5-7-2015 12:00 AM

A Software Framework For Task Based Performance Evaluation

Justin J. Mackenzie, *The University of Western Ontario*

Supervisor: Roy Eagleson, *The University of Western Ontario*
Joint Supervisor: Sandrine de Ribaupierre, *The University of Western Ontario*
A thesis submitted in partial fulfillment of the requirements for the Master of Engineering Science degree in Electrical and Computer Engineering

A SOFTWARE FRAMEWORK FOR SIMULATOR TRAINING EVALUATION AND PERFORMANCE

(Thesis format: Monograph)

by

Justin Mackenzie

Graduate Program in Electrical and Computer Engineering

A thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Engineering Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

It is difficult to objectively measure performance of complex tasks such as a surgical operation and surgical simulators require the ability to evaluate performance whether to predict surgical outcome, determine competence, provide learning feedback, etc. With no standard software framework for collecting, analyzing and evaluating performance data for complex tasks in simulations, it is investigated whether a solution can be implemented that allows for custom data collection schemes, all while being general enough to be used across many simulation platforms and can be used in a simple simulator. It is also investigated whether the implemented framework can perform its functionality while leaving a small performance footprint on the simulator.

Hierarchical task analysis is investigated as a means to decompose complex tasks into their simpler sub-tasks, where data can be collected for each task and evaluated. The framework is based on hierarchical task representation to allow robust performance data of a complex task to be collected and evaluated for any type of application. A client application is developed and allows for the generation of custom scenario parameters for the task, robust performance data collection and the ability to playback previous performances for evaluation purposes. It is shown that the implemented framework has a small peformance footprint and does not affect the performance of the simulator that is using the framework for performance data collection and evaluation.

Keywords: Surgical Simulation, Performance, Human Computer Interface, Hierarchical Task Analysis,Software Architecture, Design Patterns

Acknowlegements

I would like to thank all members of my examination committee: Dr. Jagath Samarabandu, Dr. Luiz Capretz, Dr. Ali Khan and Dr. Serguei Primak for taking time out of their busy schedules to attend the examination and provide me with a great amount of input to improve not only this thesis, but my research in general.

I would like to thank my supervisors: Dr. Roy Eagleson and Dr. Sandrine de Ribaupierre. They have always been there with feedback and support during my two year masters journey. Many times they have supplied ideas and feedback that has pushed me along to this point. I would like to thank Roy for his clarification and excellent ideas that have gotten going when I was stuck. I would like to thank Sandrine for her invaluable clinical experience and expertise.

I would like to all members of Dr. Eagleson's and Dr. de Ribaupierre's lab for their feedback, help and support during my last two years. These people include: Matt Kramers, Jing Jin, Saeed Bakhshmand, Lauren Allen, Oleksiy Zakia, Dayna Noltie, Ngan Nguyen and Ryan Armstrong. I would like to especially thank Shaun Carnegie; we started our masters together and spent a lot of time working together on our research; you helped me greatly during these past two years. I wish you good luck with your thesis defense and congratuations on the birth of your children.

I would like to thank my friends who have always been there for me; Jesse Gaccione, Tyler Desplenter, Drew Redick, Allan McCulloch, Colin Ladanchuk, Brendan Logan and many others who have helped me. Even though our lives become busier and we see each other less than we used to, I know that you guys will always be there for me. I would like to especially thank my best friend and best man Scott Van Heesch who always gives me crazy software and business ideas as well as awesome stories to tell my children one day.

I would like to thank my family for their love and support; especially my mother, Judith, has always believed in me no matter what the circumstances and my sisters, Dawn and Samantha who have always been there for me when I needed them. I would also like to thank Billy, Lorena, Billy Jr. and Pamela for their love and support.

I would like to thank from the bottom of heart my uncle Jimmy, who without there is no way that I could have achieved any of this. You have influenced my life so much for the better and there is no amount of words that could describe the gratitude I have towards you. I know you're still helping me every day.

I would like to thank the two loves of my life; my beautiful daughter Isabella and my loving fiancee Brenda. Without them none of this would have happened. Bella, you make my life so much brighter and happier; you're getting so big, I can't believe how fast the time is flying by. Brenda, without your constant help and support, there is no way that I could accomplish this; you're an amazing woman and I am very lucky to have you.

Lastly, I would like thank my hero and idol, my father, John. You always gave everything you had to your family and taught me what it meant to be a man. I know that you're still helping me everyday and witnessing all of my achievements.

I dedicate this thesis to you dad. I will always keep my eyes on the prize.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Using Simulators for Surgical Training and Evaluation

Surgical simulators can take all shapes and forms; there can be inanimate artificial tissue or organs, fresh tissue or animal models and virtual reality simulation.[41, 40, 18] Inanimate simulators represent simulators that use inanimate objects [22], artificial tissue or organs to represent patients; these simulators can either fall into the low fidelity category or the high fidelity category. Fidelity is a term used to define how realistic a simulator is; a low fidelity simulator [30, 37] may not be as realistic as a high fidelity simulator, but they are usually much less expensive to research and develop. Many studies have been conducted to investigate if there is a correlation between fidelity and overall performance of the simulator and its ability to help transfer, teach and provide safety, but no results show that they do correlate.[10]

Surgical simulation offers many benefits, but simulators are still far from being greatly incorporated into surgical training. In order for simulators to be recognized as legitimate training and assessment tools that can partially replace or work in conjunction the existing master-apprentice model, simulators must be proven valid. There are a number of validities associated with simulators, these include: construct validity, content validity, concurrent validity and predictive validity. Construct validity represents the extent to which an intended trait is measurable, content validity represents the appropriateness of measures tested in the simulator to the task being trained, the concurrent validity represents the correlation between the performance in the simulator to the performance in the real operating room and predictive validity represents the correlation between the future surgical performance and the surgical performance in the simulator. [24]

One of the main goals of all simulators is to prove that a given simulator is concurrently valid, which means that if a trainee performs well in the simulator, they should perform well in the operating room. The concurrent validity goes hand-in-hand with task transfer from the surgical simulator to the real environment, in other words, if a trainee learns the skills in the simulator, they should learn those skills for the real operation as well. One of the key goals of simulators is to improve the transfer of skills from the simulator to the operating room; while, studies have not shown any real transfer [4], other more recent studies have begun to show more promising results. [15] Performance on a simulator can be a great assessment tool if it can be proven that performance on a simulator is correlated to the performance in the operating room. Studies have shown that trainees that have trained in a simulator have skills transferred to the real life operative environment.[15] Simulator-trained trainees have been shown to score significantly higher performance in surgical competency than trainees who did not train on simulators.[13] Laparoscopic simulators have been shown to improve the surgical skills as well as the quality of tasks and speed of the tasks. [21]

An over-arching goal of surgical simulators is to provide a safe and convenient means for surgical residents to train practice surgical techniques. One of the great benefits of surgical simulation is the removal of patients from the training process; this alone is incredible as it will help lower the risk to patients in the operating room. Keeping with the previous point, the use of simulators in training is the reduction in the need for patients for training.[15] The reduction in need for the patients leads to greater patient safety as well as more flexible training for trainees. With more flexibility offered to the trainees, the trainees can train more often and even in their own home on their own time, if the simulator is portable enough. In keeping with the convenience, simulators can provide instant feedback to the trainee and would not require an experienced teacher or any evaluator for that matter to be present during the training session.[4] One of the great benefits for using simulators for surgical training is that it allows the trainee to perform the training and practice at their own convenience.[4] It has been shown that trainees who use portable simulators in conjunction with the surgical training curriculum, perform better than trainees who did not. [21]

Simulators with predictive validity essentially allow for the prediction of real operations based off of the simulation. A goal of simulators is to provide the ability to be used as an assessment tool, where simulators can predict real-life surgical procedure performance. Surgical simulator results have been shown to be able to predict the surgical outcome [4], that is where the trainees performed well in the simulator, they also performed well in the real surgical procedure; thus simulators as assessment tools provide great potential.

Surgical perceptual and motor skills, just like any other perceptual and motor skills [16], can be represented through time and accuracy metrics. Examples of accuracy metrics in the surgical domain include position, trajectory, force, or unintended contacts.[24]

One weakness of the current training techniques is the lack of exposure to rare or extreme cases in the operating room. There are complications that a resident may never witness or need to experience in a surgical procedure that the resident may have to deal with after they have finished their training in a real operation. Simulation can provide trainees the abilities to experience and practice rare complications and events in a non-life threatening matter.[41] The ability to practice rare or extreme cases that otherwise the resident would never experience, they will be better equipped to deal with the complications if they arise in a real operation.

1.2 Fitts' Law: Evaluating Human Performance

In 1954, Paul Fitts investigated the human motor system's ability to perform basic movement tasks.[16] In this revolutionary paper, Fitts' conducted three experiments; the first was the reciprocal tapping task, where the user was given the task to touch the center plate, without touch the error plate that was on either side of the center plate. The second task was the disk transfer task, which required the user to transfer washers from one pin to the other pin. The final task was the pin transfer task, which required the user to transfer pins from one hole to another. From these results, Fitts was able to formulate a model for determining the difficulty of a task and the performance of a task.

Index of difficulty, ID is value that represents how difficult a particular task is to perform; the units of index of difficulty are bits. The higher the index of difficulty, the higher difficulty the task is to perform. To determine the index of difficulty the following formula was derived:

$$ID = log_2 \frac{2D}{W} \tag{1.1}$$

Where D is the distance to the target and W is the width of the target or otherwise known as the error tolerance area.

The index of performance, IP, represents how well a user performed a given task, where index of performance is in bits per second. The higher the index of performance, the better the performance by the user. Next one can calculate the index of performance, IP in the following manner:

$$IP = \frac{ID}{MT} \tag{1.2}$$

Where MT is the mean time to complete the task.

There are a few key observations to make here; firstly, the index of difficulty is directly proportional to the width of the target and the distance to the target. As the target becomes further away from the starting point, intuitively, the task becomes harder and this reflects in the index of difficulty. Similarly, when the target's error tolerance area becomes smaller, the index of difficulty increases. Index of performance is relational to the index of difficulty, where the greater the index of difficulty, the greater the index of performance; this supports such cases where the performance of two tasks are compared with one another, if the tasks are performed in the same time, but one is much more difficult, than that performance is more impressive. Index of performance is also dependent on the time it takes to complete the task, where the longer the time, the lesser the performance. The time versus performance relationship creates the speed versus accuracy trade-off dynamic. The more difficult a task becomes, the more attention the trainee pays to accuracy, the greater the time it takes to perform the task and vice versa; this leads to interesting questions such as; will the user perform the task more quickly and sacrifice accuracy or the other way around?

Another form of Fitts' law that has been widely used by researchers was proposed by Scott MacKenzie [26]:

$$MT = a + bID = a + blog_2 \frac{2D}{W} \tag{1.3}$$

Where a and b are model parameters.

This form is derived from the practice of linear regression. Given a set of performance times and index of difficulty (which can be derived from the distance and error tolerance parameters) for each time, linear regression can be used to derive a formula similar to equation 1.3 Typically researchers will vary the distance or the error tolerance to build a more robust model.

Up to this point, Fitts' law has only been used for a 1-dimensional task, but very often, tasks are more than 1 dimension. In 1997 Accot and Zhai, developed a model that was situated in the 2-dimensional domain and focused on trajectory-based human computer interface tasks.[2]

In the study, four experiments are conducted to study difficulty and performance of the tasks: the goal passing task, where the user must navigate through a number of goal posts, second was the increasing constraints task, where the user was required to navigate through a tunnel that widened, third was the narrowing tunnel task, where the user navigated through a tunnel that was narrowing as they progressed, and lastly was the spiral tunnel task, where the user required to navigate through a sprial tunnel. In all of these tasks, the users had to keep in mind both dimensions when navigating the various paths and tunnels, where the tasks in Fitts' experiments did not contain two dimensions. The following is the general form of the steering law that was found:

$$T = a + b \int_C \frac{ds}{W(s)} \tag{1.4}$$

Where T is the time it takes to navigate through the path, C that is parameterized with s. For simpler paths, such as a straight tunnel, the steering law can be simplified to the following:

$$T = a + b\frac{A}{W} \tag{1.5}$$

The paper also investigated the difficulty and performance of navigating cascading menus that are common to graphical user interfaces. When navigating a cascading menu, the user can go through multiple routes and the time it takes for each position and selection contributes to the overall time. For example, if the user needs to navigate through three menu options, the total time, T will be the sum of all of the sub-tasks (reaching each checkpoint). There has been much research in applying Fitts' law to more complex tasks such as three dimensional targeting tasks, exploring how to model index of difficulty and index of performance for three dimensional tasks.[12, 33]

Fitts' law and the following works by other researchers over the past 60 years have established an empirical and objective method to measure performance of targeting tasks. These methods only allow for the evaluation of performance on simple targeting tasks that involve only position and selection. How can a complex task such as surgery be evaluated using the methods described in this section?

1.3 Hierarchical Task Analysis

Evaluating human performance with Fitts' Law provides a means to evaluate index of difficulty and index of performance for pointing tasks. Recall, that pointing tasks are essentially composed of two types of actions; position and selection, where the user must move a device to control the position element and perform a selection to signify the end of a task. Utilizing Fitts' law, there is a way to derive an objective methodology to evaluate performance of a given task; given an accuracy and speed for the given task. There is one main issue, it is impossible to calculate an objective performance value of a complex task such as surgery. Fitts' law was designed around positioning and selection tasks, but surgery is much more complicated than that. It is possible calculate the speed of a complex task, as it is just the time required to complete that given task, but how can accuracy be calculated? How does one calculate accuracy of a surgical procedure? Hierarchical task analysis is based on the idea of developing a hierarchy of goals, where the goal represents the end means of a task. A complex task such as surgery is represented by a collection of sub-tasks that compose that complex task. A parent or complex task is decomposed into its sub-tasks to promote simplification of tasks and defining their plans and goals. Each task or goal has a plan that indicates how the child goals are carried out; the plans co-ordinate the unit (complex task composed of smaller simpler tasks)[36] Figure 1.1 displays the hierarchical definition and figure 1.2 illustrates the path or flow definition of a simple example hierarchical task analysis.

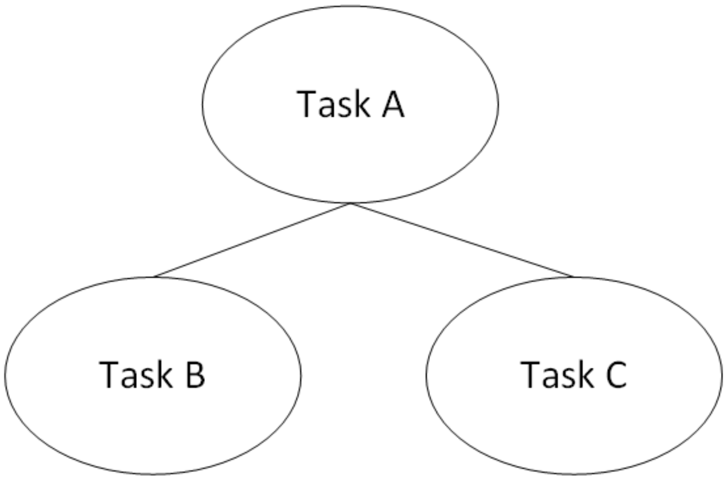


Figure 1.1: An example task hierarchy.

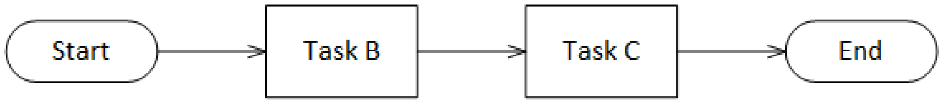


Figure 1.2: An example flow definition of a hierarchical task anaylsis.

What exactly is a task? Shepherd mentions tasks can be seen from the perspective of a problem that needs to be solved; the problem that needs to be solved will contain an initial state, responses that affect the state and an environment.[36] Hierarchical task analysis is designed to be the starting point of a task analysis and then other techniques can extend the analysis. [39] Viewing a task from this perspective illustrates one key concept; that if the purpose of a task is to complete a specific goal or solve a specific problem, then determining how well a user performed the task can be determined from whether they completed the goal or how close they were to completing the goal. Representing the tasks as a problem to be solved, simplifies both the performance and the evaluation process as both the performer and the evaluator has a clear a goal in mind on what to complete or evaluate.

A hierarchical task analysis is composed of specified task analysis, their sub operations and the plans that lay out the order of the operations.[35] The tree of tasks represents the structure of the hierarchical task definition, but the plan is what defines the flow of the task. Seeing how a hierarchical task definition is composed of both a hierarchical structure and a flow chart component (for each task), the hierarchical task definition is drawing many similarity to state charts. [20] Introducing hierarchical task analysis to the state chart domain allows for real time use of the definition as the definition will now contain a state or set of states. To make the transition from a hierarchical task definition to a state chart definition requires two main connections; one is the transformation of a task, and thus problem, to a state. It is natural to think that when a performer is performing a task to complete the goal, they are in the act or state of performing that task. For example, if the task is to insert a tool in the patient's head, before the performer completes that goal, they are in the state of inserting the tool in the patient's head. The completion of the goal of the task represents the desired input for the state to transition to the next task. Transitions in the state chart representation would be transformed from the path definition; where the source task is the completed task, the input to trigger the transition is the completion of the task and the destination is the next task in the path definition. Since state charts support hierarchy of states, the hierarchical relationships between tasks in the hierarchical task definitions are maintained. The link between hierarchical task analysis and state-space combination with transition matricies has been proposed.[9] Figure reffig:StateChartRepresentation illustrates the state representation of a hierarchical task definition.

Hierarchical task analysis definitions are mainly constructed through research in the domain and consultation with domain experts and task performers. [34, 23, 35] Software toolkits have been developed to create hierarchical task analysis definitions. [42] These tools typically involve a method in define the hierarchical task tree structure, that will contain the name and information for each node. (or task) The tools will also support the ability to define paths for the definition. The tools also offer the ability to generate the tabular representation as well.[8] Hierarchical task analysis can be used to break down a very complex task into smaller sub tasks.

1.4 Hierarchical Task Analysis for Surgical Procedures

Given the ability to break down complex tasks and goals into their sub-tasks produces many new opportunities to a variety of industries. Surgical procedures are very complex tasks that contain many sub-tasks and phases that must be completed during the surgical procedure and since the field is always looking to improve the analysis and performance of tasks, hierarchical task analysis has been researched as a viable tool to aid in analysis of surgical procedures. Hierarchical task analysis can be used to decompose a surgical operation into smaller operation tasks, where these tasks are able to be simply evaluated and provide a link between actions and errors.[14] Various research studies have been conducted around the idea of applying hierarchical task analysis to the field of surgery.[34, 35, 5] Hierarchical task analysis has also been applied outside the surgical context in regard to medication application in hospitals.[23]

A 2006 study by Sarker et. demonstrated that a hierarchical task description can be formulated for a laparoscopic surgical procedure.[34] In the experiment, expert consultant surgeons were recorded and research task analysts watched the videos, while developing a hierarchical task description of the surgical procedures. The expert surgeons were also asked to create a hierarchical task description and the results showed a very high similarity between the definitions from both sides. The study showed that the hierarchical task analysis of the surgeons

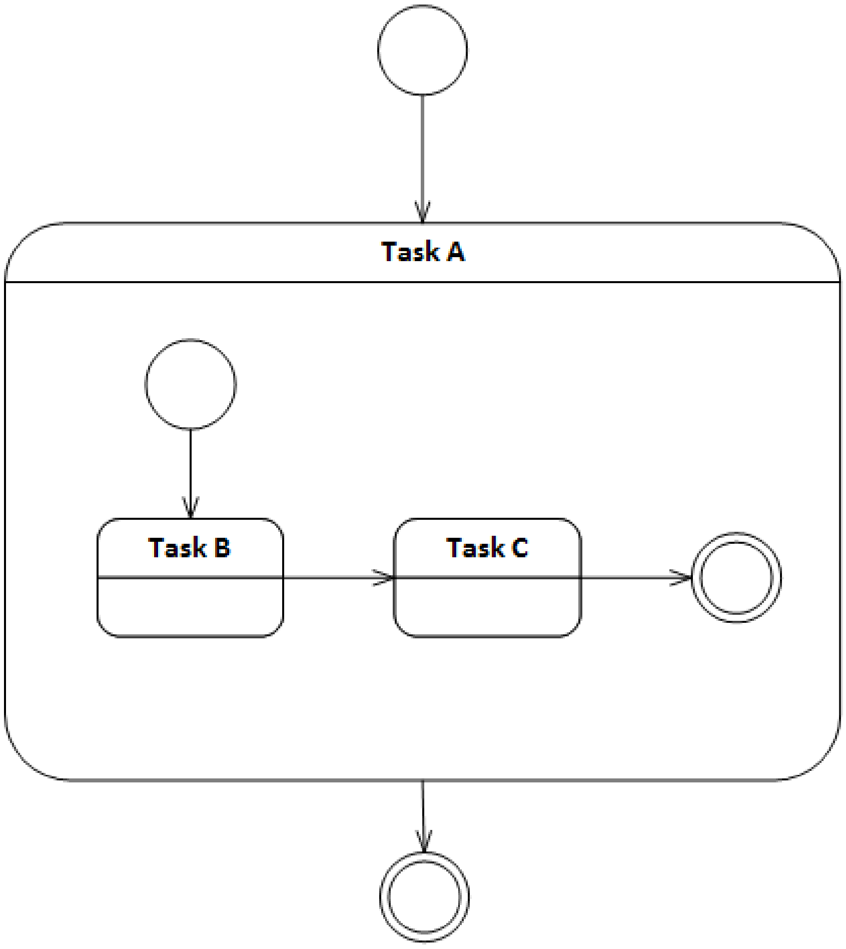


Figure 1.3: A state chart representation of a hierarchical task definition.

had content and face validity. An interesting concept was that the hierarchical task analysis provided a definition of the tasks that correlate to the interaction between team members in the operating theatre. A multiple role task analysis allows for the performance of each member of the team to be measured, but also allows for the performance of the interaction between two members to be evaluated as well.

Later in 2008, Sarker et. all once again released a study where they generalized their results to work on a number of different surgical procedures.[35] A surgical procedure is not performed single-handedly by a surgeon and the interaction between team members can be represented through hierarchical task analysis. Once again, they touched upon the idea of hierarchical task analysis its use for team interaction definition and how hierarchical task analysis can be used to illustrate how the team members will interact and work together to achieve a goal. It is proposed that key components of a surgical hierarchical task analysis can be used to develop a tool to assess the technical skill of the operator. In the paper, they proposed an eight phase that can be used to construct a hierarchical task analysis definition that effectively represents a surgical procedure.

1.5 Evaluating Human Performance Using Hierarchical Task Analysis

The main issue that was mentioned in evaluating human performance was that methods such as Fitts' Law are designed for simple targeting tasks. A complex task such as ETV or other surgical procedures cannot be simply measured using a single score, as it is not objective. Hierarchical task analysis has provided a way for task analyzers to break down a complex task into smaller tasks. It has been shown that surgical procedures can be represented as a hierarchy of tasks using the methods of hierarchical task analysis.[35] The first step of the evaluation paradigm is task decomposition, where task at hand must be decomposed into a number of sub-tasks. This decomposition is recursive and creates a tree of tasks where the nodes higher up in the tree are more abstract and complex tasks and the leaf nodes are the basic and simple tasks.

It is a principle of the analysis of human-computer interaction that tasks can be decomposed into basic components, which at their bottom level are one of the following types: Selection, Quantification, Position and Text. The hierarchical task description of user inputs completes when the sub-tasks corresponding to user interactions are iteratively decomposed until they are low-level primitives of positioning and selection. At this point, the prescription for evaluation of performance is based on Fitts' methodology, which respects the trade-off between the users' criteria of speed versus accuracy. Their sub-task performance for each interaction is the product of their speed and accuracy, averaged over a number of trial conditions, in which the difficulty of the spatial configuration of the task is varied systematically.[38, 26]

Although Fitts' law focuses on 1-dimensional cases, in an interactive immersive 3-dimensional environment, the same is true when one considers extending quantification and position from 1-dimensional or 2-dimensional cases to 3-dimensional positions, or by extension to 6 degree of freedom systems. Selection can take the form of button presses, but also within interactive virtual environments can be events where objects are made to be touched by the user (such as selecting a region, selecting a menu item from a virtual menu or by expressing a single recognizable gesture.) Text can be input from a keyboard, voice recognition, or sequences of gestures intended as the symbols for an input stream.

Let it be assumed that the HTA of a surgical procedure such as ETV can produce a hierarchy of tasks that produces a task tree structure where the leaf nodes are such simple tasks that they are essentially positioning and selection tasks. These low level tasks that are simple positioning and selection tasks can be evaluated and models can be derived using Fitts' Law. This leads to the following finding: The basic and simple tasks at the bottom of the tree can be measured quantifiably with Fitts' Law: position and selection. The over-arching performance of a task, as related to training scenarios, must be aggregated over systematically controlled variations of the index of difficulty for the sub-phases of the training task scenario categories. When assessing the performance of each sub-task, we do not advocate aggregating these performance metrics into a single overall task score for training scenarios. Rather, we advocate keeping the sub-component scores as separated so that they can be exposed to the trainee as indicators for subsequent learning or training exercises (ie. they indicate the precise aspects of a task which may require focused remedial or tutorial work). The complex and abstract tasks higher up in the tree cannot be measured in such a way, because of their complex nature. There is still one issue at hand, the performance of the low level tasks can be evaluated, but the evaluation of the higher level tasks still remain in question.

Part of the process of analysis in the construction of a hierarchical task description is that certain low-level aspects of the tasks will invariably include perceptual detection, localization, and spatial reasoning. In these phases of the task, the evaluation of user performance is formulated using experimental paradigms from psychophysics. For detection tasks, staircase paradigms can be used to establish just-noticeable difference (JNDs) [25, 7] of features that are salient to the task, in the graphical or live stream media.

By extension, aspects of the tasks which require decision-theoretic criteria to be applied, or to perform spatial reasoning in order to assess topological properties within the scene, then receiver operating characteristic models [19, 43] from signal detection theory are utilized to assess user performance in terms of the area under the curve (AUC) of the users' perceptual receiver operating characteristic (ROC) and the logarithmic transformation from the ROC to the detection error trade off (DET) curve. [27, 3]

In each case, the accuracy of detection or spatial reasoning is also an entropy measure, which is a function of the difficulty of the task in terms of its effective signal to noise ratio; similar to that of Fitts' Law. Accordingly, the user performance in the perceptual components of the task is the product of speed and accuracy, just as for the perceptual-motor components.

In order to evaluate performance of a complex, overarching task, one needs to aggregate the performance over its sequence of sub-tasks. Following this approach, any complex task in the hierarchy can be evaluated based on its sub tasks. (Which can be complex tasks as well) Given a task A that is composed of two tasks B and C, then the mean time to accomplish A will be the mean time to accomplish both B and C:

$$MT_A = MT_B + MT_C \tag{1.6}$$

Now, suppose that task B has two child tasks, D and E; the mean time to complete B would be the sum of the mean times to complete D and E.

$$MT_A = MT_B + MT_C = MT_D + MT_E + MT_C \tag{1.7}$$

By extension, any task T that is composed of a number of child tasks can be represented as such:

$$MT_T = \sum_{i=1}^{N} MT_i \tag{1.8}$$

Where N is the number of leaf nodes in the subtree under task T. Taking this equation and substituting the formula from equation 1.3, the following is obtained:

$$MT_T = \sum_{i=1}^{N} a_i + b_i ID_i \tag{1.9}$$

$$MT_T = \sum_{i=1}^{N} b_i ID_i + A \tag{1.10}$$

Where $A = \sum_{i=1}^{N} a_i$. Given this definition, it is possible to determine the mean time to complete a complex task by using it's simple position and selection sub-tasks. This representation also demonstrates how any sub-task in the hierarchical task definition affects the time to complete the complex task. Using a hierarchical representation allows the ability to investigate the data collected and evaluate the performance of all tasks in the hierarchy and to investigate the affect of various child tasks on the performance of their parent tasks.

1.6 Software Framework

The representation for hierarchical tasks, including software-based aspects that encode the interactions with methods that assess interaction task times, to subsequently assess the speed component of performance, and which aggregate each trial's position, to subsequently assess the accuracy component of performance by comparing inter-subject and intra-subject variability is the basis by which the developed simulator modules are instrumented in order to assess users' performance across their training regime.

It is natural to assume that a software-based framework for interleaving the graphics-based and physics-based interactions in the virtual training environment needs to be interleaved with the accompanying objective metrics of performance. It would be ideal that the software framework would be able to support a variety of different applications, where it can be a simulator, a simple desktop targeting task or a simulator on a mobile device. The software framework would present the ability to users to input a hierarchical task definition into custom scenarios and performance will be calculated and analyzed by the framework. Trainees and other users would be able to perform the tasks that were defined in the hierarchical task definition and while doing so, the framework will be collecting the data from the performance and calculate metrics such as speed and accuracy, such that Fitts' Law index of difficulty and index of performance values can be calculated over time. The motivation behind each consumer of the framework can vary, a simulation designer and developer will want to collect performance for the purpose of training and evaluation of a surgeon's ability to perform a real surgery.

1.7 Research Questions

Can a simulation software framework that allows the custom data collection and evaluation of the performance of a complex hierarchical task be implemented?

Can the simulation software framework be proven to robustly collect data without leaving a large performance footprint?

Can the simulation software framework be integrated into a simple surgical simulation scenario to collect meaningful and evaluate performance data for that scenario?

1.8 Surgical Simulation Frameworks

Several surgical simulation frameworks have been actively researched developed with the goal to facilitate the development of surgical simulators; these include SPRING [29], GiPSi [11] and SOFA [6] to name a few. These simulation frameworks mainly focus on the management of physics and graphics and don't provide much functionality in the realm of performance evaluation.

SPRING [29] was developed in the early 2000s by a group from Stanford. SPRING is implemented in C++, while using OpenGL for rendering and aimed to provide a general base of functionality that would be required in any simulators such as visual rendering, physics calculations, input, etc. The SPRING framework's architecture was divided into several main components, these being: visual rendering, the core, which would handle the physics and collision calculations and logic, the object and asset management system and user input, which supported haptics, network and other types of input schemes. There is no mention of any type of data collection or user performance review and evaluation in the SPRING framework.

GiPSi [11] is a simulation framework that came after SPRING and was also developed in the early 2000s. GiPSi is also implemented in C++ and provides functionality such as visual rendering, haptic interfacing, collision detection and the majority of attention to object models. The framework provides a great amount of flexibility when defining computational models to represent simulation objects, for example, heart tissue deformation. There is no mention of any type of data collection or user performance review and evaluation in the GiPSi framework.

SOFA [6] is a simulation framework that was developed in the mid-2000s is completely open source and developed in C++. SOFA's main goal is to provide a very flexible and customizable surgical simulation framework. The framework offers a very modular architecture where the functionality of the framework can be highly flexible and extendable. The main focus of this modularity is in the modelling of simulation objects, somewhat similar to the goal of the GiPSi framework. SOFA allows the ability to combine multiple modelling components through a mapping component, such that they may work together to provide the overall behaviour of an object, for example, a behaviour model may contain a collision model, a visual model, a haptic model, etc. SOFA is still open source and is still actively developed to this day and many new features have been contributed by the community that supports SOFA. SOFA did not originally have any mention of performance data collection or evaluation, but upon further review of the current state of the project, it does contain a monitoring component, that provides some monitoring and data collection functionality. The monitoring component allows the ability to visualize positions, trajectories, velocities and forces of particles in the GUI or output them to a file. This monitoring provides the ability to collect property data about objects in the scene. The performance is not calculated and evaluation is not determined by the monitoring module itself, just the ability to log properties of objects.

1.9 Contributions

The main contributions of this thesis are:

- An approach to determine human performance of a complex task using principles of hierarchical task analysis and Fitts' Law.

- An algorithm for representing a hierarchical task analysis as a state chart.

- An abstract representation of a task-based scenario is developed.

- A software component that provides flexible, extendable and robust hierarchical task performance data collection to simulator applications with very minimal overhead that is controlled by the scenario definition.
- A software component that provides playback of previous scenario performance in a variety of formats.
- A scenario for a brain tumour removal task is developed and a simulator application is developed in Unity3D for the task.
- The simulator utilizes the software components to provide performance data collection and playback functionality.

1.10 Layout of Thesis

This document will discuss a proposed software framework to conduct scenario simulations, collect data and playback a performance, as well as an example client application in a number of sections. The document contains many illustrations to aid in clarification of ideas that are written in text; types of illustrations include: UML class diagrams, UML sequence diagrams, UML use case diagrams, state charts, basic illustration diagrams, charts and screen shots. The document also contains many source code snippets; these code snippets will aid in understanding how the system is able to implement the ideas that are discussed in text. The source code of the framework and client application are written in the C# programming language and thus the code snippets reflect that.

The remaining part of this thesis is broken into four main sections:

- The Framework Overview and Domain
- Data Acquisition
- Performance Playback and Review
- Design of the Ellipsoid Orientation Matching Application
- The Implementation of the Application in Unity

The framework overview and domain will give an overview of the proposed task-based performance evaluation software framework and discuss the core module in detail. The section includes discussion on the actors of the domain and how they interact with the proposed framework, high level architecture of the system, the various modules and applications in the framework. Lastly, the section concludes with a detailed description and analysis of the core module of the framework.

Data acquisition will discuss in great deal the scenario simulator module, which is the module responsible for collecting data from a scenario performance. Module requirements, design decisions and implementation details are discussed and analyzed. Interfacing details and explainations of how the module is utilized are also presented.

Performance playback and review will discuss the playback module, which allows the ability to playback a previous performance to a viewer for the purposes of review and evaluating performance. The principles and design behind the playback are presented as well how the module interfaces with other components.

Design of the ellipsoid orientation matching application will discuss the design of an application that will require the trainee to perform a simple targeting task while using the framework to collect data and provide playback functionality. The section will discuss the hierarchical task analysis of a basic targting task and how to formulate a scenario from the analysis. Scenario set generation is discussed to provide a means for generating many scenarios for a trainee to perform. The section will also discuss the performance calculations that will be used to evaluate performance of the task.

The implementation of the ellispoid orientation matching application in Unity will discuss the implementation of the designed application using the Unity3D game engine. The implementation of re-usable Unty3D components are analyzed and discussed. Specific examples are given on how to incorporate and use the various framework functionality, such as simulated scenarios, extracting the data and playing back the performance.

Chapter 2

Scenario Simulator Framework

2.1 Framework Overview

The framework caters to many different roles in the training and performance evaluation domain. There are a few roles that exist in the domain:

- Trainee
- Experimenter
- Evaluator
- Scenario Author
- Curriculum Designer

The trainee is the actor that performs a scenario and will be a part of evaluation of the performance process. The trainee will interact with the domain through the scenario simulator application to perform a scenario and a playback application to review an performance. The experimenter is the actor that conducts the scenario that the trainee will perform. The experimenter interacts through the scenario simulator application to conduct the scenario performance sessions with the trainee. The evaluator is the actor that evaluates the performance with the trainee. The evaluator interacts through the playback application to review a performance with the trainee and they will use a data analysis application to evaluate the performance. The scenario author is the actor that creates the scenarios that are designed by a curriculum designer and performed by the trainee. The scenario author will interact with the domain through a scenario creation application, which would enable them to create scenarios with tasks, complications, etc. The curriculum designer is the actor that designs the elements to be authored in the scenarios; they interact with the playback and data analysis applications to discover short-term and long-term trends to seek improvements in the curriculum.

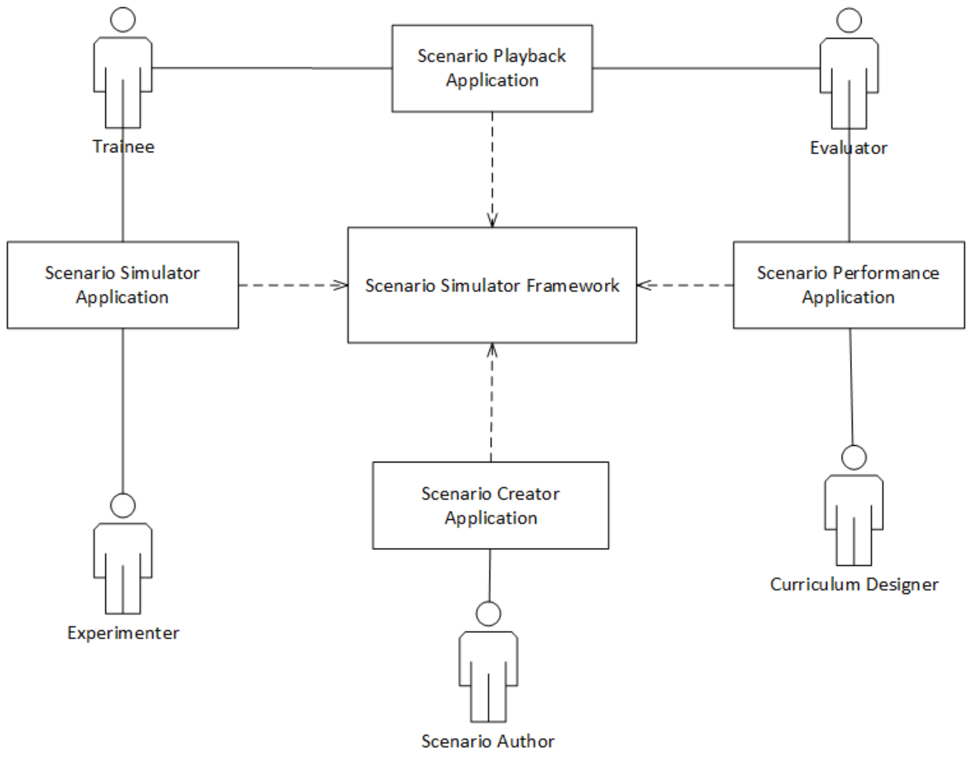

Trainee
Scenario Playback Application
Evaluator
Scenario Simulator Application
Scenario Simulator Framework
Scenario Performance Application
Scenario Creator Application
Experimenter
Curriculum Designer
Scenario Author

Figure 2.1: The actors and the domain.

2.2 Framework Architecture

The architecture of the framework is split into three main modules: the simulation, the playback and the core modules. Refer to figure 2.2 for a package diagram of the framework; which displays the three main modules and their dependencies on one another.

The core module represents all of the core entities of the framework; these entities are POCO (Plain Old C# Objects) that do not have any dependencies on any other module or framework. The core module is used by the other modules of the framework and client applications that use the framework. The benefit of using only POCOs in core module is that since they do not have any dependencies they will not have to change for any reason other than high level business policy changes. It will also be simpler to incorporate the core domain module into any other module or application, since the consumer of the package will not need to consume any other packages as well. The simulator module represents all of the components of the framework that are responsible for the functionality of the data acquisition of the framework. The simulator module depends on the core module and a third party state chart framework. The playback module represents all of the components of the framework that are responsible for the functionality of the playback of the framework. The playback module depends on the

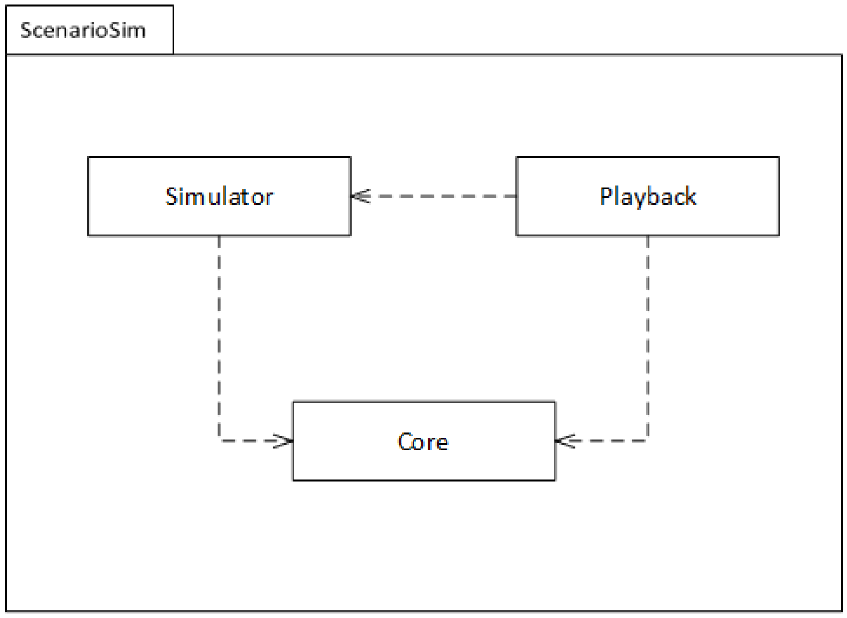


Figure 2.2: The high level architecture of the framework.

core module and the simulator module for replaying previous performances.

In section 2.1, it was illustrated and described how a variety of actors interact with different applications. The following applications would interact with the framework like so:

- Scenario Creator Application: The creator application is responsible for authoring the scenarios and thus would rely on the core module of the framework. Since the creator application does not need any simulator or playback functionality, the core is the only module of the framework that it would rely on.
- Scenario Simulator Application: The simulator application can take many forms such from a simulator to a mobile application. These applications will contain the scenarios that the user performs and will collect data through the scenario simulator module.
- Scenario Playback Application: The playback application can be used as an evaluation and learning tool. The playback application will use the playback module to review and replay previous performances.
- Scenario Performance Application: The performance application can be used to analyze and review data results obtained by the simulator module. The performance application will use the playback module to access the ability to review previous performances.

The architecture of the framework conforms to an onion architecture[32], similar to that shown in figure 2.4. The benefit of an onion architecture is that layers of the framework only depend on items that are on the same layer or interior layers. With external frameworks and details on the outside of layers of the architecture, they will not affect the interior of the architecture: the business objects (Core) and services. (Simulator and Playback) The motivation

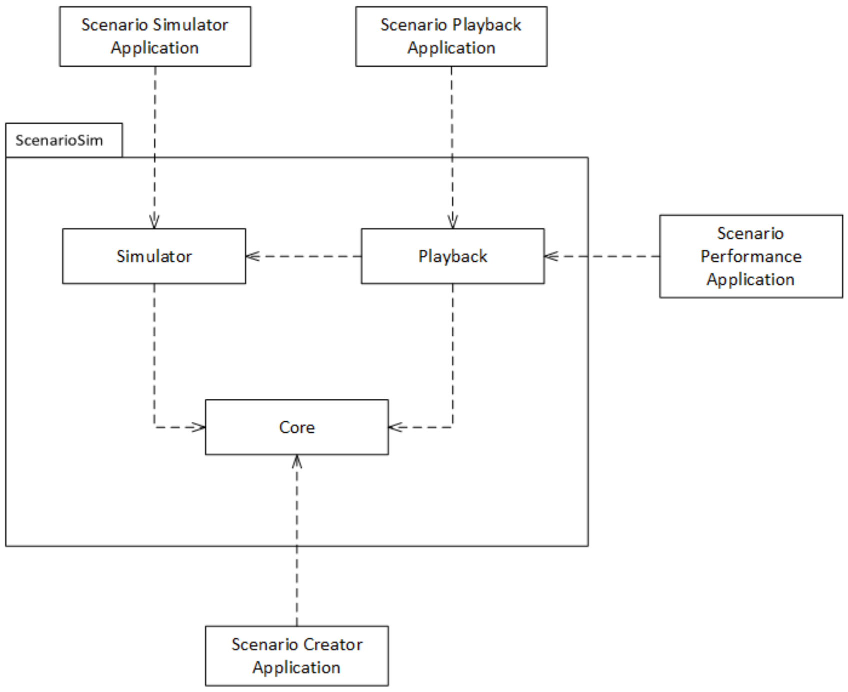

Scenario Simulator Application
Scenario Playback Application
ScenarioSim
Simulator
Playback
Scenario Performance Application
Core
Scenario Creator Application

Figure 2.3: The interaction between interior modules and exterior applications.

behind this is the Dependency Inversion Principle (DIP), which states that high level policies should not depend on the details.[28] With high level modules such as the core module and even the simulator and playback services not depending on the details of the outside layers of the architecture, the outside layers can simply be swapped out as if they are plugins. The elegance of the proposed architecture means that the framework would not care about care about details such as the persistent storage mechanisms; it would be trivial to swap out the support for SQL Server database to MongoDB to XML serialized files. Another benefit of placing implementation details on the outside of the architecture is that it allows custom simulation components, which will be discussed in section 3.6, to be easily implemented and plugged into the framework with no additional development effort.

2.3 The Core Module

One of the main goals of the framework is to be abstract enough to allow any type of scenario to be executed, for example an endoscopic third ventriculostomy surgical procedure scenario, a laparoscopic surgical procedure scenario or even non-surgical related scenario such as entering data into a web-based application. With abstraction and re-usability being a key goal for the framework, the presentation to the user must go through a client application. A client application can take many forms and it may be a simulator that is based in a 3-dimensional virtual environment, it may be an augmented reality application, a mobile application or even a form-based web application. The input from the user shall go through the client application into the framework, then the input is processed by the framework and finally, the feedback is pushed

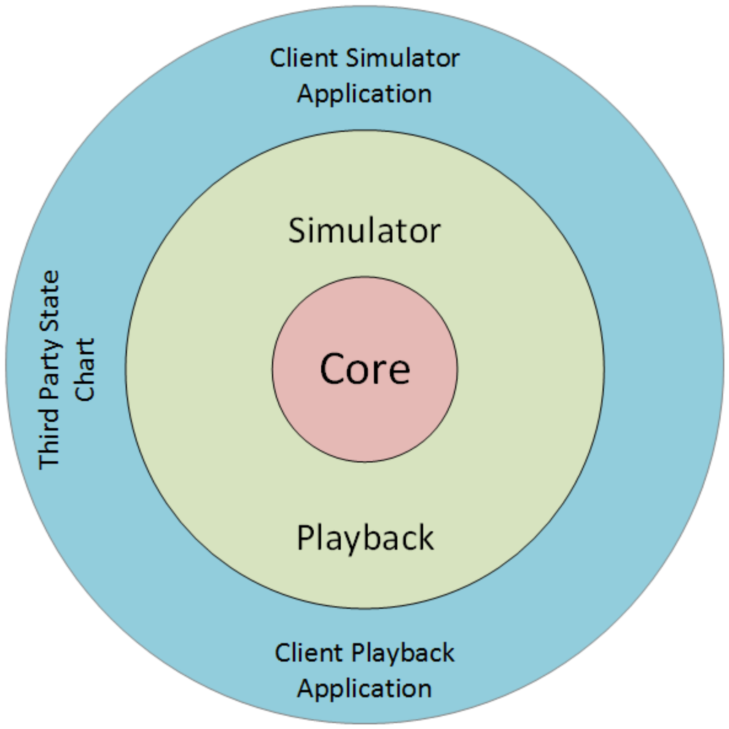


Figure 2.4: The onion architecture of the framework.

from the framework to the client application and presented to the user. While the scenario is being performed by the user, the correct flow and hierarchy of the tasks must be presented to the users through the feedback mechanism, which will be the client application.

Please note for the rest of the chapter, terms such as actors and scenarios relate to the domain of task performance analysis and not with software engineering.

2.4 Scenario

A scenario in the contexts of the framework, which is a hierarchical task-based performance evaluation framework, contains a few elements: an actor, a sequence of events and a context. The actor represents the role that submits or responds to events in the scenario, the events are a compilation of the actions performed by the user, as well as complications. The context describes the environment that the scenario takes in. Refer to figure 2.5 for the scenario class structure.

The scenario contains a task that the actor needs to perform, a list of transitions between tasks, a list of entities that represent physical objects in the environment and a list of complications that allow for custom events to be fired.

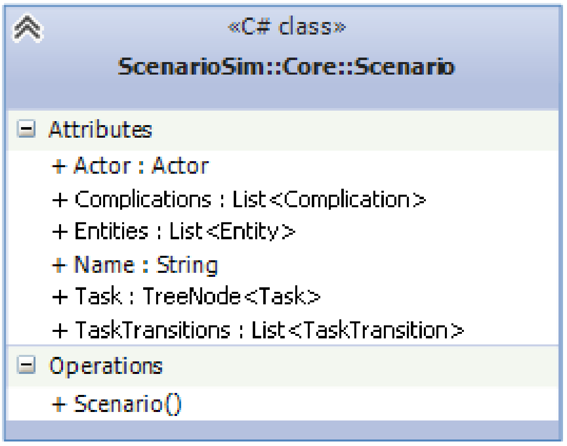

«C# class»
ScenarioSim::Core::Scenario
Attributes
+ Actor : Actor
+ Complications : List<Complication>
+ Entities : List<Entity>
+ Name : String
+ Task : TreeNode<Task>
+ TaskTransitions : List<TaskTransition>
Operations
+ Scenario()

Figure 2.5: The scenario class.

2.5 Actors

An actor is a role in the scenario that initiates or participates in the flow of events of the scenario. An actor can be controlled by the trainee or by an artificial intelligence-controlled agent. Actors in surgical simulator scenarios for example, are represented most often by 3-dimensional avatars that have the ability to move around in the 3-dimensional world and interact with the virtual world around them. Actors in other environments can be represented by 2-dimension avatars in a 2-dimensional environment, or not at all where there is no avatar for the user. The trainee's interactions alone represent an actor in the system. Thus, an actor can simply be abstracted to the role of an entity that can submit or receive events to/from the system. In the current implementation, each actor will have its own scenario. Figure 2.6 illustrates the actor class.

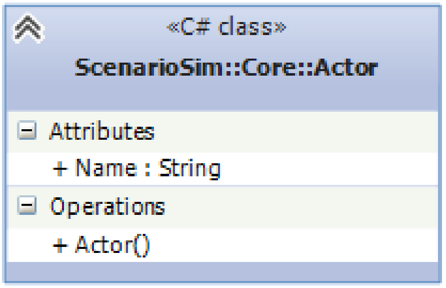


Figure 2.6: The actor class.

2.6 Task Hierarchy

To allow the ability to track performance on a task-by-task basis, hierarchical task analysis (HTA) was proposed as a solution in section 1.5. Since, HTA is based around tasks, it is intuitive to create an object to represent tasks and this leads to the *Task* class. The task class contains a name for the task and a list of accuracy metrics, which coincide with accuracy also discussed in section 1.5. Another key proponent to HTA is the hierarchical structure of the tasks, which follows an N-ary tree structure. A tree node generic data structure, with the *Task* class as its given type, where each node is a task, allows for the hierarchical task structure to be represented in the framework; for example:

```
Task matchOrientationTask = new Task() { Name = "Match Ellipsoid Orientation" };
TreeNode<Task> matchOrientationNode = new TreeNode<Task>(matchOrientationTask);
```

Given this structure, there a few properties to observe; the root node task will be the task in the scenario; an example task could be "Perform Endoscopic Third Ventriculostomy Procedure". The generic tree structure allows for simple traversal, retrieval of child objects and insertion of child objects; for example:

```
matchOrientationNode.AppendChild(positionToolTask);
```

The decision to create a generic tree node class was made to accommodate any future requirements that would require a tree-like structure. The tree node structure will contain a reference to all of its child nodes and the task node, where the task node will hold the task specific data. The tasks are essentially the goals or items to complete that will contribute to the success of the overall task. [36]

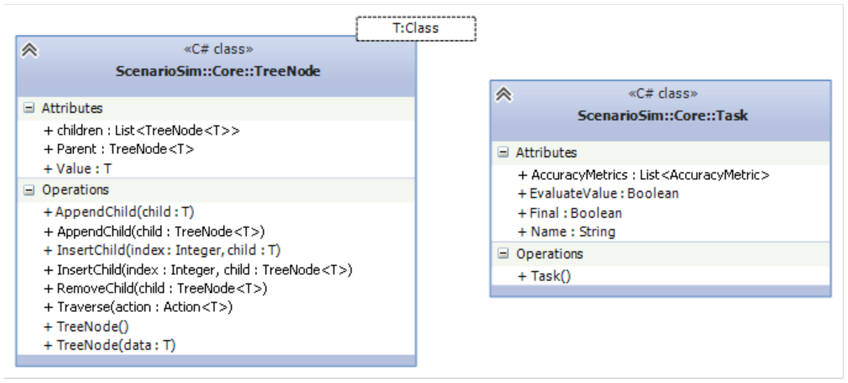

T:Class
«C# class»
ScenarioSim::Core::TreeNode
Attributes
+ children : List<TreeNode<T>>
+ Parent : TreeNode<T>
+ Value : T
Operations
+ AppendChild(child : T)
+ AppendChild(child : TreeNode<T>)
+ InsertChild(index : Integer, child : T)
+ InsertChild(index : Integer, child : TreeNode<T>)
+ RemoveChild(child : TreeNode<T>)
+ Traverse(action : Action<T>)
+ TreeNode()
+ TreeNode(data : T)
«C# class»
ScenarioSim::Core::Task
Attributes
+ AccuracyMetrics : List<AccuracyMetric>
+ EvaluateValue : Boolean
+ Final : Boolean
+ Name : String
Operations
+ Task()

Figure 2.7: The task and tree node classes.

2.7 Task Transitions

The task hierarchy represents how tasks are organized in a structural way, but not how the tasks interact with one another. There are two definitions to keep in mind with HTA: one is the structure of the tasks (this is handled by the *TreeNode* class with the *Task* type) and the flow from one task to another. HTA proposed a plan, which would organize the tasks into a chronological order, such that a flow can be established from one sub task to another. The *TaskTransition* class governs how the tasks flow from one to another. Each task transition contains the source task's name, the destination task's name and the identification number of the event or action that triggers the control to flow from one task to another. This structure resembles an entry in a state transition table.

```
TaskTransition transition = new TaskTransition() { EventId = 2, Source = "Translate Tool",
    Destination = "Rotate Tool" }
```

Refer to figure 2.8 for the class diagram of the task transition class.

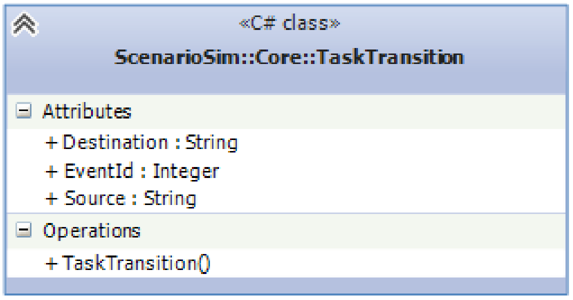


Figure 2.8: The task transition class.

2.8 Entities

The context of the scenario is the geometrical definition of the scenario, or otherwise known as the scene. The context contains the physical items in the scenario; examples of this would be a patient, the trainee or a tool. In a 3-dimensional virtual environment, 3-dimensional mesh models can represent these items in the geometric context. In a 2-dimensional environment, sprites or event UI elements such as buttons can represent entities. The entity class contains a reference to the *Transform* structure which contains three 3-dimension vector objects (*Vector3f*); position, rotation and scale.

```
Transform tool = new Transform(new Vector3f(0, 0.8f, -90.3f), new Vector3f(270, 180, 0), new
    Vector3f(0.25f, 0.25f, 0.25f));
Entity toolEntity = new Entity() { Id = 2, Name = "Tool", transform = tool }
```

The position controls the position of the entity in the context, the rotation controls the euler angles of rotation and the scale controls the scaling of the entity in the three directions. Refer to figure 2.9 for the structure and relationship between these structures.

2.9 Complications

Complications are a form of feedback mechanism from the HTA [36]; a complication represents an event that arises based on the simulation performance of a scenario. Complications

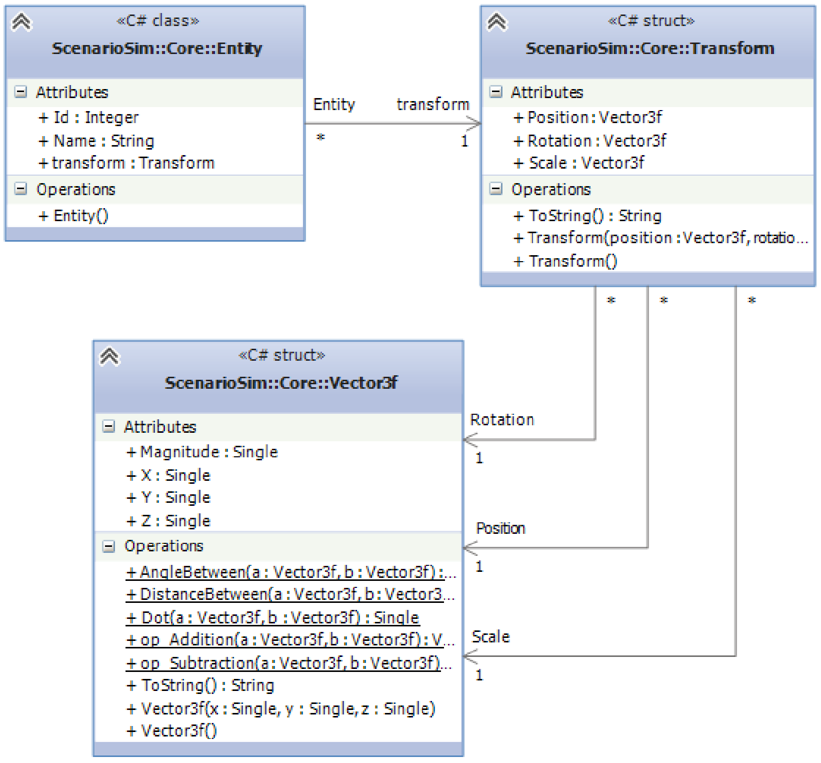

«C# class»
ScenarioSim::Core::Entity
Attributes
+ Id : Integer
+ Name : String
+ transform : Transform
Operations
+ Entity()
Entity
transform
*
1
«C# struct»
ScenarioSim::Core::Transform
Attributes
+ Position : Vector3f
+ Rotation : Vector3f
+ Scale : Vector3f
Operations
+ ToString() : String
+ Transform(position : Vector3f, rotatio...
+ Transform()
«C# struct»
ScenarioSim::Core::Vector3f
Attributes
+ Magnitude : Single
+ X : Single
+ Y : Single
+ Z : Single
Operations
+ AngleBetween(a : Vector3f, b : Vector3f) :...
+ DistanceBetween(a : Vector3f, b : Vector3...
+ Dot(a : Vector3f, b : Vector3f) : Single
+ op_Addition(a : Vector3f, b : Vector3f) : V...
+ op_Subtraction(a : Vector3f, b : Vector3f)...
+ ToString() : String
+ Vector3f(x : Single, y : Single, z : Single)
+ Vector3f()
Rotation
Position
Scale

Figure 2.9: The entity class structure.

allow scenario authors to inject custom events into an ideal scenario to introduce new obstacles to be faced by the user. Complications are the elements that separate the base or template scenario from the custom and specific scenarios. There are a variety of complications:

1. Task Entry complications: Complications that are triggered when entering the phase of completing a new task. This will be an event that is triggered from another event.

2. Task exit complications: Complications that are triggered when exiting the phase of completing a task This will be an event that is triggered from another event.

3. Timed Task Entry complications: Complications that are triggered after a certain time value after entering a new task.

4. Timed task exit complications: Complications that are triggered after a certain time value after exiting a task.

5. Random timed complications: These complications are triggered at a random time. Random events.

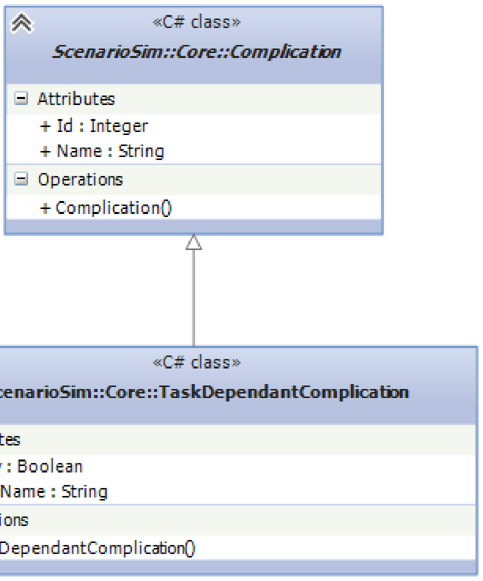


Figure 2.10: The complication class hierarchy.

2.10 Metrics

To be able to calculate accuracy metrics for a task as described in section 1.5, an accuracy metric structure was created. The *AccuracyMetric* is a abstract base class that follows the template method design pattern, where the *CalculateError* method is the abstract method implemented by the derived child classes. The accuracy metric is responsible for holding the expected value, the name of the metric, the place to find the actual value of the metric (this is in the form of a event-parameter pair). Currently, there are two sub-classes that follow the template method pattern, these are the *DirectionAccuracyMetric* and the *PositionAccuracyMetric*. The template method design pattern is used to abstract common behaviour between the position accuracy metric class and the direction accuracy metric class to a common inherited class, while keeping the different behaviour in the sub-classes though the *Calculate* method. These two metrics and their calculation will be discussed in great detail later in section 5.4.1. Refer to figure 2.11 for the accuracy metric class structure.

Once a scenario has been performed by a trainee, the results of a task need to be recorded. The *TaskResult* class is responsible for storing the results of a task; it stores the speed and a collection of accuracy metric results. The accuracy metric results stores the ideal value, actual value and error of a metric. The task result class will correspond one-to-one with the task class and thus can be used in the *TreeNode* class to store a hierarchy of results. Refer to figure 2.12 for a structure of the task result class.

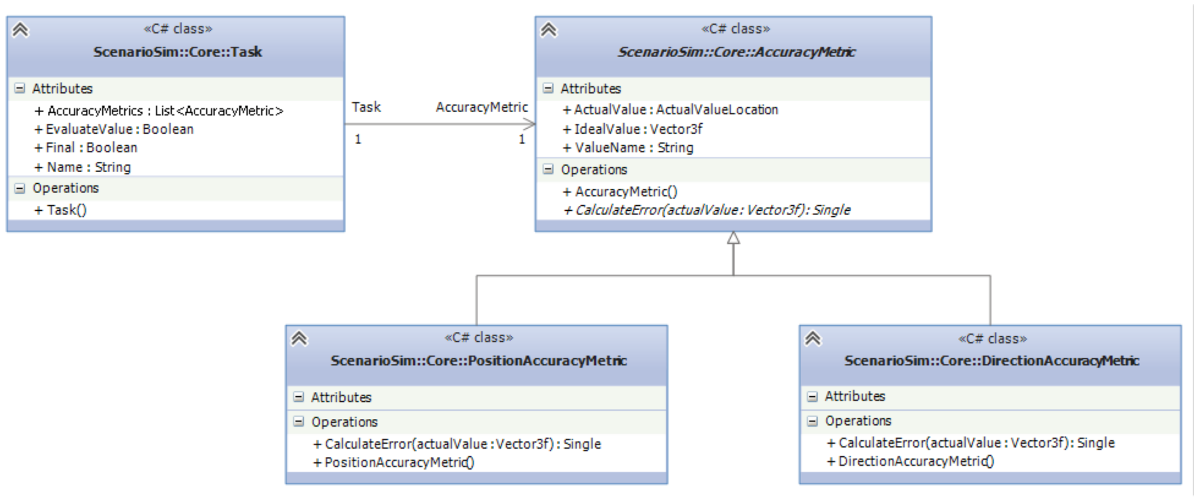

«C# class»
ScenarioSim::Core::Task
Attributes
+ AccuracyMetrics : List<AccuracyMetric>
+ EvaluateValue : Boolean
+ Final : Boolean
+ Name : String
Operations
+ Task()
Task
AccuracyMetric
1
1
«C# class»
ScenarioSim::Core::AccuracyMetric
Attributes
+ ActualValue : ActualValueLocation
+ IdealValue : Vector3f
+ ValueName : String
Operations
+ AccuracyMetric()
+ CalculateError(actualValue : Vector3f) : Single
«C# class»
ScenarioSim::Core::PositionAccuracyMetric
Attributes
Operations
+ CalculateError(actualValue : Vector3f) : Single
+ PositionAccuracyMetric()
«C# class»
ScenarioSim::Core::DirectionAccuracyMetric
Attributes
Operations
+ CalculateError(actualValue : Vector3f) : Single
+ DirectionAccuracyMetric()

Figure 2.11: The accuracy metric structure.

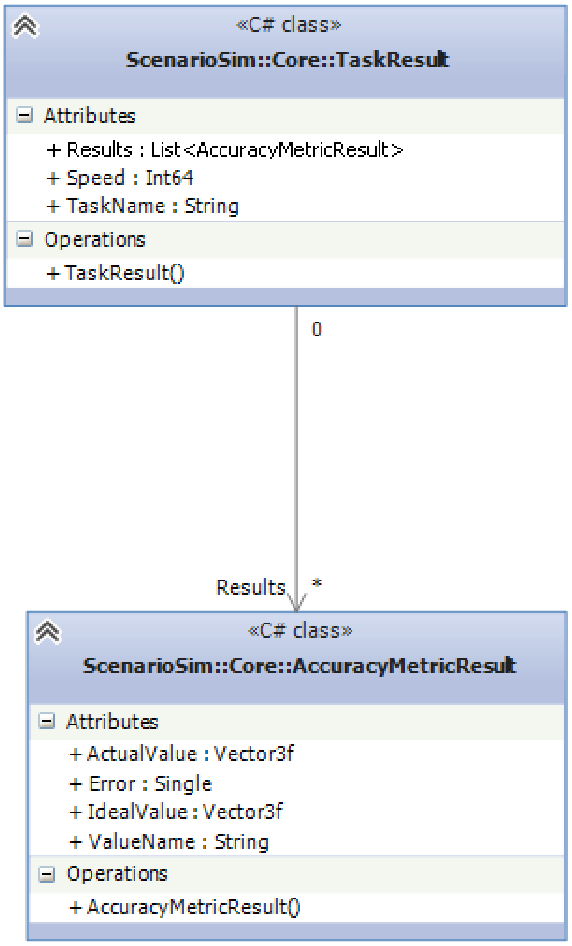


Figure 2.12: The task result structure.

Chapter 3

Data Acquisition: The Simulation Module

Recall in chapter 1 it was established through the methods of hierarchical task analysis that a complex task can be broken down into a hierarchical tree of tasks. The flow of the tasks from one to another was defined in the plan of the hierarchical task analysis. In chapter 2, the hierarchical task tree and the task transitions are defined in the scenario inside of the core domain of the framework. The scenario domain definition is great for defining a hierarchical task definition, a task flow definition, the environment of the scenario, the actors participating in the scenario and the events that arise in a scenario.

The next step is to establish the link from the hierarchical task analysis and scenario domain to the evaluation domain. In order to evaluate performance of a scenario by a user, the data of a scenario performance must be collected; the data acquisition functionality of the framework is handled by the simulation module, which will be explained in detail in this chapter. The simulator module allows a hierarchical-task-based-scenario to be played through by a user in a client application, who assumes the role of an actor in the scenario.

3.1 Requirements

The following list is composed of the current requirements of the module; these have grown over time and originally started as only an initial few.

- The complications in the scenario must be sent as feedback to the user at the correct time.
- The correct parameter data must be collected in the correct moments based on what the scenario author specifies.
- The correct metrics and calculations must be performed based on the correct events and tasks.
- To keep track of the performance of tasks, time spent performing each task must be kept. This will help with the evaluation of the speed of each task.
- When analyzing the requirements, it was difficult to envision a hierarchical structure that would essentially interact with the user through a client application.

- The component would operate as follows:
- Accept an input from the user via the client application.
- Calculate any metrics based on the current task.
- Determine if the current task was completed.
- Start the new task if it was.

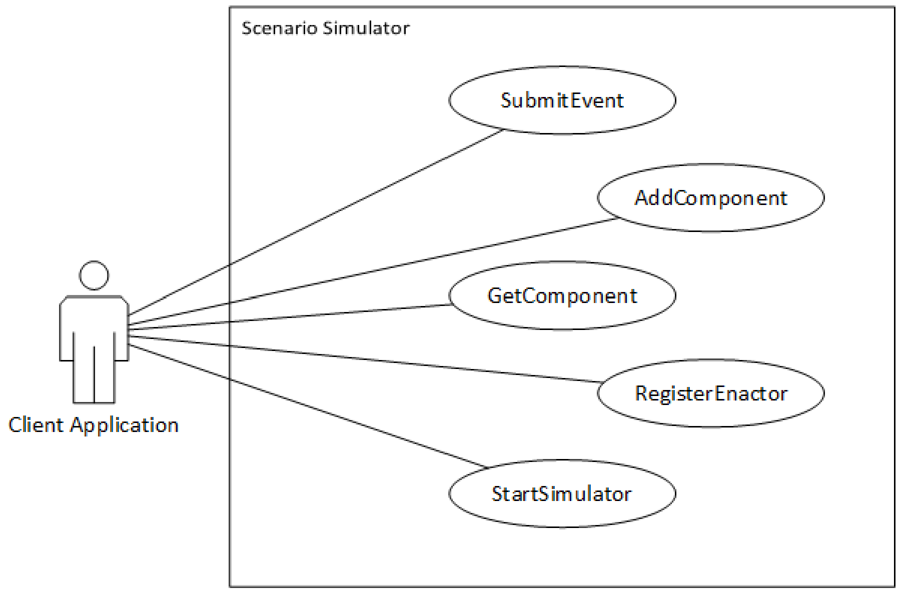


Figure 3.1: A use case diagram of the scenario simulator module

3.2 The State Chart Module

State charts were used to provide a component that would trigger complications and collect data based on the scenario and hierarchical task definition. The benefits of using state charts are the following:

- State charts allow the ability to input events into the state chart and the proper transitions will be taken.
- Intuitively a task and state correspond to one another. When the user is performing a task, they are in the state of performing that task.
- This is very similar to the flow of tasks from one to the other defined in the plan in HTA.
- It is easy to tell if a task is being started or has just finished because the corresponding state will be entering or exiting.

- It is easy to tell if a task is currently being performed, by checking if that state is currently active.
- It is easy to spawn complications based off of tasks by using the state entry or exit actions.
- The hierarchy of tasks is preserved and once again, it is simple to see what tasks are currently being performed (including the parent tasks) because it is easy to see what states are active in a state chart, even the hierarchical states.
- It is simple to produce entry, do and exit actions that are synonymous with state charts to inject complications into the task hierarchy execution.
- It is simple to create events to send into the state chart to trigger state changes.
- Parallel tasks can be represented with concurrent states in the state chart.
- Control of the behavior of the application can be based on the current task(s), since all of the active tasks (states) are known.

3.2.1 The State Chart Interface

The state chart implementation follows that of UML state chart notation, a variation of Harel state charts. With modularity and loose coupling in mind, the implementation of the state chart module is hidden behind a series of interfaces that are used by the rest of the scenario simulator module.

The motivation behind this design decision is the dependency inversion principle (DIP) [28], which states that high level modules such as the scenario simulator module should not depend on low level modules such as the UML state chart module. Both modules should depend upon abstractions. Figure 3.2 illustrates a typical architecture where a high level module such as the scenario simulator module will depend on a low level module, such as the third party UML state chart framework.

To conform to the DIP, one simply has to invert the dependency of the low level module on the high level module; this is done by creating an abstraction layer in the high level module, which the high level module's components will depend on. Next, a wrapper module or middle man module is developed outside of the high level module, which will depend on both the abstraction layer and the low level module. The wrapper module will implement the abstractions from the high level module and use the low level module to implement the abstractions. Refer to figure 3.3 for the architecture after applying these changes.

Keeping this principle in mind, allows the scenario simulator module to switch out state chart implementations or make changes to the state chart implementation with ease. Figure 3.4 illustrates the interfaces between the state chart module and the scenario simulator module.

3.2.2 The UML State Chart Module

The UML state chart module is a small wrapper module that implements the state chart interface and wraps around the third party UML state chart framework. The first class is the *UmlStateChartEngine* class that implements the *IStateChartEngine* interface and wraps around the

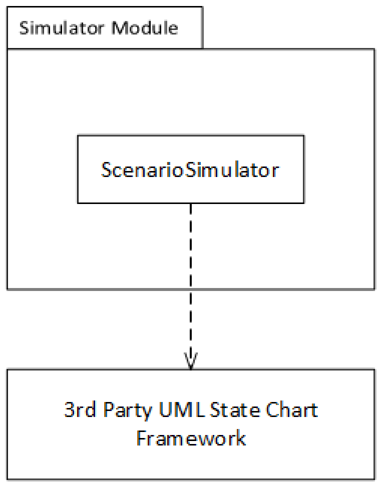


Figure 3.2: A typical architecture between high level modules and low level modules before DIP.

state chart behaviour of the third party framework. The *UmlStateChartEvent* class implements the *IStateChartEvent* interface and wraps around the *StateChartEvent* class from the third party framework. Lastly, the *UmlStateChartBuilder* class implements the *IStateChartBuilder* interface and is responsible for creating the state chart from the scenario. Refer to figure 3.5 for an illustration of the module.

3.2.3 Hierarchical Task Definition to State Chart

The state chart building algorithm takes in a scenario definition, which contains the following: a task hierarchy, which is in the form of a generic tree structure, with a task as the data type and complications, which is a generic collection of complication objects. The algorithm will produce a state chart based on the task hierarchy and the complications. The algorithm is designed around a depth-first approach, where the algorithm will recursively crawl down the hierarchy until it finds a leaf node, or simple task, and creates it. The steps below outline the algorithm:

1. Create the state chart structure based on the root task node in the hierarchical task definition.
2. Assign the name and other properties of the task to the state chart object.
3. For each child task of the root task, create a new state for that task.
4. Check if the given task has children, if it does, create a hierarchical state to represent the task; if it does not have children skip to step 6.
5. For each child of the task, go to step 4 with that given task.
6. Create a simple state object.

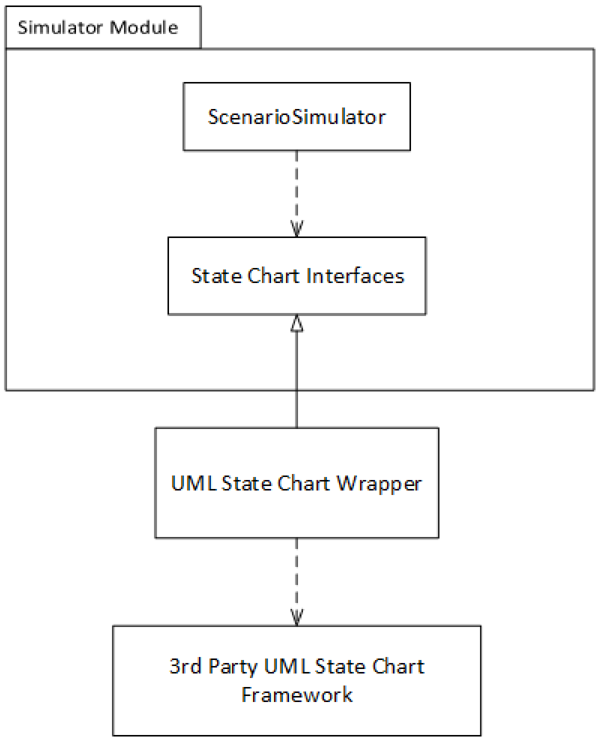


Figure 3.3: The architecture after inverting the dependency.

7. Assign the properties to the state and add it to the parent hierarchical state.

The following C-Sharp code snippet is from the build method in the *UmlStateChartBuilder* class:

```
public IStateChartEngine Build(Scenario scenario)
{
string name = scenario.Task.Value.Name;

StateChart stateChart = new StateChart(name);

List<TreeNode<Task>> childrenNodes = scenario.Task.children;

// Add each child state.
foreach (TreeNode<Task> taskNode in childrenNodes)
AddState(taskNode, stateChart);

// Add start state node and history node to state chart.
PseudoState startState = new PseudoState(string.Format("{0} Start", stateChart), stateChart,
    PseudoStateType.Start);
string startTaskName = childrenNodes.First().Value.Name;
Transition historyTransition = new Transition(startState, states[startTaskName]);

states.Add(scenario.Task.Value.Name, stateChart);

// Add all the transitions to the statechart.
foreach (TaskTransition transition in scenario.TaskTransitions)
new Transition(states[transition.Source],
states[transition.Destination], new StateChartEvent(transition.EventId));

// Add complications to the state chart.
AddComplications(scenario.Complications);
```

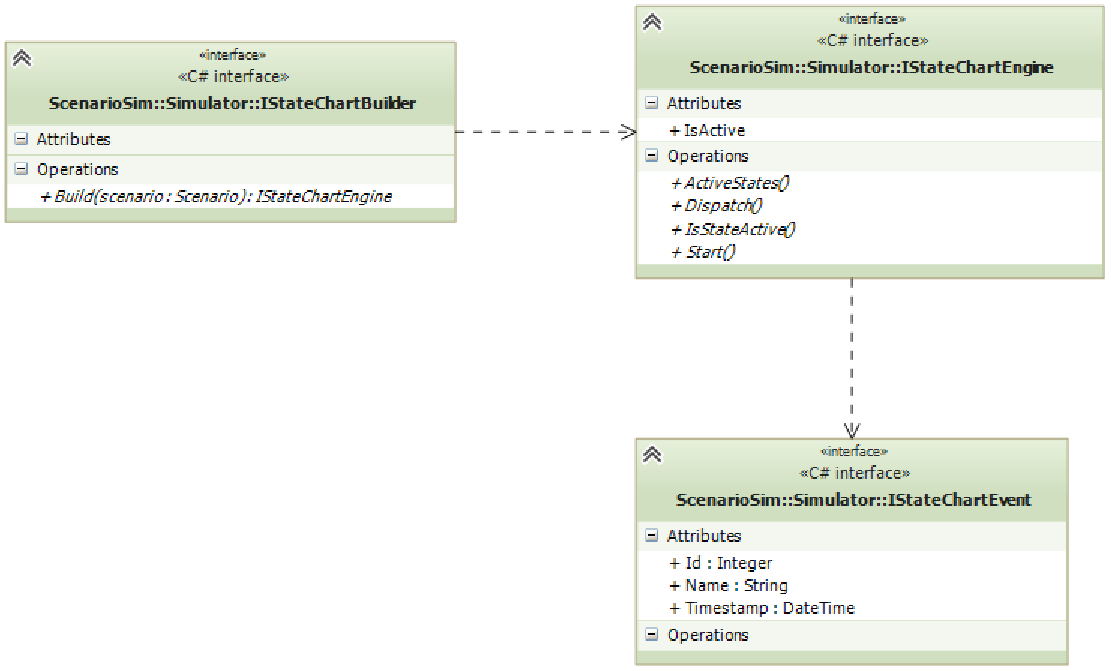


Figure 3.4: A class diagram of the state chart interface.

```
return new UmlStateChartEngine(stateChart);
}
```

The next method is the add state method, which will recursively add the tasks as states in the state chart:

```
private void AddState(TreeNode<Task> taskNode, Context parent)
{
string name = taskNode.Value.Name;

// If it is a final task, add it as a final state.
if (taskNode.Value.Final)
{
FinalState state = new FinalState(name, parent);
states.Add(name, state);
return;
}
List<TreeNode<Task>> childNodes = taskNode.children;

// Check if there are children.
if (childNodes.Count > 0)
{
// Create a hierarchical state, since this task has children.
HierarchicalState state = new HierarchicalState(name, parent, null, null);

// Recursively add the child tasks.
foreach (TreeNode<Task> childNode in childNodes)
AddState(childNode, state);

// Add the history and start state nodes for the hierarchical task.
PseudoState historyState = new PseudoState(state.Name + " History", state,
    PseudoStateType.History);
PseudoState startState = new PseudoState(state + " Start", state, PseudoStateType.Start);
```

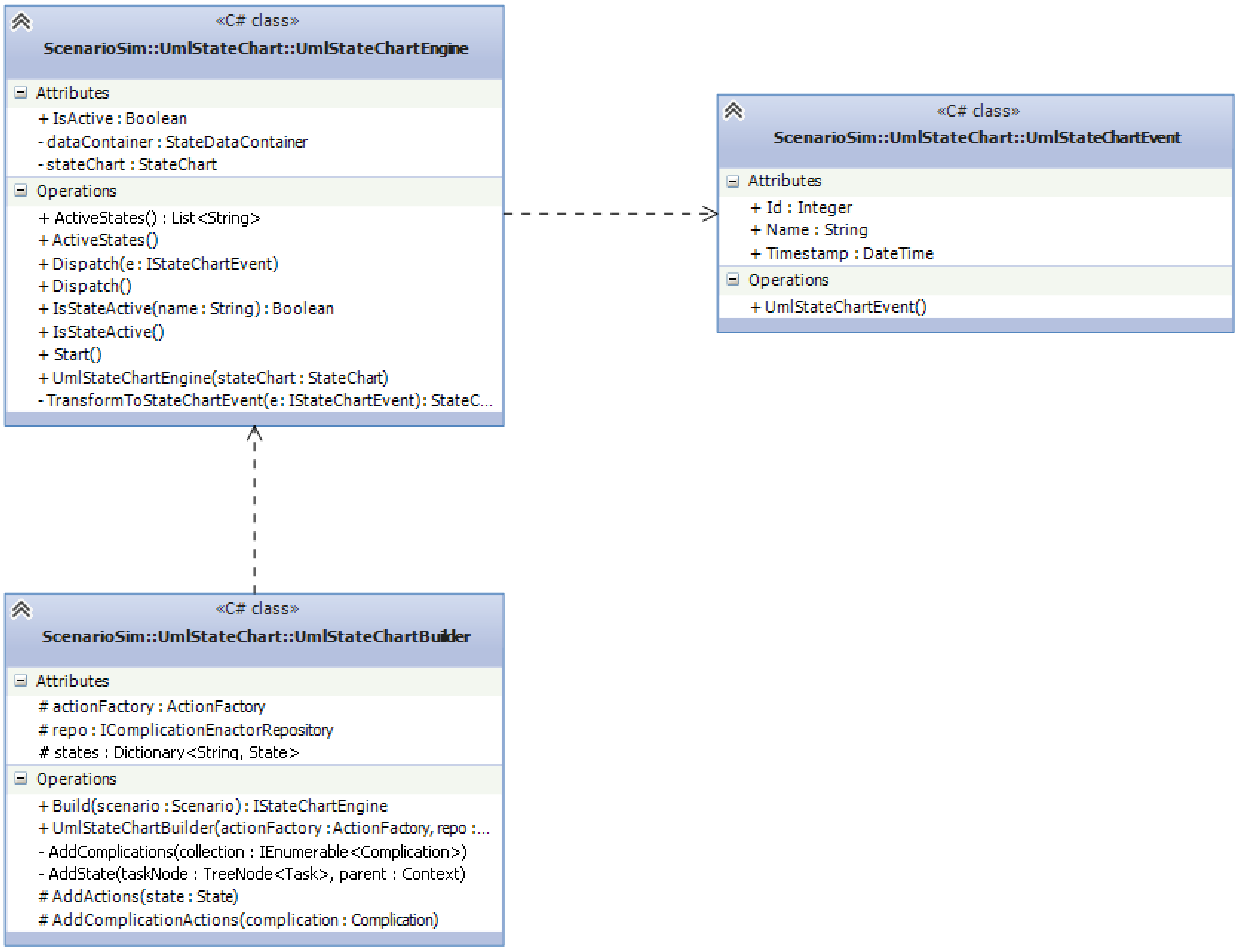

«C# class»
ScenarioSim::UmlStateChart::UmlStateChartEngine
Attributes
+ IsActive : Boolean
- dataContainer : StateDataContainer
- stateChart : StateChart
Operations
+ ActiveStates() : List<String>
+ ActiveStates()
+ Dispatch(e : IStateChartEvent)
+ Dispatch()
+ IsStateActive(name : String) : Boolean
+ IsStateActive()
+ Start()
+ UmlStateChartEngine(stateChart : StateChart)
- TransformToStateChartEvent(e: IStateChartEvent): StateC...
«C# class»
ScenarioSim::UmlStateChart::UmlStateChartEvent
Attributes
+ Id : Integer
+ Name : String
+ Timestamp : DateTime
Operations
+ UmlStateChartEvent()
«C# class»
ScenarioSim::UmlStateChart::UmlStateChartBuilder
Attributes
# actionFactory : ActionFactory
# repo : IComplicationEnactorRepository
# states : Dictionary<String, State>
Operations
+ Build(scenario : Scenario) : IStateChartEngine
+ UmlStateChartBuilder(actionFactory : ActionFactory, repo :...
- AddComplications(collection : IEnumerable<Complication>)
- AddState(taskNode : TreeNode<Task>, parent : Context)
# AddActions(state : State)
# AddComplicationActions(complication : Complication)

Figure 3.5: The UML state chart wrapper module.

```
string startTaskName = childNodes.First<TreeNode<Task>>().Value.Name;
Transition startTransition = new Transition(startState, historyState);
Transition historyTransition = new Transition(historyState, states[startTaskName]);
states.Add(name, state);
AddActions(state);
}
else
{
// Add a simple state, since this task did not have children.
State state = new State(name, parent, null, null);
states.Add(name, state);
AddActions(state);
}
}
```

Lastly, to add the complications to the state chart, the *EnactComplicationAction*, which derives from the UmlStateChartAction is added to the entry or exit action of the desired state/-task. How complications are sent back to the client application is discussed in greater detail in section 3.9. The following code illustrates the addition of the enact complication action to the state chart:

```
private void AddComplications(IEnumerable<Complication> collection)
{
foreach(Complication complication in collection)
AddComplicationActions(complication);
}

protected virtual void AddComplicationActions(Complication complication)
{
if (!(complication is TaskDependantComplication))
return;

TaskDependantComplication c = (TaskDependantComplication) complication;
if (c.Entry)
states[c.TaskName].EntryAction = new EnactComplicationAction(repo, c.Id);
else
states[c.TaskName].ExitAction = new EnactComplicationAction(repo, c.Id);
}
```

3.3 Simulator Module Interface

The scenario simulator and data acquisition module contains a single boundary interface to client applications, *IScenarioSimulator*. This interface is responsible for providing a means for client applications to interact with the simulation module. The main functionality that the interface provides is the following:

- Adding a simulator component to customize the moduleâĂŹs behavior.
- Adding an enactor to provide custom feedback for complication events.
- Starting the simulation.
- Submitting a scenario event to the module.

Please refer to figure 3.6 for the definition of the *IScenarioSimulator* interface. The decision to place all of the functionality of the module into a single interface was made to provide the developers of the third party application with one single and easy-to-use interface. This will result in a developer only needing to become familiar with one main interface, instead of having to learn many interfaces. The interface at this point is still rather slim, so it does not violate the interface segregation principle, the 'I' in the SOLID software design principles.[28] As the interface continues to grow, it would be in the best interest to find related functionalities to break into separate interfaces.

3.4 Scenario Events

A scenario event is an event received from the client application, such as the Ellipsoid Matching client described in chapter 5. The event will mark any event that pertains to the scenario. An example of this would be if a tool was translated, a tool was rotated or even a button was pressed. The scenario events are sent to the boundary interface from client, which in turn will submit the event to the state chart component and subsequently all simulator components. The scenario event consists of several data fields used to keep track of data:

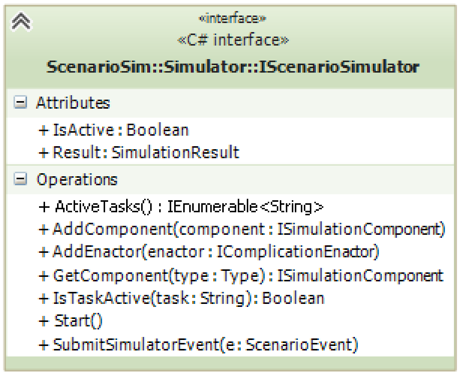

«interface»
«C# interface»
ScenarioSim::Simulator::IScenarioSimulator
Attributes
+ IsActive : Boolean
+ Result : SimulationResult
Operations
+ ActiveTasks() : IEnumerable<String>
+ AddComponent(component : ISimulationComponent)
+ AddEnactor(enactor : IComplicationEnactor)
+ GetComponent(type : Type) : ISimulationComponent
+ IsTaskActive(task : String) : Boolean
+ Start()
+ SubmitSimulatorEvent(e : ScenarioEvent)

Figure 3.6: The IScenarioSimulator interface.

- Timestamp: When the event occurred.
- Name: The name of the event.
- Id: the unique identifier of the event.
- Description: A more detailed description of the event.
- Parameters: A collection of parameters (their name and value) that represent the data of the event. These would be similar to event arguments.

When the scenario event comes through the boundary interface to the simulator object, it is sent to the state chart component for processing. The event is then sent to all of the simulator components that are registered in the module; these components are mentioned later in section 3.6. These events act as a road map of the scenario. If one would go through each event that is submitted to the module, they would know exactly what the user did each step of the way.

3.5 Scenario Simulator

The *ScenarioSimulator* is the class that implements the *IScenarioSimulator* interface; it is the component that orchestrates the logic of the simulator module. It acts as a container for the simulator components, which are described in detail in section 3.6. The scenario simulator class began as a class contained many responsibilities and only delegated a select few to other components; it handled much of the functionality of the module. Such responsibilities included:

- Orchestrating all of the module's logic.
- Directly calling the logging of the events.
- Directly calling collection of scenario events.
- Directly calling the serialization and saving of the scenario events.

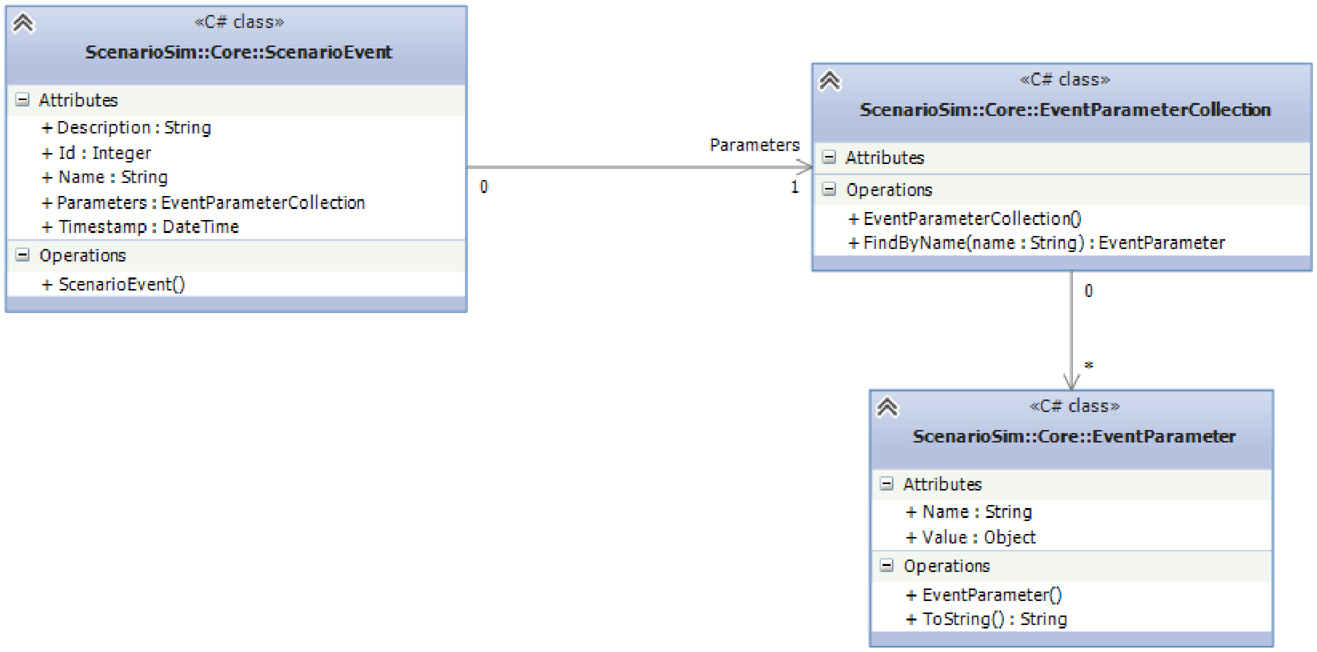


Figure 3.7: The IScenarioSimulator interface.

- Compilation and writing of result files.

The scenario simulator class was refactored over time based on the single responsibility principle [28] to reduce the number of reasons the class had to change. Now delegates all of the previous functionality to simulator components that implement the *ISimulationComponent* interface. This refactoring was successful by incorporating the strategy design pattern [28, 17]. These components will handle all custom behavior of the simulator such as:

- Collection and saving of scenario events submitted to the simulator.
- Tracking the time spent in each task.
- Logging of scenario events to a log file.

The scenario simulator contains a reference to a simulator component repository that is responsible for managing the storage of components. Now with all of the functionality delegated to simulator component concrete implementations through a layer of abstraction, the scenario simulator object is oblivious to the implementation details of itself. To include the functionality, the simulator simply has to forward a submitted event to all of its components as shown below.

```
public virtual void SubmitSimulatorEvent(ScenarioEvent e)
{
if (!IsActive)
throw new Exception("Simulator has not been started. Please call Start() before submitting
    events.");
stateChart.Dispatch(TransformSimulatorEvent(e));

foreach (ISimulationComponent c in componentRepository.GetAllComponents())
c.SubmitEvent(e);

if (!IsActive)
Complete();
}
```

3.6 Simulation Components

The simulation components implement concrete behavior for the scenario simulator class, without the simulator class knowing about the implementation details. As discussed in section 3.5, the simulator components follow the strategy pattern and they are required to implement the *ISimulatorComponent* interface. The interface has three methods that can be used by the component to perform certain behaviors at different times of the scenario simulation. The three are as follows:

- The start method will be called when the scenario simulator recieves its start call from the client application.
- The submit event method will be called when the scenario simulator recieves a submit event call from the client application.
- The complete method will be called when the scenario simulation has completed and the end goal of the scenario has been achieved.

These three methods allow the component to perform initialization during the start method, before the simulation has begun behavior during the simulation whenever an event is received and to perform any clean up or process intensive functionality at the end when the simulation has completed. The *ISimulatorComponent* interface is shown in figure 3.8.

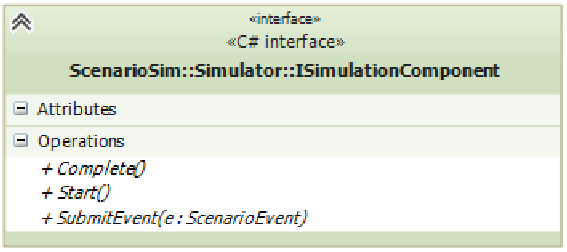

«interface»
«C# interface»
ScenarioSim::Simulator::ISimulationComponent
Attributes
Operations
+ Complete()
+ Start()
+ SubmitEvent(e : ScenarioEvent)

Figure 3.8: The ISimulationComponent interface.

3.6.1 Time Keeper Component

The time keeper component was designed to keep track of the time spent in each task during the simulation of the scenario. The behavior of this component was originally located in a time keeper class that was directly referenced to by the *ScenarioSimulator* class, but after the simulation component pattern was adopted, the functionality was moved into a concrete simulation component class. The following is the submit event method implementation of the time keeper component class:

```
public void SubmitEvent(ScenarioEvent e)
{
long currentTime = e.Timestamp.Ticks;
long deltaTime = currentTime - previousTime;

foreach(string task in previousTasks)
times[task] += deltaTime;

previousTasks = simulator.ActiveTasks();
previousTime = currentTime;
}
```

The method takes the time stamp of the event as the current time and calculates the time since the previous event was received; this produces the delta time value. The delta time value is appended to all tasks that were active in the previous time, since there is no way that they ended between events. (Tasks can only end on event submission) After the times are updated on the previously active tasks, the previous time and previous tasks are updated to the current to aid in the method call on the next event submission. The start method initializes the component and sets the initial starting time and starting tasks for the first submit event method call to use. The complete method simply updates any active times that were active just before the simulation completed.

3.6.2 Parameter Tracking Component

The parameter tracking component is designed to keep track of parameters that are sent with scenario events over the time of the scenario simulation. This allows data to be recorded in relation to time over the course of the simulation, for example the orientation of a surgical tool over the time of a surgical simulation client or the position of the user's cursor in an interactive web-based learning tool client. The parameter tracking component is built three main concepts:

- The tracking registry and tracking parameter registration. The component needs to be aware of what parameters to track and this information is contained in a data structure. When a desired parameter is to be tracked, a tracked event parameter object will be added or registered to the structure so the component knows to track the desired parameter.
- The tracking process. First, the component accepts the incoming event and iterates through each parameter. If the event and parameter pair is in the registry structure, then it is added to the parameter holding structure with the time stamp of the event.
- The result output process. The result output process simply serializes a list of the tracked event parameter objects which contain the parameter and the timestamp of when the parameter was received. This serialized list can then be placed into a result structure that will be loaded at another time.

The following code shows the submit event method, which is responsible for checking if there are any tracked parameters in the submitted event and will save them with the correct time if there are any that are found.

```
public void SubmitEvent(ScenarioEvent e)
{
foreach (EventParameter p in e.Parameters)
if (IsTracked(p, e.Id))
trackedEventParameters.Add(new TrackedEventParameter()
{
Parameter = p,
Timestamp = e.Timestamp
});
}

private bool IsTracked(EventParameter parameter, int eventId)
{
return (from EventParameterPair p in trackedParametersRegistry
where p.ParameterName == parameter.Name && p.EventId == eventId
select p).Any();
}
```

3.6.3 Logging Components

The logging components are a group of components used to log the scenario events sent by the client application to the scenario simulator module. Currently there are two logging components that write to text log files, one is designed to write in basic human-readable text, *TextLoggingComponent* and the other is designed to write in a comma-separated format, *CsvLoggingComponent*.

Originally, these two classes were separate and contained redundant code, since both used the *StreamWriter* class provided in the .NET framework, which is used to write text to a stream. Folliowing the DRY principle, the classes' similar functionality was extracted to an abstract base class, *LoggingComponent* and the resulting structure, shown in figure 3.9, follows the template design pattern. The *LoggingComponent* class will handle all of the writing logic and contains a generate log entry method, which derived logging classes will implement with their own custom log entry strings; this structure allows for client application developers to easily implement their own custom formatting for log files by simply implementing one method that returns the string.

```
public void SubmitEvent(ScenarioEvent e)
{
writer.WriteLine(GenerateLogEntry(e));
}

protected abstract string GenerateLogEntry(ScenarioEvent e);
```

To implement custom event logging functionality, simply derive a class that inherits the *LoggingComponent* class and implements the generate log entry method. Below are two examples, the first is a basic human readable text entry and the second is a comma-separated text entry for csv files.

```
protected override string GenerateLogEntry(ScenarioEvent e)
{
return string.Format("{0} recieved at {1}.", e.Name, e.Timestamp);
}

protected override string GenerateLogEntry(ScenarioEvent e)
{
return string.Format("{0},{1},{2}",
e.Timestamp, e.Id, e.Name);
}
```

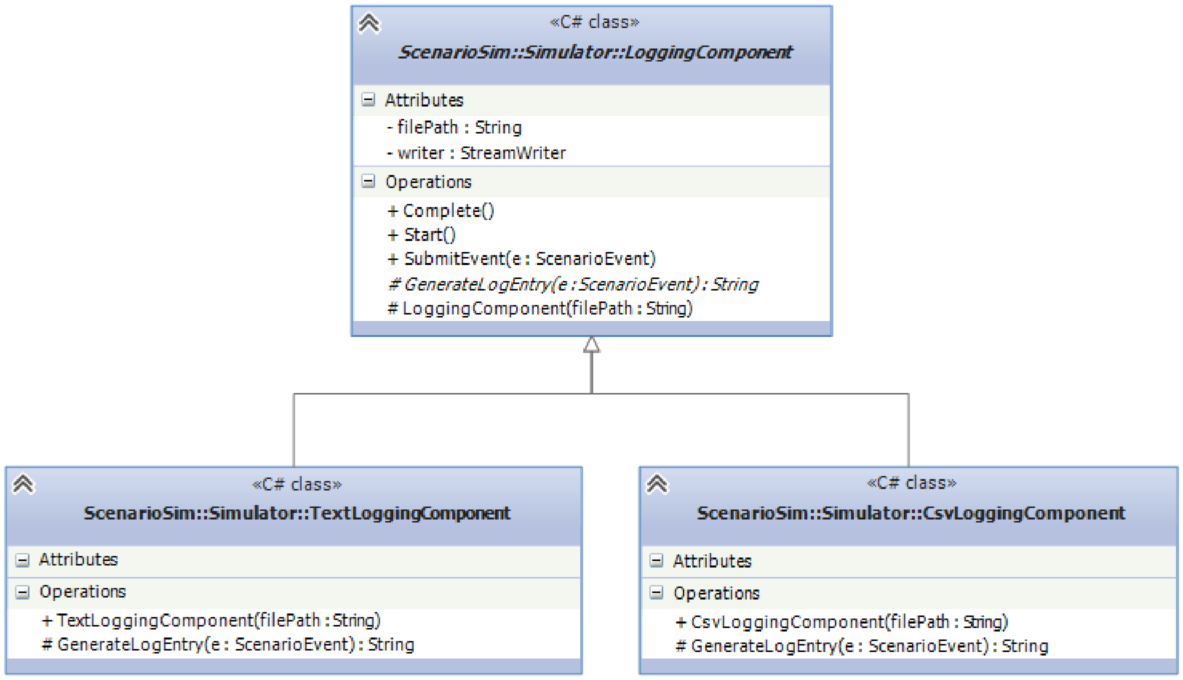

«C# class»
ScenarioSim::Simulator::LoggingComponent
Attributes
- filePath : String
- writer : StreamWriter
Operations
+ Complete()
+ Start()
+ SubmitEvent(e : ScenarioEvent)
# GenerateLogEntry(e : ScenarioEvent) : String
# LoggingComponent(filePath : String)
«C# class»
ScenarioSim::Simulator::TextLoggingComponent
Attributes
Operations
+ TextLoggingComponent(filePath : String)
# GenerateLogEntry(e : ScenarioEvent) : String
«C# class»
ScenarioSim::Simulator::CsvLoggingComponent
Attributes
Operations
+ CsvLoggingComponent(filePath : String)
# GenerateLogEntry(e : ScenarioEvent) : String

Figure 3.9: The Logging component structure

3.6.4 Scenario Event Collection Component

The scenario event collection component is responsible for storing all events that are submitted to the scenario simulator module and serializing them for later use. The purpose of this component is collect the events, so they are serialized at the end of the simulation and may be used in other modules and applications; the use of the events is described in greater detail in chapter 4. As the events are submitted to the scenario simulator and thus submitted to the scenario event collection component, the events are saved in list structure. In the completion of the simulator and thus the component, the scenario event list is serialized to the given file.

The component uses the *IFileSerializer* interface to serialize the list of the scenario events to a file; this interface currently has two implementations; the *XmlFileSerializer* and the *JsonFileSerializer*, which will serialize the given object to or from xml and json file formats respectively. The file serializer structure is shown in figure 3.10. As a result of the serialization abstraction, the component is oblivious to how the scenario event collection is serialized; this ignorance is maintained because of the dependency injection [28], where the serializer is passed to the component and the component only knows about the abstraction. The complete method of the component is shown below to illustrate the serialization ignorance:

```
public void Complete()
{
serializer.Serialize(filePath, eventCollection);
}
```

Any serializer implementation that implements the *IFileSerializer* interface can easily be passed to the component, allowing once again, for customization to the framework behaviour.

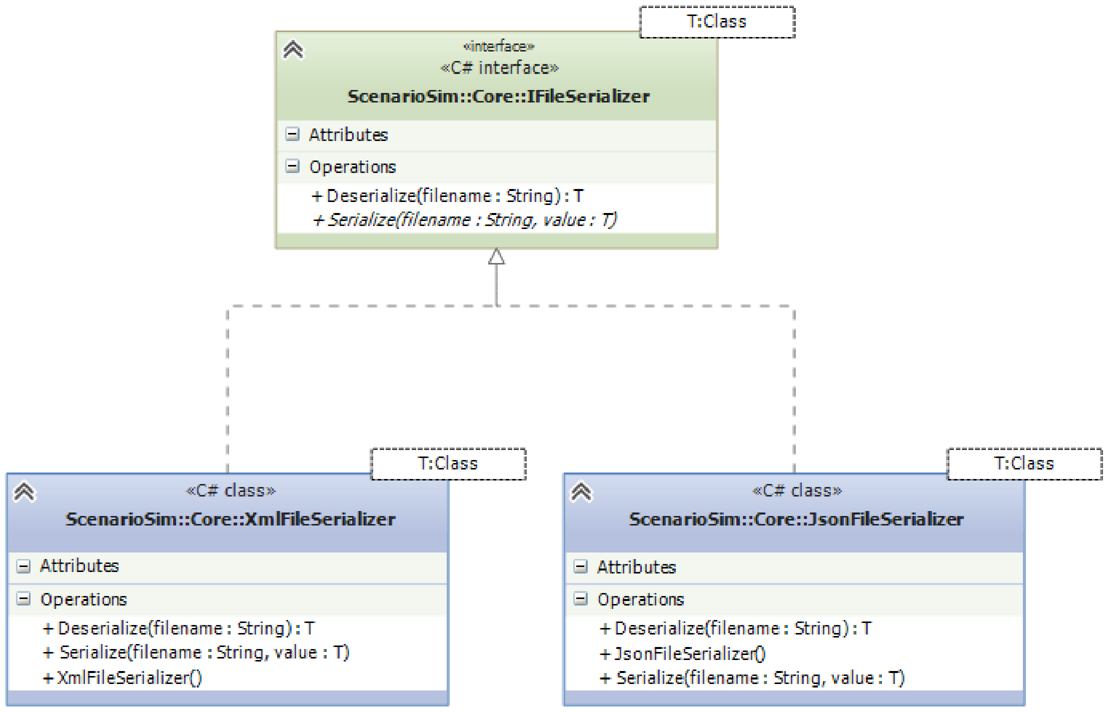

T:Class
«interface»
«C# interface»
ScenarioSim::Core::IFileSerializer
Attributes
Operations
+ Deserialize(filename : String) : T
+ Serialize(filename : String, value : T)
T:Class
«C# class»
ScenarioSim::Core::XmlFileSerializer
Attributes
Operations
+ Deserialize(filename : String) : T
+ Serialize(filename : String, value : T)
+ XmlFileSerializer()
T:Class
«C# class»
ScenarioSim::Core::JsonFileSerializer
Attributes
Operations
+ Deserialize(filename : String) : T
+ JsonFileSerializer()
+ Serialize(filename : String, value : T)

Figure 3.10: The file serializer structure

3.7 The Entity Placement

The entity placer is abstraction that is responsible for initializing the scene of the scenario. Since the framework supports any type client and does not care about the presentation to the user, the initialization of the scene must occur behind an abstraction. The *IEntityPlacer* is an interface that contains one method, *Place(Entity)*; the method is designed such that concrete implementations of the entity placer shall take the given entity and thus its transform and create the correct entity in the client application with the given data from the transform. Refer to figure 3.11 for a class diagram of the entity placer interface. The entity placer abstraction is contained within the *ScenarioSimulator* class and the place method is called for every entity that is contained in the scenario:

```
foreach (Entity entity in scenario.Entities)
placer.Place(entity);
```

3.8 The Enactor Pattern

To achieve all of the complication functionality in the simulator module, there needs to be complication behaviour specific to the client application that is triggered from the simulation of the scenario. Not only do we need specific behaviour, but we need it to be specific to the

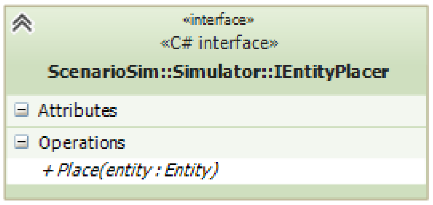


Figure 3.11: The entity placer interface.

type of complication. An important constraint is that we know nothing of the implementation of the complication's presentation in the client application.

Problem Need to present client specific behaviour for specific framework objects without knowing anything about the implementation of the behaviour.

Discussion The enactor pattern builds upon two patterns; the command pattern to abstract the implementation of an action from the framework and the presenter from the model-view-presenter pattern. The enactor is built around an abstraction that holds an Enact method. The enactor abstraction will exist in the framework, but will be implemented in the client application, since the client application is the only piece of the system that is aware of its behaviour. The enactor abstraction will also contain an attribute defining the specific object that it represents. The enact method will take in the specific details of the object that it is acting behaviour for.

Structure The structure of the enactor pattern contains the following elements:

- The enactor abstraction.
- The object type being enacted.
- The invoker of the enactor.
- The concrete enactor.
- The client application that creates the concrete enactor.

Example The following is a simple example code in C-Sharp that will have two types of enactors for the enacted object. The enacted object will be a new day event object, which will be triggered on each new day. Refer to figure fig:EnactorPattern for class diagram illustrating the structure.

Firstly, there is the object being enacted, in this case it is an event that will be sent when a new day begins. The event will contain a field that indicates what day it is.

```
class NewDayEvent
{
public DayOfWeek Day { get; set; }
}
```

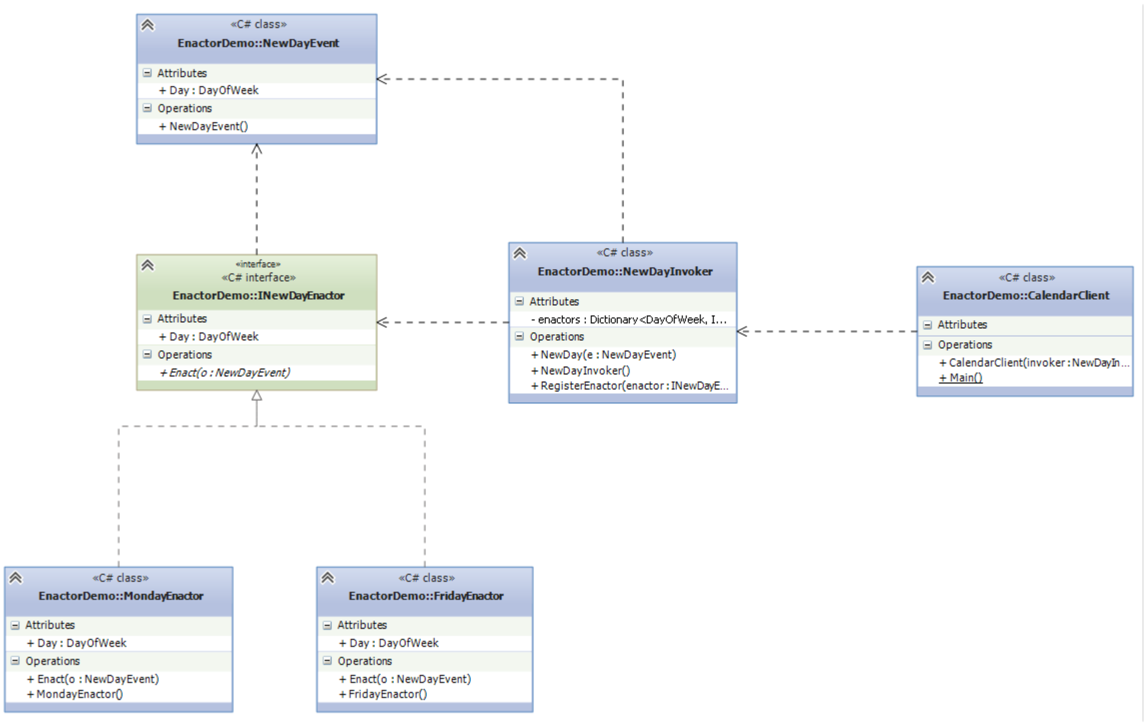


Figure 3.12: The enactor pattern.

Next is the enactor abstraction itself, it will contain the enact method, which will be implemented by implemented by concrete enactors and a property to indicate which event (based on the day) that the enactor will enact for.

```
interface INewDayEnactor
{
void Enact(NewDayEvent o);
DayOfWeek Day { get; }
}
```

Next there are the concrete enactors that will implement the enact behaviour and the property of the enactor abstraction.

```
class FridayEnactor : INewDayEnactor
{
public void Enact(NewDayEvent o)
{
Console.WriteLine("TGIF!");
}

public DayOfWeek Day
{
get { return DayOfWeek.Friday; }
}
}

class MondayEnactor : INewDayEnactor
{
public void Enact(NewDayEvent o)
{
Console.WriteLine("Oh no it's Monday!");
}
public DayOfWeek Day
{
get { return DayOfWeek.Monday; }
}
}
```

Next there is the invoker of the enactor abstraction. This object will hold the enactors and when it receives the new day event, it will call the enact method on the correct enactor. It is completely oblivious to the implementation details of the concrete enactors themselves.

```
class NewDayInvoker
{
Dictionary<DayOfWeek, INewDayEnactor> enactors;

public NewDayInvoker()
{
enactors = new Dictionary<DayOfWeek, INewDayEnactor>();
}

public void RegisterEnactor(INewDayEnactor enactor)
{
enactors.Add(enactor.Day, enactor);
}

public void NewDay(NewDayEvent e)
{
if(enactors.ContainsKey(e.Day))
enactors[e.Day].Enact(e);
}
}
```

Lastly, there is the client application, which will register its custom enactors into the separate package and is oblivious to how they are called.

```
class CalendarClient
{
public CalendarClient(NewDayInvoker invoker)
{
invoker.RegisterEnactor(new MondayEnactor());
invoker.RegisterEnactor(new FridayEnactor());
}
}
```

Remarks

- The enactor is similar to the command, observer and chain of responsibility in that it decouples the sender from the receiver.
- The enactor decouples a behavior of an object from the implementation of the object.
- Chain of responsibility and enactor can be used to link multiple enactors together.
- Similar to the command, undo can be implemented with the enactor pattern, by adding a second method to the Enactor interface to perform the undo.
- Using a dictionary with an identifier and enactor key-value pair can be used to store enactors to be used for later.

3.9 Complications

Recall that the setup of complications in the state chart was discussed in section 3.2 and that there were state chart actions, *EnactComplicationAction*, linked to either the entry or exit actions of a state, depending on the definition of the complication. The actual implementation of the complications will be discussed in detail in this section. The complication implementation is based off of the enactor pattern that was proposed and discussed in great detail in section 3.8, where the complication has an enactor abstraction, *IComplicationEnactor*, the *EnactComplicationAction* is the enactor invoker, the enacted object is the *Complication* class and the concrete enactors are completely unknown within the scenario simulator module. The *EnactComplicationAction* implements the execute action method of the *UmlStateChartAction* abstract class; this method implementation is as follows:

```
protected override void ExecuteAction(StateDataContainer container)
{
if (!enactorRepository.Contains(complicationId))
return;
IComplicationEnactor enactor = enactorRepository.GetEnactor(complicationId);
enactor.Enact();
}
```

Where the identification number of the complication and the enactor repository is passed through the constructor of the action. Refer to figure 3.13 for an illustration of the structure of the complication enactor component.

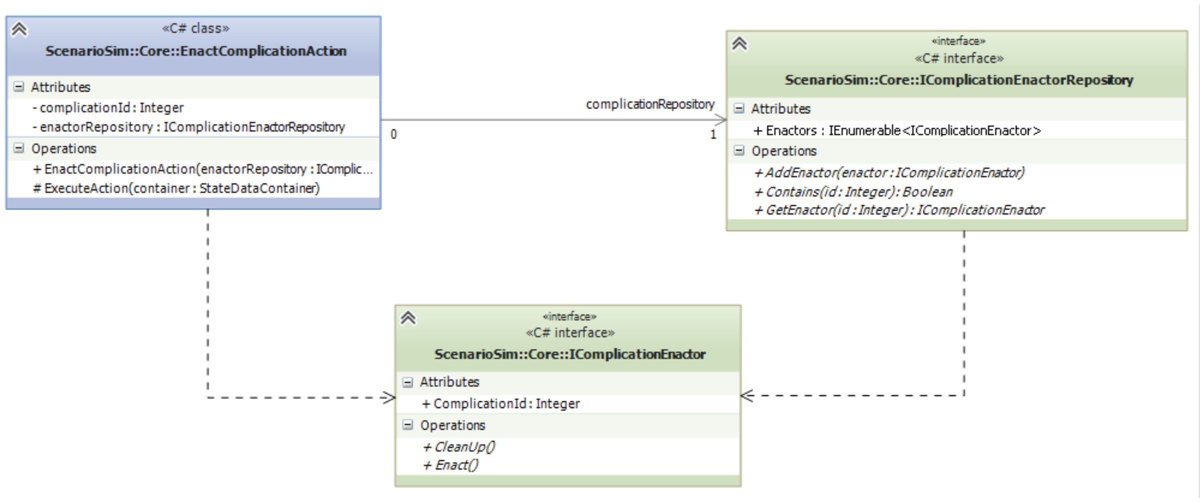

«C# class»
ScenarioSim::Core::EnactComplicationAction
Attributes
- complicationId : Integer
- enactorRepository : IComplicationEnactorRepository
Operations
+ EnactComplicationAction(enactorRepository : IComplic...
# ExecuteAction(container : StateDataContainer)
complicationRepository
0
1
«interface»
«C# interface»
ScenarioSim::Core::IComplicationEnactorRepository
Attributes
+ Enactors : IEnumerable<IComplicationEnactor>
Operations
+ AddEnactor(enactor : IComplicationEnactor)
+ Contains(id : Integer) : Boolean
+ GetEnactor(id : Integer) : IComplicationEnactor
«interface»
«C# interface»
ScenarioSim::Core::IComplicationEnactor
Attributes
+ ComplicationId : Integer
Operations
+ CleanUp()
+ Enact()

Figure 3.13: The complication enactor component structure.

3.10 Module Performance

Table 3.1 contains the performance (speed) of the submit event method for each data collection component. There was a sample size of 100,000 tests performed for each component and the mean speed, variance and standard deviation were reported. The tests were performed on a

Component	Mean Speed (μs)	Variance	Standard Deviation
Simulator	0.0489	0.0339	0.184
Parameter Tracking	0.0724	0.414	0.643
CSV Logging	4.70	77.7	8.82
Text Logging	3.81	54.2	7.36
Time Keeper	0.0446	0.143	0.378
Event Collection	0.289	0.173	0.416
State Chart	0.755	68.0	8.24

Table 3.1: Scenario Simulator Performance

Class	Cyclometric Complexity	Depth of Inheritance	Class Coupling	Lines of Code
ScenarioSimulator	28	1	18	39
TimeKeeperComponent	9	1	12	18
ParameterTrackingComponent	11	1	13	13
LoggingComponent	5	1	5	5
TextLoggingComponent	2	2	3	3
CsvLoggingComponent	2	2	3	3
ScenarioEventCollectionComponent	5	1	6	7

Table 3.2: Scenario Simulator Code Metrics

late 2011 Macbook Pro with 64 bit Windows 7 OS installed, 4 GB RAM and an Intel Core i5 2.4 GHz Dual Core processor. It is observed that the components that logged to files on the disk are the slowest. (4.70 μs for CSV logging and 3.81 μs for plain test logging) The non-I/O data collection components required much less CPU time, where the highest is the state chart component. (0.755 μs) For comparison sake, for a 60 frames per second system, one frame or loop can take up to approximately 16,667 μs to complete, thus even the I/O data collection components as-is, 4.70 and 3.81 μs, take up only 0.0282 % and 0.0228 % of the simulation loop respectively.

3.11 Code Analysis

The following table reports the code metrics of the scenario simulator module described in this chapter. For each class in the module, the cyclometric complexity, the depth of inheritance, the class coupling and lines of code metrics were analyzed and reported in the table 3.2 A key goal in the implementation of the framework was to pay attention to one main metric, the lines of code. It is observed that the breakup of the functionality into the several different data collection components allowed the lines of code of each functionality to be reduced, where the average lines of code for the components is 8.17. A smaller lines of code reduces the amount of bugs and defects that can possibly be present in a component and the metric is carried with great weight when determining the quality of the code.

Chapter 4

Performance Evaluation: The Playback Module and Evaluation Components

A vital aspect to evaluating user performance in a training regime or experimental scenario is the evaluation process. There are a few different stakeholders in the evaluation process, one could be the original user that performed the scenario and another could be an evaluator of the performance; this could be a teacher of a course or curriculum. Both of these roles have different purposes for evaluating the performance, the former would be for learning purposes, so that the user can perform better next time. The latter could be evaluating the user, who could be a student, for their grade in a course. Another type of evaluator could be an individual that is evaluating user performance for a research study.

A trainee who performed the scenario would look to improve their performance for the next time they perform the scenario. They would review how they performed for each task and see which tasks they need to practice or focus more on. An external evaluator of the performance would look to evaluate the performance for testing or accreditation purposes. They need to not only look at the empirical results, but also the abstract results of the performance.

The playback module offers the ability to users and evaluators alike to view a playback of a previous performance by a user. Typically when reviewing performance, an evaluator will review the data of the performance, this could be some sort of scoring mechanism or concrete data value. Another reviewing mechanism is the reviewing of a video of the performance. This offers the evaluator the visual perspective of the performance by the user. The visual perspective is very helpful to evaluating performance, because it gives information on the performance that simple data would not give. Even though the data acquisition module gives a very detailed breakdown of the data from the scenario performance, a visual perspective on the performance can give evaluators and user many helpful benefits.

4.1 Objective

The objective of the playback module is to provide an application program interface to allow client applications to provide the ability to playback previous scenario performances to evaluators of the performance. The module will offer the following features:

- A visual playback of the scenario performance.

- Data pertaining to the performance available during the visual playback:
- Task(s) currently being performed.
- Time spent in the current task.
- Current value of all metrics of current tasks being performed.
- Ideal values of all metrics of current tasks being performed.

4.2 Module Design

The module is designed off of an event-driven approach, which takes the events from the performance and replays them to the evaluator. The event-driven replay is based off of the same idea that revision control systems use to track changes. A revision control system starts with a file or a set of files and then the changes to those files are tracked, which allows a user to select any version of the file and then system can construct it by taking the original file and sequentially applying all of the changes that pertain to that file, until it is to the requested version. The same approach is used in the design of the scenario performance replay. First we start with an initial scenario state and scenario context; exactly the same to that of when the scenario is performed by the user. As the original scenario is performed by the user, they input a various number of commands to the client application and in turn, those commands are sent as events to the framework. These events are similar to the revision changes in a revision control system. Every event has an effect on the state of the scenario performance.

The simulation and data acquisition module collects the sequence of events from the user and this provides the set of state changes that are needed to recreate the scenario performance. To simulate the performance in automated matter, we can submit these events to the scenario simulator in a similar manner that the user submitted the events to the simulator. Recall, the simulator interface accepts a scenario event and handles all of the logic behind the scene. The playback module will provide a way to automate the event submission process by submitting them to the simulator itself.

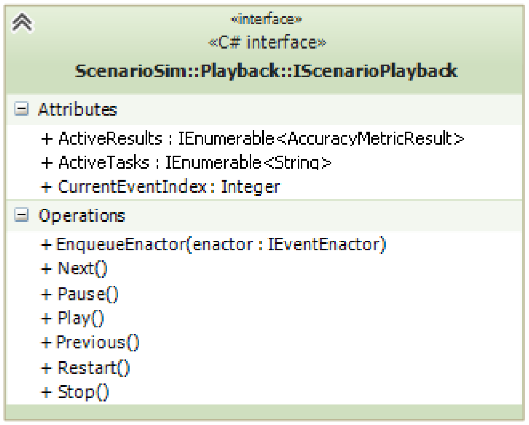
4.3 Scenario Playback Interface

The playback module, like the simulator module provides a simple and intuitive interface to client applications that allow the playback of a performance. The interface provides the following functionality:

- Play the real time playback of the performance.
- Pause the real time playback of the performance.
- Stop the real time playback of the performance.
- Restart the real time playback of the performance from the beginning.

- Register an enactor for the playback.
- Provides a list of the tasks that the user was actively performing at that moment.
- Provides a list of the current metrics that pertain to the current active tasks.
- Step to the next event that was submitted.
- Jump back to the previous event that was submitted.

Please refer to figure 4.1 for the class diagram of the *IScenarioPlayback* interface.



```
«interface»
«C# interface»
ScenarioSim::Playback::IScenarioPlayback
Attributes
  + ActiveResults : IEnumerable<AccuracyMetricResult>
  + ActiveTasks : IEnumerable<String>
  + CurrentEventIndex : Integer
Operations
  + EnqueueEnactor(enactor : IEventEnactor)
  + Next()
  + Pause()
  + Play()
  + Previous()
  + Restart()
  + Stop()
```

Figure 4.1: The IScenarioPlayback interface

4.4 Scenario Playback

The playback implementation uses the method mention above: the event driven playback. Once we have the events, we can simply submit them as if it was the user submitting them to the client. During the scenario performance, the client application will create a new simulator, start it and then proceed to submit events to the simulator based on user input to the client application; figure 4.2 illustrates this activity.

During the playback though, the playback component is responsible for submitting the event, thus eliminating the userâĂŹs involvement in the scenario performance. During the playback, the client application creates a new playback object and the playback component is responsible for starting the simulator and submitting the events to simulator. The playback component has an internal timer, which when the interval elapses, will submit the appropriate events to the simulator and enact those events on the client; figure 4.3 illustrates this activity.

As mentioned earlier, the real time playback is based off of the scenario events of a previous performance. A performance has a set of events that a user performed and each of one of the events has a timestamp. The timestamps are a type of *DateTime*, which keeps track of both the date and the time. To be able to playback the events, the timestamps must be converted to a time span that is referenced to the beginning of the scenario performance; this is done

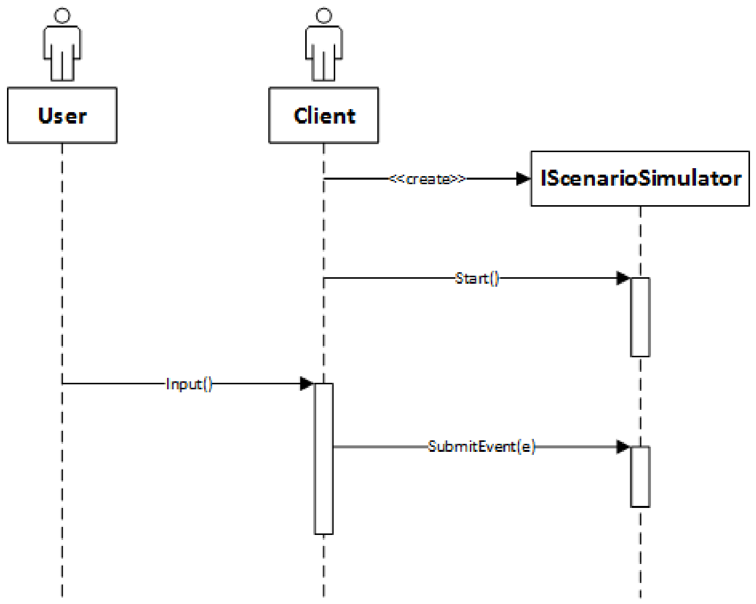


Figure 4.2: The scenario simulator sequence diagram.

by subtracting the timestamp of the very first event off of all of the events. The result is that the first event will have a time of zero, since it marks the beginning of the performance and all subsequent events will be the time spans since the beginning of the performance. The following code demonstrates this initialization:

```
private void ShiftEventTimes(ScenarioEventCollection collection)
        {
            events = new List<KeyValuePair<long, ScenarioEvent>>();
            long startTime = collection[0].Timestamp.Ticks;

            foreach (ScenarioEvent e in collection)
                events.Add(new KeyValuePair<long, ScenarioEvent>(e.Timestamp.Ticks -
                    startTime, e));
        }
```

The next step is to play through the events as if they are happening again. To achieve this, the start time of the playback is recorded and a timer is set up to elapse every 60th of a second. During each timer elapse event, the delta time is calculated by taking the current time and subtracting the start time of the playback. The delta time value is compared against the delta times that were calculated in the initialization phase. If the next event's delta time is less than the playback delta time, the event is enacted. The main loop of the real time playback is shown in the following code:

```
private void timer_Elapsed(object source, ElapsedEventArgs e)
        {
            long deltaTime = e.SignalTime.Ticks - startTime.Ticks;

            lock (events)
            {
                while (nextEventIndex < collection.Count &&
                    events[nextEventIndex].Key < deltaTime)
                {
                    ScenarioEvent se = events[nextEventIndex].Value;
                    simulator.SubmitSimulatorEvent(se);
```

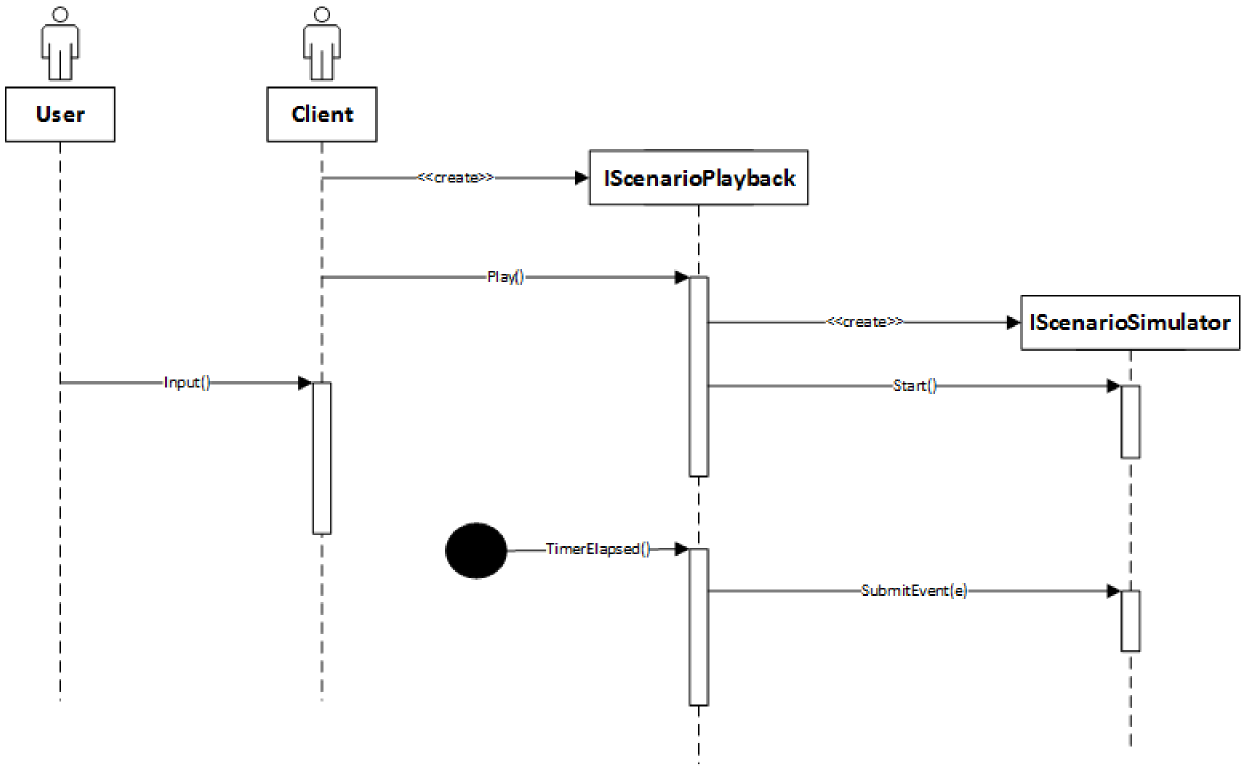


Figure 4.3: The scenario playback sequence diagram.

```
                    if (enactors.ContainsKey(se.Id))
                        enactors[se.Id].Enact(se);
                    nextEventIndex++;
                }
            }
        }
```

The result is a smooth playback that will enact the events in the same sequence and time frame that they were originally submitted.

4.5 Event Enactor

Recall from section 3.8 the enactor pattern was described in great detail. In order to achieve the ability to replay events that a user performed in the original scenario performance, the play back module must be able to replicate the client application-specific behaviour. Complications also had client application-specific behaviour and relied on the enactor pattern to achieve this functionality, so it is natural to re-use the same pattern again to achieve the ability to play back events in a client application. The enactor pattern was applied to events and the event enactor abstraction is shown in figure **??**.

The overall structure of the playback module is illustrated in figure 4.5

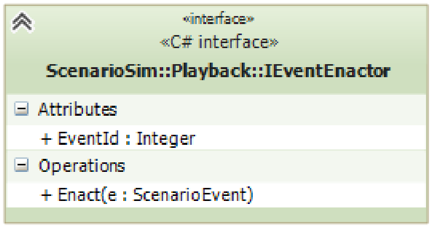


Figure 4.4: The event enactor abstraction.

Class	Cyclometric Complexity	Depth of Inheritance	Class Coupling	Lines of Code
ScenarioPlayback	24	1	23	59

Table 4.1: Scenario Playback Code Metrics

4.6 Code Analysis

The following table reports the code metrics of the scenario playback module described in this chapter. For each class in the module, the cyclometric complexity, the depth of inheritance, the class coupling and lines of code metrics were analyzed and reported in the table 4.1 It is observed that there is only one class in the playback module and it is 59 lines of code. As mentioned in the code analysis of the simulator module, it is desired to breakup classes into smaller classes to reduce the lines of code of all classes. It is recommended and a future task to break up the scenario playback class into smaller pieces, as there are several responsibilities of this class that can be broken up into smaller classes.

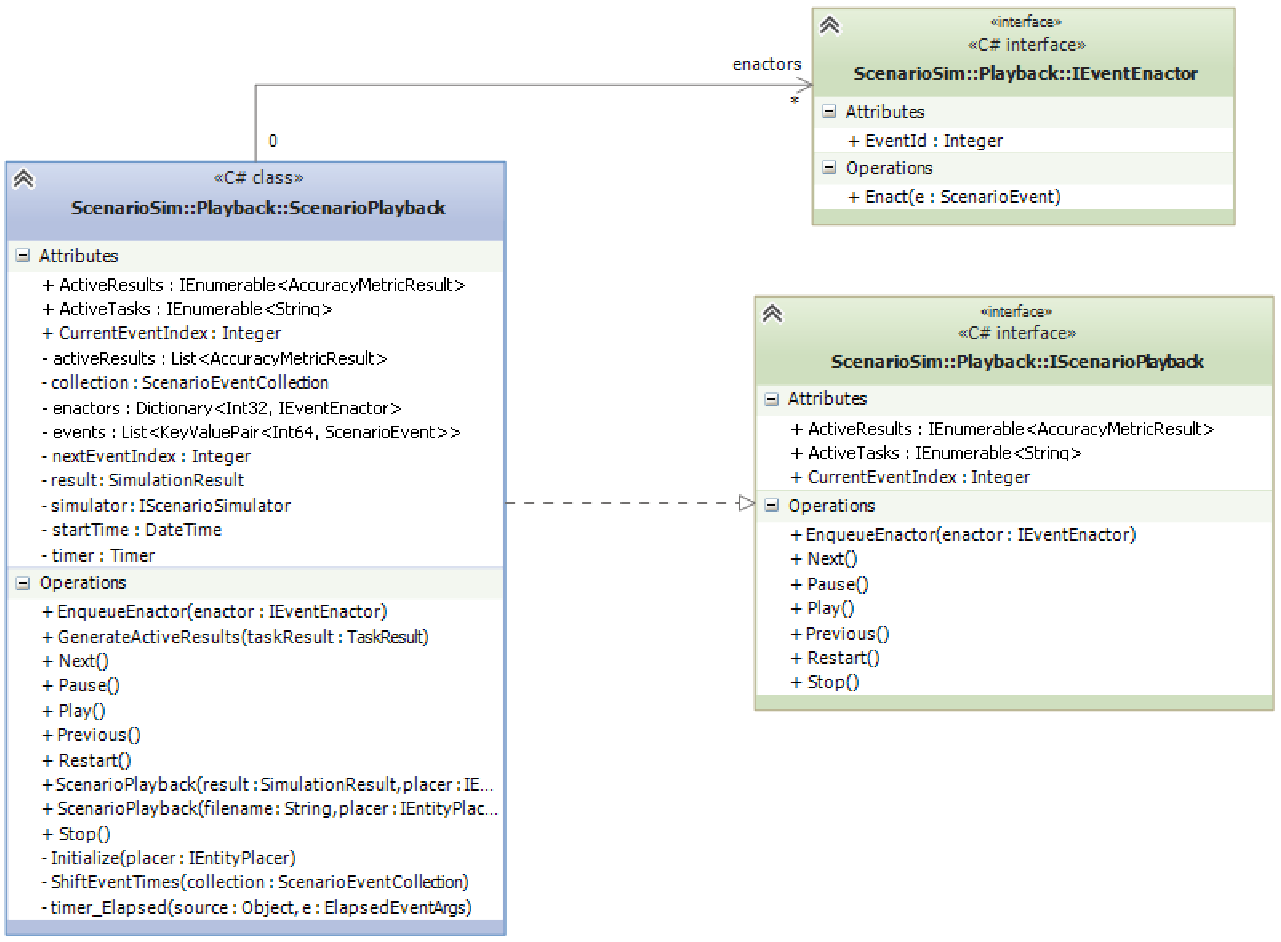

enactors
*
0
«C# class»
ScenarioSim::Playback::ScenarioPlayback
Attributes
+ ActiveResults : IEnumerable<AccuracyMetricResult>
+ ActiveTasks : IEnumerable<String>
+ CurrentEventIndex : Integer
- activeResults : List<AccuracyMetricResult>
- collection : ScenarioEventCollection
- enactors : Dictionary<Int32, IEventEnactor>
- events : List<KeyValuePair<Int64, ScenarioEvent>>
- nextEventIndex : Integer
- result : SimulationResult
- simulator : IScenarioSimulator
- startTime : DateTime
- timer : Timer
Operations
+ EnqueueEnactor(enactor : IEventEnactor)
+ GenerateActiveResults(taskResult : TaskResult)
+ Next()
+ Pause()
+ Play()
+ Previous()
+ Restart()
+ ScenarioPlayback(result : SimulationResult, placer : IE...
+ ScenarioPlayback(filename : String, placer : IEntityPlac...
+ Stop()
- Initialize(placer : IEntityPlacer)
- ShiftEventTimes(collection : ScenarioEventCollection)
- timer_Elapsed(source : Object, e : ElapsedEventArgs)
«interface»
«C# interface»
ScenarioSim::Playback::IEventEnactor
Attributes
+ EventId : Integer
Operations
+ Enact(e : ScenarioEvent)
«interface»
«C# interface»
ScenarioSim::Playback::IScenarioPlayback
Attributes
+ ActiveResults : IEnumerable<AccuracyMetricResult>
+ ActiveTasks : IEnumerable<String>
+ CurrentEventIndex : Integer
Operations
+ EnqueueEnactor(enactor : IEventEnactor)
+ Next()
+ Pause()
+ Play()
+ Previous()
+ Restart()
+ Stop()

Figure 4.5: The scenario playback module.

Chapter 5

The Ellipsoid Orientation Matching Task: Performance Evaluation

5.1 Motivation

A brain tumor is a disease in which abnormal cells form in the tissues of the brain; there are two types of brain tumors: malignant, which are cancerous and can spread to other tissues in the body and benign, which are not cancerous.[1] In the United States, there is an estimated 68,470 new cases of primary and non-malignant brain and CNS tumors expected to be diagnosed in 2015. [31] The incidence rate of all primary malignant and non-malignant brain and CNS tumors is 21.42 cases per 100,000. (7.25 per 100,000 for malignant and 14.17 per 100,000 for non-malignant) [31]

One of the available treatments for tumors is the resectioning of or the removal of the tumor itself from the patient. The removal of the tumor requires a surgeon to create an opening in the head of the patient, move the brain tissue in between the skull and the tumor and then remove the tumor. One of the tasks in the tumor resectioning procedure is the orientation task, where the surgeon must orient themselves in the most efficient manner to remove the tumor. This requires the surgeon to align themselves with the major or longest axis of the tumor; this will allow the surgeon to reduce the amount of manipulation of the normal brain tissue around the tumor and to avoid creating any neurological deficits. The shortest distance is usually desired in order to avoid transecting a larger layer of normal brain, but also a trajectory is the direction of the longest axis of the tumor to decrease the amount of retraction necessary on adjacent brain in order to see the whole tumor when debulking it. Alignment with the longest axis of the tumor also provides the minimal amount of movement required of the surgeon's tools.

5.2 Overview

The ellipsoid orientation matching task simulator is a multi-faceted application built using the software framework described in chapters 3 and 4 and the Unity game engine. It involves a targeting task that the trainee is to perform and attempt to achieve the lowest possible error values. The task will require the trainee to orientate a tool to intersect through the longest axis of an ellipsoid that will be situated inside of a translucent head. The purpose to this application was to build an application that utilizes the proposed framework, as this would help develop and refine the framework at the same time. The framework is built to help collect data from scenario performances and review those performances for training and evaluation purposes and this was a perfect environment for it.

5.3 Designing the Scenario

5.3.1 Hierarchical Task Analysis

The first step is to develop a task hierarchy of the ellipsoid matching task. When a trainee is attempting to position a tool such that it will intersect through the longest axis of an ellipsoid, one of the sub-tasks is the positioning of the tool. Depending on the position and orientation of the target ellipsoid and the trainee's view of the scene, performing the task may be very difficult or impossible. It would be of great benefit to allow the trainee to change their view of the scene; this will allow the trainee to obtain better performance. The "change of view" task will be another sub-task of the overarching task and this will now result in two sub-tasks: the change of view task and the position tool task. The current task hierarchy takes into account the trainee's tasks to optimize their view and to position the tool, but what about the selection task? The trainee must let an evaluator or simulator know that they finalize their decision and submit their selection as their final answer so to speak; without this, it is not possible to know when the trainee is finished and has made a decision. Accounting for the selection task, this results in a task hierarchy illustrated in figure 5.1.

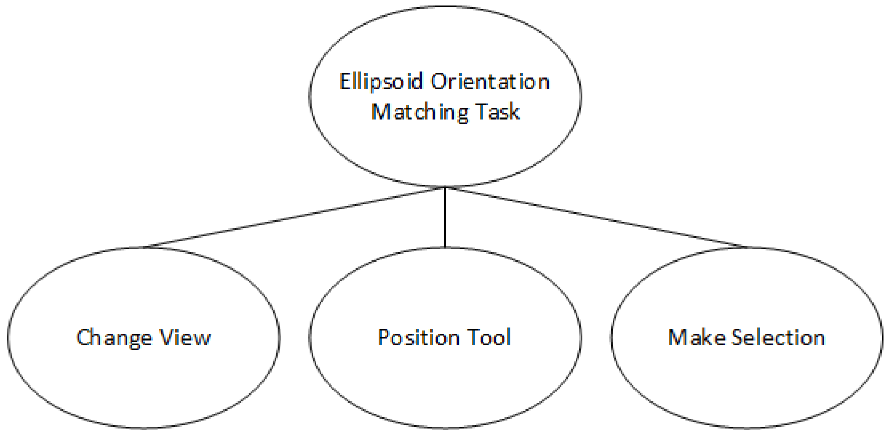


Figure 5.1: A preliminary task hierarchy for the ellipsoid orientation matching task.

Since the tool starts at a position away from the head, the trainee is required to move the tool in all three axes x, y and z to reach the ideal position. The current task hierarchy is a great start, but there is one glaring issue at the moment; the trainee requires three degrees of freedom (x, y and z position), but there are only two degrees of freedom available through a mouse or joystick. (x and y axes)

To solve this problem, a two-input approach where the mouse x and y axes will control the translation of the tool in an orbiting-type function around the head and a third axis, s, will control the translation of the tool inwards to and outwards from the head. The third axis, s will be mapped to the scroll wheel of the mouse for ease-of-use and minimal hand movement on the part of the trainee. To illustrate this approach, it is more intuitive to think of it in terms of the spherical coordinate system; figure 5.2 illustrates a spherical coordinate system representation.

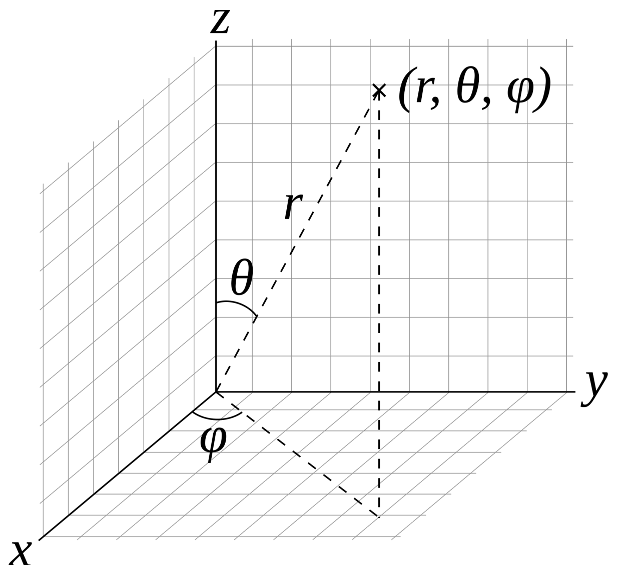


Figure 5.2: A spherical coordinate system representation. [source: Wikipedia Commons]

The movement of the tool in the azimuthal direction (φ) will be a function of the x axis input of the mouse:

$$\varphi = f(x) = Ax \tag{5.1}$$

Where x is the x axis input of the mouse and A is a scaling constant. The same can be applied to the polar angle (θ):

$$\theta = f(y) = By \tag{5.2}$$

Where y is the y axis input of the mouse and B is a scaling constant. Lastly the radius (r) is defined:

$$r = f(s) = Cs \tag{5.3}$$

Where s is the scroll axis of the mouse and C is a scaling constant. There are two separate inputs using this approach: the mouse translation, which affects the x and y axis, and the scrolling wheel, which affects the s axis. The same position solution for the tool can be used for the camera as wel in the change of view task.

Recall that there are two measurements to take into account, the position of the tool and the direction of the tool after the trainee's selection. The performance of the position can be fully influenced by the translation solution that was proposed, but the direction of the tool will always be pointed at the center of the head. What if the longest axis doesn't go through the center of the head? The trainee will never have the opportunity to achieve a perfect direction alignment. The next component of the task is that which involves fine-tuning the trajectory of the tool.

After the trainee has translated the tool such that it intersects with the head, the position of the tool has essentially been selected. The trainee will now have the chance to rotate the tool around the intersection of the tool with the head, to improve the direction of the tool. This two-step process will decompose the position tool task into two sub-tasks: the translate tool (orbiting of tool and moving tool inwards/outwards) and the rotate tool tasks. This concludes the task hierarchy for the task defined in the requirements; the final task hierarchy is shown in figure 5.3

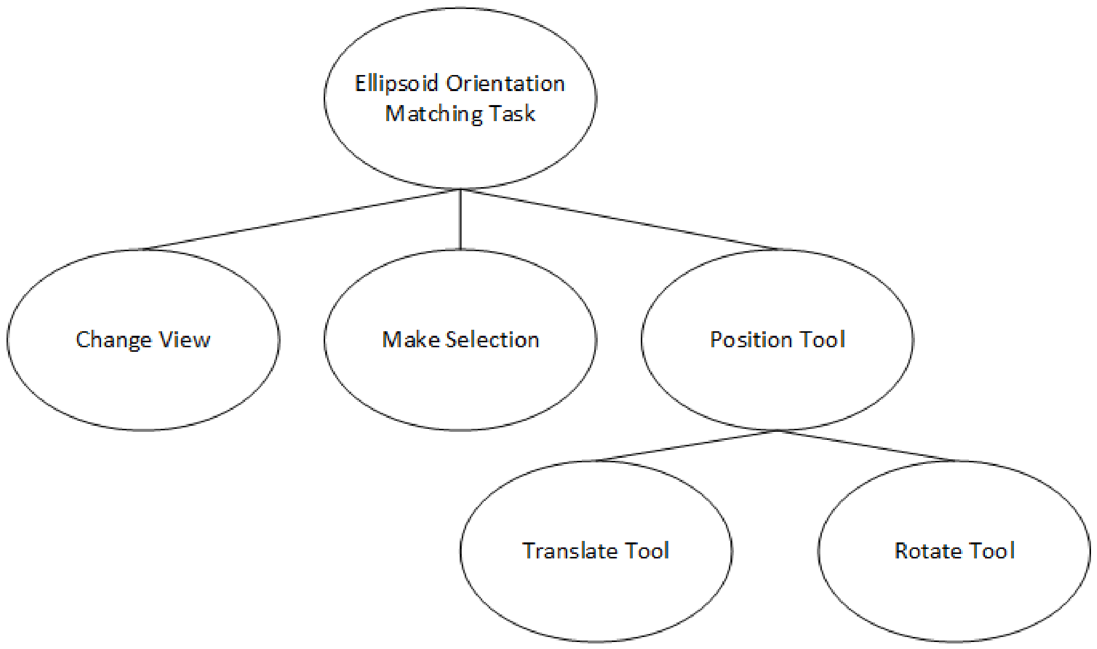


Figure 5.3: The task hierarchy after decomposing the position tool task.

5.3.2 Hierarchical Task Paths

Next, the path of the task will be defined; this is essentially how the tasks interact with each other and how the state of the task flows from one state to another One way to think of how paths work is by representing a task with a flow chart comprised of its sub-tasks. The transitions from one node to another will be a path or transition from one task to another. To begin with the overarching task: the "Ellipsoid Orientation Matching Task"; this task is composed of three sub-tasks: change of view, positioning the tool and selection. The task will begin with positioning the tool; from there the trainee will decide whether the view is optimal, if the view is not optimal the trainee may change the view. The trainee will make their selection when they decide that the tool's position is optimal. This leads to a few paths: firstly the trainee can switch between changing the view and positioning the tool, secondly the trainee can make a selection at any time.

The next complex task in the hierarchy is the positioning tool task. The positioning tool task is composed of two sub-tasks: the translate tool and rotate tool tasks. The trainee will begin performing the translate tool task and after the tool has collided with the head, the trainee will then perform the rotating tool task to fine-tune the tool's orientation.

5.4 Generating a Scenario Set

The placement of the target will be described in the following parameters:

1. The position, which will be represented in *x*,*y*,*z* coordinates.
2. The polar(pitch) and azimuth (yaw) angles.
3. The eccentricity and radius of the ellipsoid.

Unity manages the transformations of objects by utilizing two 3-dimension vectors; one for position, one for scaling and a quaternion for the rotation, which can also be represented by the Euler angles. (3-dimensional vector) Thus, the transformation of an object can be represented by three 3-dimensional vectors. The first step is to devise a transformation from the set of parameters outlined above to the structure that Unity utilizes. The position of the ellipsoid is fairly straightforward, because both Unity and the parameter set use a 3-dimensional vector for position.

$$\vec{p} = \vec{p_0} = [\ x_0\ y_0\ z_0\ ] \tag{5.4}$$

Where p_0 is the position vector from input parameters. The next parameters represent the orientation of the target and those are the pitch and yaw. With the assumption that the local x axis of the target will always be the longest axis, the orientation parameters are straight forward, if we use the Euler angles:

$$\vec{r} = [\ 0\ yaw\ pitch\ ] \tag{5.5}$$

Where $\vec{r}$ is the vector of Euler angles; *yaw* and *pitch* are directly from the input parameters.

Lastly to properly scale the target, the input parameters of eccentricity and radius must be transformed to a 3-dimensional scaling vector. Firstly the longest axis of the ellipsoid is equal to the radius parameter, therefore the x component of the scaling vector will be the radius from the parameters.

$$\vec{s} = [\ r\ b\ b\ ] \tag{5.6}$$

Next, to find the radius of the minor axes of the ellipsoid (y and z), it is required to determine the length from the focus point to the center:

$$c = \varepsilon r \tag{5.7}$$

Where c is the length from the focus point to the center of the ellipsoid, ε is the eccentricity and r is the radius. Next determine the radius of the minor axis:

$$b = \sqrt{r^2 - c^2} \tag{5.8}$$

Where b is the radius of the minor axis, r is the radius and c is the length from the focus point to the center of the ellipsoid. With that final 3-dimensional scaling vector transformed, the transformations are complete.

To generate a set of scenarios based off of input parameters, a scenario generator class was created to take a template scenario, a set of expected parameter values and a set of parameter variances. The generator will copy the template scenario and then randomly generate a transform for the target inside the head, based on randomly generated parameters. The reading and copying of a scenario template is implemented by the following code:

```
Scenario template = serializer.Deserialize(templateFile);

        Scenario result = new Scenario();
        result.Name = string.Format("{0} {1}", template.Name, current);

        result.Task = template.Task;
        result.Entities = template.Entities;
        result.Complications = template.Complications;
        result.TaskTransitions = template.TaskTransitions;
```

The parameters are generated by taking an expected value and a variance value and generating a random value from within the variance value from the expected value. For example, to generate a random radius:

```
float actualRadius = Random.Range(expectedRadius - radiusVariance,
            expectedRadius + radiusVariance);
```

Using the range method in the *Random* class with all parameters mentioned above will produce a set of random parameter values. The next step is to take these generated parameter values and to assign them to the Unity transform 3-dimensional vectors for position and scale and the quaternion for the rotation:

```
        Vector3 position = new Vector3(actualX, actualY, actualZ);
        Quaternion rotation = GenerateRotation(actualPitch, actualYaw);
        Vector3 scaling = GenerateScale(actualRadius, actualEccentricity);

        Quaternion GenerateRotation(float pitch, float yaw)
        {
                Quaternion result = new Quaternion();
                result.eulerAngles = new Vector3(0, yaw, pitch);
                return result;
        }

        Vector3 GenerateScale(float radius, float eccentricity)
        {
                Vector3 result = new Vector3();

                float c = radius * eccentricity;

                float b = Mathf.Sqrt(radius * radius - c * c);

                result.x = radius;
                result.y = b;
                result.z = b;

                return result;
        }
```

After generating random parameter values for the scenario and assigning them to the target ellipsoid's transformation, there is one and crucial step left to perform and that is to determine the perfect or ideal tool position and direction for the task. Since evaluation of the performance is required, the ideal or perfect values must be found for each randomly generated scenario. To determine the ideal direction of the tool such that it intersects with the longest or major axis of the ellipsoid would simply mean that the direction vector of the tool must be the same as the direction vector that passes through the major axis of the target. To determine the direction vector that runs through the major axis, two game objects were positioned at each end of the major axis of the ellipsoid, refer to figure 5.4 for an illustration. The direction vector could easily found by taking the resulting vector of the negative game object's position subtract the positive game objects position.

```
Vector3 direction = axis.AxisDirection;
Vector3 reverse = -direction;
IdealValue = Vector3ToVector3f(reverse);

public Vector3 AxisDirection
    {
        get { return (PositiveEdge.position - NegativeEdge.position).normalized; }
    }
```

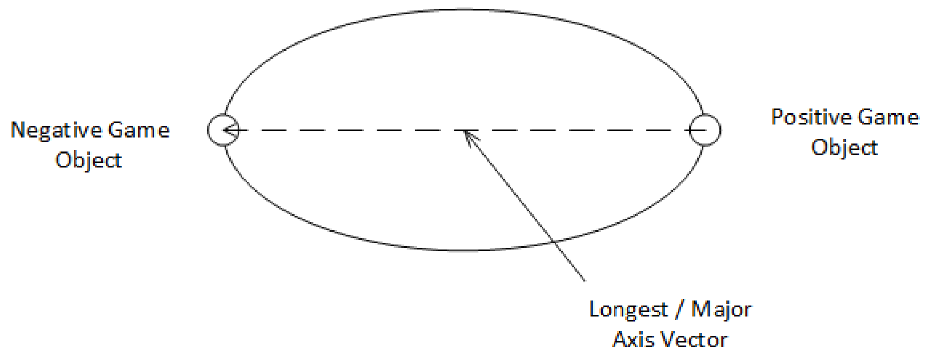


Figure 5.4: The ellipsoid's longest axis.

With the ideal direction vector found, now it is time to determine the ideal position where the tool's tip should touch the head. Finding this ideal position is more difficult than it is to find the ideal direction; as it requires to find the intersection of the ideal direction vector from the target's position with the head. To determine the ideal position the initial approach was to take the ideal direction and the position of the target and to construct a ray and to determine where that ray intersected with the head mesh. Refer to figure 5.5 for an illustration of the approach. Unity provides a *RaycastAll* method which will cast a ray from a given position in the given direction and return an array of *RaycastHit* objects which contain collision information:

```
Vector3 direction = axis.AxisDirection;
RaycastHit[] hits = Physics.RaycastAll(axis.PositiveEdge.position, direction);
```

Unfortunately, this approach did not work, because the built in functionality of the *RaycastAll* method will not return collisions where the ray cast originates from inside a collider. Since the ray was being casted from the target's position inside of the head, it would not return the collision with the head's collider. The next idea was to cast a ray from outside the head back onto the head. This was accomplished by casting a ray from the target along its major axis direction vector and taking a point that was outside the head along that ray. Next, cast a

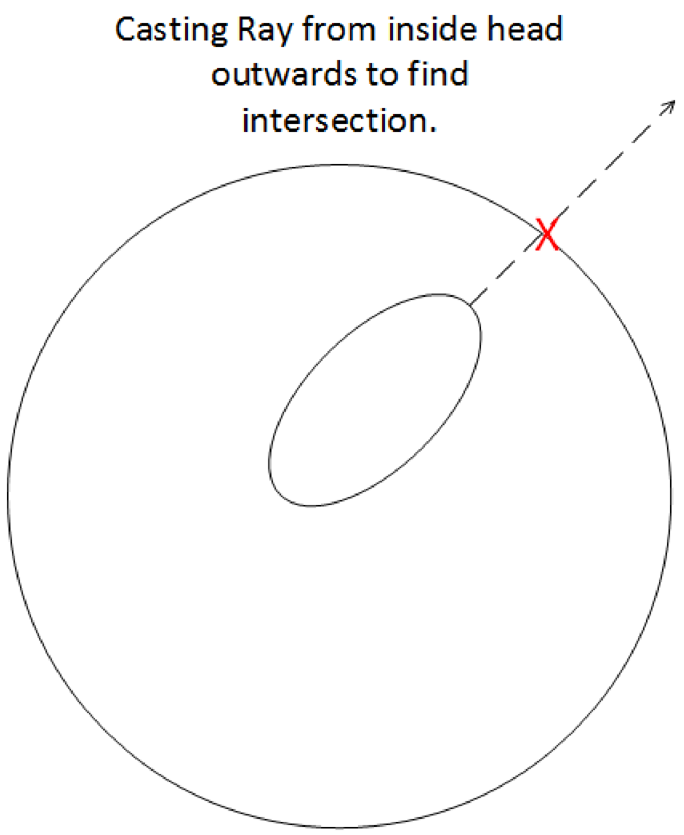


Figure 5.5: The first attempt at finding ideal position.

ray from the new determined position in the negative direction of the major axis, thus targeting the target from outside the skull just as the tool would. Finally, collect the collision hit from the new ray cast on the head; the intersection would be the perfect or ideal position. Refer to figure 5.6 for an illustration of the approach.

```
Ray ray = new Ray(axis.PositiveEdge.position, direction);
        Vector3 newPosition = ray.GetPoint(100);
        Vector3 reverse = -direction;


        RaycastHit[] hits = Physics.RaycastAll(newPosition, reverse);

        Vector3 idealPosition = (from RaycastHit hit in hits
                                 where hit.collider == collider
                                 select hit.point).First<Vector3>();
```

Now that the ideal position and direction for a given scenario can be found, the ideal values are assigned in the *PositionAccuracyMetric* and *DirectionAccuracyMetric* classes respectively:

```
AccuracyMetric directionMetric = new DirectionAccuracyMetric()
        {
            ValueName = "Tool Direction",
            IdealValue = Vector3ToVector3f(reverse),
            ActualValue = new ActualValueLocation(1, "Tool Direction")
        };
        AccuracyMetric positionMetric = new PositionAccuracyMetric()
        {
```

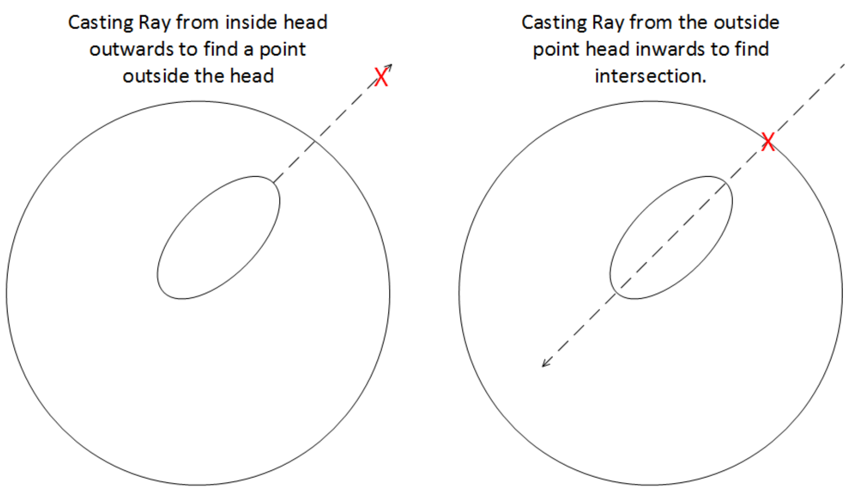

Casting Ray from inside head outwards to find a point outside the head
X
Casting Ray from the outside point head inwards to find intersection.
X

Figure 5.6: The second attempt at finding ideal position.

```
        ValueName = "Tool Position",
        IdealValue = Vector3ToVector3f(idealPosition),
        ActualValue = new ActualValueLocation(1, "Tip Position")
    };

    result.Task.Value.AccuracyMetrics.Add(directionMetric);
    result.Task.Value.AccuracyMetrics.Add(positionMetric);
```

5.4.1 Data Collection

For this set of scenarios, there is a wide range of data that will be collected by the scenario simulator module. Firstly, the time spent on each phase of the task will be measured; fortunately, the *TimeKeeperComponent* discussed in section 3.6.1 can track the time spent on each task during the scenario performance. The parameter values of the target in the task will need to be recorded to compare performance versus the input parameters; these parameter values will be directly taken from the input parameters file that is generated during the scenario generation process. The position of the tip of the tool and the direction of the tool will be tracked over the time of the experiment; this can be accomplished by passing the position and direction as parameters in various events and using the *TrackedParameterComponent* that was discussed in section 3.6.2. To track the parameters, register them with the parameter tracker:

```
(simulator as LoggingScenarioSimulator).AddTrackedParameter(new EventParameterPair()
            {
                EventId = 3,
                ParameterName = "Tip Position"
            });
        (simulator as LoggingScenarioSimulator).AddTrackedParameter(new EventParameterPair()
        {
            EventId = 3,
            ParameterName = "Tool Direction"
        });
        (simulator as LoggingScenarioSimulator).AddTrackedParameter(new EventParameterPair()
        {
            EventId = 9,
            ParameterName = "Tool Direction"
});
(simulator as LoggingScenarioSimulator).AddTrackedParameter(new EventParameterPair()
{
    EventId = 9,
    ParameterName = "Tip Position"
});
(simulator as LoggingScenarioSimulator).AddTrackedParameter(new EventParameterPair()
{
    EventId = 4,
    ParameterName = "Tip Position"
});
(simulator as LoggingScenarioSimulator).AddTrackedParameter(new EventParameterPair()
{
    EventId = 4,
    ParameterName = "Tool Direction"
});
```

The position error and direction error of the tool will also be tracked over time to analyze the trainee's error versus time in the scenario. To calculate the position error, the Euclidean distance between two vectors will be used. Recall that the ideal position was determined during scenario generation and the actual position will be determined during the scenario performance by the trainee. The following is the formula for calculating the error between the ideal and actual position:

$$P_E = ||P_I - P|| \tag{5.9}$$

Where P_E is the position error, P_I is the ideal position and P is the actual position selected by the trainee. Since the error is being tracked over the time of the experiment, it is possible to express the position error as dependent on time:

$$P_E(t) = ||P_I - P(t)|| \tag{5.10}$$

The direction error will be calculated by determining the angle between two vectors; this can be done by utilizing the dot product:

$$D_E = cos^{-1}(\frac{D_I \cdot D}{|D_I||D|}) \tag{5.11}$$

Where the D_Err is the direction error, D_I is the ideal direction and D is the direction of the tool selected by the trainee. Similar to the position error, the direction error can be represented as a function of time:

$$D_E(t) = cos^{-1}(\frac{D_I \cdot D(t)}{|D_I||D(t)|}) \tag{5.12}$$

The two calculations, with time will produce a plot of the error over time. This plot in an ideal case, will have a negative slope, because the trainee will produce less error as the performance continues and the actual value becomes closer to the ideal value. Refer to figures 5.7 and 5.8 for example error versus time plots.

While these two plots give valuable information, there is one issue, they cannot be placed on the same plot effectively, because they use two different units. Position error is measured in distance (For example, meters) and direction error is measured in degrees. A new error calculation is proposed such that both position and direction error can be placed on the same plot: normalized error. Normalized error is aimed to remove the units from the error calculation

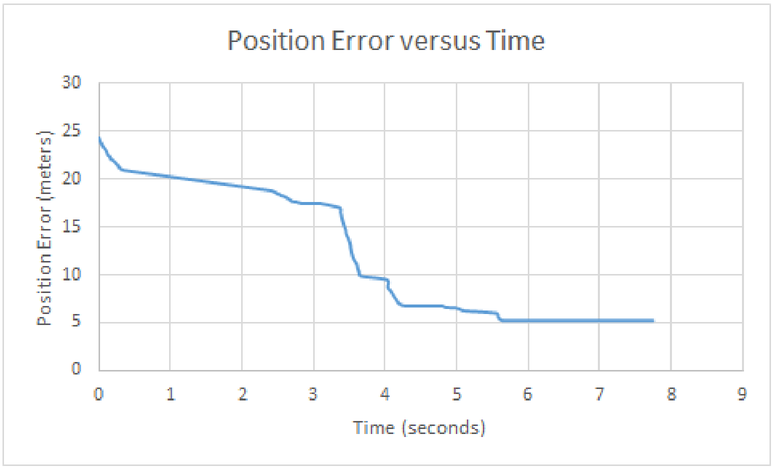


Figure 5.7: An example position error graph that is produced.

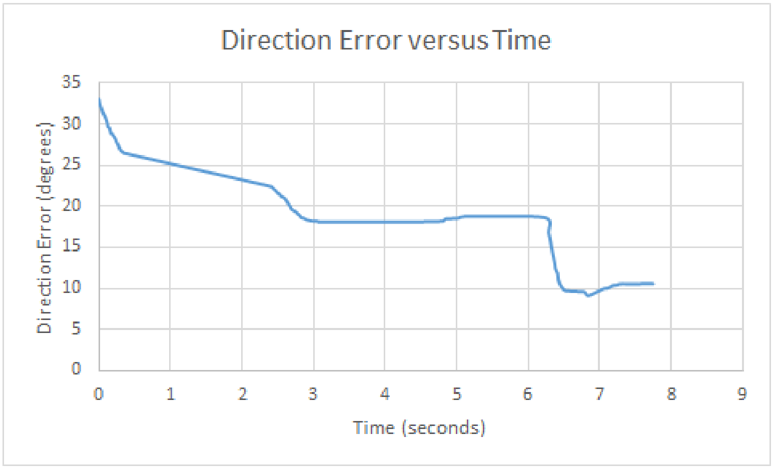


Figure 5.8: An example direction error graph that is produced.

and provide a standard scale to show the error over time. When actual value is equal to ideal value, the error calculation should result in 0, as there is no error; it is the perfect score. When the actual value is not equal to the ideal value, the error calculation should result in some value greater than 0, but to what max value? Direction error in degrees has a fixed range, being 0-180. The question becomes what should the opposite end of the scale be? In the case of position error, there is no opposite end of the scale, because the trial operates in an infinite space. A possibility could be to add a fixed range of the tool to have a fixed scale. Another possibility and the proposal is to provide an upper limit and that the beginning position and direction of the tool versus the ideal position and direction of the tool will produce an initial error. The normalized error at this initial error will be the upper limit. Thus, if an upper limit of 100 is used, the normalized error for both position and direction will be 0 when the toolâĂŹs position/direction will be equal to that of the idea position/direction. The normalized error will be 100, when position/direction will be equal to the starting position/direction. The following equation is the generic version of the normalized error:

$$N_E(t) = L(1 - \frac{Err(0) - Err(t)}{Err(0)}) \tag{5.13}$$

Where L is the upper limit value, $Err(t)$ is the error at time t and $Err(0)$ is the initial error. The following is the normalized error calculation for position:

$$P_{NE}(t) = 100(1 - \frac{P_E(0) - P_E(t)}{P_E(0)}) \tag{5.14}$$

The following is the normalized error calculation for direction:

$$D_{NE}(t) = 100(1 - \frac{D_E(0) - D_E(t)}{D_E(0)}) \tag{5.15}$$

The normalized error calculation allows for the plotting of the normalized error over time, where both the direction and position will have the same initial normalized error (100) and the same goal normalized error (0). A value of over 100, signifies that the user performed movements that actually made the error worse than the initial error. Both direction and position will share the same units, since the units are canceled in the calculation. Both direction and position can be plotted on the same graph to illustrate the error over time in the same context. The following plot is an example of the direction and position normalized error over time:

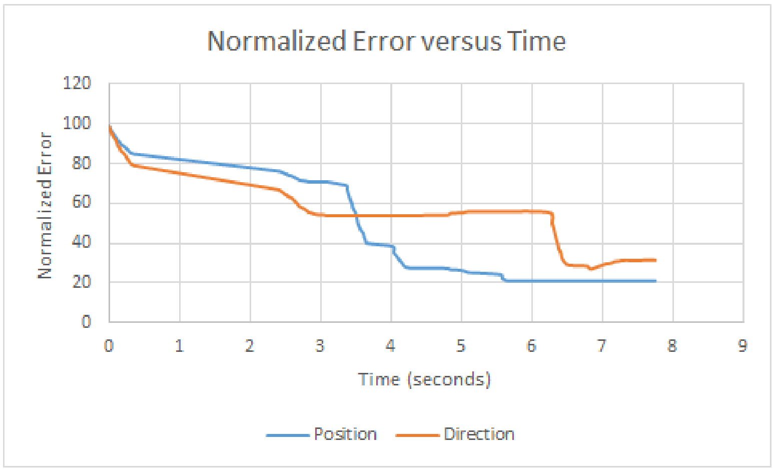


Figure 5.9: An example normalized error graph that is produced.

5.4.2 Error Tracking Components

To perform the tracking of the position error, direction error and their normalized counter parts, the component extension structure of the scenario simulator shall be utilized. Recall in section 3.6, that the scenario simulator offers the ability to add custom behaviour through simulation components. The structure of the error tracking components follows the template design pattern; the base template class is the *ErrorTrackingComponent* component, which implements the *ISimulationComponent* interface. The decision to follow the template pattern was made because of the common functionality between the tracking of position and direction parameters. Figure 5.10 illustrates the error tracking component structure. The base *ErrorTrackingComponent* component class is responsible for managing the error values over time and this is handled in the *Submit* method:

```
public void SubmitEvent(ScenarioEvent e)
        {
            var parameter = from EventParameter p in e.Parameters
                            where p.Name == parameterName
                            select p;

            if (parameter.Count() == 0)
                return;

            errors.Add(new ErrorMetricEntry()
            {
                Timestamp = e.Timestamp,
                Error = Calculate((Vector3f)parameter.First().Value, idealValue)
            });
        }

protected abstract float Calculate(Vector3f actual, Vector3f idealValue);
```

This simply finds the correct parameter, calculates its error value and adds it to the collection with the timestamp. The calculate method is an abstract method that will be implemented by the position and direction error tracking components respectively:

```
protected override float Calculate(Vector3f actual, Vector3f idealValue)
        {
            return Vector3f.DistanceBetween(idealValue, actual);
        }

protected override float Calculate(Vector3f actual, Vector3f idealValue)
        {
            return Vector3f.AngleBetween(idealValue, actual);
        }
```

For the normalized error tracking components, they simply inherit their base error tracking component and override the *Calculate* method like so:

```
protected override float Calculate(Vector3f actual, Vector3f idealValue)
        {
            if (errors.Count == 0)
                return scalingFactor;

            return scalingFactor * (1 - (errors[0].Error - base.Calculate(actual,
                idealValue)) / errors[0].Error);
        }
```

With these component classes, it is now possible to calculate the position and direction over time, as well as the normalized counterparts.

5.5 Scene Design

5.5.1 Trial Scene

The trial scene is the scene in the application where the trainee will perform a number of scenarios in a row, while the application will collect the data from the performance in the

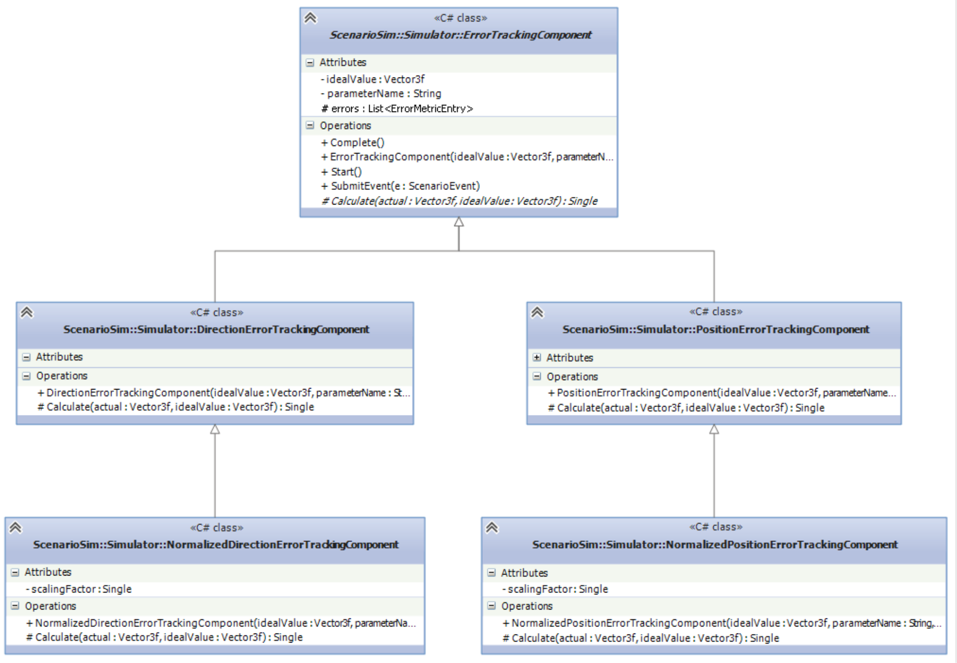

«C# class»
ScenarioSim::Simulator::ErrorTrackingComponent
Attributes
- idealValue : Vector3f
- parameterName : String
# errors : List<ErrorMetricEntry>
Operations
+ Complete()
+ ErrorTrackingComponent(idealValue : Vector3f, parameterN...
+ Start()
+ SubmitEvent(e : ScenarioEvent)
# Calculate(actual : Vector3f, idealValue : Vector3f) : Single
«C# class»
ScenarioSim::Simulator::DirectionErrorTrackingComponent
Attributes
Operations
+ DirectionErrorTrackingComponent(idealValue : Vector3f, parameterName : St...
# Calculate(actual : Vector3f, idealValue : Vector3f) : Single
«C# class»
ScenarioSim::Simulator::PositionErrorTrackingComponent
Attributes
Operations
+ PositionErrorTrackingComponent(idealValue : Vector3f, parameterName...
# Calculate(actual : Vector3f, idealValue : Vector3f) : Single
«C# class»
ScenarioSim::Simulator::NormalizedDirectionErrorTrackingComponent
Attributes
- scalingFactor : Single
Operations
+ NormalizedDirectionErrorTrackingComponent(idealValue : Vector3f, parameterNa...
# Calculate(actual : Vector3f, idealValue : Vector3f) : Single
«C# class»
ScenarioSim::Simulator::NormalizedPositionErrorTrackingComponent
Attributes
- scalingFactor : Single
Operations
+ NormalizedPositionErrorTrackingComponent(idealValue : Vector3f, parameterName : String,...
# Calculate(actual : Vector3f, idealValue : Vector3f) : Single

Figure 5.10: The error tracking components structure.

background. The scene starts with the head and tool in place and a countdown timer alerting the trainee to when the first scenario will begin, refer to figure 5.11 for some screen shots.

As the scenario begins, the target is placed in the head based off of the input parameters discussed in section 5.4.

The trainee is now able to move the tool around and perform the scenario. Figure 5.13 shows a few screen shots of the scenario being performed.

As per our scenario, the trainee is able to switch control modes and change the view of the camera in the same fashion as moving the tool. Figure 5.14 shows the view of the scenario changed.

Recall that after the tool would collide with the head, the tool would be fixed in that position and the trainee will be able to fine-tune their selection by rotating the tool. Figure 5.15 shows the tool being rotate in the final part of the positioning tool task.

Finally, when the trainee has made their selection, the scenario completes and a countdown timer begins to alert the user of when the next scenario will begin.

5.5.2 Playback Scene

The playback scene allows for the viewer (trainee or evaluator) to select a previous performance result and replay it back to them. The scene begins with a file browser prompt for the viewer to select a result file they wish to playback. Once the file is selected, the performance is played back to the viewer on the screen. There are additional feedback mechanisms given to the viewer in the form of displaying which task is currently being performed, as well as display the ideal result to the viewer. The viewer will be able to see how the trainee positioned the tool and

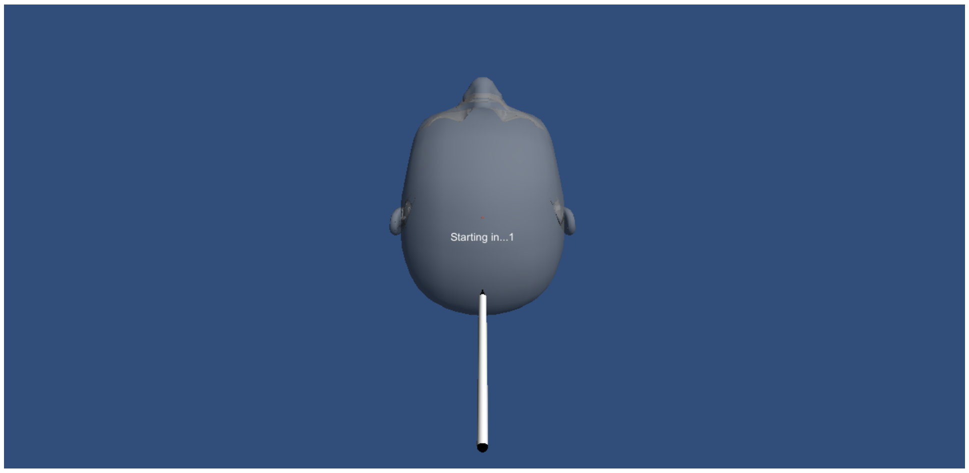


Figure 5.11: The initial screen.

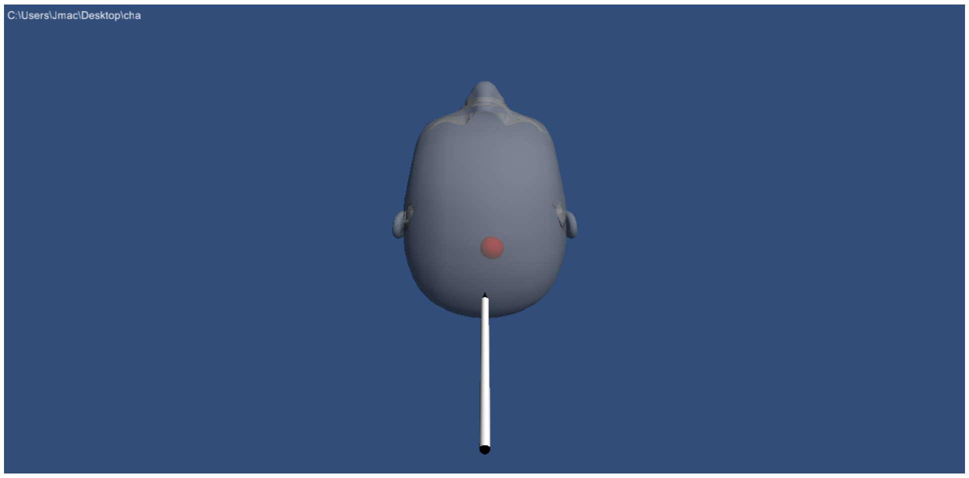


Figure 5.12: The generated target.

changed the view of the camera, while seeing the ideal result all at the same time. Figure 5.16 shows the playback scene that the viewer would see.

5.5.3 Ghosting Scene

An interesting possibility with the playback module is the ability to not only playback a previous performance for viewing, but also to playback a previous performance while the trainee is performing a scenario. A second tool can be placed in the scene and it will be controlled by the playback module, all while the trainee is performing the scenario with their own tool and control of the camera. This feature is referred to as ghosting, where the trainee will have a second "ghost" tool that will be performing the scenario with them while they perform the scenario. Ghosting offers the ability to act as a teaching tool, potentially providing an expert's performance as the ghosted playback while the trainee watches and performs the scenario at

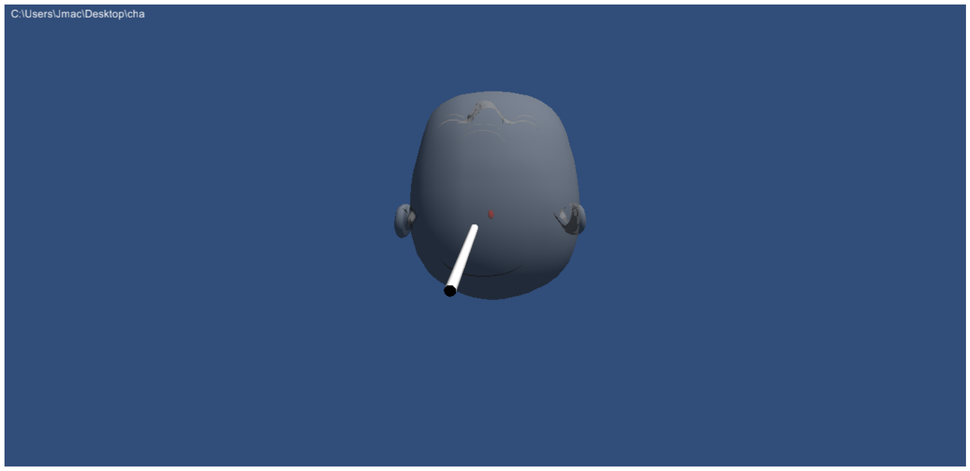


Figure 5.13: The trainee moving the tool.

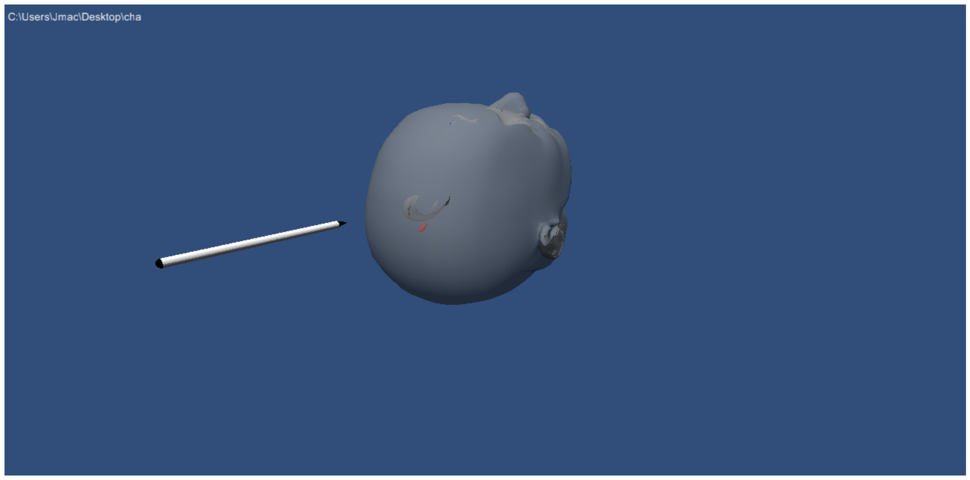


Figure 5.14: The trainee changing the view.

the same time. Refer to figure 5.17 for screenshots of the ghosting scene.

5.5.4 Operating Room Scene

The underlying goal of the framework was to provide the ability to robustly collect data and analyze performance for surgical simulator and more specifically the ETV procedure. The trial scenario scene offers a simple targeting task, which relates to the insertion of the trocar task of the ETV procedure, but a floating head in the air is not very realistic. The operating room scene provides a more realistic environment to the user, while using the same mechanics and scenarios from the basic targeting scene. Since the operating room is built using the exact same logic, it supports the playback and ghosting features as well. Figure 5.18 shows screen shots of the operating room scene.

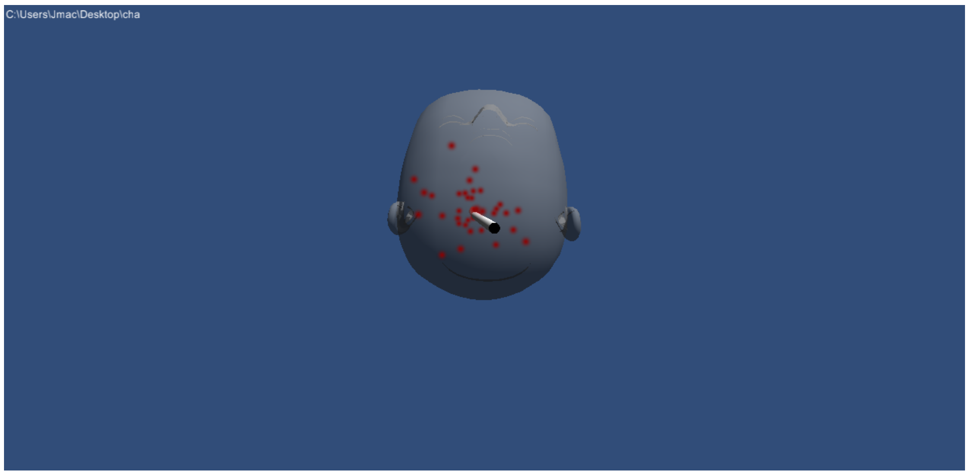
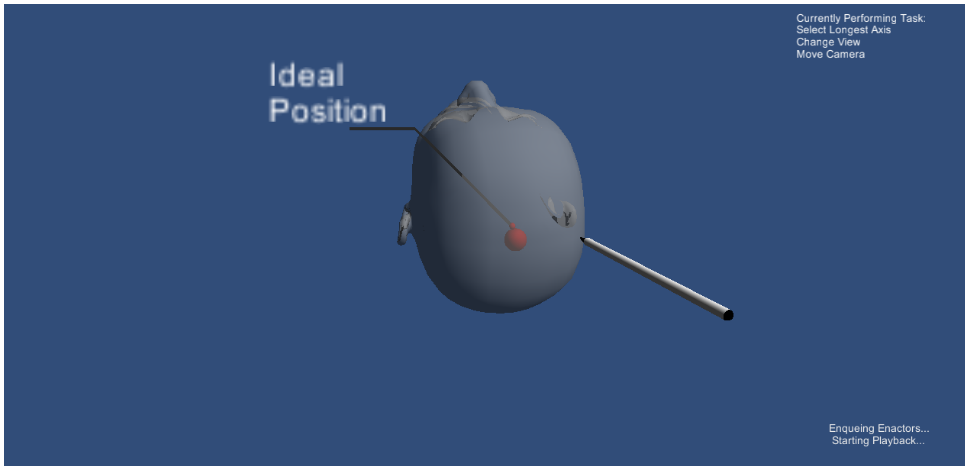


Figure 5.15: The trainee rotating the tool.



Currently Performing Task:
Select Longest Axis
Change View
Move Camera

Enqueing Enactors...
Starting Playback...

Figure 5.16: The playback screen.

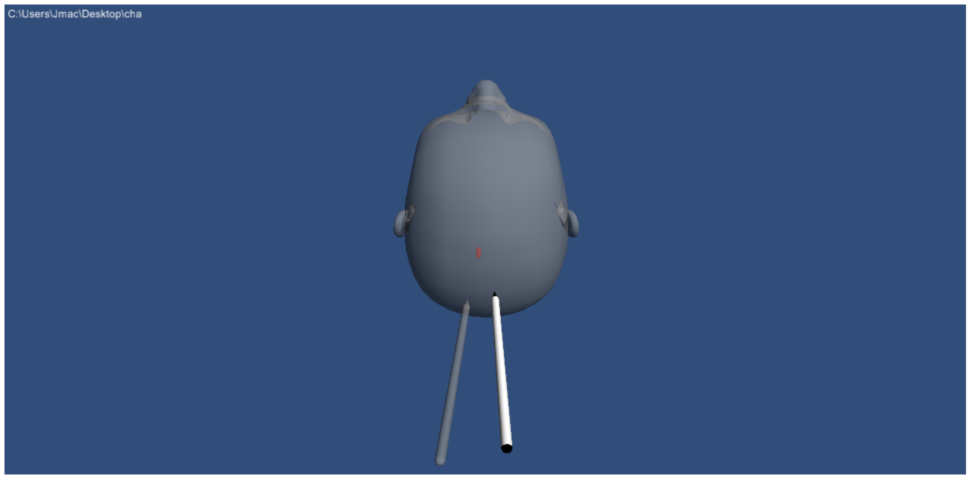


Figure 5.17: The ghosting screen.

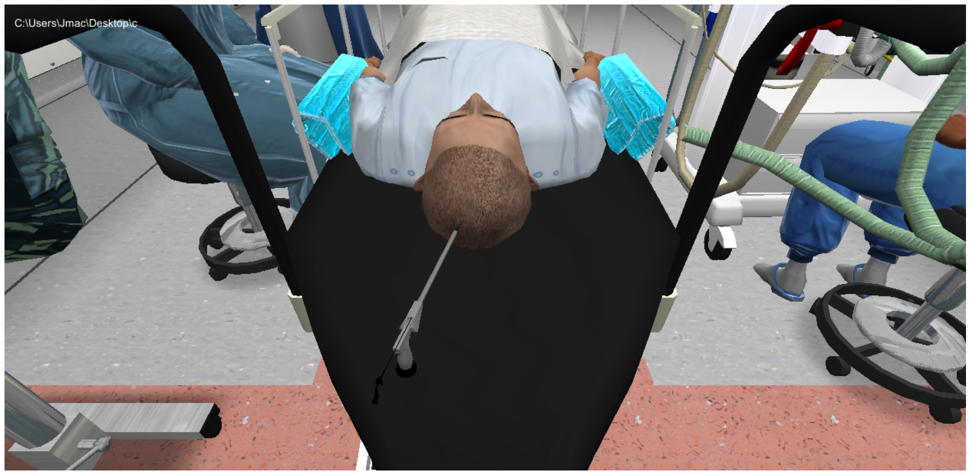


Figure 5.18: The operating room scene.

Chapter 6

The Ellipsoid Orientation Matching Task: Implementation in Unity3D

6.1 The Trial Manager

To organize and conduct a series of scenarios in sequence, there must be some sort of flow control from one scenario to another. The trial manager is the component that acts as the overall controller of the application. The trial manager is responsible for reading in a number of scenario files and constructing a collection of scenarios that will be performed by the trainee.

```
 DirectoryInfo info = new DirectoryInfo(scenarioFolderPath);

        FileInfo[] files = info.GetFiles("*.scenario");

        foreach (FileInfo f in files)
            scenarioCollection.Add(f.FullName);
```

Given a folder path, the trial manager will retrieve the file path of all scenario files in the given directory.

Once the trials have reached the starting point, the trial manager will communicate with the scenario manager to start the given scenario.

```
manager.StartScenario(User, scenarioCollection[currentScenario], resultFolderPath);
```

Once a trainee has completed the current scenario, the trial manager will save all necessary results into a folder, zip the folder and upload through FTP to a remote location where an evaluator can evaluate the results. Using the Ionic Zip library (https://dotnetzip.codeplex.com/), zipping a folder programmatically is fairly straight forward:

```
void UploadFolder(string folder)
    {
        string zipFile = string.Format("{0}\\{1}.zip", outputFolder, resultFolder);
        using (ZipFile file = new ZipFile())
        {
            file.AddDirectory(folder);
            file.Save(zipFile);
        }

        UploadFile(zipFile);
    }
```

6.2 The Scenario Manager

The scenario manager component is responsible for communicating with the scenario simulator interface provided by the framework. The relationship between the scenario manager and the scenario simulator framework is illustrated in figure 6.1. The scenario manager primarily provides methods to other components that mirror those of the scenario simulator interface to allow other components indirect access to the scenario simulator. The design decision for this set up was to keep all references to the scenario simulator interface nestled away into only one component; this way if the interface for the scenario simulator ever changes, then only one component of the client application would need to change. Another responsibility of the scenario manager is that it will handle the creation and starting of the scenario simulator module; this includes:

- Creating a new scenario simulator object.
- Registering the complication enactors.
- Registering the tracked parameters.
- Starting the scenario simulator object.

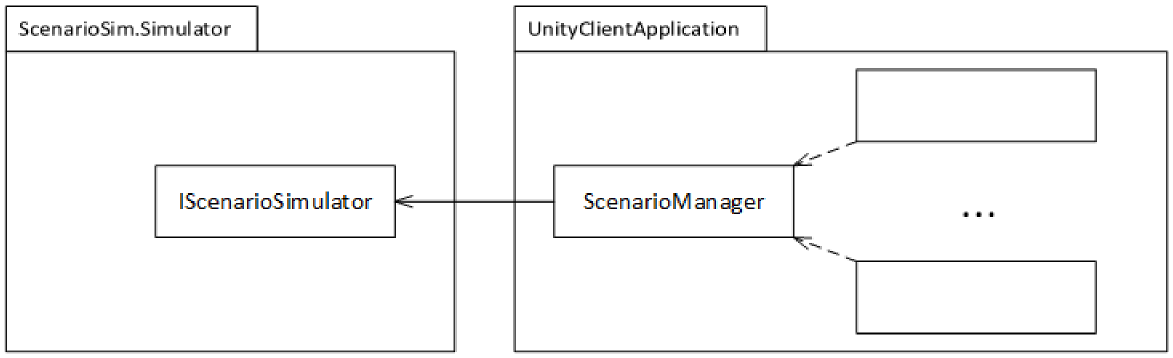


Figure 6.1: The scenario manage relationship with the framework.

[Code example here]

6.3 The Playback Manager

To provide the playback functionality to the user in the simulator, the playback module of the scenario simulator module will be incorporated. The scenario playback provides a single and easy-to-use interface to client applications and to communicate with this interface, the playback manager component was created. The responsibility of the playback manager is to interface with the scenario playback module on behalf of the rest of the simulator. Figure 6.3 illustrates this relationship.

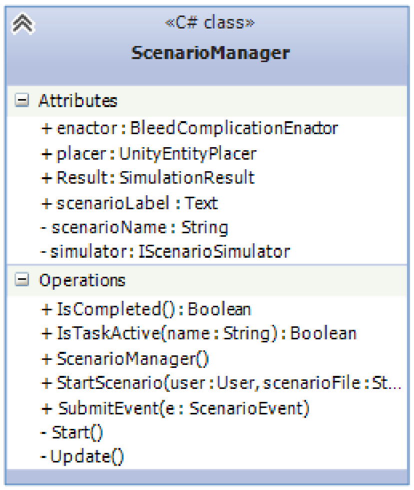

«C# class»
ScenarioManager
Attributes
+ enactor : BleedComplicationEnactor
+ placer : UnityEntityPlacer
+ Result: SimulationResult
+ scenarioLabel : Text
- scenarioName : String
- simulator: IScenarioSimulator
Operations
+ IsCompleted() : Boolean
+ IsTaskActive(name : String) : Boolean
+ ScenarioManager()
+ StartScenario(user : User, scenarioFile : St...
+ SubmitEvent(e : ScenarioEvent)
- Start()
- Update()

Figure 6.2: The scenario manage class.

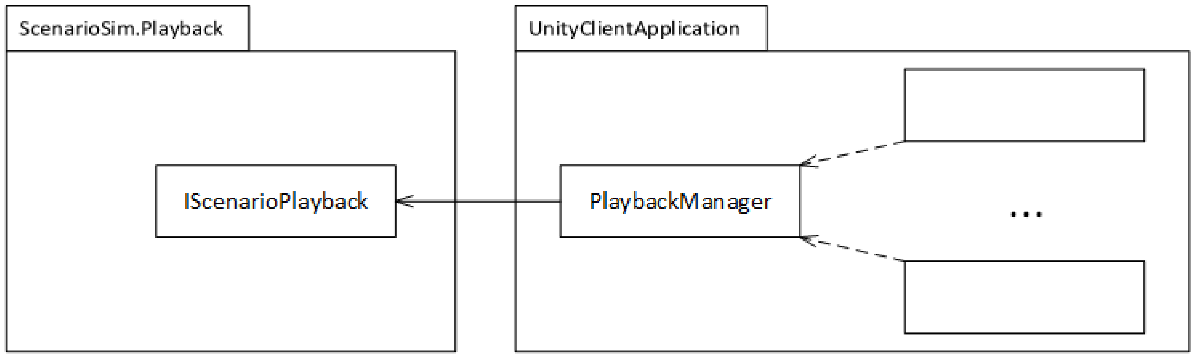


Figure 6.3: The playback manager relationship.

6.4 The Input Manager

The input manager is a component that is responsible for taking the user input and executing the proper command based off of that input. Since the input manager derives the Unity *MonoBehaviour* class, it's update method will be called every frame. The update method is key to the input manager's functionality; every update frame, the input manager scans all of the potential inputs such as the mouse and keyboard keys for activation. If an activation is found, the input manager will then create a command and execute that command to perform the proper functionality. The command that the input manager creates is a derivation of the submit event command, which will be discussed in section /refsec:submitEventCommand.

In Unity, the *Input* class provides a set of static methods to easily access any input queries. The get axis method is used for retrieving the intensity value of a given axis, where an axis is defined by some input devices that has an analog state, such as a mouse or joystick. A button is a discrete state, because it is either down or not. To retrieve the axis intensity value, simply call the *GetAxis* method and pass the axis name as a string, for example: "Mouse X" to get the intensity change in the mouse in the x-direction or "Mouse Y" to get the intensity change in the mouse in the y-direction. The following code will retrieve the x-axis intensity of the mouse at the given moment:

```
float xAxis = Input.GetAxis("Mouse X");
```

Retrieving the state of buttons is also done through the Input class, through the *GetButtonDown* method. The following code will retrieve the button down state of a button named "Switch Control":

```
bool buttonDown = Input.GetButtonUp("Switch Control");
```

Using these tools offered by the *Input* class, the *InputManager* component will determine what the user input is at the given frame. Essentially, the input manager will check the range of inputs such as the mouse axis, the mouse scroll wheel access, the switch mode key and the selection key. If a condition returns true for a key down request or returns a non-zero value for an axis value, the input manager will create the appropriate command and execute it. For example, if the scenario is currently in the position tool phase and the user has moved the mouse in the x-axis and/or y-axis, the input manager will create a submit translate tool event command to be executed. If instead the scroll axis is a non-zero value, a submit move tool in command will be created. The following code illustrates this creation of the submit translate tool event command:

```
if (scenarioManager.IsTaskActive("Translate Tool"))
            {
                if (xAxis != 0 || yAxis != 0)
                    command = new SubmitTranslateToolEventCommand(scenarioManager,
                        DateTime.Now, xAxis, yAxis, Stylus, Head);
                else if (scrollAxis != 0)
                    command = new SubmitMoveToolInEventCommand(scenarioManager, DateTime.Now,
                        scrollAxis, Stylus, scalingFactor);
            }
```

Lastly, at the end of update method, the command is executed:

```
if (command != null)
            command.Execute();
```

6.5 The Entity Placer

The entity placer is a class that implements the *IEntityPlacer* interface in the scenario simulator module that was discussed in section 3.7. Recall, that the purpose of the entity placer was to initialize the scenario by positioning all of the entities in the scenario in the proper locations in the environment. The entity placer class in Unity is a concrete implementation of the entity placer abstraction that scenario simulator module depends on.

The implementation of the entity placer is fairly straightforward, but extremely important. The *IEntityPlacer* abstraction has one method that needs to be implemented and that is the place method, which has an *Entity* object as a parameter. The duty of the entity placer is to take that entity and place it in the correct position. The *UnityEntityPlacer* implementation contains a dictionary that contain the game objects as values to be placed and their corresponding entity names as keys; for example the tool's game object will be paired with the name "Tool", which is the name of the tool entity in the scenario. When the placer receives the place method call

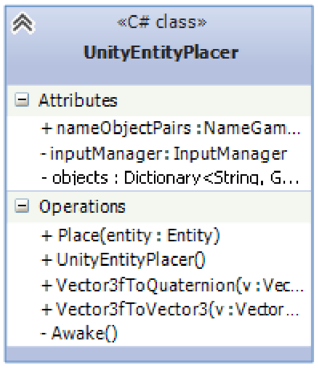

«C# class»
UnityEntityPlacer
Attributes
+ nameObjectPairs : NameGam...
- inputManager : InputManager
- objects : Dictionary<String, G...
Operations
+ Place(entity : Entity)
+ UnityEntityPlacer()
+ Vector3fToQuaternion(v : Vec...
+ Vector3fToVector3(v : Vector...
- Awake()

Figure 6.4: The unity entity placer.

with the tool entity, it will search the dictionary for the object with the key "Tool" and set the position, rotation and scaling of the corresponding game object to that of the entity in the scenario. The following code illustrates this process:

```
public void Place(Entity entity)
    {
        objects[entity.Name].transform.position =
            Vector3fToVector3(entity.transform.Position);
        objects[entity.Name].transform.rotation =
            Vector3fToQuaternion(entity.transform.Rotation);
        objects[entity.Name].transform.localScale = Vector3fToVector3(entity.transform.Scale);
    }
```

Where objects is the dictionary of name-game object pairs, *Vector3fToVector3* is a helper function to translate from the scenario vector implementation (*Vector3f*) to the Unity vector implementation (*Vector3f*) and *Vector3fToQuaternion* is a helper function to transform the Euler angles in the entity rotation definition to a Unity quaternion definition for rotation. The objects dictionary is initialized from the Unity editor inspector, where the designer of the Unity application can simply drag the desired game objects to be placed into the placer component and assign the entity name to pair with the game object.

6.6 Event Commands

The event commands are classes that execute the functionality of events in the simulator. The commands follow the command design pattern [28, 17], which features a command abstraction with an execute method that will execute all of the functionality of that command. One of the great advantages of using the command pattern is that it allows the separation of the invoker of the command from the receiver of the command. Since the simulator will be used as a platform for both the performance playback and the scenario simulation itself, it is necessary to separate the invoker from the receiver. The invoker for the playback will be an enactor object that implements the *IEventEnactor* interface from the playback module, where the invoker in the simulator will be the input manager, which is responsible for taking user inputs. Figure **??** illustrates the high level structure of the process to execute a command while the user is performing the scenario. Figure **??** illustrates the high level structure of the process to execute a command while the playback module is replaying the performance. The *ICommand* interface is illustrated below in figure 6.7

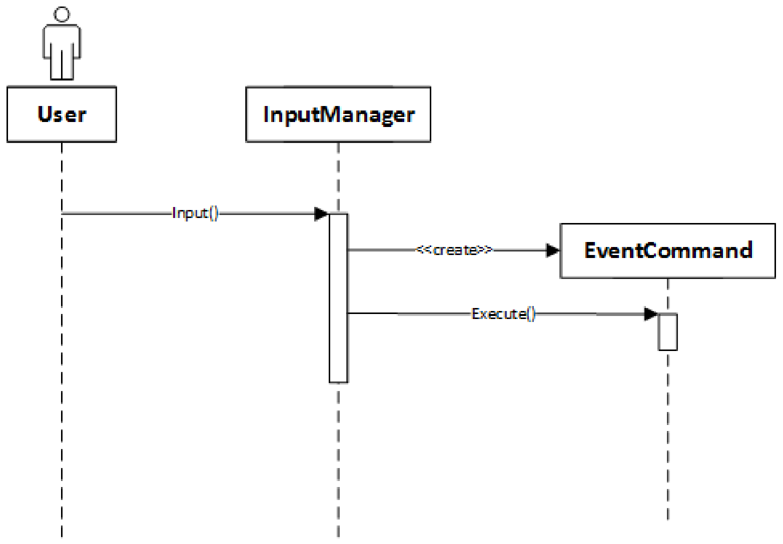


Figure 6.5: The command invokaton from the user input.

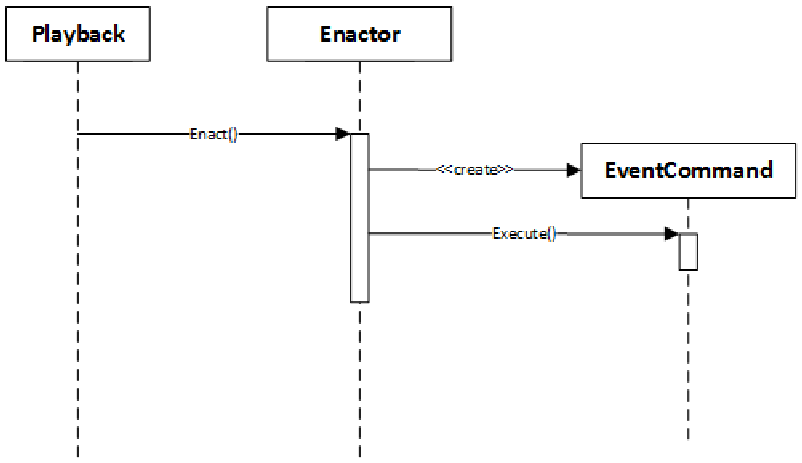


Figure 6.6: Command invocation from the play back.

6.6.1 Rotate Object Command

The rotate object command is used to translate the too or cameral based on the input from the invoker. The command constructor accepts an x-axis intensity, the object to rotate, the object to rotate around and a y-axis intensity. The object to rotate and the object to rotate around are game objects from the unity scene and the x-axis and y-axis intensity are values that can be passed from the user input script or from the scenario playback via the event enactor invoker

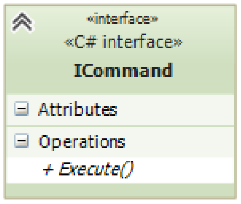


Figure 6.7: The ICommand interface.

script. The surgical tool and game camera are examples of objects that are rotated around. The patient's head and the tip of the tool (where it makes contact with the patient's head) are examples of game objects that are being rotated around. The x-intensity value and y-intensity value will control how much the object is being rotated. Since the *RotateObjectCommand* class conforms to the command design pattern, all of the functionality lies in the *Execute()* method. The following is the execute method of the translate tool command:

```
public void Execute()
    {
        rotatedObject.transform.RotateAround(referenceObject.transform.position, new
            Vector3(0, 1, 0), xIntensity * -1);
        rotatedObject.transform.RotateAround(referenceObject.transform.position, new
            Vector3(1, 0, 0), yIntensity);
    }
```

The method will take the rotated game object and rotate it around the reference game object, around the y axis by an angle based on the x-intensity value multiplied by negative one. It will also perform the same rotation, but around the x-axis and using the y-intensity. This will give two degrees of freedom to the user, when invoked through the simulation, to allow them to use the mouse to rotate based on the x and y translation of the mouse. This command will give the functionality to rotate the rotated object in an orbital fashion around the reference object.

6.6.2 Move Object In Command

The move object in command is used to move an object along a given axis, a certain amount. When looking at the simulator, this command would be used to move the surgical tool or the in-game camera inwards to or outwards from the skull. The command constructor accepts a game object to be translated, a scaling factor, an intensity and an axis to move along. The formula for translating the object is shown in equation 6.1

$$\vec{p} = \vec{p_0} + scalingFactor * intensity * \vec{axis} \tag{6.1}$$

The following is the execute method of the move object in command:

```
public void Execute()
    {
        movingObject.transform.Translate(scrollIntensity * scalingFactor * axis, Space.World);
    }
```

The code will simply call the translate method of the moving object to move it along the given axis, scaling it by the intensity and a scaling factor, which will control how much movement is caused. The scaling factor is mainly to handle the different scaling of scenarios. The basic targeting task context for instance has a much larger scale than the operating room context, so the operating room context will receive a much lower scaling factor to eliminate tools and cameras moving in and out too quickly. The second parameter, "Space.World", will tell unity to perform the transform in world space instead of local space.

6.7 Submit Event Commands

The submit event commands are a collection of commands used by the simulator to submit events to the scenario simulator module and invoke the event commands discussed in section 6.6. The submit event commands follow the command design pattern and all implement the same *ICommand* interface. There is currently a submit event command for each type of event that can be submitted to the scenario simulator for the designed scenario explained in section 5.3. All of the commands follow a similar structure; they each inherit the abstract base class, *SubmitEventCommand*, which contains all of the common logic in submitting an event to the scenario simulator module. The individual derived commands handle all of the specific behaviour to that command, which includes defining the parameters to be submitted with that event, along with creating and executing the correct event command that was described in section 6.6. Each command will be explained in more detail in the following subsections.

6.7.1 Submit Event Command

The submit event command is the base command class for all of the commands that deal with submitting an event. The command contains the logic and implementation dealing with creating the scenario event with the proper data and submitting it to the scenario simulator to be processed. The structure of these commands follows the template method pattern, where the common behaviour is placed into a template class, which in this case is the *SubmitEventCommand* and the specialized behaviour is placed into the derived classes. The following is the execute method from the submit event command class:

```
 public virtual void Execute()
    {
        ScenarioEvent e = new ScenarioEvent()
        {
            Id = id,
            Name = name,
            Description = description,
            Timestamp = timestamp,
            Parameters = GetParameters()
        };

        manager.SubmitEvent(e);
    }

protected abstract EventParameterCollection GetParameters();
```

The method above creates a scenario event from data that is injected into the command from the constructor. The get parameters abstract method is implemented by the derived, specialized submit event classes, because the parameters that each event submit is different. The submit hierarchy class is illustrated in figure 6.8

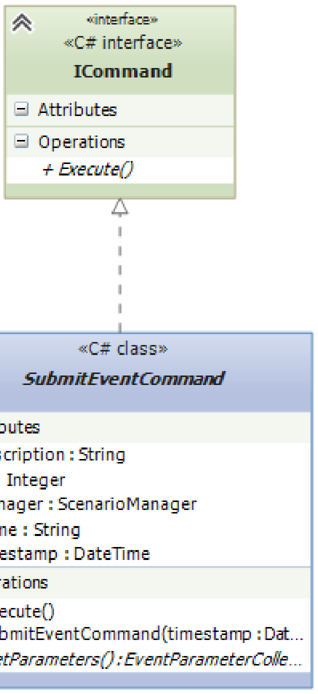


Figure 6.8: The submit event command.

6.7.2 The Derived Submit Event Classes

The derived and specialized submit event classes all follow the same implementation, since they all conform to the *SubmitEventCommand* abstract base class. All of the derived classes will implement the *GetParameters* method, which will return a collection of parameters based on data that is injected into the command. These classes will also execute any event command that was described in section 6.6. The following is a list of all of the submit event command classes:

- *SubmitChangeViewEventCommand* : Will be executed when the user selects to change to view mode from positioning tool mode.
- *SubmitCollisionEventCommand* : Will be executed when the tool collides with the patient's head.
- *SubmitMoveCameraInEventCommand* : Will be executed when the user zooms in or out the camera.
- *SubmitMoveToolInEventCommand* : Will be executed when the user moves the tool inwards to or outwards from the patient's head.
- *SubmitRotateToolEventCommand* : Will be executed when the user is rotating the tool around the collision point

- *SubmitPositionToolEventCommand* : Will be executed when the user selects to change to position tool mode from change view mode.
- *SubmitSelectionEventCommand* : Will be executed when the user has made their selection and the scenario will be complete.
- *SubmitTranslateCameraEventCommand* : Will be executed when the user elects to orbit the camera around the patient's head.
- *SubmitTranslateToolEventCommand* : Will be executed when the user elects to orbit the tool around the patient's head.

Since all of the commands perform the same functionality, just with the parameters differing, along with which event command to execute, only the *SubmitTranslateToolEvent* will be discussed in detail.

6.7.3 Translate Tool Command

As mentioned in section 6.7.1, the translate tool command is created and executed when the user has elected to orbit the tool around the patient's head. The command has two responsiblities: to create the parameter collection that will be submitted by the base template class, *SubmitEventCommand* and to create and execute the *RotateObjectCommand*, which was discussed in detail in section 6.6.1. The following is the execute method from the translate tool command class:

```
public override void Execute()
    {
        translateToolCommand.Execute();
        base.Execute();
    }
```

This implementation is very simple, all it does is execute the translate tool command, which is an instance of the *RotateObjectCommand* class. Next, it will submit the event to the scenario simulator, by calling the base (*SubmitEventCommand*) execute method. The following is the get parameters method, which will supply the list of parameters for the base submit event command to package with the scenario event to be submitted to the scenario simulator module.

```
protected override EventParameterCollection GetParameters()
    {
        EventParameterCollection result = new EventParameterCollection();
        result.Add(new EventParameter()
        {
            Name = "X-Axis Intensity",
            Value = xIntensity
        });

        result.Add(new EventParameter()
        {
            Name = "Y-Axis Intensity",
            Value = yIntensity
        });

        result.Add(new EventParameter()
        {
            Name = "Tip Position",
            Value = tool.transform.FindChild("Tip").position.ToVector3f()
        });

        result.Add(new EventParameter()
            {
                Name = "Tool Direction",
                Value = (tool.GetComponent(typeof(LongestAxis)) as
                    LongestAxis).AxisDirection.ToVector3f()
            });

        return result;
    }
```

All of the derived command class perform this same behaviour, the main difference is the set of parameters that the individual command will return in the *GetParameters* method. Another small difference is the command that will be executed in the execute method; for instance, in the above command, a *RotateObjectCommand* is executed, where in another command it will be different. For example, the *SubmitMoveToolInEventCommand* will execute a *MoveObjectInCommand*. The submit translate tool event command will submit four parameters to the scenario simulator module:

- X-Axis Intensity: The intensity of the mouse movement in the x-axis direction. (Floating Point Number)
- Y-Axis Intensity: The intensity of the mouse movement in the y-axis direction. (Floating Point Number)
- Tip Position: The position of the tip of the tool. (3-Dimensional Vector)
- Tool Direction: The direction that the tool is pointing in. (3-Dimensional Vector)

There is motivation behind collecting all of these parameters, as they will be used during scenario performance evaluation and analysis. The x and y axis intensity values are collected to show user movement over the time of the scenario and are also used to replay the performance in the playback module after they have been serialized with the event. The tip position and tool direction are collected to once again demonstrate the position and trajectory of the tool over time; this is used to chart the error and normalized error over time, which is described in section 5.4.1. The benefit of using this parameter collection structure is that it allows for very custom behaviour, if there are more parameters that are desired to be submitted, simply add them to this method. As discussed in section 3.4, the parameter is composed of a name and a value, where the value is an object, thus any type of value can be placed in that property.

Upon further inspection, it is possible to compile all of these commands into one class. One of the differences between them is the command that they execute inside of their execute method. Since all commands implement the *ICommand* interface, an *ICommand* object can be injected into the individual submit event commands and they will execute the given command.

6.8 The Complication Enactors

In order for complications to be displayed on the screen to the trainee, the client application must implement and register the enactors they wish to display on the screen. The client application has one concrete enactor that it implements and registers with the scenario simulator and that is the *BleedComplicationEnactor* class; figure 6.9 illustrates this class. The bleed complication enactor will create a bleed particle system in the scene at position of the tool tip. The enact method implementation is shown below:

```
public void Enact()
    {
        Quaternion q = new Quaternion();
        Vector3 rotation = new Vector3(0, 180, 0);
        q.eulerAngles = rotation;
        bleed = Instantiate(bleedParticle, tip.position, q);
        bleedAudioObject = Instantiate(bleedAudio, tip.position, q);
    }
```

This code simply creates a bleed particle at the positon of the tool's tip and also plays a sound.

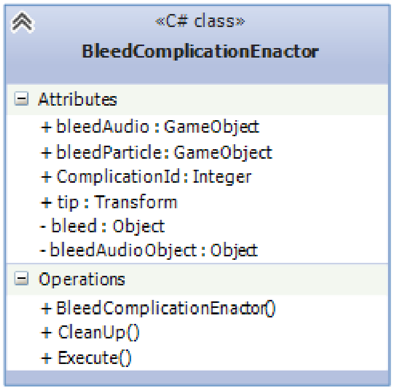

«C# class»
BleedComplicationEnactor
Attributes
+ bleedAudio : GameObject
+ bleedParticle : GameObject
+ ComplicationId : Integer
+ tip : Transform
- bleed : Object
- bleedAudioObject : Object
Operations
+ BleedComplicationEnactor()
+ CleanUp()
+ Execute()

Figure 6.9: The bleed complication enactor class.

6.9 Enacting Events for Playback

Recall from section 4.5, the playback module of the framework relies on the enactor pattern to playback events to the use in the client application. It is the client application's responsibility to implement the concrete enactors that will playback the events to the user. This section will describe the implementation of the enactors that provide the ability to playback the scenario performance on the unity simulator platform. Firstly, the command invoker component, which is responsible for invoking the event commands discussed in section 6.6, will be discussed. Secondly, the event enactors themselves, will be discussed.

6.9.1 The Command Invoker

The command invoker is the class that is responsible for executing the event commands that are sent by the event enactors through the playback module. The original design of the enacting of events in the playback client did not contain a command invoker class, its creation was a result of a multi-threading issue in unity. The enactors are called from a separate thread because the playback uses a timer elapse event to enact the correct events at the correct time and this timer elapse event is fired from a new thread. Originally, the implemented enactor would create the event command with the proper parameters and then execute it, but this would not work because the unity's restriction of access to certain elements from different threads, similar to how .NET windows forms cannot be manipulated from another thread.

To counter this issue, the command invoker class was created. The invoker has two main responsibilities:

- Collect and store a temporary buffer of commands.
- To execute and remove the commands from the buffer on every update call.

With the new design, the enactors will queue the desired command inside the command invoker instead of executing the command. On every update call from Unity's engine, the command invoker will empty its queue and execute every command it dequeues. This approach works because the new thread that is created from the timer elapse event inside of the scenario playback module will not be manipulating any game objects; it will simply add the command to a queue and forget about it. When the main thread is in its update cycle and comes to the invoker class, it will pick up the command and execute it with no issue; figure 6.10 illustrates this process. The following code is the update method in the command invoker class:

```
// Update is called once per frame
    void Update()
    {
        lock (commands)
        {
            while (commands.Count > 0)
            {
                ICommand command = commands.Dequeue();
                command.Execute();
            }
        }
    }
```

This is one of the greatest benefits of the command pattern, it allows for the encapsulation of a request to be used at a later time.

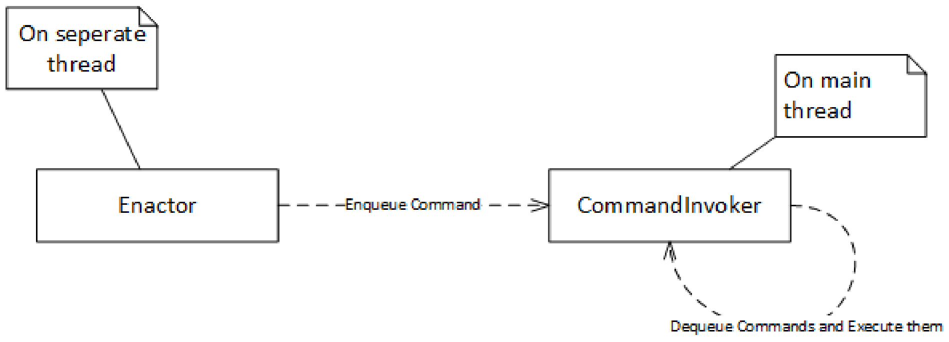


Figure 6.10: The command invoker process.

6.9.2 The Event Enactor

The event enactors are a set of enactors that assume the role of the concerete enactor in the enactor pattern discussed in section 3.8. There is an abstract base enactor, the *EventEnactor*, which implements the enact method. In its enact method, the base enactor simply retrieves the proper command from its derived classes and enqueues this command in the command invoker.

The following source code demonstrates this functionality:

```
public void Enact(ScenarioEvent e)
    {
        invoker.EnqueueCommand(GetCommand(e));
    }

protected abstract ICommand GetCommand(ScenarioEvent e);
```

Refer to figure 6.11 for the template pattern of the event enactor structure.

The derived enactor class will supply the specific command and inject the proper data to those commands from the scenario event data. The following is an example of the *GetCommand* implementation in the *TranslateToolEnactor* class:

```
protected override ICommand GetCommand(ScenarioEvent e)
    {
        return new TranslateToolCommand(
            (float)e.Parameters.FindByName("X-Axis Intensity").Value,
            (float)e.Parameters.FindByName("Y-Axis Intensity").Value,
            tool, head);
    }
```

Once this command is enqueued in the command invoker and the command invoker will execute it, the tool will be translated around the head by itself, as if the user was performing the task again.

There is one enactor for each type of event that needs to be visually enacted in a scenario; these would include:

- MoveToolInEnactor : Will be created and enacted when a move tool in event is to be perfromed.
- RotateToolEnactor : Will be created and enacted when a rotate tool event is to be perfromed.
- MoveCameraInEnactor : Will be created and enacted when a move camera in event is to be perfromed.
- TranslateToolEnactor : Will be created and enacted when a translate tool event is to be perfromed.
- TranslateCameraEnactor : Will be created and enacted when a translate camera event is to be perfromed.

The events that do not need a visual display, such as a selection event or a change of mode event, don't necessarily need an enactor, but can be given one. An example enactor would be displaying a label on the screen alerting the evaluator that a change of mode was made and this enactor would be very simple to implement. The developer would need to implement a command class that would change a label's text, derive a child class of *EventEnactor* that

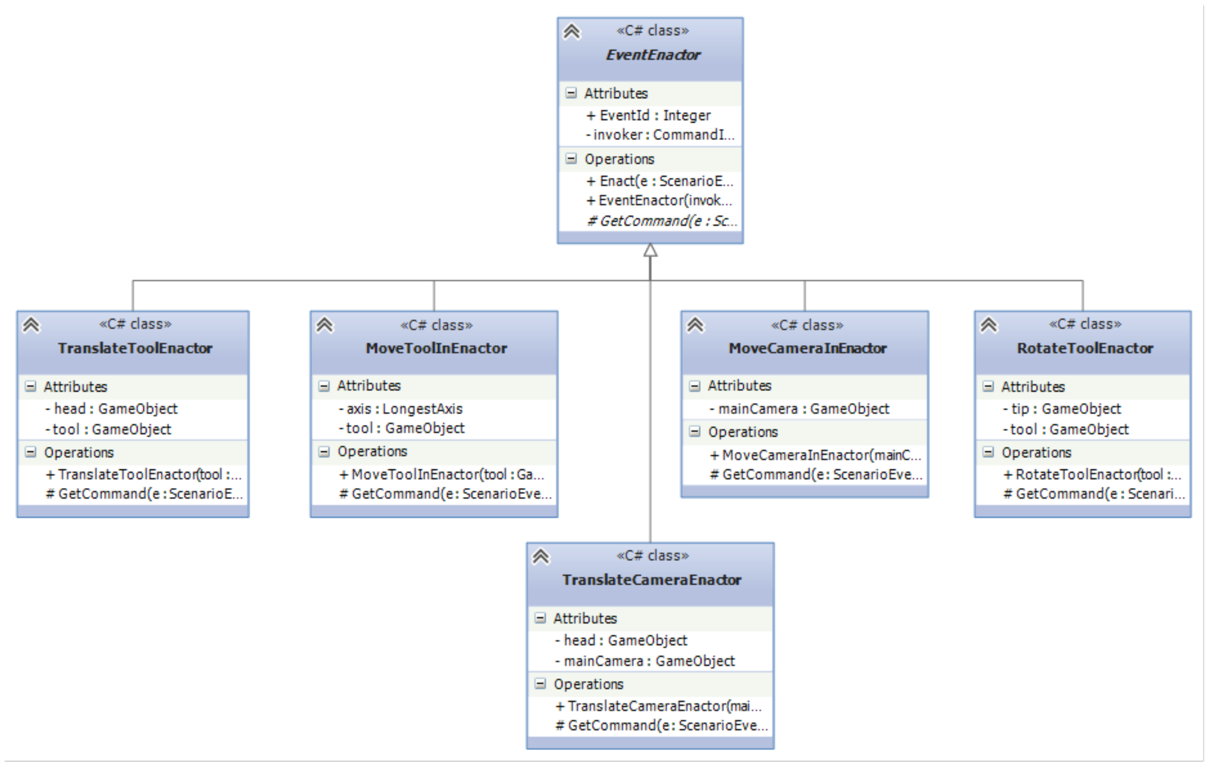


Figure 6.11: The command invoker structure.

would implement the *GetCommand* method to return newly created command. Finally, the developer would simply need to register the enactor with the playback and that is it.

The entire playback process has been fully discussed throughout this chapter and chapter 4. To recap, the following is the process for enacting a translate tool event:

1. The playback module timer elapses.
2. The playback checks if any events need to be played at the moment. There is a translate tool event.
3. The playback checks its event enactor repository for an enactor that is registered for the translate tool event. It finds one.
4. The playback retrieves the enactor and calls the enactor method.
5. Now on the Unity side, the concrete enactor will create a translate tool command with the correct tool game object, head game object and the data from the event.
6. The translate tool command will be submitted to the command invoker.
7. On the main thread, when the command invoker is executing its update method, it will dequeue the command and execute it.

8. The command will perform the translation of the tool around the head based on the parameters from the event.

Chapter 7

Closing Remarks

7.1 Discussion and Conclusion

Recall the research questions proposed in section 1.7:

- Can a simulation software framework that allows the custom data collection and evaluation of the performance of a complex hierarchical task be implemented?

- Can the simulation software framework be proven to robustly collect data without leaving a large performance footprint?

- Can the simulation software framework be integrated into a simple surgical simulation scenario to collect meaningful and evaluate performance data for that scenario?

This thesis began with the problem of how to calculate performance of complex tasks such as a surgical procedure. The literature provided answers to both human performance evaluation and hierarchical task analysis. Hierarchical task analysis can be used to decompose a complex task such as surgery into smaller simpler tasks that are simpler to evaluate. Fitts' law and its derivatives can be used to evaluate difficulty and performance of simple targeting tasks. After utilizing hierarchical task analysis to break down the complex task into simple tasks that can be evaluated using Fitts' law, the performance of the complex task can be aggregated and calculated. The performance of every task can be calculated, which allows for detailed feedback to trainees for each task they perform in a surgical procedure. A Fitts' law model can be derived by varying the D and W parameters to define different index of difficulty values. With the mean time to complete the tasks and the varying index of difficulty values, the a and b parameters can be derived.

$$MT = a + bID = a + blog_2 \frac{2D}{W} \tag{7.1}$$

It was shown that the mean time to complete a complex task can be represented by a weighted sum of the index of difficulties of the descendant leaf nodes in the hierarchical task tree:

$$MT_T = \sum_{i=1}^{N} a_i + b_i ID_i \tag{7.2}$$

Using this approach, it is possible to predict the mean time to complete a complex task based on the index of difficulty of the simple descendant tasks.

A number of simulator frameworks have been developed; these include SOFA, GiPSi and SPRING. SOFA is the only framework of the three that offers any sort of out of data collection or monitoring, but it is rather limited with no customizability built in without the need for extensive development. SOFA offers no hierarchical task representation of a simulation scenario, so there is no data collection on a task-based level and therefore cannot be used to evaluate performance on a task-based level. A software framework is developed with reusability and extendibility in mind; it can be used with any type of client application that would like to evaluate user performance in an abstract way, all while custom data collection schemes can be easily added. The framework's architecture allows for high maintainability and loose coupling between different layers; the abstractions and the details are separated through the dependency inversion principle and results in an onion architecture. Reducing the coupling between layers in the architecture produces higher modularity and allows for implementations to be swapped in and out, thus increasing the maintainability of the system.

A simple core domain module is developed to act at the center of the entire framework without any external dependencies. The elements of hierarchical task analysis and performance evaluation are realized in the core domain. A hierarchical task representation was used to realize the ability to evaluate performance of sub-tasks in a broader complex task such as a surgical procedure. It is believed that a breakdown of a surgical performance and empirical results of the sub-tasks provides a more objective measure of performance, rather than a weighted or subjective-based scoring mechanism. It is also believed that determining performance of sub-tasks allows a more useful and robust evaluation that provides both the evaluator and the trainee better feedback for each sub-task and phase of a complex task. Evaluation of each sub-tasks allows the trainee to focus on improving the exact parts of the overall task that were given a weaker performance score rather than guessing what they can improve upon. Providing an automatic and numerical evaluation method for the performance of sub-tasks allows for trainees to view performance results without the need of an expert to evaluate performance; this allows a trainee to perform surgical simulation tasks at anytime and anywhere, depending on the portability of the simulator. Producing data and evaluating performance on the simple task level is a concept that is new and will be researched further in the future and it is ideal to design a realistic and full surgical procedure scenario, where trainees will perform the procedure and the individual tasks' performance in the procedure will be derived.

To perform the data collection in a customized way as mentioned in the research questions, a component pattern is utilized to provide the simple injection of custom data collection functionality. The scenario simulator module allows the ability to robustly collect performance data of performed scenarios. The module offers the ability for custom components to be plugged in and are based to allow for custom behaviour in the module. Initially, there was no component structure implemented and all of the functionality for time collection, event collection, logging and such were hard coded into the simulator module. The result of this initial implementation was that the simulator module required modification whenever custom data collection functionality was required, which violates the open/closed principle. It was observed that all of the data collection functionality that was implemented in the initial simulator module contained behaviour at three separate points: the beginning, the end and when an event is submitted by the trainee. From the three different points, an abstraction was generated and all data collection components followed this abstraction, the *ISimulationComponent*. If a simulator developer wishes to incorporate their own custom functionality for the specific simulator, it can be accomplished by implementing a component that implements the interface and adding an instance of it to the simulator module instance. The module allows for custom events to be fired and enacted in the client application, while utilizing the proposed enactor design pattern. Two components that are implemented that can be used for evaluating performance are the time keeping component, which is responsible for tracking the amount of time spent performing each task and the custom accuracy component, which is responsible for determining how accurate a trainee was in performing a specific task. Utilizing the data collected from these two components can be used in the future to determine the performance of a specific task by a trainee The playback module provides a way to means for better evaluation of a performance; it proposed a new way to replay performances based off of the events that are submitted to the simulator during the performance. Playback provides a means to evaluate performance through the eye test and allows experts, evaluators and trainees to review a task performance while providing custom data visualization at the same time as the playback. The purpose of providing custom data visualization playback is to provide additional evaluation and learning cues all in one place to improve the evaluation and learning process. It is planned that in the future, experiments be designed and executed to determine if the additional data visualization can provide a benefit to the learning and/or evaluation process.

Recall that it was a research question and goal to investigate if the proposed simulator data collection framework could operate while leaving a low overhead footprint. A series of tests were performed to evaluate the performance hit produced by the various data collection components. The results showed that the data collection components did not require much CPU time and executed their functionality quickly and would leave little to no effect on the simulation loop, where the graphics, input and physics were calculated. The components that logged to files on the disk were the slowest, (4.70 μs for CSV logging and 3.81 μs for plain test logging) but these components can be improved to access the disk only once at the end of the simulation performance, instead of N times during the performance, where N is the number of events performed by the trainee. Utilizing the second approach would require only saving the logs in memory during the execution the task and writing the logs to disk at the end of the performance. The non-I/O data collection components required much less CPU time, where the highest was the state chart component. (0.755 μs) For comparison sake, for a 60 frames per second system, one frame or loop can take up to approximately 16,667 μs to complete, thus even the I/O data collection components as-is, 4.70 and 3.81 μs, take up only 0.0282 % and 0.0228 % of the simulation loop respectively.

A client application was developed to demonstrate the ability of the framework in the Unity3D game engine. The application allowed users to perform a simple targeting task and the performance data is collected while the user performs the task. The task was to align the tool with the longest axis of the ellipsoid target; this task resembles the task for a surgeon to orient themselves with the longest axis of a brain tumor during the brain removal surgical procedure. Meaningful data was collected by utilizing the provided time keeping component in the framework and developing a custom accuracy evaluation component that was plugged into the simulation data collection module. It was shown that accuracy (position and direction error) could be tracked and reported over the time of the task, as well as the final position and direction error of the selection made by the trainee. The error values and time values alone could be used to evaluate the performance of a task as they provide an objective score of how well the trainee performed. The client application allows for the generation of custom scenario parameters for the task to vary the target's position and thus varying the difficulty of the task. The generation of custom scenario parameters will be utilized in the future when trials will be used to develop a Fitts' law model of the task. The client application allows for the ability to playback previous performances with additional data overlaid on the screen utilizing the playback module of framework. It can be investigated in the future to determine whether the data overlay provides additional benefit to the learning and evaluation process. The client application allows for the "ghosting" feature, which allows for an additional tool to be shown that would be controlled by the playback while the user is also able to perform the task at the same time. It is a possibility to design and execute an experiment in the future to investigate the effect the ghosting feature may have on the learning process in the future.

This thesis had a few research questions that revolve around an idea that hasn't been developed and implemented before and has shown that it is possible to develop a customizable and yet general performance evaluation framework for simulators. It was shown that the performance footprint left by the framework's data collection components was small and did not affect the performance of the simulator in question. A simple targeting task simulation was implemented using the Unity3D game engine and utilized the developed framework to collect meaningful data for performance evaluation and provided playback capability to evaluate performance through the eye test. A lot of additional ideas are proposed in this thesis that are not validated and if there was more time, it would have been great to be able to validate and test these ideas. It is a goal in future work to develop tests to validate the various ideas proposed in this thesis, such as the effect of the playback data visualization overlay as a learning and evaluation tool, the ghosting feature as a learning tool and the ability to predict time to complete the task based on the index of difficulty of the simple descendant tasks in a task hierarchy to name a few. This thesis provides the foundation of a new and expandable idea to the field of simulation task performance evaluation and it is the goal of future work to expand this idea and validate it to the point where it is a viable option to simulation developers to be able to collect, analyze and evaluate task performance data.

7.2 Future Work

A desired feature of the framework would be able to work clients of any programming language. Currently, the framework is developed in C# and .NET and it works with Unity3D through the C# scripting engine. A plan to create a web API to host the C# simulator module that would communicate over HTTP has been proposed, so any client language or framework can be used with the scenario simulator module.

A web application to create and author the scenarios that will be simulated is greatly desired and a C# Windows Forms prototype has been developed. Ideally the prototype will be moved to a MVC5 web application to allow greater portability. Client simulator applications would be able to load scenario files from this web application.

Bibliography

[1] National cancer institute: Pdq adult brain tumors treatment. http://www.cancer.gov/types/brain/patient/adult-brain-treatment-pdq.

[2] Johnny Accot and Shumin Zhai. Beyond fitts' law: models for trajectory-based hci tasks. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 295–302. ACM, 1997.

[3] Andy Adler and Michael E Schuckers. Calculation of a composite det curve. In *Audio- and Video-Based Biometric Person Authentication*, pages 860–868. Springer, 2005.

[4] G Ahlberg, T Heikkinen, L Iselius, C-E Leijonmarck, J Rutqvist, and D Arvidsson. Does training in a virtual reality simulator improve surgical performance? *Surgical Endoscopy and Other Interventional Techniques*, 16(1):126–129, 2002.

[5] Latif Al-Hakim, Tanaphon Maiping, and Nick Sevdalis. Applying hierarchical task analysis to improving the patient positioning for direct lateral interbody fusion in spinal surgery. *Applied ergonomics*, 45(4):955–966, 2014.

[6] Jérémie Allard, Stéphane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Hervé Delingette, and Laurent Grisoni. Sofa-an open source framework for medical simulation. In *MMVR 15-Medicine Meets Virtual Reality*, volume 125, pages 13–18. IOP Press, 2007.

[7] Sonya Allin, Yoky Matsuoka, and Roberta Klatzky. Measuring just noticeable differences for haptic force feedback: implications for rehabilitation. In *Haptic Interfaces for Virtual Environment and Teleoperator Systems, 2002. HAPTICS 2002. Proceedings. 10th Symposium on*, pages 299–302. IEEE, 2002.

[8] Graeme Walters Neville Stanton Andrew Bass, John Aspinall. A software toolkit for hierarchical task analysis. *Applied Ergonomics*, 26(2):147–151, 1995.

[9] C Baber and N Stanton. Analytical prototyping. *C3 Industrial Control Computers and Communications Series*, 16:175–194, 1998.

[10] J M Beaubien and D P Baker. The use of simulation for training teamwork skills in health care: how low can you go? *Quality and Safety in Health Care*, 13 Suppl 1:i51–6, Oct 2004.

[11] M Cenk Cavusoglu, Tolga G Goktekin, Frank Tendick, and Shankar Sastry. Gipsi: an open source/open architecture software development framework for surgical simulation. *Studies in health technology and informatics*, 98:46–8, 2004.

[12] Yeonjoo Cha and Rohae Myung. Extended fitts' law for 3d pointing tasks using 3d target arrangements. *International Journal of Industrial Ergonomics*, 43(4):350–355, 2013.

[13] Jonathan Cohen, Seth A Cohen, Kinjal C Vora, Xiaonan Xue, J Steven Burdick, Simmy Bank, Edmund J Bini, Henry Bodenheimer, Maurice Cerulli, Hans Gerdes, et al. Multicenter, randomized, controlled trial of virtual-reality simulator training in acquisition of competency in colonoscopy. *Gastrointestinal Endoscopy*, 64(3):361–368, 2006.

[14] Alfred Cuschieri. Human reliability assessment in surgery–a new approach for improving surgical performance and clinical outcome. *Annals of the Royal College of Surgeons of England*, 82(2):83, 2000.

[15] S R Dawe, G N Pena, J A Windsor, J A J L Broeders, P C Cregan, P J Hewett, and G J Maddern. Systematic review of skills transfer after surgical simulation-based training. *The British Journal of Surgery*, 101(9):1063–76, Aug 2014.

[16] Paul M Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of experimental psychology*, 47(6):381, 1954.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[18] Maya M Hammoud, Francis S Nuthalapaty, Alice R Goepfert, Petra M Casey, Sandra Emmons, Eve L Espey, Joseph M Kaczmarczyk, Nadine T Katz, James J Neutens, Edward G Peskin, et al. To the point: medical education review of the role of simulators in surgical training. *American journal of obstetrics and gynecology*, 199(4):338–343, 2008.

[19] James A Hanley and Barbara J McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, 143(1):29–36, 1982.

[20] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[21] TJ Johnston, B Tang, A Alijani, I Tait, RJ Steele, J Ker, G Nabi, Surgical Simulation Group at the University of Dundee, et al. Laparoscopic surgical skills are significantly improved by the use of a portable laparoscopic simulator: Results of a randomized controlled trial. *World journal of surgery*, 37(5):957–964, 2013.

[22] Wendela Kolkman, MAJ Van de Put, R Wolterbeek, JBMZ Trimbos, and FW Jansen. Laparoscopic skills simulator: construct validity and establishment of performance standards for residency training. *Gynecological Surgery*, 5(2):109–114, 2008.

[23] Rhonda Lane, Neville A Stanton, and David Harrison. Applying hierarchical task analysis to medication administration errors. *Applied ergonomics*, 37(5):669–79, Sep 2006.

[24] Alan Liu, Frank Tendick, Kevin Cleary, and Christoph Kaufmann. A survey of surgical simulation: applications, technology, and education. *Presence: Teleoper. Virtual Environ.*, 12:599–614, December 2003.

[25] R D Luce and W Edwards. The derivation of subjective scales from just noticeable differences. *Psychological review*, 65(4):222–37, Jul 1958.

[26] I Scott MacKenzie. Fitts' law as a research and design tool in human-computer interaction. *Human-computer interaction*, 7(1):91–139, 1992.

[27] Alvin Martin, George Doddington, Terri Kamm, Mark Ordowski, and Mark Przybocki. The det curve in assessment of detection task performance. In *Fifth European Conference on Speech Communication and Technology*, 1997.

[28] Micah Martin and Robert C Martin. *Agile principles, patterns, and practices in C#*. Pearson Education, 2006.

[29] Kevin Montgomery, Cynthia Bruyns, Joel Brown, Stephen Sorkin, Frederic Mazzella, Guillaume Thonier, Arnaud Tellier, Benjamin Lerman, and Anil Menon. Spring: A general framework for collaborative, real-time surgical simulation. *Studies in health technology and informatics*, pages 296–303, 2002.

[30] Stephen J Motowidlo, Marvin D Dunnette, and Gary W Carter. An alternative selection procedure: The low-fidelity simulation. *Journal of Applied Psychology*, 75(6):640, 1990.

[31] Quinn T Ostrom, Haley Gittleman, Peter Liao, Chaturia Rouse, Yanwen Chen, Jacqueline Dowling, Yingli Wolinsky, Carol Kruchko, and Jill Barnholtz-Sloan. Cbtrus statistical report: primary brain and central nervous system tumors diagnosed in the united states in 2007–2011. *Neuro-oncology*, 16(suppl 4):iv1–iv63, 2014.

[32] Jeffrey Palermo. The onion architecture: part 1. http://jeffreypalermo.com/blog/the-onion-architecture-part-1/.

[33] Erik Prytz, Michael Montano, and Mark W Scerbo. Using fitts' law for a 3d pointing task on a 2d display: Effects of depth and vantage point. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 56, pages 1391–1395. SAGE Publications, 2012.

[34] S K Sarker, R Hutchinson, A Chang, C Vincent, and A W Darzi. Self-appraisal hierarchical task analysis of laparoscopic surgery performed by expert surgeons. *Surgical endoscopy*, 20(4):636–40, Apr 2006.

[35] Sudip K Sarker, Avril Chang, Tark Albrani, and Charles Vincent. Constructing hierarchical task analysis in surgery. *Surgical endoscopy*, 22(1):107–11, Jan 2008.

[36] A Shepherd. Hta as a framework for task analysis. *Ergonomics*, 41(11):1537–52, Nov 1998.

[37] Irwyn A Shepherd, Cherene M Kelly, Fiona M Skene, and Karin T White. Enhancing graduate nurses' health assessment knowledge and skills using low-fidelity adult human simulation. *Simulation in Healthcare*, 2(1):16–24, 2007.

[38] R William Soukoreff and I Scott MacKenzie. Towards a standard for pointing device evaluation, perspectives on 27 years of fittsâĂŹ law research in hci. *International journal of human-computer studies*, 61(6):751–789, 2004.

[39] Neville A Stanton. Hierarchical task analysis: developments, applications, and extensions. *Applied Ergonomics*, 37(1):55–79, June 2006.

[40] Shaun Shi Yan Tan and Sudip K Sarker. Simulation in surgery: a review. *Scottish medical journal*, 56(2):104–109, 2011.

[41] J Torkington, SG Smith, BI Rees, and A Darzi. The role of simulation in surgical training. *Annals of the Royal College of Surgeons of England*, 82(2):88, 2000.

[42] Ateet Vora. A hierarchical task analysis software tool based on the model-view-controller architecture pattern. Master's thesis, Rutgers, January 2011.

[43] M H Zweig and G Campbell. Receiver-operating characteristic (roc) plots: a fundamental evaluation tool in clinical medicine. *Clinical chemistry*, 39(4):561–77, Apr 1993.

Curriculum Vitae

Name:	Justin Mackenzie
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2009-2013 B.E.Sc.
	University of Western Ontario London, ON 2013 - 2015 M.E.Sc
Honours and Awards:	OGS 2014-2015
Related Work Experience:	Teaching Assistant The University of Western Ontario 2013 - 2015

Publications:

MacKenzie, J.; Carnegie, S.; Schmalz, J.; Schmalz, M.; de Ribaupierre, S.; Eagleson, R., "Surgical simulation workflow representation using hierarchical task analysis and statecharts: Implementation on the evolution engine," Games Media Entertainment (GEM), 2014 IEEE , vol. 1, no. 2, pp. 22-24, Oct. 2014