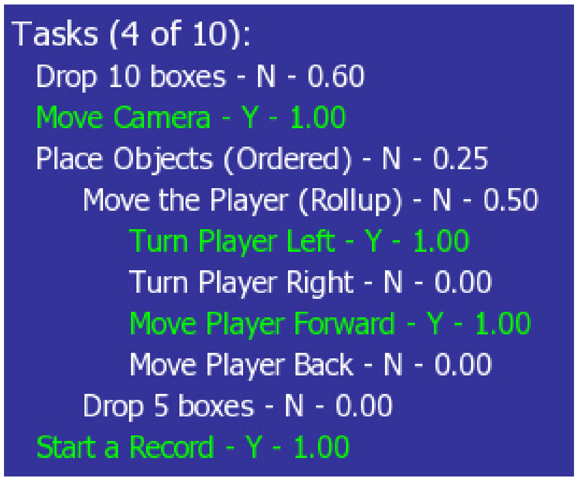
Events and event tracking will play a critical role in making tasks easier to define and process. In reality, a game event just defines a possible behavior; it has no actual behavior. Since Events are really just an item with a string name, we will need a general way to send and receive events throughout the Delta3D Game Manager. For that, we’ll use the Game Event Message.

1.2 Tasks

Tasks are the objects being analyzed and tracked within the Delta3D AAR module. Tasks are a generic concept; there are many synonyms for tasks defined in other After Action Review systems or Learning Management Systems (LMS). A task could be an objective, an interaction, an event, a step, a subtask, or a job.

In designing the task tracking component of Delta3D, we choose to refer to tasks in a generic fashion and not assign them to any specific meaning. Rather, the tools using this system shall attach meaning through the context of the tool itself. For example, a top level task may be presented as a learning objective by the tool; however, to Delta3D, a “learning objective” is merely a task. Section 3 discusses the default tasks available to the Delta3D AAR module and how each task interacts with the system.

The following image shows an example of how tasks might be defined and used. It shows the ability to mark tasks complete, have a score, and have children. It also shows the concept of root tasks, event tasks, rollup tasks, and ordered tasks. This screenshot snippet was taken from the HUD of the testAAR application.



```
Tasks (4 of 10):
  Drop 10 boxes - N - 0.60
  Move Camera - Y - 1.00
  Place Objects (Ordered) - N - 0.25
      Move the Player (Rollup) - N - 0.50
          Turn Player Left - Y - 1.00
          Turn Player Right - N - 0.00
          Move Player Forward - Y - 1.00
          Move Player Back - N - 0.00
      Drop 5 boxes - N - 0.00
  Start a Record - Y - 1.00
```

Figure 1 Task Use Case Example (from testAAR)

2 Game Events

Game Events are atomic interactions that can happen in the simulation. Each event represents a single action such as 'Apple Found', or 'Hostage Rescued'. At the simplest level, game events are just string identifiers. However, we can't just pass a bunch of strings around the system. Therefore, this section discusses the individual components and classes necessary to integrate the concept of game events with the existing Delta3D engine. Game events leverage the Dynamic Actor Layer (DAL) and the Game Manager (GM) architecture, both of which are available through the Delta3D API (See Referenced Documents). The diagram below depicts a high level overview of the Game Event system.

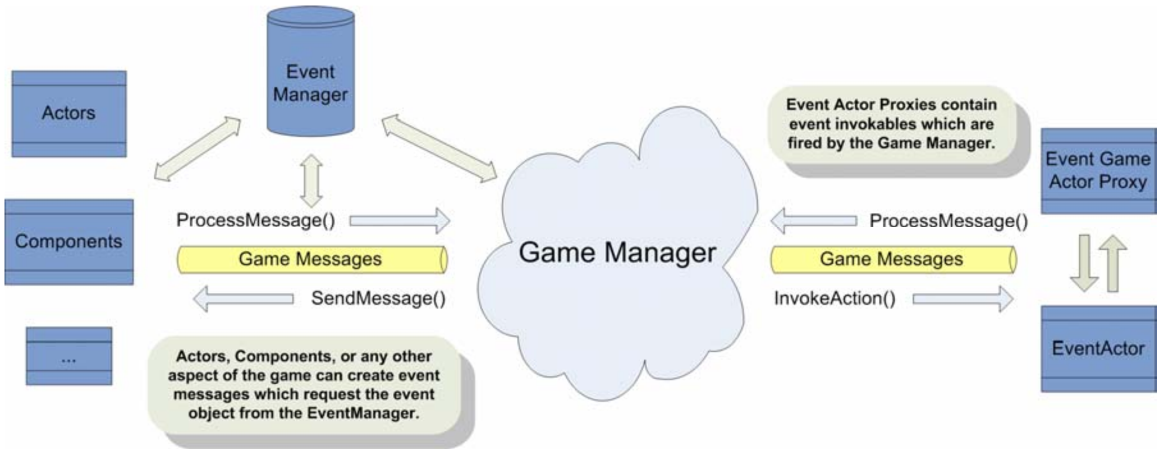

Actors
Event Manager
Components
...
ProcessMessage()
Game Messages
SendMessage()
Actors, Components, or any other aspect of the game can create event messages which request the event object from the EventManager.
Game Manager
Event Actor Proxies contain event invokables which are fired by the Game Manager.
ProcessMessage()
Game Messages
InvokeAction()
Event Game Actor Proxy
EventActor

Figure 2 Game Event System Overview

2.1 Base Game Event and Game Event Manager

A game event is a new data-type used by the Delta3D engine. To make it work, we need two new classes: a game event and an event manager to track the game events.

2.1.1 Class Diagram

The following class diagram depicts the design and functionality of the game event and game event manager classes.

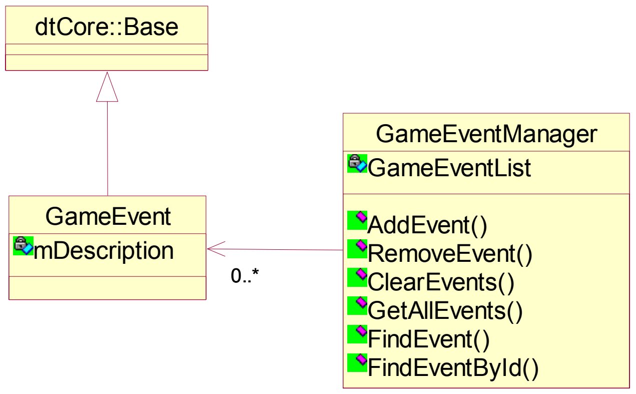


Figure 3 Game Event Class Diagram

2.1.2 Class Descriptions

The following class descriptions describe the above class diagram.

2.1.2.1 GameEvent

This is a simple class which represents a single event. Although a game event is simply a string, this class exists to provide additional meta-data for a game event. As a subclass of Base, it has a unique id that is trackable. Game Events are similar in concept to enumerations, but they can be created dynamically. For example, a map will likely have specific events that will be loaded when that map is loaded.

2.1.2.2 GameEventManager

The *GameEventManager* is a simple singleton (single instance class) which provides a central data store for the game events that are currently registered. As shown in the above class diagram, it contains methods for accessing the current list of game events, and the ability to register ("add") new game events. The *FindEvent()* method allows you to search the game event list for game events whose name matches the given string. This method supports the notion of wildcard characters; calling *FindEvent ("LightSwitch*Toggle")* would find all events where the game event name begins with "LightSwitch" and ends with "Toggle".

2.2 Game Event Property

Although game events are represented by a simple data class and managed by the *GameEventManager*, game events still need a flexible and generic method of accessing them from within the game environment. Actors need to reference them and users need to assign events to different actors. In addition, game events need to be stored in a persistent manor. The Dynamic Actor Layer (DAL) module of Delta3D provides a generic mechanism for accessing such elements using an ActorProperty. The following class diagram and class descriptions explain this concept further.

2.2.1 Class Diagram

The following class diagram illustrates the event actor property design.

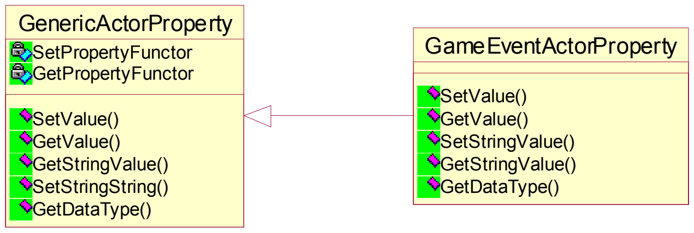


Figure 4 Game Event Actor Property Class Diagram

2.2.2 Class Descriptions

The following class descriptions describe the above class diagram.

2.2.2.1 GameEventActorProperty

This class is a subclass of *GenericActorProperty* which is the base class for most of the generic properties in the DAL. The *GameEventActorProperty* follows the same pattern as other properties in the DAL; it uses the same Functor getter/setter mechanisms exposed through *GenericActorProperty*. The method signatures for both the setter parameter and the getter return value should be of type "const GameEvent *".

In addition to defining the new property class a new data-type enumeration must be added to the DAL. This data-type will be used to identify the *GameEventActorProperty* class and will therefore be returned by the *GetDataType()* abstract method on the *GenericActorProperty* base class. The new data-type enumeration should be called "GAME_EVENT". This new data type will allow events to be assigned to Actor properties in the Simulation, Training, and Game Editor (STAGE); loaded and saved from the map; and packaged up and streamed across a network.

2.3 Game Event Message

GameEventActorProperties support generic access to events from actors and the game environment; however, a similar mechanism must be designed allowing similar access to game events from the Game Manager's messaging system. The Game Manager sends many game messages to various components and actors during the lifetime of the application. For example, the networking component receives messages that it must serialize and de-serialize across the network. The logging component logs most of the game messages that occur during the game or simulation. Therefore, it is important that game events are exposed to these systems. This section describes the game event message class and associated message parameter class which provides this functionality.

2.3.1 Class Diagram

The following class diagram illustrates the game event message and message parameter additions to Delta3D.

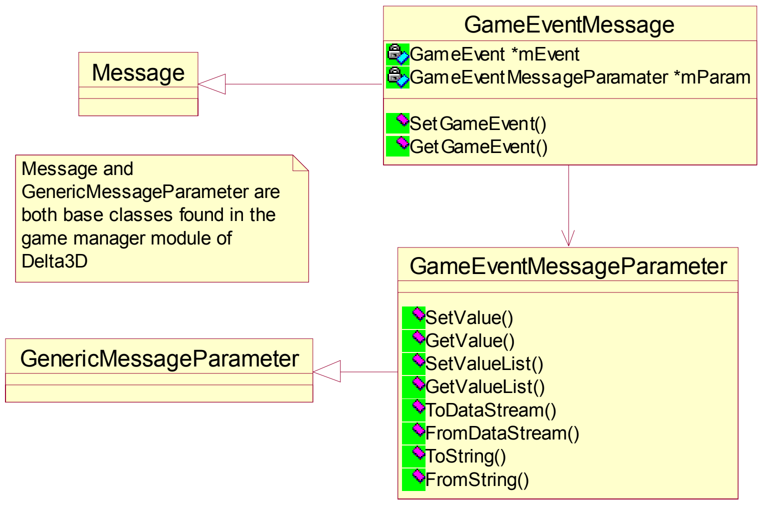


Figure 5 Game Event Message Class Diagram

2.3.2 Class Descriptions

The following section describes the above class diagram.

2.3.2.1 GameEventMessageParameter

This class is a subclass of *GenericMessageParameter* which is the base class for most of the generic message parameters in the Game Manager system. The *GameEventMessage-Parameter* follows the same pattern as the other message parameter subclasses. The *SetValue()* and *GetValue()* methods should both operate on GameEvent pointers where as the to/from DataStream and to/from String should operate on the GameEvent's unique identifier; this reduces network traffic since the *GameEventManager* contains references to all game events currently registered in the system.

2.3.2.2 GameEventMessage

The *GameEventMessage* is a simple message class extending the base *Message* class located in the Game Manager framework. It exists to make it easier to wrap GameEvents into messages which are to be processed by the Game Manager. It has two methods which set and retrieve a *GameEvent* instance. These methods then manage the underlying *GameEventMessageParameter* accordingly.

3 Tasks

The heart of this design is the concept of Tasks. As discussed above, a task in the Delta3D sense is an all encompassing term and structure for representing all tracked activities. The crux of the task design is centered on the concept of Actors and ActorProxies. The following diagram demonstrates a basic overview of the task design.

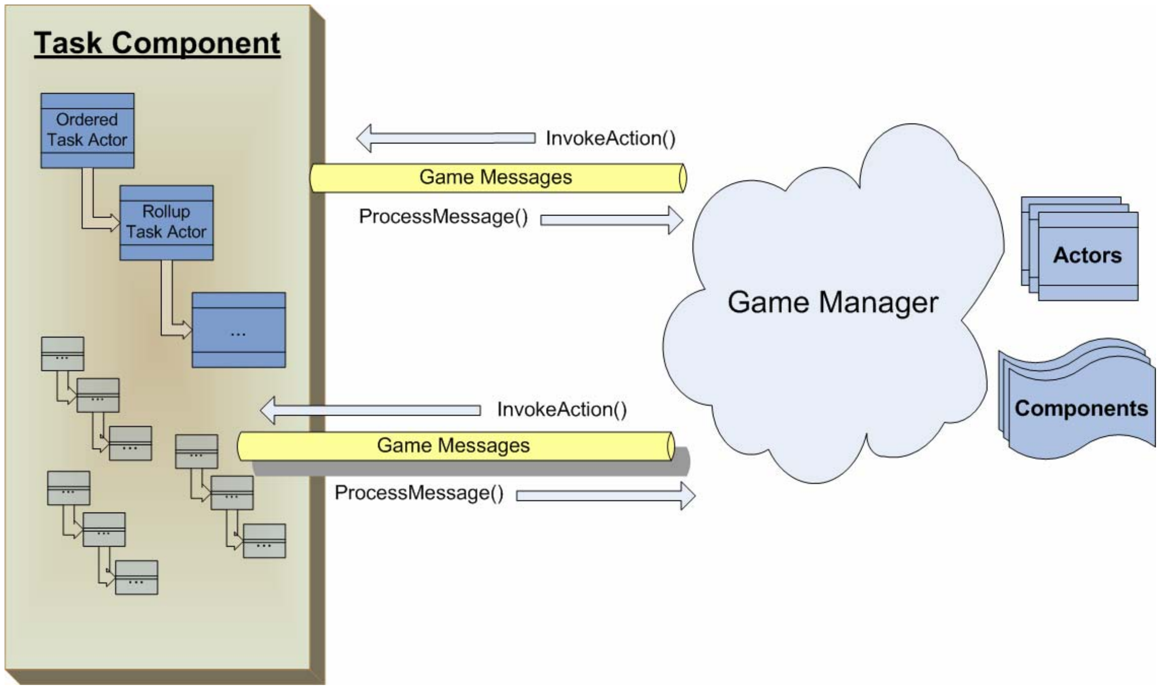


Figure 6 Task Component and Task Actor Overview

A task is a new type of Actor. The GameManager manages Actors and provides inter-actor communication. A map is composed of Actors. Making each Task into an Actor provides a perfect entry into the whole Actor framework. As an Actor, tasks will be managed by the GameManager and any registered components. This lets tasks leverage the existing messaging system, network transparency, and general object management capabilities already existing within the Delta3D engine.

3.1 Task Design

As discussed above, tasks are centered on the concept of Actors. The following class diagrams and descriptions describe this concept in more detail. In addition, this section discusses the base class for all Delta3D tasks. Note, all task actors are located in the dtActors namespace and actor library that ships with Delta3D.

3.1.1 Class Diagram

The following class diagram illustrates the base task design and the mechanisms by which it is incorporated into the Delta3D Actor concept.

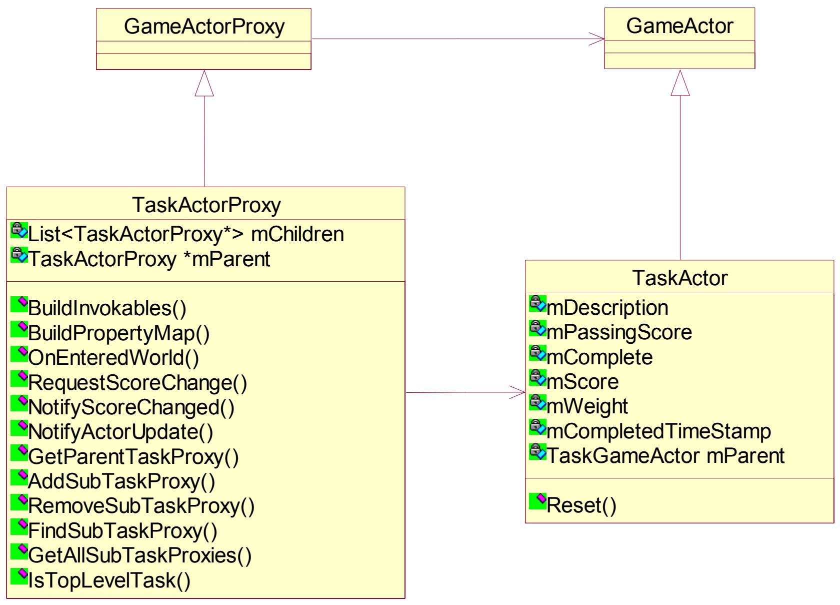


Figure 7 Base Task Actor Class Diagram

3.1.2 Class Descriptions

The following class descriptions describe the above class diagram.

3.1.2.1 TaskActorProxy

The *TaskActorProxy* is a proxy which exposes the properties of the underlying *TaskActor* discussed below. This class provides the hooks into the Game Manager architecture. It also provides the ability to register “invokables” or callbacks into the game messaging system. The task actor proxy also maintains a parent child relationship. In this manor, each task is capable of both becoming a subtask and containing subtasks.

There are two methods of particular interest with regards to the parent/child relationship amongst tasks. The following methods: *NotifyScoreChanged()* and *RequestScore Changed()* are invoked by any task which needs to update its score. The task must first get permission from all parent tasks before a score change may occur. This is to allow a parent task to control its subtasks by providing a set of rules and ensuring that those rules are correctly followed. The *OrderedTaskActor* in the section below utilizes this behavior to control the order in which tasks may be completed or modified. Once permission has be obtained, the task may adjust its score and/or mark itself complete after which the method *NotifyScoreChanged()* informs any parent tasks that a change was made to its children.

3.1.2.2 TaskActor

The *TaskActor* is the base class for all specialized task actors in the framework. This class has all the general attributes (properties) needed by a task. These define attributes such as score, weight, complete status, etc. These properties are exposed but are managed primarily though the specialized subclasses of tasks discussed below. In this way, specialized subclasses of the base task actor contain logic for evaluating completeness and scoring.

3.2 Task Component

The task component provides basic access to all tasks in the system. Its purpose is to track the existence of tasks within the Game Manager in 2 lists: root tasks and all tasks. The task component is a subclass of the base Game Manager Component (GMComponent) as shown in Figure 8. The task component is discussed in further detail in the following class description section.

3.2.1 Class Diagram

The following class diagram illustrates the task component design.

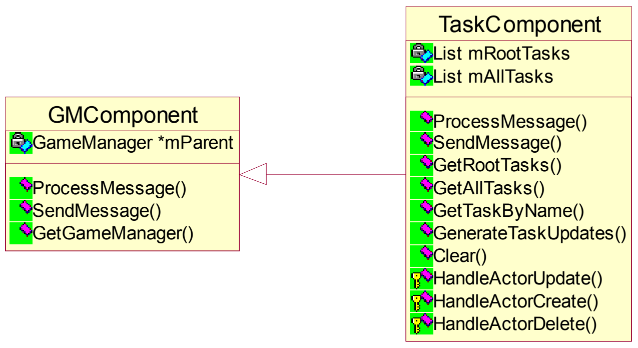


Figure 8 Task Component Class Design

3.2.2 Class Descriptions

The following section describes the above class diagram.

3.2.2.1 TaskComponent

Since the task component is a subclass of GMComponent, it is exposed to all of the messages sent and processed by its parent Game Manager. The task component uses this information to track all tasks in the system. It provides a common location for accessing the current tasks active in the game or simulation. The task component contains methods to retrieve the list of tasks currently live in the simulation. Note, there are two separate lists of tasks being tracked by the component. The first list is the root task list which represents all tasks containing no parent task. The second list is a list of all tasks - essentially a flattened version of the task hierarchy.

In order to maintain these task lists, the task component must handle three game manager messages. These messages are the actor update message, the actor create message, and the actor delete message; they are intercepted in the component's process message method and handled by the handler methods named after the game message they are to handle.

In addition to processing the above three game messages, the task component also provides an interface for broadcasting the current state of all tasks currently residing in the system. This behavior is invoked using the *GenerateTaskUpdates()* method. Upon invocation of this method, the task component loops through all of its tasks, instructs the task actor to generate an "ActorUpdateMessage" and instructs the game manager to both send and process the message.

3.2.3 Sequence Diagram

In order to understand the principles of the task component, the following sequence diagram describes the different processes for listening to Actor messages as they propagate from the Game Manager to the task component.

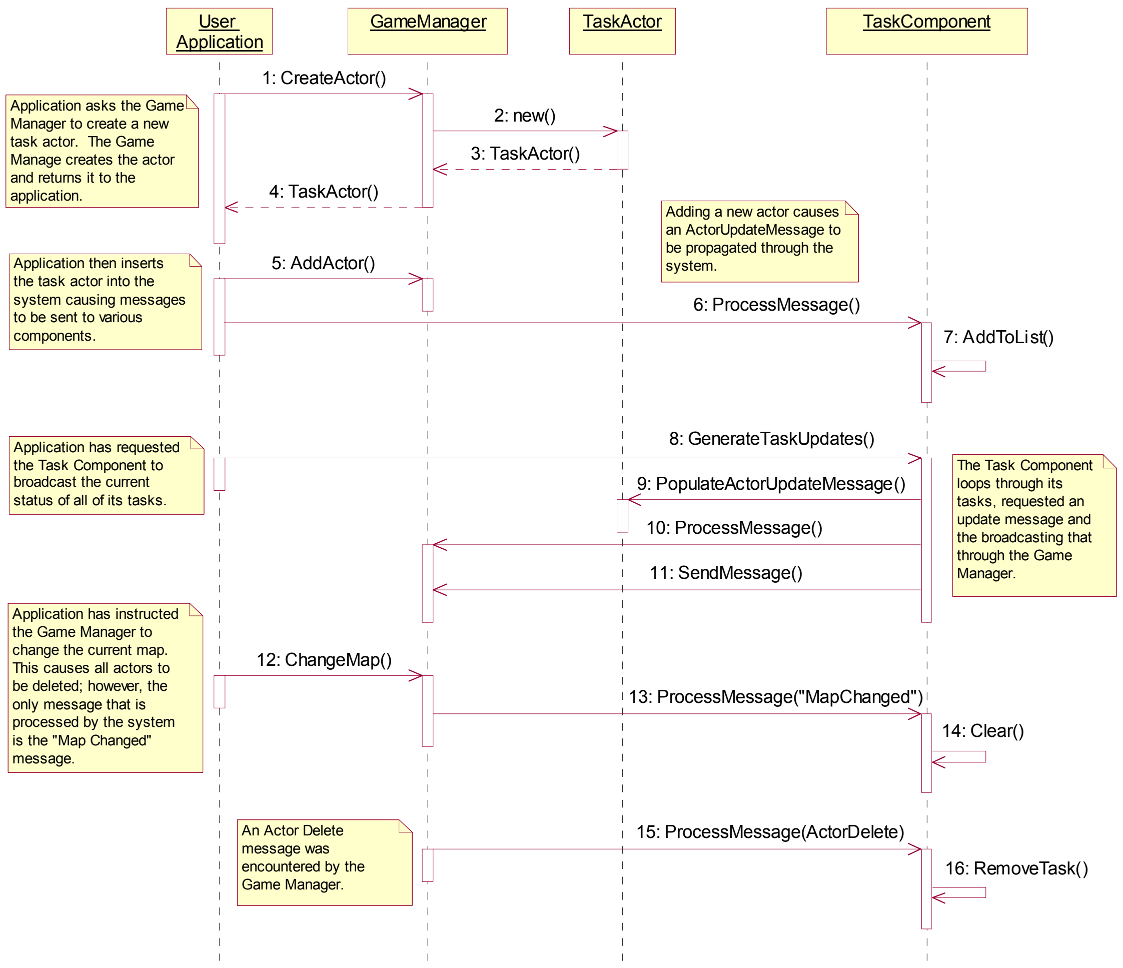

User Application
GameManager
TaskActor
TaskComponent
1: CreateActor()
2: new()
3: TaskActor()
4: TaskActor()
Application asks the Game Manager to create a new task actor. The Game Manage creates the actor and returns it to the application.
Adding a new actor causes an ActorUpdateMessage to be propagated through the system.
5: AddActor()
Application then inserts the task actor into the system causing messages to be sent to various components.
6: ProcessMessage()
7: AddToList()
8: GenerateTaskUpdates()
Application has requested the Task Component to broadcast the current status of all of its tasks.
9: PopulateActorUpdateMessage()
10: ProcessMessage()
11: SendMessage()
The Task Component loops through its tasks, requested an update message and the broadcasting that through the Game Manager.
Application has instructed the Game Manager to change the current map. This causes all actors to be deleted; however, the only message that is processed by the system is the "Map Changed" message.
12: ChangeMap()
13: ProcessMessage("MapChanged")
14: Clear()
An Actor Delete message was encountered by the Game Manager.
15: ProcessMessage(ActorDelete)
16: RemoveTask()

Figure 9 Sequence Diagrams – Task Component Actions

3.3 Rollup Task

The *RollupTaskActorProxy* is a subclass of the base *TaskActorProxy,* as noted in the class diagram below. The primary role of this class is to utilize the parent/child task relationship as a container. A Rollup Task is complete when all (or most if passing score does not equal one) of its children are complete.

In order for the rollup task proxy to determine its "completeness", it must implement the appropriate logic in its *NotifyScoreChanged()* method. This method shall be invoked by a child task when its state has been updated. Upon invocation of this method, the rollup task must check the passing score attribute found in the task actor super class. The passing score is interpreted by the rollup task as the number of child tasks that must be complete in order for it to be considered complete. The passing score is represented as a percentage of the total number of child tasks. For example, if the rollup task contains 10 child tasks and the passing score is 50% (0.5), then the rollup task's *NotifyScore Changed()* method can only mark the task complete if 5 out of the 10 child tasks are considered "complete". This is of course assuming that the task weight is equal to 1.0. As mentioned in the Task Overview section, tasks may have weights associated with them. Therefore, the weight of each task contained by the rollup task is used in the final calculation to determine the total score. Note - rollup tasks can also be nested.

The *RollupTaskActor* extends the base task actor as noted below. The rollup task has no additional properties.

3.3.1 Class Diagram

The following class diagram illustrates the rollup game task.

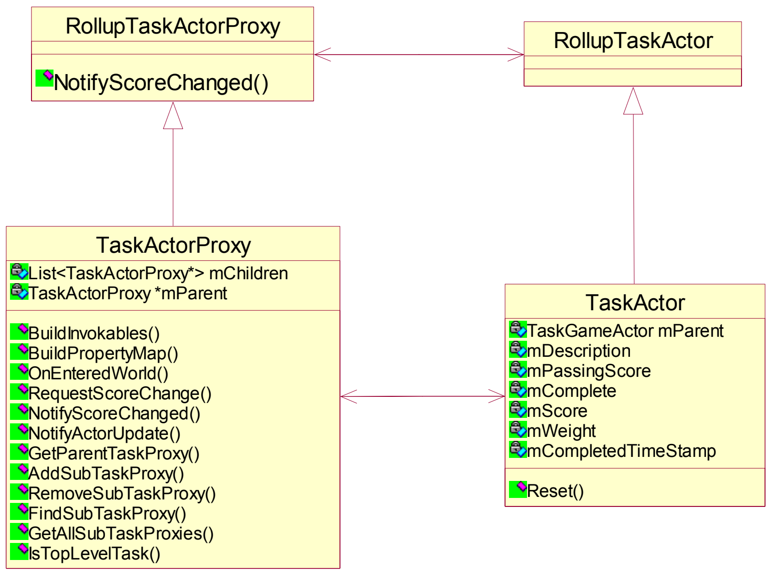


Figure 10 Rollup Task Class Design

3.4 Ordered Game Task

The *OrderedTaskActorProxy* is a subclass of the base task actor. This task specialization is similar to the rollup task with a few significant differences. The implementation of *NotifyScoreChanged()* not only uses the passing score logic similar to the rollup task, but it also enforces the concept of order. Child tasks contained within an *OrderedTaskActor Proxy* must be completed in the order specified when the child task is added. Each child task is assigned an index signaling the order with an index corresponding to their place in the ordered task's list of subtasks.

As noted in the Section 3.1, the *RequestScoreChange()* method should return a Boolean value indicating whether or not the child task "requesting" a state update should in fact be authorized to do so. This class uses that functionality when verifying order. In actuality, there are two scenarios that must be taken into consideration: block and fail. The current scenario is defined by the failType property.

The first scenario involves the case where even though the child task was "completed" out of order, the ordered task itself, still has the potential to be complete. This scenario corresponds to the enumeration FailureType::BLOCK. For example, the two tasks (for simplicity's sake) required to open a car door are: 1) Locate the car keys, and 2) open the door. If a player attempts to open the door without the car keys, he/she must simply stop that operation and go find the car keys. This implies that the ordered task can still be completed at a later time when the player locates the car keys. Child tasks that are attempted out of order will be prevented from changing their score (or any score related attribute) by having RequestScoreChange() return false.

The second scenario to be considered is an ordered task where a child task must occur in a specific order, period. This scenario corresponds to the enumeration FailureType::CAUSE_FAILURE. The ordered tasks may only be available for a certain time, or maybe doing them out of order would result in death or harm. In this case, if the child task is not completed when it should have been, the ordered task is immediately marked as a failure. Once a failure occurs on an ordered task, it can never be made complete; its children will no longer be allowed to change their complete/failure status. The *OrderedTaskActorProxy* also contains a method to retrieve the task that caused a failure. While the causing task may not be a direct child, it must be a sibling.

The *OrderedTaskActor* should extend the base *TaskActor* and in addition, contain an attribute for the enumerated type of failure the task should enforce.

3.4.1 Class Diagram

The following class diagram illustrates the ordered game task.

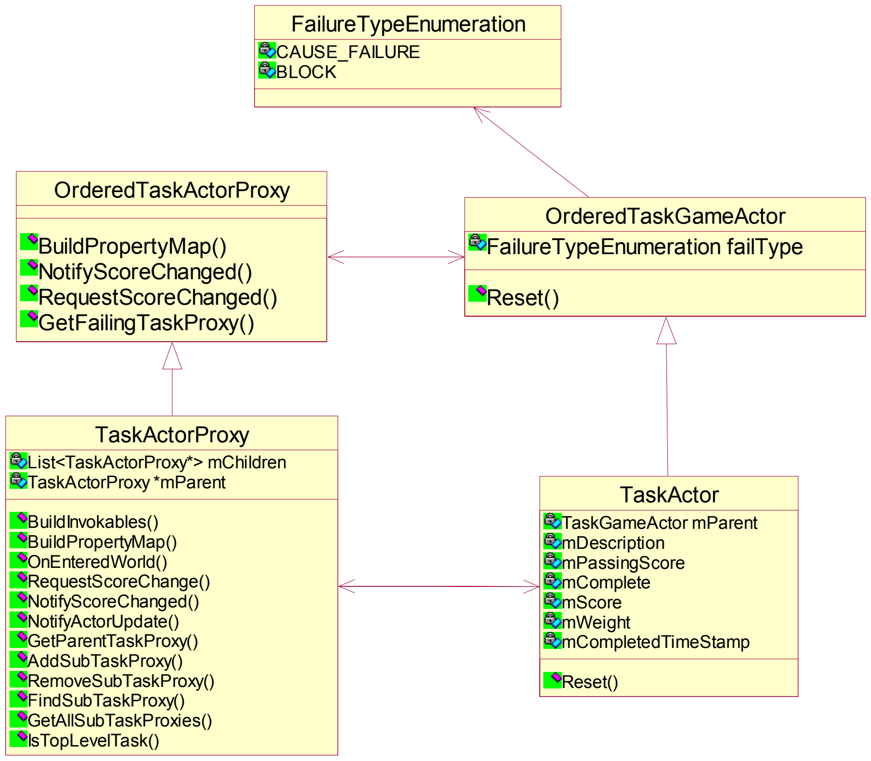


Figure 11 Ordered Task Class Design

3.5 Event Game Task

The *EventGameTaskActor* is a subclass of the base task game actor. Its sole responseability is tracking whether or not the event it contains has been fired. The event game task accomplishes this behavior by listening for game event messages discussed in Section 2.3. As noted in the sequence diagram below, upon receipt of event messages, the event task actor must verify the incoming event and update its event counters as necessary. Upon receiving a game event of interest, the actor must notify its parent using the *RequestScoreChanged()* and *NotifyScoreChanged()* methods located on its corresponding proxy.

The event task also contains the logic for tracking the number of occurrences of the particular event of interest. Using the number of occurrences and minimum number of occurrences attributes, the event task may be marked complete only if the specified minimum number of events has occurred. For example, a player may have the task of rescuing civilians from a burning building. The task states that the player must rescue five people before he/she can continue. The task can be represented using this event task class by tracking a “CIVILIAN_RESCUED” game event message and setting the minimum number of occurrences to five. This causes the event task to be marked complete when the “CIVILIAN_RESCUED” message has been fired five times indicating that in fact five civilians were saved from the burning building.

The *EventTaskActorProxy* should extend the base *TaskActorProxy* and expose the specialized attributes via actor properties found on the event task actor. It is important to note that the event task proxy must register an invokable in order for its underlying task actor to receive event messages. This is done in the *OnEnteredWorld()* method on the *EventTaskActorProxy*. This should only be registered if the actor is local.

3.5.1 Class Diagram

The following class diagram illustrates the event game task.

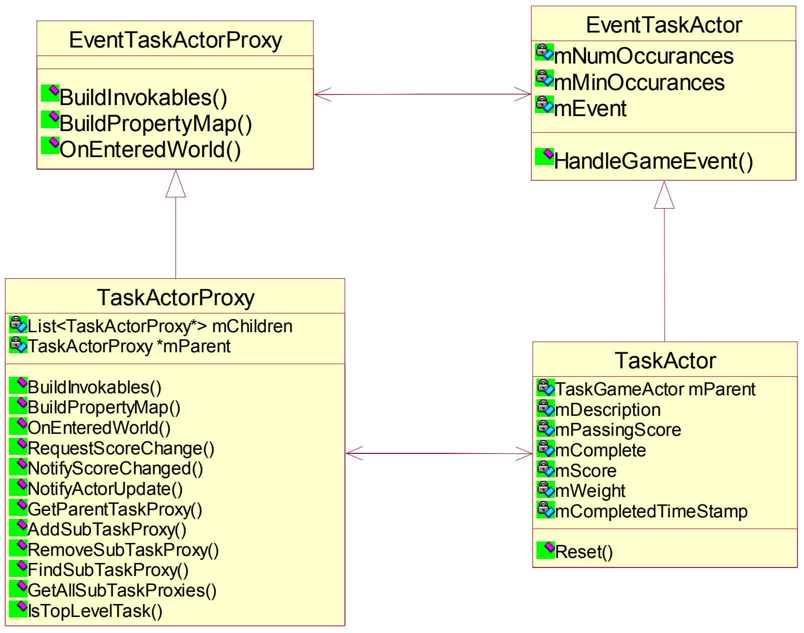


Figure 12 Event Task Class Design

3.5.2 Sequence Diagram

The following sequence diagram illustrates the process by which an event task actor gets updated and flagged for potential completion.

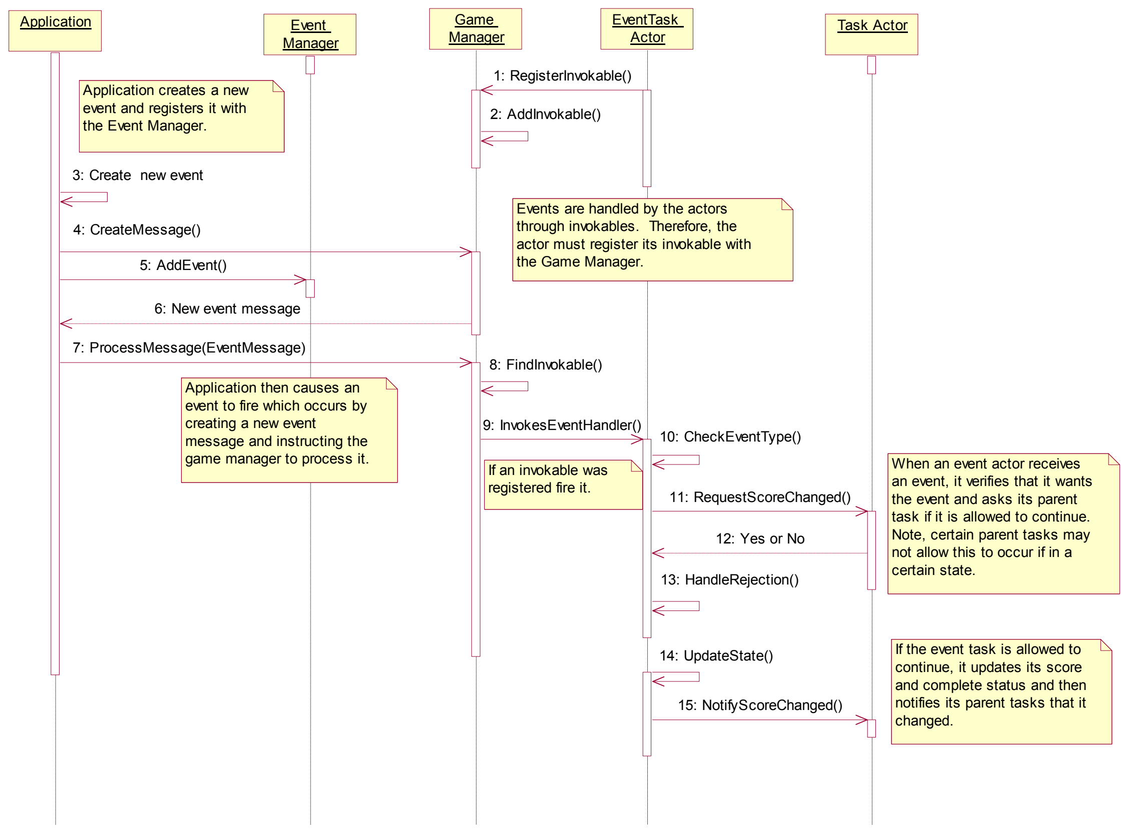

Application
Event Manager
Game Manager
EventTask Actor
Task Actor
1: RegisterInvokable()
2: AddInvokable()
Application creates a new event and registers it with the Event Manager.
3: Create new event
4: CreateMessage()
5: AddEvent()
6: New event message
7: ProcessMessage(EventMessage)
8: FindInvokable()
Events are handled by the actors through invokables. Therefore, the actor must register its invokable with the Game Manager.
Application then causes an event to fire which occurs by creating a new event message and instructing the game manager to process it.
9: InvokesEventHandler()
10: CheckEventType()
If an invokable was registered fire it.
11: RequestScoreChanged()
12: Yes or No
13: HandleRejection()
When an event actor receives an event, it verifies that it wants the event and asks its parent task if it is allowed to continue. Note, certain parent tasks may not allow this to occur if in a certain state.
14: UpdateState()
15: NotifyScoreChanged()
If the event task is allowed to continue, it updates its score and complete status and then notifies its parent tasks that it changed.

Figure 13 Event Task Actor – Sequence Diagram

4 Demonstration Application and Unit Tests

The demonstration application plays an important role in both showing the task system and fulfilling customer requirements. The demo application should show all aspects of the task system design above. More specifically it should include the following:

- Custom events that are registered by the event manager.
- Custom tasks to be tracked by the system through their complete lifecycle. i.e. Added to the system and flagged as complete or not though the actions of the user.
- Use an instance of the ordered task. (Ex: a task such as walk left, then walk right, etc.)
- Use an instance of the event task. (Ex: Drop 10 crates.)
- Demonstrate a scored task directly leveraging the passing score property.
- Use an instance of a general rollup task.
- Provide the ability to dump task status either to the console or through a user-interface.

See Section 1.2 for a picture of the tasks used in the final test application.

Note, in addition to the above list of demonstration points, all aspects of this design should be thoroughly covered with unit tests.

5 Notes

- Although not represented in this design document, game events can eventually be stored in game map (dtDAL::Map). This will enable them to be edited and loaded from STAGE.
  - This implies that the map load/save functionality needs to support the event table. In addition, the map needs to add and remove game events from the GameEventManager.
  - The GameEventManager also needs to listen for map change events.
- GameEventActorProperties need to be added to the map load/save cycle.
- There could be some potential problems with actor library dependencies. This will need to be resolved.
- Statistics are a low priority, therefore are not referenced in this document.
- All properties of each task should be supported by STAGE. The integration of this design with STAGE will occur in a different phase of this effort.

Appendix A: Glossary

Acronym	Definition
AAR	After Action Review
API	Application Programming Interface
DAL	Dynamic Actor Layer
GM	Game Manager
JNTC	Joint National Training Capability
LMS	Learning Management System
SCORM	Sharable Content Object Repository Model
SDD	Software Design Document
SRD	System Requirements Document
STAGE	Simulation, Training, and Game Editor
UI	User Interface