Paper Presented at AlaSim 2014

Multiagent Test Range: Fostering Disciplined Software Engineering Practices in Students via Modeling, Simulation, Visualization, and Analysis

Dan Tappan
Department of Computer Science, Eastern Washington University, Cheney, WA, USA
dtappan@ewu.edu

Keywords: software engineering, experiment-based testing and evaluation, pedagogy

ABSTRACT: *This pedagogy-oriented system complements modeling, simulation, visualization, and analysis with software engineering and software quality assurance, as well as scientific method, to provide students a hands-on, holistic experience of real-world software development and evaluation. It provides a highly extensible virtual test range for designing, building, and evaluating a variety of military platforms — airplane, ships, and submarines — with rich combinations of munitions, sensors, and fuzes.*

1 Introduction

The testing strategy of typical undergraduate software engineering students is a shotgun approach of unfocused, nonrepeatable tests of questionable rigor and value. Testing is an ad hoc afterthought because they have no experience with developing a disciplined test plan, a formal methodology to carry it out, and a persuasive means to demonstrate the results. This pedagogy-oriented system mitigates these problems through a richly extensible, student-friendly Java integrated modeling-and-test environment for discrete-event simulation of component-based agents within a virtual test range. It allows students to define, build, manipulate, and evaluate simplified real-world platforms (airplanes, ships, and submarines) with a wide variety of smart and dumb munitions, tracking sensors, and triggering fuzes.

Computational modeling, simulation, visualization, and analysis (MSVA) rely heavily on object-oriented programming, design patterns, and software engineering, but the converse is rarely the case. Software engineering — at least at the undergraduate level — is traditionally taught as practical top-down problem-solving. For logistical reasons, the process is often linear as mostly analysis, design, and especially implementation, with some testing, but without much regard to the holistic role the system is intended to play and how to establish how well it does so. The advanced concepts of software quality assurance are often relegated to the graduate level. Many undergraduates thus have little exposure to the critical end stages of verification, validation, accreditation, and certification. Insight into them could contribute to better understanding and more targeted decisions in the earlier stages.

To this end, the primary goal of this system, as well as its overarching pedagogical approach, is to integrate the perspectives of both MSVA and software engineering in such a way that students can learn to understand and apply them to their own problems. It capitalizes on yet a third perspective, which is required of students in their studies but often considered irrelevant: the study and application of science. Scientific method is the foundation of modeling and simulation to determine the behavior of virtual systems that correspond to counterparts in the real world [1]. It should be equally useful for assessing software quality and performance, if done strategically [2]. Therefore, design and execution of controlled experiments, sensitivity analysis, performance metrics, and other formal techniques can be applied to software development as a persuasive, defensible way to present results and establish confidence in them.

2 Pedagogical Foundation

This work indirectly derives from the author's decade of experience as lead systems engineer and software architect for accredited modeling and simulation projects at the U.S. Army Materiel Systems Analysis Activity on the Future Combat Systems program at Aberdeen Proving Ground, Materiel Test Directorate and Systems Test and Assessment Directorate at White Sands Missile Range, Electronic Proving Ground at Fort Huachuca, and elsewhere. These teams had predominantly young, inexperienced members who found the guidance offered by the precursors to this work to be very helpful.

The academic product here, based on almost two decades of teaching computer science and engineering at the university level, is targeted toward helping students transition from the bottom-up "nuts and bolts" study of computer science in lower-division coursework to the top-down contextual problem-solving process of real-world, practical software engineering at the upper-division and graduate levels, as well as in professional work environments.

2.1 Critical Thinking

Students often want to hit the keyboard running and start coding a task upon first sight. Many think computer science is coding, and software engineering is just coding more. They generally resist with great effort the academic notion of thinking before doing, or formal analysis and design leading to implementation and evaluation. Background research into the subject matter is often regarded as "busy work," when in fact this grounding is critical to proper understanding and execution in software engineering [3]. To this end, this system plays an ideal role because students are expected to have no background in the subject matter, or what they think they know is likely wrong or misleading. Pushing them out of their comfort zone forces them to embrace this formalized approach instead of their own familiar, but limited, ad hoc ones of dubious rigor and value. These skills will prepare them to approach any new problem.

Problem-solving for any domain has been studied in endless detail. The approach of multidisciplinary critical systems thinking has long been effective in traditional engineering, and has more recently become an emphasis in software engineering [4,5]. This work considers a multidimensional approach of forcing students to decompose the pieces of a problem into what they are (data), what they can do (control), and what they actually do or have done to them (behavior) by critical analysis with the W^{5}H question words of *who*, *what*, *when*, *where*, *why*, and *how*. This low-level analysis then combines to form associative structures that connect the dots in a DIKW network, as in Figure 1.1 [6]:

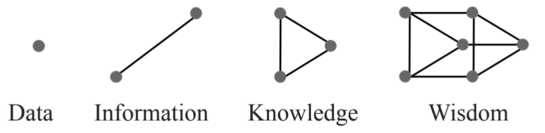

Figure 1.1: DIKW Learning Associativity

- ***D****ata*: raw values with no associativity or context
- ***I****nformation*: values in one context
- ***K****nowledge*: values in multiple contexts
- ***W****isdom*: creation of generalized principles by connecting a network of contexts from different sources for predictive, anticipatory, proactive understanding

Aspects of this approach overlap and complement the classic Bloom's Taxonomy of Educational Objectives, which rank cognitive activities from low to high level: *remember*, *understand*, *apply*, *analyze*, *create*, and *evaluate* [7]. Building and testing a software system of any complexity requires skillful manipulation of all these levels. The education community debates the order of the last two, but for the modeling-and-simulation community, this one is the norm [1].

2.2 Software as Surrogate

The basis of using modeling and simulation for software engineering is to build practical software components that demonstrably correspond to their real-world counterparts. There are two aspects: testing that the underlying code works as specified, and evaluating how well it works under various conditions of interest to learn from it. Both are closely related in reality, but in the software development process, they often radically diverge with little warning.

The process of mapping from specifications — at least in the form of coursework assignments — to programmatic solutions is familiar to students even in beginning courses. By and large they do a decent job, too. However, the opposite direction is almost never a consideration, even for experienced professionals: if someone unaffiliated with the problem were asked to hypothesize from its solution alone what the original problem space looked like, the two would likely bear little resemblance. In both directions, valuable details are lost or mangled, and extraneous ones are picked up. In a simulation environment where the program is a surrogate for the real world under virtual study, any misalignment in mapping may undermine the conclusions drawn from it [8]. It is therefore critical to build and maintain strong correspondences and be able to demonstrate them convincingly. The argument here is that this premise also holds true for software engineering in general.

To this end, software quality assurance should be a consideration from the very start, not an afterthought as testing at the end, as it is often practiced: design to build and test simultaneously. This approach entails determining what components can and cannot do, and then building an architecture to enforce these constraints. In fact, the mantra of the author's teaching philosophy is: *does what it is supposed to do; does not do what it is not supposed to do*. The amount of code dedicated to preventing, detecting, and handling errors often eclipses what actually does the intended work [3].

2.3.1 Code Considerations

The low-level goal of testing that students' code works as specified is the realm of traditional object-oriented programming and software engineering. This architecture provides two complementary opportunities: analysis for investigating how existing components function and interact, and synthesis for adding new ones. The emphasis in both cases is on clean, orthogonal solutions with well-designed, inherently defensive structure. In other words, with minimal effort, the architecture permits acceptable actions and prohibits unacceptable ones. This goal is critical in such a large, highly compositional, dynamic, plug-and-play system like this one, with its 320 classes. If done well, students do not have to expend time coding and testing for illegal combinations if they can show that these cases cannot happen. Unfortunately, most students work quite differently and love to hardcode uniquely for every little perceived special case, which leads to code that is bloated, brittle, and difficult to maintain. For example, one student practically bragged that in his development process, he "kept throwing more code at the compiler until it shut up." Section 4 discusses how the pedagogical approach here helps instill better discipline.

2.3.2 Simulation Considerations

The high-level goal of evaluating and learning from how well components work under various conditions is the realm of traditional modeling and simulation. This architecture explicitly supports scientific method as the primary means of investigation. In particular, it expects students to understand the nature of their problem space well enough to state what the expected results of their experiments should be before conducting them. It then provides the opportunity afterwards to reflect on any differences before proceeding to the next experiment. If the results were wrong, then the next attempt should be in a different direction; if they were correct, next should come incremental refinement in the same general direction.

Controlled experiments are the foundation. Students run a baseline experiment and record the results. They then intentionally perturb one — and only one — parameter and rerun the experiment under the same conditions. Any differences can then be directly attributed to this single change, which helps elicit sound cause-and-effect relationships. This strategy avoids the typical undisciplined student approach of indiscriminately changing a whole bunch of things at once and then having no idea what actually played a role, nor when, where, why, and how.

3 Architecture

The software architecture combines traditional model-view-controller modules that clearly separate the main concerns of the system [9]. While they are still interconnected, the dependencies are kept to a minimum such that different versions of the modules may be swapped in and out without undue burden. Such flexibility allows the system to be extended into other related domains, such as air traffic control, aircraft-carrier operations, and aircraft fly-by-wire control systems, which are recent adaptations investigated by students in other offerings of the author's software engineering courses.

3.1 Model

The model defines what agents are in terms of their data and control—what they are and are capable of doing, respectively. In object-oriented programming, this breakout maps directly to class member data and methods.

3.1.1 Agents

Agents are any component of the simulation that may be created, manipulated, and deleted dynamically. They include the three types interacting within the battlespace — actors, munitions, and sensors/fuzes — as well as graphical views of it. Section 4 addresses the acceptable combinations.

3.1.1.1 Actors

Actors populate the world. Their physical state is defined by three-dimensional world coordinates (latitude, longitude, and altitude or depth), course, and speed. They are primary agents because the behavioral commands in 3.3.1.2 can directly control these properties. Actors also contain an infinite supply of any appropriately defined combination of munitions:

- *Airplane*: may carry bombs, depth charges, torpedoes, and missiles.
- *Ship*: may carry main-gun shells, depth charges, torpedoes, and missiles.
- *Submarine*: may carry only torpedoes.

Each actor also has performance characteristics for its minimum and maximum speed, acceleration and deceleration rates, rate of turn, crush depth, and so on.

3.1.1.2 Munitions

Munitions populate actors. Their physical state is also defined by world coordinates, course, and speed, but they are secondary agents because the behavioral commands cannot directly control them.

The unguided munitions are dumb. After deployment, their lifespan is dictated by ballistic trajectories that cannot change.

- *Shell*: follows a parabolic arc based on the azimuth and elevation specified in the firing command in 3.3.1.1 and terminates at sea level or upon hitting a ship or surfaced submarine.
- *Bomb*: falls from the release altitude and also terminates at sea level or upon hitting a ship or surfaced submarine. The horizontal velocity of the airplane is imparted on the trajectory.

- *Depth charge*: if dropped from an airplane, falls from the release altitude with the imparted horizontal velocity until reaching sea level, where it then behaves as if it had been released from a ship. The depth charge then sinks straight down at a slower rate until detonating based on its fuze or reaching the sea floor. It cannot detonate at sea level, even if dropped onto a ship or surfaced submarine.

The guided munitions are smart fire-and-forget weapons. Their trajectories depend on the performance of their sensor and fuze and on the actions of the target.

- *Missile*: uses its sensor to track targets and its fuze to detonate only after exceeding a specified travel distance.
- *Torpedo*: uses its sensor to track targets and its fuze to detonate only after exceeding a specified arming time. If dropped from an airplane, it falls like a depth charge.

Each munition has performance characteristics for its minimum and maximum speed, acceleration rate, rate of turn, blast radius and yield, and so on.

3.1.1.3 Sensors and Fuzes

Sensors and fuzes populate munitions. They are functionally identical, except in their role: the former tracks a target, whereas the latter decides when to detonate its host munition.

Passive sensors receive energy only. They have a sensitivity property that allows them to determine a distance or bearing to a target and whether the energy exceeds a threshold.

- *Acoustic:* operates based on sound energy, which is a function of the speed of a primary agent.
- *Sonar*: operates based on reflected sound energy from an active sonar source provided separately.
- *Thermal*: operates based on thermal energy, which is a function of the speed of a primary agent.
- *Depth*: operates based on depth below sea level.
- *Distance*: operates based on elapsed distance traveled.
- *Time*: operates based on elapsed time traveled.

Active sensors are passive sensors that also emit energy. All emitters of the same type use the same notional frequency, so receivers can detect reflections from multiple emitters, for better or worse.

- *Radar*: operates by emitting a radio signal and receiving its reflection.
- *Sonar*: operates like radar, but with a sound signal.

Radar and thermal sensors have a conical field of view (FOV) that limits where they can see. The FOV may be fixed along the forward-facing longitudinal axis of a munition, or it may sweep horizontally over a field of regard (FOR) at a set rate. The latter requires the configuration of a movable mount.

For simplicity and consistency, power and sensitivity are based on percentages, not on real-world units like decibels. Attenuation in air and water is a function of distance. Radar reflectivity is also based on the rough characteristic dimension of the target: a target in profile (broadside) produces a stronger signal than head on.

3.1.2 Datatypes

Anecdotal evidence shows that students have a huge problem with abstracting, maintaining, and manipulating data properly. Java primitives are appropriate in earlier low-level courses, but at the project level, they lead to a proliferation of problems. For example, units and magnitudes are not applied consistently, error handling is almost nonexistent, and code bloats from haphazard attempts at reimplementing similar solutions in multiple places.

To mitigate this situation, the architecture provides a rich set of self-contained concrete datatypes for every kind of relevant data; e.g., `Airspeed`, `Altitude`, `Attenuation`, `Azimuth`, `WorldCoordinate`, `Course`, `Depth`, `Distance`, `FieldOfRegard`, `FieldOfView`, `Groundspeed`, `Heading`, `Identifier`, `Latitude`, `Longitude`, `Percent`, `Pitch`, `Power`, `Sensitivity`, `Time`, `Yaw`, and many dozens more not directly in play in this paper. Each maintains its own error checking and helper methods for manipulating and converting it appropriately. This approach lends itself to convenient unit testing in isolation. It also reduces the burden of documentation; e.g., avoiding having to state everywhere that horizontal angles are in navigational degrees because `Azimuth` always use this form.

Another incessant problem students have is with indiscriminate coupling and undisciplined, unprotected sharing of objects. Changing a mutable object in one place may have countless unexpected consequences throughout an entire system. To mitigate this problem, datatypes employ a functional paradigm, which makes them immutable. Any mutable action on them produces a new object via copy-on-write semantics [10].

3.2 View

The view module of the architecture manages how the user sees the output. While not considered agents in the traditional simulation sense, view windows are actually treated as such because they can be created, manipulated, and deleted dynamically through six commands. They play an integral role in testing and evaluation, so they are part of the world. For simplicity, it is a flat-earth model. The plug-and-play nature of the view allows any visualizer to connect to the architecture, provided that it follows the specified protocols.

3.2.1 Two-Dimensional Visualization

The three-dimensional world is presented as any number and combination of two-dimensional views from different top, front, and side perspectives, as in Figure 3.1. The position, size, scale, and grid-line configuration of each is independent. Agents are represented by various glyphs, which include explicit state information like identifier, speed, heading, and altitude, as well as metadata like track and predicted impact point. The display can be zoomed, dragged, resized, and locked onto an agent, among other useful features for evaluation.

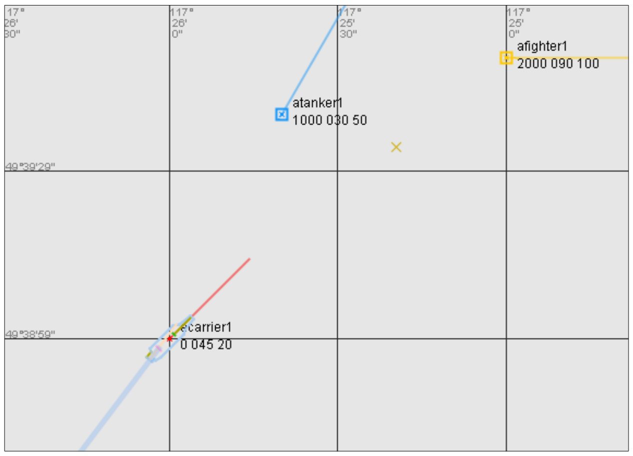

Figure 3.1: Two-Dimensional Top Perspective

The user may also insert himself or herself into the world as a nonparticipating agent any number of times as fixed reference points for meta-analysis. This feature allows the user to narrate the execution of a test plan from a specific vantage; e.g., looking northeast from the southwest corner of the world 10 kilometers from the ship, the missile passed from right to left at low altitude.

3.2.2 Three-Dimensional Visualization

Three-dimensional visualization, both for dynamic runtime analysis and static postanalysis, is also available through a Java 3D plug-in. Figure 3.2 shows visualizations for a variety of views on a test world. It also includes depictions of otherwise unseen aspects like fields of view and regard. This visualizer has seen extensive use in the author's artificial intelligence courses, related pedagogical research, and industry work as a general-purpose world viewer [11,12,13]. Gnuplot is also supported as an export format, but for postanalysis only.

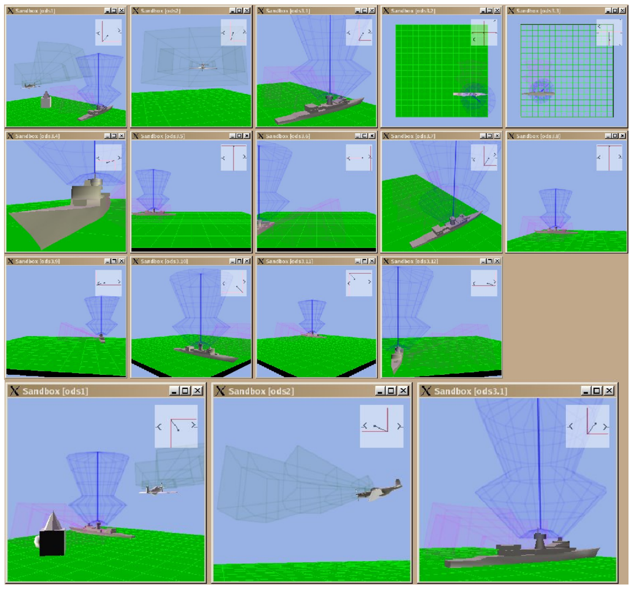

Figure 3.2: Three-Dimensional Perspectives

3.2.3 Logging

Visualization is informative for observing qualitative behavior in real time at runtime, or playing it back later, but more detailed quantitative postanalysis requires the underlying data. The logging system records over two dozen parameters for every event, which export directly to Excel.

3.3 Controller

The controller module of the architecture manages how the user defines and controls the model and simulation, and as well as how the simulation itself executes. This paper addresses only the first part.

3.3.1 Input

All user input (except view manipulation via the mouse) is entirely from the command line. This text form is very convenient for adding or modifying commands as programming exercises. The plug-and-play nature, based on the Command and Interpreter design patterns and implemented as a context-free grammar with JavaCC, also accommodates any input form that could produce the communication protocols that the controller processes [9]. This approach decouples the input from the processing in the same way that the output is decoupled from it, thereby reducing the temptation to hack solutions.

3.3.1.1 Creational and Structural Commands

Creational commands specify the agents in the model. They partition the process into the separate stages of first defining agent families, then declaring agent instances from them. In the object-oriented sense, the corresponding process is defining classes and then instantiating objects, which students often conflate into the same actions. By keeping these concerns separate, it becomes clear that definitions address data and control (potential for work), whereas declarations address behavior (actual work). As definitions may contain, or be contained by, other definitions, these commands are also considered structural [9].

In order of dependency, sensor and fuze families are defined first because they contain no agents. In the structural sense, they are leaves in a compositional tree. The typical form of these 14 commands is:

```
define sensor radar id with field of view fov
 power power sensitivity sensitivity
```

where the italicized fields translate directly into the datatypes in 3.1.2.

Munition families are defined next because they may contain sensor and fuze agents. The typical form of these 11 commands is:

```
define munition missile id1 with sensor id2
 fuze id3 arming distance distance
```

Finally, the actors are defined with munitions:

```
define ship id1 with munition[s] idn+
```

The declaration process is limited to two actions that either create an actor:

```
create actor id1 from id2 at coordinates with
 course course speed speed
```

where id_1 is the actor instance and id_2 is the actor family, or create a munition:

```
load munition id1 from id2
```

At this point, the munition instance is listed on the activity scoreboard as ready to fire. If it is a smart munition, its entry continuously updates its sensor state to determine whether a target is within its launch acceptance region for tracking.

The second action deploys the munition accordingly:

```
deploy munition id
deploy munition id at azimuth azimuth
 elevation elevation
```

where the latter variant is for shells fired from main guns.

3.3.1.2 Behavioral Commands

Behavioral commands instruct actors to assume a new state gradually according to their performance characteristics (e.g., acceleration, rate of turn or climb):

```
set id course course
set id speed speed
set id altitude altitude
set id depth depth
```

3.3.1.3 Miscellaneous Commands

Miscellaneous commands allow specialized control over the controller and model to facilitate repeatable experiments. They pause, resume, wait, and change the clock speed. They also load scripted commands of any type from text files and define maneuvers that may be executed on a parameterized agent; e.g.,

```
execute maneuver climb_left on my_fighter
```

where maneuver `climb_left` would have been defined earlier as a sequence of behavioral commands to climb and change course by 90 degrees counterclockwise.

Finally, it is possible to force any agent to assume coordinates, course, or speed instantaneously to establish test conditions outside the normal legal channels:

```
set id state at coordinates with course course
 speed speed
```

4 Preliminary Results

Ironically, this test-and-evaluation framework does not lend itself to convenient test and evaluation with its student subjects. It was not feasible to set up a controlled experiment that compared a test group to a control group because the entire class of 33 students had to do the same project the same way. As a result, these preliminary results are informal. They are based on anecdotal observation, eight individual assignments, 10 anonymous weekly assessments of course content, 16 project status reports (individual and team), a final project evaluation, and a course evaluation.

Although the project was already complete in advance, the students had to go through the analysis and design stages without access to it, as if they themselves were in charge of its outcome. Background research got everyone up to speed on the subject matter. It also provided valuable insight into their generally limited critical-thinking skills. For example, the entire class misinterpreted which horizontal direction on a map increasing longitude corresponds to in North America, despite several obvious opportunities to crosscheck their understanding. Many students expressed that it was a shocking reality check demonstrating the importance of connecting the dots and actually understanding what the connections mean. Such cases are so common that a related exercise with potential "gotchas" has become part of every project.

Students next had to apply the multidimensional slicing and dicing of the problem space from Section 2 to tease out relationships among the agents in a bottom-up manner. The first step entailed building the compatibility matrix in Table 4.1 to show which munitions may use which sensors and fuzes. A significant number of students misunderstood these constraints and acknowledged that they would have implemented an incorrect solution.

	Sensor							
Munition	Acoustic	Depth	Distance	Radar	Sonar, passive	Sonar, active	Thermal	Time
Bomb								
Depth Charge	✓	✓			✓	✓	✓	✓
Missile			✓	✓			✓	✓
Shell								✓
Torpedo	✓	✓	✓		✓	✓	✓	✓

Table 4.1: Compatibility Matrix

Similarly, the next level up entailed the applicability matrix in Table 4.2 to show which actors could deploy which munitions against other actors under which conditions (where *A* and *B* for submarines designate above and below water, respectively, and the other letters correspond to the munitions in Table 4.1).

	Target			
Source	Airplane	Ship	Submarine (A)	Submarine (B)
Airplane	M	B,M,T	B,T	D,T
Ship		M,S,T	S,T	D,T
Submarine (A)		M,T	T	T
Submarine (B)		T	T	T

Table 4.2: Applicability Matrix

From this point, students transitioned to the solution domain through traditional object-oriented analysis on composition, inheritance, and communication relationships expressed in UML. Their subsequent programmatic solutions were plug-in components to the architecture, which required them to learn how to understand and use it from an earlier analysis assignment. Their components replaced the existing ones in the provided solution.

Section 2.2 discussed the goals of testing that students' code works and then evaluating how well. The architecture itself facilitated the former because it kept their component solutions independent. Even this relatively small set of agents and properties would have led to a combinatorial explosion of testing requirements if students had been permitted to hack their solutions together with molecular-level coupling as usual. Most recognized the value of this partitioning in their low-level unit and functional (white and black-box) testing, mid-level compositional testing, and high-level integration testing [14].

Evaluating model performance was the final aspect of the project. It was a predominantly superficial exercise in design and execution because the purpose was to expose students to the process. They used the author's solution instead of their own for consistency. The task was to execute 18 required experiments, and then to select from several sets of options, with a rationale for the choices. Each of the 32 total experiments addressed a representative set of metrics of interest, such as which munition was most effective against which target type, or what the maximum effective range of a missile was. One set of options addressed sensitivity analysis to establish reasonably optimal low-level performance characteristics like sweep rate of the field of view over the field of regard on a missile radar sensor.

The project supported no explicit countermeasures, but certain tactical maneuvers were tested to try to outwit smart munitions by causing them to exceed their performance limitations. One test even had a torpedo acquire and destroy its own firing submarine. Students said they had a lot of fun.

4.1 Test Report

The final deliverable was a formal report describing the test plan and its results. Each experiment addressed eight points, where 1–4 related to planning, 5–6 to execution, and 7–8 to presenting the results:

1. The rationale behind the test; i.e., what it was testing and why it mattered.
2. A general English description of the initial conditions of the test.
3. The commands for (2).
4. An English narrative of the expected results of executing the test.
5. The actual results with at least one screenshot of the most representative view.

6. A snippet of the actual results from the log file with a supporting explanation, including statistics, metrics, and charts.
7. A brief discussion on how the actual results agreed with the expected results, or if they disagreed, a hypothesis of why.
8. A suggestion for how to extend this test to address related aspects of potential interest.

4.2 Examples

Point 5 was the most informative aspect of each discussion. The flexibility of the world viewer allowed the students to select the most representative perspectives to support the rest of their discussion. For example, Figure 4.1 illustrates the side view of dropping a bomb from an airplane flying to the right. Note the horizontal velocity imparted on the bomb.

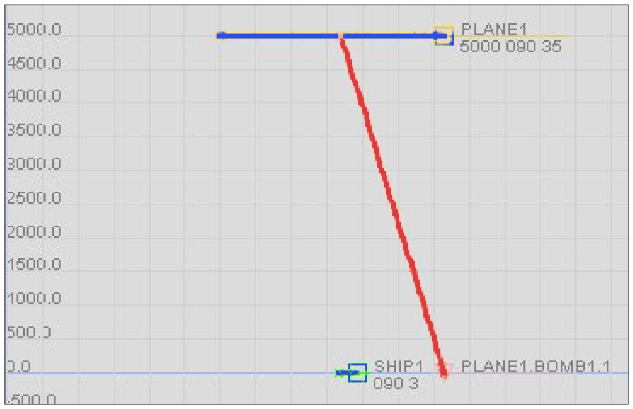

Figure 4.1: Bomb Release

Figure 4.2 presents a top view of two torpedoes fired from a submarine that then track a ship trying to outrun them.

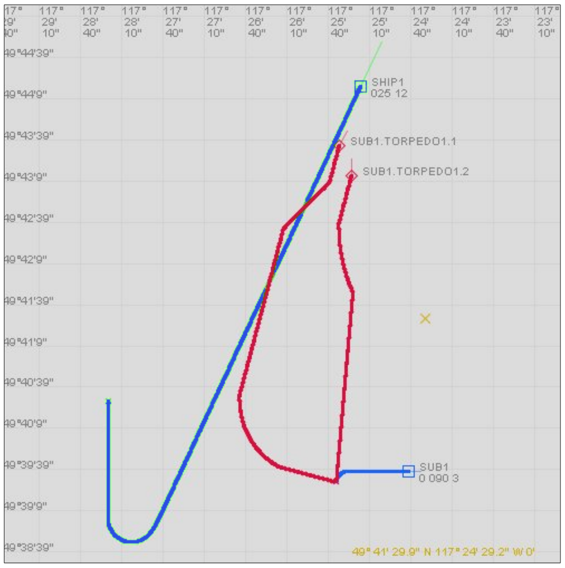

Figure 4.2: Torpedo Tracking

5 Future Work

This virtual test range accommodates a wide variety of plug-and-play components. At the low level, there are endless options for extending it to other actor platforms and weapon systems with different technologies. At the high level, more advanced strategic and tactical scenarios like acquisition, lethality, survivability, and engagement could be investigated.

Although designed for a stochastic methodology to determine ranges of performance experimentally, the current system does not take advantage of it. This technique could support rich sensitivity analysis to tease out countless aspects of component behaviors. It could also allow students to apply their knowledge of discrete mathematics, statistics, and probability, which all are required to study, but most mistakenly consider to be irrelevant to their degree. Practical application of these otherwise abstract concepts could enlighten their views on quantitatively demonstrating performance and showing confidence in the results.

6 Conclusion

Modeling, simulation, visualization, and analysis are inherently at the heart of most processes in software engineering and software quality assurance, yet these subjects are not traditionally taught together from a unified perspective. Not only did this work consolidate so many of their essential elements into a digestible package delivered over a fast-paced 10-week academic quarter, but the students overwhelmingly loved it. The average course evaluation was 4.7 out of 5 (outstanding). Furthermore, the many laudatory comments have since contributed to refining both this system and how the course is delivered.

References

[1] Sokoloski, J. and Banks, C.: Principles of Modeling and Simulation, Wiley, Hoboken, 2009.

[2] Denning, P.: "The Science in Computer Science" Communications of the ACM, Vol. 56, No. 5, May 2013.

[3] McConnell, S.: Code Complete: A Practical Handbook of Software Construction, Microsoft, Redmond, 2004.

[4] Sommerville, I.: Software Engineering, Addison-Wesley, Boston, 2011.

[5] Irish, R.: "Engineering Thinking: Using Benjamin Bloom and William Perry to Design Assignments" Language and Learning Across the Disciplines 3(2):83–102, 1999.

[6] Rowley, J.: "The wisdom hierarchy: representations of the DIKW hierarchy" Journal of Information Science 33(2):163–180, 2007.

[7] Bloom, B.: Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain. David McKay, New York, 1956.

[8] Zeigler, B., Praehofer, H., and Kim, T. G.: Theory of Modeling and Simulation, Academic Press, San Diego, 2000.

[9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Upper Saddle River, 1994.

[10] Okasaki, C.: Purely Functional Data Structures, Cambridge University Press, New York, 1999.

[11] Tappan, D.: "A Pedagogical Framework for Modeling and Simulating Intelligent Agents and Control Systems" Technical Report WS-08-02, AAAI Press, 2008.

[12] Tappan, D.: "A Pedagogy-Oriented Modeling-and-Simulation Environment for AI Scenarios" in Proc. of WorldComp International Conference on Artificial Intelligence, Las Vegas, NV, 2009.

[13] Tappan, D.: "Student-Friendly Java-Based Multiagent Event Handling" in Proc. of Association for the Advancement of Artificial Intelligence, Bellevue, WA, 2013.

[14] Pressman, R.: Software Engineering: A Practitioner's Approach, McGraw-Hill, 2009.

Author Biography

DAN TAPPAN is an Assistant Professor of Computer Science at Eastern Washington University. He has been a professor for nine years, before which he spent a decade as a defense contractor, mostly involved in the modeling and simulation of weapon systems at White Sands Missile Range and Aberdeen Proving Ground. His other main research areas are artificial intelligence, especially natural language processing, as well as intelligent systems, aviation, and STEM education.