Peer Reviewed Paper Presented at AlaSim 2016

A Meta-Case Study of Modeling, Simulation, Visualization, and Analysis for Real-World Software Systems Engineering Education

Dan Tappan
Department of Computer Science, Eastern Washington University, Cheney, WA
Keywords: software engineering, systems engineering, pedagogy

ABSTRACT: *The foundation of modern systems of systems is computer systems controlling electrical systems in turn controlling mechanical systems. Despite the core role computers play, computer science students do not generally see or appreciate this perspective because few classroom projects demonstrate it. This work showcases eight recent projects that employ a systems-engineering approach to teaching software engineering. Specifically, it shows how modeling, simulation, visualization, and analysis serve as a powerful toolkit for the analysis, design, implementation, testing, and evaluation of engaging real-world projects related to aviation, military, construction, and railroad applications.*

1. Introduction

Modern technology is a complex system of systems composed of mechanical systems controlled by electrical systems controlled by software systems. Software engineering is not just about software anymore. The systems-engineering processes of analysis, design, implementation, testing, evaluation, verification, validation, and accreditation demand far more than the typical classroom environment can address. This paper presents an overview of a highly successful reusable Java-based software architecture and corresponding holistic pedagogical approach that utilize modeling, simulation, visualization, and analysis at all levels with an overarching focus on software quality assurance. It uses multiagent continuous time-stepped simulations for interactive virtual worlds that capture a vast breadth and depth of multidimensional exposure to realistic systems while still being manageable for students and the instructor. This overview highlights commonalities and results from a survey of eight recent projects:

- **AAR**: aircraft accident reenactment environment for creating, recreating, and analyzing events
- **ACO**: aircraft carrier operations with fighters taking off, landing, and repositioning, and refueling from tankers
- **ATC**: air traffic control with airplanes operating on the ground and in the air in various airspace configurations and contexts
- **FBW**: fly-by-wire control system with networked control surfaces and external components of an airplane on a test stand
- **HCE**: heavy construction equipment toolkit with sensors and electrical, mechanical, hydraulic, and pneumatic actuators
- **MTR**: military test range with airplanes, ships, and submarines using sensors and weapons
- **RLM**: railroad layout manager with tracks, cars, engines, and signaling and safety systems
- **UAV**: unmanned aerial vehicle remote cockpit with instrumentation and flight data recording

This approach forces students to develop and apply critical-thinking and technical-communication skills by pushing them out of their comfort zone into overwhelmingly unfamiliar real-world environments. It helps establish the endless dots and their interconnections and interrelationships to learn about the problem domain of the subject matter, to translate it into the solution domain, and to evaluate the results. Modeling and simulation here uses software as a surrogate for the real world to investigate what-if scenarios from countless perspectives. It dovetails with the scientific method as a disciplined approach for envisioning, building, and conducting repeatable controlled experiments in support of developing quality software systems of systems. Finally, it emphasizes an array of underutilized visualization techniques as an expressive yet intuitive means of conveying information to all stakeholders in the development process.

2. Software Systems Engineering

The term *software systems engineering* as a combination of *software engineering* and *systems engineering* is not mainstream yet. In fact, it produces only 340 thousand hits on Google versus 34 and 16 million for the other two, respectively. However, despite the lack of terminological recognition, the fusion of these fields is indeed how professionals develop complex systems of systems. Although the students referenced in this paper are studying computer science as their major discipline, they cannot be completely oblivious to the central role that it plays in the larger world where they plan to spend their careers. The multidisciplinary nature of these projects

fosters an understanding and appreciation of such a holistic perspective.

2.1 Software Engineering

Software engineering is a complex, multidimensional, multifaceted process. There are countless ways to conduct it. This paper considers the following traditional stages: *analysis* is the process of understanding the problem domain; *design* is mapping the many real-world elements of the analysis to the corresponding virtual-world elements of the solution domain; *implementation* is building the solution with appropriate tools and techniques; and finally, *testing and evaluation* is demonstrating that the solution works and is consistent with the original problem, as well as refining and optimizing it.

In reality, the process invariably ends up looking like a variant of the popular joke in Figure 2.1, which apparently has been floating around the public domain since the advent of software engineering. The panes correspond from top left to bottom right as: *how the customer explained it*; *how the project leader understood it*; *how the analyst designed it*; *how the programmer wrote it*; *what the beta testers received*; *how the business consultant described it*; *how the project was documented*; *what operations installed*; *how the customer was billed*; *how it was supported*; *what marketing advertised*; *and finally what the customer really needed*.

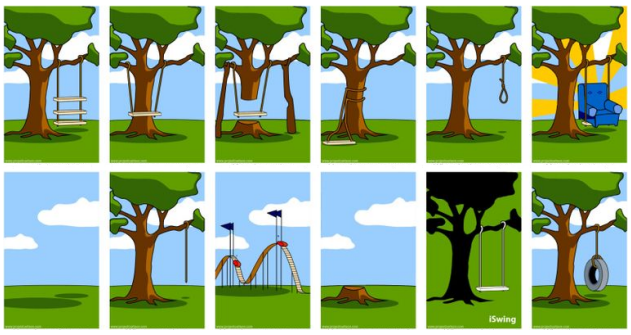


Figure 2.1: Software Engineering in Practice

The particulars of each pane are irrelevant. What really matters is the larger perspective that every manner of absurdity happens from one step to the next. While there are unquestionably many genuinely unavoidable pitfalls in software engineering, many need not become problems with some reasonable care. The approach throughout these projects aims to reduce the endless disconnects as students translate from one stage to the next. Far too often their "strategy" is to try anything that comes to mind with the hope that it works. In fact, one student blatantly admitted that he "kept throwing more code at the compiler until it shut up." In the world of physical engineering, developing what-if mockups and prototypes is commonplace and extremely useful, but because of the investment in actually building something, engineers put more thought into the design. In the virtual world of programming, students develop the bad habit of believing that haphazard trial and error is an actual strategy to problem-solving because it appears to come at no cost. In reality, they often do not know why their solutions fail to work, or if the solutions do actually work, they cannot articulate why. Neither perspective is conducive to producing quality software.

2.2 Systems Engineering

Systems engineering is a superset of software engineering that involves a vast array of systems of systems of all types. While systems engineering often tends to be a higher-level, more managerial and less technical perspective, this work focuses on the engineering aspects of multidisciplinary systems. In fact, the breadth and depth of subject matter throughout these projects aligns quite well with *mechatronics*, which is an amalgamation of at least the disciplines represented in Figure 2.2.

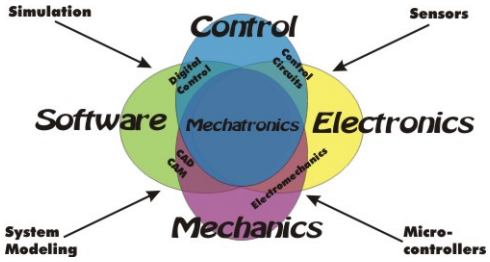


Figure 2.2: Systems Engineering Convergence [1]

Computer science students are naturally not expected to have a background in all of these areas. These projects, especially in the analysis stage, offer many opportunities for students to familiarize themselves with the subject matter to the degree necessary to do something computational with it. This approach provides good training because in the working world, professionals are always being immersed into unfamiliar environments. The ability to adapt and learn quickly is essential.

3. Pedagogical Foundation

The pedagogical foundation is extensive and covered in great detail in [2]. The goal here is to provide just an overview of how modeling and simulation relate to thinking and doing for software systems engineering.

3.1 Modeling

Modeling can be considered the process of translating a problem in the real physical world to a solution in the virtual computer world, as depicted by the right arrow in

Figure 3.1. By and large, students do understand this direction because they are accustomed to receiving problems to solve. What they rarely recognize is the inverse direction depicted by the left arrow. In this case, if their solution were given to someone with no knowledge of the original problem, it is very unlikely that this person would be able to recreate it correctly. The reason relates to the cartoon in Figure 2.1, which reflects an endless parade of translation errors where important details are lost or mangled, and new unfounded ones mysteriously appear. The result is poor software, which Weinberg [3] characterizes eloquently: "If builders built buildings the way programmers write programs, then the first woodpecker that came along would destroy civilization."

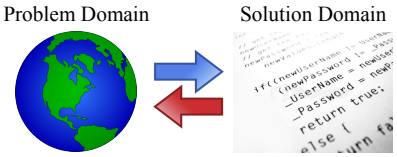


Figure 3.1: Real to Virtual-World Correspondence

The pedagogical emphasis in this paper is on how to teach students to translate the problem domain to the solution domain appropriately and to verify the translation. Section 5.1 covers this process in much more detail. Here it suffices to define the approach as "slicing and dicing" the problem domain into increasingly smaller pieces that ultimately have clear translations, such as in Figure 3.2. In particular, students need to be able to articulate what they want, how to get it, and how to know that they got it. These steps correspond generally to analysis, design, and testing, respectively, but for small, bite-sized pieces that are easier to understand and process. They also capture both directions in Figure 3.1. One of the simplest approaches plays a commanding role: posing and getting resolution on any number of *who*, *what*, *when*, *where*, *why*, and *how* (W5H) questions, which form the backbone of mental models for understanding anything in the world [4,5].

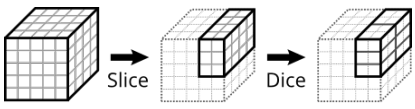


Figure 3.2: Domain Decomposition [6]

Establishing small pieces helps combine them meaningfully into ever-larger ones, which ultimately leads to systems of systems. Figure 3.3 shows the data-information-knowledge-wisdom (DIKW) hierarchy, which helps guide this process by establishing these pieces as dots and providing a framework for connecting them appropriately [7]. This process reflects learning by accumulating experience.

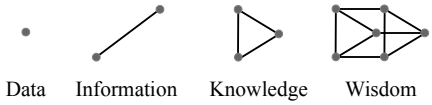


Figure 3.3: DIKW Hierarchy

- Data: raw entities with no context
- Information: entities in one context
- Knowledge: entities in multiple contexts
- Wisdom: generalized principles created by connecting a network of contexts from different sources for predictive, anticipatory, proactive understanding

Finally, Bloom's Taxonomy of Educational Objectives plays the overarching role of helping foster critical thinking by leading students upward from the low-level, data-oriented learning activities of *remembering*, *understanding*, and *applying* to the high-level, knowledge-oriented activities of *analyzing*, *creating*, and *evaluating* [8]. In many respects, this flow also corresponds to analysis, design, implementation, and testing in software development.

3.2 Simulation

The role of simulation in these projects is two-fold. First, it makes them interesting, which helps entice students to take the process of learning to develop them seriously. Second, it provides a disciplined way of evaluating whether their solutions work correctly, and if so, then how well. The basis is the scientific method, which is common to all sciences except ironically computer science [9,10]. Figure 3.4 captures the typical process flow, which for software development tends to reflect the following steps:

- Determine what needs to be tested.
- Define an appropriate test.
- Run controlled experiments.
- Collect and interpret results.
- Report on whether the test passed. If not, make proposed corrections to the program and run the same test again. If so, refine the program to make the results better until meeting a specified level of performance.

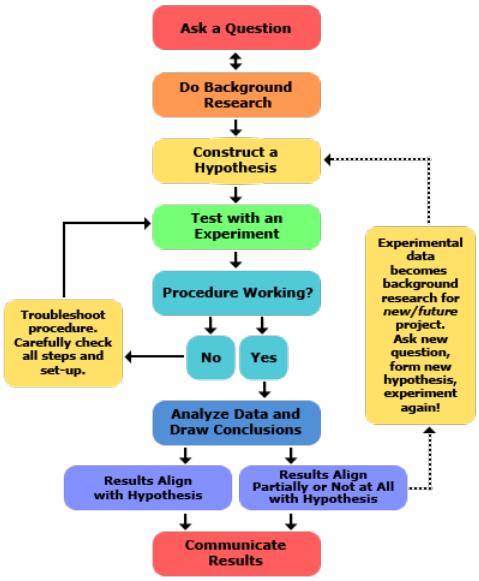


Figure 3.4: Scientific Method [11]

A third use of simulation is the traditional purpose for developing such software: to evaluate what-if scenarios about the problem domain by using the software as a surrogate. This use plays only a minor role here, primarily for demonstration and discussion, because the students are not studying to become subject-matter experts.

4. Project Showcase

Each project investigates a rich breadth and depth of aspects that exercise important elements of software engineering. The overview here, however, is of the general characteristics of each.

4.1 Unmanned Aerial Vehicle

Project UAV involved the architecture for interacting with the flight dynamics model of an unmanned aerial vehicle, as well as receiving and interpreting navigation information from ground stations [12]. It also involved implementing parts of the instrumentation in Figure 4.1 to present the results of this processing to the pilot.

Figure 4.1: UAV Viewer

4.2 Air Traffic Control

Project ATC involved a large-scale world of arbitrary aircraft, navigation systems, airports, and airspace under the command of various air traffic controllers [13]. Figure 4.2 depicts the respective views of ground, approach, and enroute controllers, each with a different perspective on the same world and different goals and procedures.

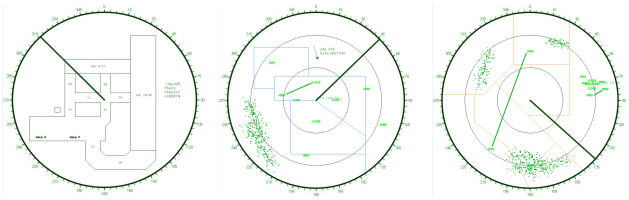


Figure 4.2: ATC Controller Viewers

The same underlying display accommodated all variants. Figure 4.3 shows a composite view with almost every option enabled simultaneously. A hallmark of good software design is being able to apply the same solution to many related problems without undue effort [14].

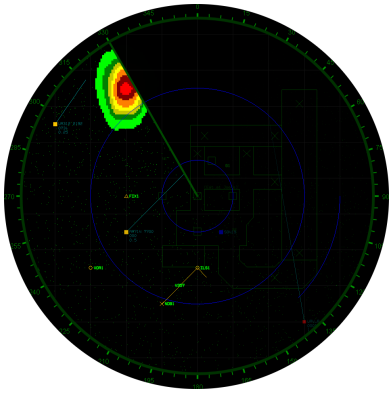


Figure 4.3: ATC Viewer, Composite

4.3 Fly-by-Wire Aircraft Control

Project FBW involved a hierarchical network of networks that coordinated controllers, sensors, and actuators on a fly-by-wire aircraft on a test stand [15]. Figure 4.4 depicts the flight control surfaces and other components like engines and landing gear, which had very specific behaviors that had to be ensured. (See Figures 5.7 and 5.8.)

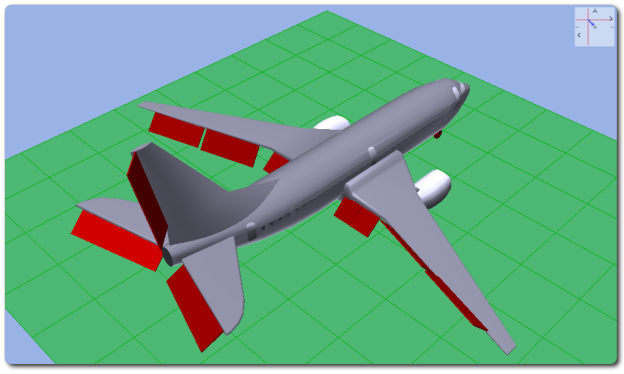


Figure 4.4: FBW Viewer

Figure 4.5 depicts the corresponding fly-by-wire network, which the architecture utilized through rich communication protocols.

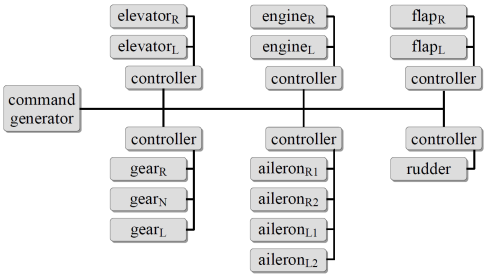


Figure 4.5: Fly-By-Wire Network Architecture

4.4 Aircraft Accident Reenactment

Project AAR involves a combination of projects UAV, ATC, and FBW to define, execute, and analyze a wide variety of failures that lead to aircraft accidents. Figure 4.6 depicts a mockup of the expected final form, which is still under development.

Figure 4.6: AAR Viewer Mockup [16]

4.5 Aircraft Carrier Operations

Project ACO involved a very dynamic environment for aircraft carrier operations [2]. It included fighters on board and in the air. Carriers maintained components like catapults, blast barriers, arresting wires, and optical landing systems. The fighters interacted with them and airborne tankers to carry out simple training missions by taking off, refueling, and landing. Figure 4.7 depicts top, side, and front views of a launch.

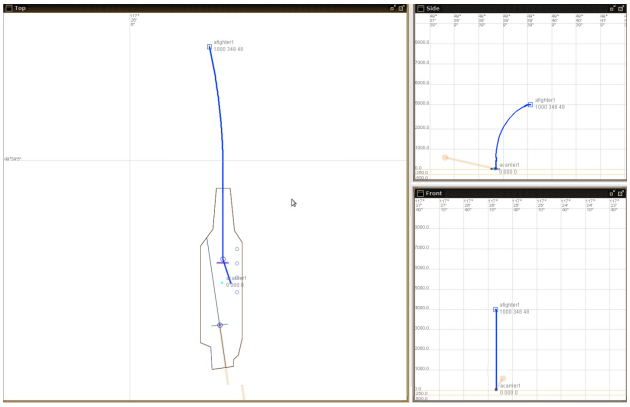


Figure 4.7: ACO Viewer

4.6 Military Test Range

Project MTR involved the most complex dynamic world [17]. It provided an evaluation environment for a wide variety of weapon systems on different platforms. Munitions supported the specific combinations of sensors and fuzes in Table 4.1.

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

Table 4.1: Compatibility Matrix

Similarly, Table 4.2 shows which platforms could engage each other with which munitions. (Submarines A and B are above and below water, respectively. The other letters correspond to the first letter of each munition.)

Source	Target: 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

Acquisition, lethality, engagement, countermeasures, and other considerations played out in a two-dimensional top-view world, as in Figure 4.8.

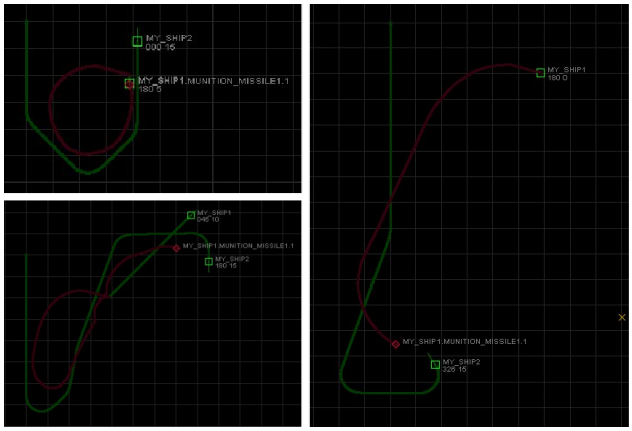


Figure 4.8: MTR Viewer, 2D Perspective

Some of the output, as in Figure 4.9, naturally exported to the three-dimensional visualizer that Section 5.3 covers.

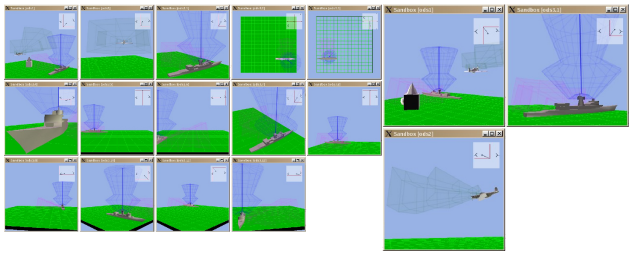


Figure 4.9: MTR Viewer, 3D Perspective

4.7 Heavy Construction Equipment

Project HCE involved the design and evaluation of heavy construction equipment. Despite major differences in appearance, as in Figure 4.10, the underlying model is quite similar to the fly-by-wire architecture in project FBW. Here, however, the actuators are electrical, mechanical, hydraulic, and pneumatic cylinders that connect fixed and variable linkages and free-body components. As with FBW, the equipment resides on a virtual test stand and does not actually perform any function in the world.

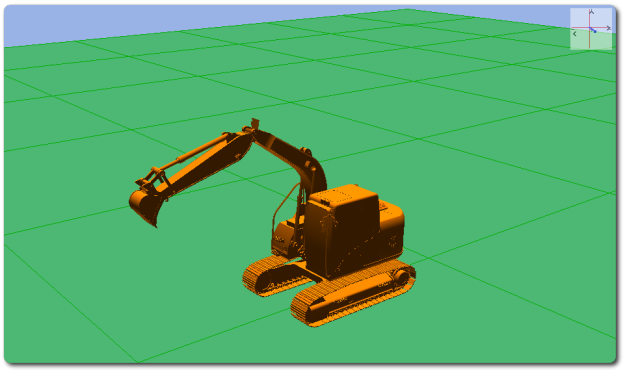


Figure 4.10: HCE Viewer

4.8 Railroad Layout Manager

Project RLM involved a railroad layout manager that captured the usual components like tracks, engines, and cars. It also supported complex signaling and safety systems. The viewer in Figure 4.11 is characteristic of many projects, which present the world from an iconified top view. Although the graphics are expressive, they are not particularly attractive. However, the architecture of these projects accommodates improvements to the model, view, and controller concerns (see Section 5.2.1) relatively independently, which is another hallmark of good software design [18].

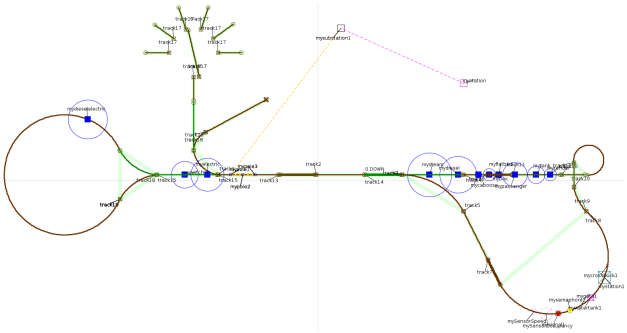


Figure 4.11: RLM Viewer

5. Architectural Framework

The architectural framework contains the elements for modeling, simulation, visualization, and analysis. For the most part, only the model and parts of the visualization differ substantially among projects.

5.1 Modeling

The first step in making sense of any project is to establish what its pieces are and consist of. The term

agent generally applies to top-level entities of study like airplanes, whereas a *component* is part of one, such as landing gear. However, from many perspectives, there is no practical difference, so the latter term is used here. Table 5.1 provides examples of both categories from each project as defined in Section I.

Project	Example
AAR	airplane, cockpit control, flight control surface, data logger
ACO	fighter, tanker, catapult, arresting gear, refueling boom
ATC	aircraft, taxiway, runway, airspace geometry, navigation aids
FBW	elevator, aileron, rudder, flap, slat, landing gear, engine
HCE	chassis, frame, linkage, joint, lever, hydraulic cylinder
MTR	airplane, ship, submarine, sensor, fuze, missile, bomb
RLM	track, switch, engine, car, sensor, gate, semaphore, signal light
UAV	airplane, instrument, navigational transmitter and receiver

Table 5.1: Components

5.1.1 Data

The next step is to define each component in terms of three aspects. The first is *data*, which captures what a component is. For example, Table 5.2 specifies representative characteristics of a component from Table 5.1. A model is always an abstraction, so not every detail is captured. Determining what to include, as well as how to represent it, is part of the model-based thinking that students need to learn [19].

Project	Example
AAR	an airplane has a callsign, latitude, longitude, and altitude
ACO	a catapult has an acceleration rate to maximum speed
ATC	an aircraft has an (x,y,z) position and a direction at a speed
FBW	a rudder has a maximum positive/negative deflection angle
HCE	a hydraulic cylinder has a minimum and maximum extension
MTR	a radar sensor has a maximum range and sensitivity
RLM	an engine has a current and maximum speed
UAV	an airplane has a yaw, pitch, and roll attitude

Table 5.2: Data

Students tend to experience surprising difficulty in representing real-world data. The emphasis in these projects in primarily on breadth, not depth, so it is an inappropriate use of time to expect students to implement complex representations themselves. Therefore, the author provides most as predefined datatypes, such as in Figure 5.1. Each captures the practical essence of its abstracted role in the project. Most are simplifications, such as a flat-earth model for latitude and longitude. Each manages its units and magnitudes and provides error checking, utility methods, logging, and other useful features.

```
Acceleration, Altitude, AngleMath, AngleNav,
Attitude, AttitudePitch, AttitudeRoll,
AttitudeYaw, Azimuth, Bearing, Callsign,
CoordCartAbsolute, CoordCartRelative,
CoordPolarMath, CoordPolarNav, CoordPolarNav3D,
CoordWorld, CoordWorld3D, Course, Distance,
Drag, Elevation, FieldOfView, FieldOfRegard,
Heading, Identifier, Interval, Latitude, Lift,
Longitude, Percent, Power, Range, Rate, Speed,
Time, Thrust, Track, Vector, Velocity, Weight
```

Figure 5.1: Datatypes

5.1.2 Control

The second aspect to define for each component is its *control*, which captures what it can do. These capabilities must be consistent with the use of the component, as in Table 5.3. They must also be consistent with the data because control operates on data to produce more data. This input-processing-output model is the basis of all computing, yet students' solutions frequently have disconnects with control operating on nonexistent or incorrect data, or with data having no corresponding control. The relationships between data and control must be clearly established before proceeding.

Project	Example
AAR	increase the altitude of the airplane
ACO	activate the catapult as configured
ATC	instruct to change the direction of the aircraft
FBW	set the target deflection angle
HCE	set the cylinder target extension distance
MTR	transmit a radar pulse
RLM	set the target engine speed
UAV	set the attitude components

Table 5.3: Control

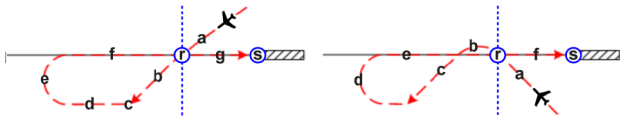
5.1.3 Behavior

Data and control are static in that they define the existence and capabilities of components. *Behavior*, on the other hand, is dynamic because it specifies how components function with respect to an operational context. For example, each action in Table 5.4 has a purpose. It translates to the control level to manipulate the data level. All levels must be bidirectionally consistent.

Project	Example
AAR	climb to avoid terrain
ACO	launch a fighter to defend the carrier
ATC	change aircraft direction to avoid conflicting traffic
FBW	deflect the rudder left to coordinate a turn
HCE	extend a cylinder to dump the load bucket
MTR	ping a target with radar to lock a missile
RLM	reduce the engine speed to arrive at a station
UAV	change the attitude to execute a landing maneuver

Table 5.4: Behavior

Figure 5.2 demonstrates two extended examples. In both cases, the goal is to align an airplane with the runway at point *S*. Depending on the arrival position and direction, the actions to carry out differ. The top-level goal decomposes into the lower-level steps *a–g* or *a–f* that reference the corresponding control.



a fly to initial approach fix ***r***
b fly for 45 seconds
c turn right to runway reciprocal
d fly for 45 seconds
e turn right to landing course
f fly to final approach fix ***r***
g fly to ***s***

a fly to initial approach fix ***r***
b turn left to 135 degrees from landing course
c fly for 60 seconds
d turn right to landing course
e fly to final approach fix ***r***
f fly to ***s***

Figure 5.2: Landing Approaches

5.1.4 Decompositional Characteristics

Establishing other characteristics of components does not exhibit the same ordered road map. Instead, it is up to the students to decide what is relevant, as well as how and why, and then to act accordingly on these decisions. This section covers three of the main breakouts.

Table 5.5 distinguishes between *independent* and *dependent* components. They generally align with the definition of top-level agents versus lower-level components, respectively, but often the world is not so clear. For example, an airplane in project FBW is independent because it operates on its own, whereas its landing gear is always dependent on it because this component would never be found separate from an airplane. Similarly, in MTR, a (fire-and-forget) missile in flight is on its own, but before launch, it was dependent on the fighter carrying it. Finally, a fighter aboard a carrier in ACO is initially dependent when parked. It becomes both independent and dependent while taxiing. Upon takeoff it becomes independent until landing. Students must recognize and understand such dynamic complexities in order to manage them properly.

Project	Independent	Dependent
AAR	airplane	flight data recorder
ACO	carrier, tanker	catapult, arresting wire
ATC	airplane, airport	radar station (in a network)
FBW	airplane	landing gear, engine
HCE	chassis	bulldozer blade
MTR	missile in flight	missile on fighter
RLM	track layout	engine and rail car
UAV	airplane	flight control surface

Table 5.5: Independent/Dependent Components

Table 5.6 distinguishes between *static* and *dynamic* components, which are generally those that do not change and those that do, respectively. In project FBW, for example, the wing is merely an attachment point. It has data defining its shape, but no control that allows it to do anything with the data. On the other hand, the landing gear can extend and retract, which changes its state over a time interval.

Project	Static	Dynamic
AAR	airport layout, terrain	air traffic, weather
ACO	parking area, taxiway	refueling booms, tailhook
ATC	taxiway, runway, airspace	airplane, weather pattern
FBW	fuselage, wing	landing gear, engine
HCE	chassis, support	linkage, actuator
MTR	sea floor and surface	bomb, missile, torpedo
RLM	straight and curved track	switch track, drawbridge
UAV	instrument panel background	instrument needle

Table 5.6: Static/Dynamic Components

Dynamic components can change state in different ways. Table 5.7 distinguishes between *discrete* events, which happen instantaneously, and *continuous* ones, which are relatively smooth transitions. In project FBW, switching the landing light on or off is instantaneous, whereas the landing gear takes time to change state. Students are familiar with discrete events because ordinary programming operates this way: calling a method executes it immediately, and the program does not proceed until execution is complete. Continuous events, on the other hand, are much more difficult to manage, especially in a controlled way for simulation purposes.

Project	Discrete	Continuous
AAR	aircraft responds to radio call	aircraft descends to altitude
ACO	fighter reports position	carrier changes direction
ATC	engines start	airplane taxis to runway
FBW	landing light illuminates	landing gear retracts
HCE	hydraulic pump activates	hydraulic cylinder extends
MTR	radar pulse propagates	torpedo tracks target
RLM	switch track changes	drawbridge goes up
UAV	navigation aid turns on	aircraft accelerates in a dive

Table 5.7: Discrete/Continuous Components

5.2 Simulation

Simulation is the realization of the operational context of behavior in Section 5.1.3 with respect to the scientific method in Section 3.2. It involves setting up and running controlled experiments and collecting results for visualization and analysis.

5.2.1 Simulation Framework

The simulation framework is based on a traditional model-view-controller architecture. This model aligns closely with the simulation model in Section 5.1, and the view aligns with visualization in Section 5.3. The controller plays two roles: to interact with the user and to execute the simulation.

5.2.2 Domain-Specific Languages

All interaction with the user (except for simple mouse manipulation of the views) is through text-based commands, which can be typed directly from a command line or read from a file. Each project has its own application-specific language, as in Figure 5.3, which plays three distinct roles based on well-established software design patterns [20].

```
/* load "/home/author/workspace/MultiagentTestbed/tests/test5.mat" */

define sensor radar        FUZE_RADAR1    with field of view 30 power 50 sensitivity 10
define sensor thermal      FUZE_THERMAL1  with field of view 30 sensitivity 1
define sensor sonar active FUZE_SONAR1    with power 20 sensitivity 2
define sensor depth        FUZE_DEPTH1    with trigger depth -250
define sensor time         FUZE_TIME1     with trigger time 5
define sensor distance     FUZE_DISTANCE1 with trigger distance 1
define sensor acoustic     FUZE_ACOUSTIC1 with sensitivity 7

define munition depth_charge MUNITION_DEPTHCHARGE1 with fuze FUZE_ACOUSTIC1
define munition missile      MUNITION_MISSILE1     with sensor FUZE_RADAR1    fuze FUZE_ACOUSTIC1
define munition torpedo      MUNITION_TORPEDO1     with sensor FUZE_ACOUSTIC1 fuze FUZE_ACOUSTIC1

define ship      ACTOR_SHIP1      with munition (MUNITION_MISSILE1 MUNITION_DEPTHCHARGE1)
define submarine ACTOR_SUBMARINE1 with munition (MUNITION_TORPEDO1)

create actor MY_SHIP1      from ACTOR_SHIP1      at 49*39'31#/117*25'34#/0 with course 270 speed 0
create actor MY_SUBMARINE1 from ACTOR_SUBMARINE1 at 49*39'30#/117*25'05#/-1000 with course 270 speed 5

set MY_SHIP1 load munition MUNITION_DEPTHCHARGE1
set MY_SUBMARINE1 load munition MUNITION_TORPEDO1

set MY_SHIP1 deploy munition MY_SHIP1.MUNITION_DEPTHCHARGE1.1
```

Figure 5.3: Script Snippet

Creational commands play the role of defining separate components at their lowest levels. The commands are highly specific to the projects, but all are of the same basic form:

```
CREATE something WITH arguments
```

For example, project MTR uses the following commands to create two sensors as radar and depth fuzes with certain characteristics:

```
DEFINE SENSOR RADAR fuze_radar1
  WITH FIELD OF VIEW 30 POWER 50 SENSITIVITY 10
DEFINE SENSOR DEPTH fuze_depth1
  WITH TRIGGER DEPTH -250
```

Structural commands combine the separate components into higher-level components or top-level agents. For example, the following command creates and assembles a missile with a previously created radar sensor and proximity fuze, plus it defines additional characteristics like a minimum flyout distance before arming:

```
DEFINE MUNITION MISSILE munition_mission1
  WITH SENSOR sensor_radar1
  FUZE fuze_proximity1 ARMING DISTANCE 0.5
```

Behavioral commands control the behavior of components. For example, the following commands change the course of a fighter, make it descend, and arm and fire its missile:

```
DO fighter1 CHANGE COURSE 315 DESCEND D-900
DO fighter1 ARM missile1
DO missile1 FIRE
```

Miscellaneous and *metacommands* control the simulation itself. For example, the following commands change the granularity of the simulation time steps and their correspondence to wall-clock time, wait 500 milliseconds, and then exit.

```
@CLOCK 100 20
@WAIT 500
@EXIT
```

One of the most useful metacommands is `@RUN`, which reads commands from a file as a script. This capability is extremely powerful for disciplined testing and evaluation because it allows students to partition separate tests into separate files. Instead of the usual approach of manipulating their programs directly to set up and execute tests and save the results, in which they generally undo or corrupt previous tests, here everything remains independent and more organized. Complex testing often involves executing different behaviors on the same initial configuration, which is easy to set up by having files call other files. This approach instills a lot of discipline in students, who would otherwise have no other practical way of performing such actions.

5.2.3 Simulation Implementation

The architecture manages a continuous time-stepped simulation. It maintains a collection of all components

and periodically updates each according to a single system clock such that every component performs its relevant actions at that instant. For example, extending landing gear in project FBW involves repeated updates to the components of the gear assembly, each of which advances the extension by a small amount. The end effect is the impression of continuous, smooth movement over time from the start of the physical interval (retracted) to the end (extended).

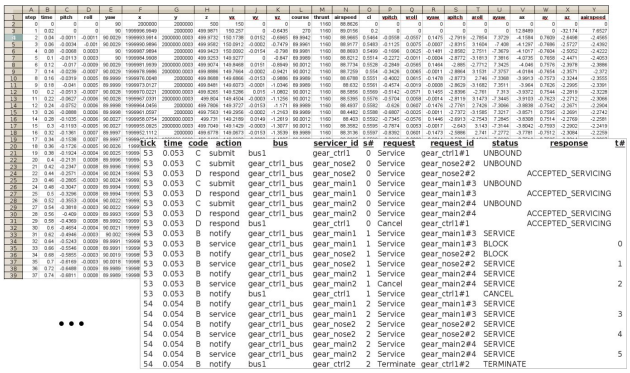
5.3 Visualization

Visualization involves far more than just graphics. It is a means of presenting complex data and information in ways that convey the content, structure, and meaning intuitively. No single way captures all aspects. At the lowest level, the compositional nature of agents and components lends itself to text output that indicates the creational and structural elements, such as for a notional fighter jet in Figure 5.4. Generating such output from an object-oriented program is straightforward and convenient. Moreover, doing so with a common data interchange format like XML results in a great benefit because other tools can do the tedious cosmetic work, thereby freeing students to focus on more appropriate tasks. For example, the Google Chrome web browser manages the indentation and color-coding here.

```
▼<component type="fighter" id="f1">
  ▼<performance>
    ▼<speeds>
       <min value="160"/>
       <max value="900"/>
       <acceleration base="32.7" coefficient="1.42" limit="18.3784"/>
     </speeds>
   </performance>
  ▼<sensors>
     <radar-passive id="r1" band="k" range="94.0" attenuation="37"/>
   </sensors>
  ▼<weapons>
     <missile type="sidewinder" id="m1" definition="@sidewinder{2,7}"/>
     <missile type="sidewinder" id="m2" definition="@sidewinder{2,7}"/>
   </weapons>
 </component>
```

Figure 5.4: XML Representation

Similarly, the value of Microsoft Excel is underappreciated as a legitimate and surprisingly powerful visualization tool. The architecture of these simulations automatically generates a wealth of low-level data about the states of the components and their intercommunication, as in Figure 5.5. Almost everything that occurs is captured somewhere in a structured text log file that by design exports effortlessly to Excel.



tick	time	code	action	bus	servicer_id	s#	request	request_id	status	response	t#
53	0.053	C	submit	bus1	gear_ctrl1	0	Service	gear_ctrl1#1	UNBOUND		
53	0.053	C	submit	gear_ctrl1_bus	gear_nose2	0	Service	gear_nose2#2	UNBOUND		
53	0.053	D	respond	gear_ctrl1_bus	gear_nose2	0	Service	gear_nose2#2		ACCEPTED_SERVICING	
53	0.053	C	submit	gear_ctrl1_bus	gear_main1	0	Service	gear_main1#3	UNBOUND		
53	0.053	D	respond	gear_ctrl1_bus	gear_main1	0	Service	gear_main1#3		ACCEPTED_SERVICING	
53	0.053	C	submit	gear_ctrl1_bus	gear_main2	0	Service	gear_main2#4	UNBOUND		
53	0.053	D	respond	gear_ctrl1_bus	gear_main2	0	Service	gear_main2#4		ACCEPTED_SERVICING	
53	0.053	D	respond	bus1	gear_ctrl1	0	Cancel	gear_ctrl1#1		ACCEPTED_SERVICING	
53	0.053	B	notify	gear_ctrl1_bus	gear_main1	1	Service	gear_main1#3	SERVICE		
53	0.053	B	service	gear_ctrl1_bus	gear_main1	1	Service	gear_main1#3	BLOCK		0
53	0.053	B	notify	gear_ctrl1_bus	gear_nose2	1	Service	gear_nose2#2	BLOCK		
53	0.053	B	service	gear_ctrl1_bus	gear_nose2	1	Service	gear_nose2#2	SERVICE		1
53	0.053	B	notify	gear_ctrl1_bus	gear_main2	1	Service	gear_main2#4	SERVICE		
53	0.053	B	service	gear_ctrl1_bus	gear_main2	1	Cancel	gear_main2#4	SERVICE		2
53	0.053	B	notify	bus1	gear_ctrl1	1	Service	gear_ctrl1#1	CANCEL		
54	0.054	B	notify	gear_ctrl1_bus	gear_main1	2	Service	gear_main1#3	SERVICE		
54	0.054	B	service	gear_ctrl1_bus	gear_main1	2	Service	gear_main1#3	SERVICE		3
54	0.054	B	notify	gear_ctrl1_bus	gear_nose2	2	Service	gear_nose2#2	SERVICE		
54	0.054	B	service	gear_ctrl1_bus	gear_nose2	2	Service	gear_nose2#2	SERVICE		4
54	0.054	B	notify	gear_ctrl1_bus	gear_main2	2	Service	gear_main2#4	SERVICE		
54	0.054	B	service	gear_ctrl1_bus	gear_main2	2	Service	gear_main2#4	SERVICE		5
54	0.054	B	notify	bus1	gear_ctrl2	2	Terminate	gear_ctrl1#2	TERMINATE		

Figure 5.5: Excel Table Representations

While this presentation contains copious raw data, it is not at all intuitive. Nobody can just look at the endless rows and columns and truly see the big picture of what is happening. However, still within Excel, judicious selection of data fields easily generates a wealth of graphs, such as in Figure 5.6, that convey information about relationships, especially causes and effects. The eye is naturally drawn to the visual form, and the brain sees patterns. Anomalies and discontinuities are far more apparent. Furthermore, this form can directly contribute to test reports as a concise depiction wrapped by brief English text for context. This approach greatly reduces the effort of writing. Students do not generally consider communication to be a significant part of computer science, but in the real world, it is actually what professionals often do the most.

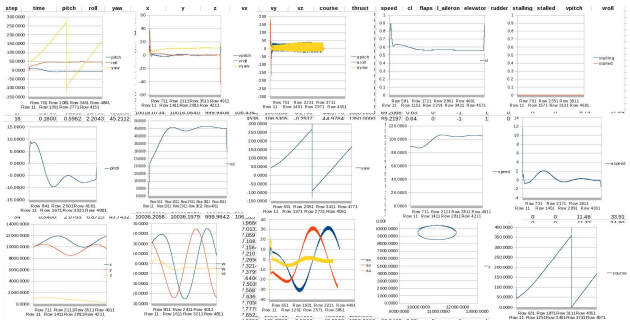


Figure 5.6: Excel Graph Representations

For more complex interactions, especially for precisely timed tests, manual annotation is worth the proverbial thousand words. For example, Figure 5.7 depicts the actions of a rudder actuator from project FBW from two perspectives at the following key time points:

1. at initial position 0º neutral; command to 45º left
2. arrives; command to 45º right
3. arrives; command to 0º
4. arrives; command to 30º left
5. at 15º left preemptively command to 45º right
6. arrives

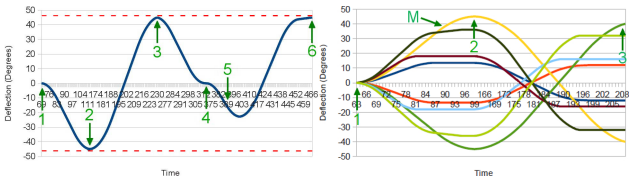


Figure 5.7: Annotated Events

Lower-level analysis using basic calculus computed within Excel produces the velocity and acceleration breakout in Figure 5.8.

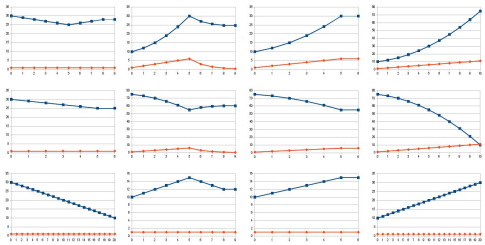


Figure 5.8: Excel Graph Representations

As many components change position within a two or three-dimensional world, plotting their tracks in freely available Gnuplot over time produces a rich perspective on their behavior. For example, the tracks in Figure 5.9 follow aircraft that were commanded to perform some actions. Again, the eye is naturally drawn to any disconnects. This high level does not provide enough detail to determine specifically what may be wrong, but it does help target any problem, which can then be diagnosed by going back into the lower-level visualizations above.

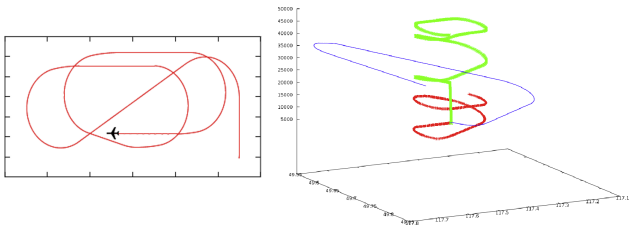


Figure 5.9: Gnuplot 2D Representations

Although a major consideration in visualization is to avoid investing costly, tangential effort into purpose-built graphical tools, at some point this perspective often becomes necessary because general-purpose tools have no inherent relationship to the problem domain. In this case, the author provides a three-dimensional visualizer written in JOGL (Java OpenGL) that is used throughout many courses, and indeed derives from similar needs in earlier work in the defense industry [21]. Figure 5.10 depicts a variety of cartoon-like, yet very informative, sequences of actions and events.

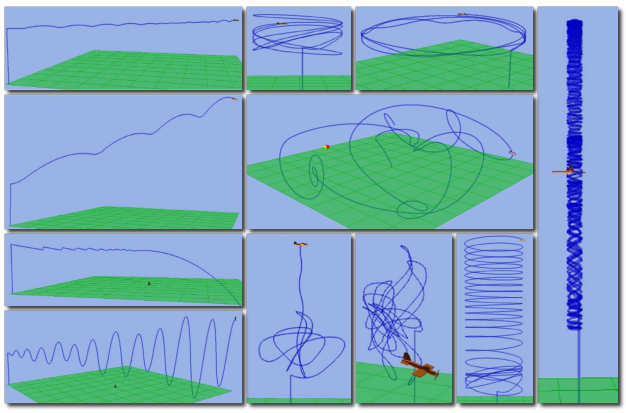


Figure 5.10: 3D Visualizer

The capability to integrate domain-specific visualization is key. Metainformation, such as fields of view and degrees of freedom in Figure 5.11, are invaluable for making sense of otherwise hidden aspects of the world.

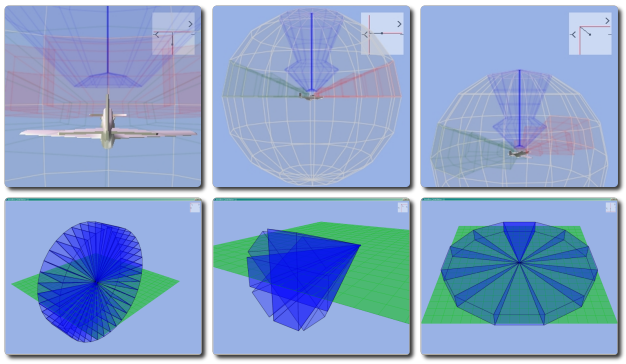


Figure 5.11: 3D Visualizer Augmentation

Finally, as many projects model components in the real world with world coordinates (albeit simplified to flat earth), their output in latitude, longitude, and altitude directly exports to tools like Google Earth, which can depict tracks overlaid onto actual terrain, as in Figure 5.12.

Figure 5.12: Google Earth Visualization

5.4 Analysis

Analysis involves making sense of the results of experiments. For subject-matter experts, simulation tools provide insight into domain-specific problems. For students within the context of an educational environment, however, the goal of analysis is primarily to establish that the software itself works appropriately.

To this end, students have to produce a professional-looking test report based on a cross-section of roughly 40 experiments that demonstrate representative aspects of the system. For consistency, since not every team's own solution was correct or functioned identically, they used the author's. Each experiment addressed eight requirements, where 1–4 relate 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.
3. The commands for (2).
4. An English narrative of the expected results.
5. The actual results with at least one graph showing the most representative view of the states.
6. A snippet of the actual results from the log file with a supporting explanation, including statistics, metrics, and graphs, as appropriate.
7. A discussion on how well the actual results agreed with the expected results, or if they disagreed, a hypothesis on why.
8. A suggestion for how to extend this test to address related aspects of potential interest.

The experiments varied wildly from project to project. The following is a subset from MTR:

- Fly an airplane on a constant course at a constant altitude and speed.
- Fly an airplane in a 360-degree clockwise turn approximated by an octagon in a climb where each leg of the octagon is a separate climb. All legs should have the same increase in altitude.
- Drop a bomb from a high-speed airplane at 8,000 feet onto a ship.
- Drop a depth charge with an acoustic fuze near a submarine, but miss.
- Fire a missile with a radar sensor and depth fuze from a ship at an airplane, detonating near the airplane.
- Fire a missile with a radar sensor and time fuze from a ship at an airplane, detonating near the airplane.
- Fire a torpedo with a sonar sensor and sonar fuze from a submarine at a fast ship.
- Fire a missile with a radar sensor and radar fuze from an airplane at a ship. Move the ship in such a way that the radar signal reflectivity goes from maximum to minimum and back as a function of aspect angle.

Snippets of the visualizations are invaluable for supporting the argument that useful tests were conducted correctly. For example, Figure 5.13 depicts dropping a bomb from a low-speed airplane flying right at 5,000 feet onto a ship. The bomb missed, but its (simplified) descent profile was as expected.

Figure 5.13: Bomb Release, Side View

Although these simulations are often cartoon-like in their simplifications, they still reflect a relatively rich set of behaviors to tease out. A small set of more complex experiments always provides this interesting opportunity.

Figure 5.14: Torpedo Engagement, Top View

For example, Figure 5.14 depicts firing two torpedoes from a submerged submarine at a ship that is broadside at launch and tries to outrun them. As the torpedoes converge on the ship, their active sonar sensors begin to interfere with each other because they are on the same frequency. The students needed to make an earnest attempt at accounting for this observation. They are not training to be subject-matter experts and thus are not held

to that standard, but by this point in the course, they should be able to articulate a reasonable hypothesis, whether correct or not. In the DIKW hierarchy, this aspects demonstrates knowledge and even hints of wisdom.

6. Results

Each project was independent with a different group of approximately 32 students. The papers cited for these projects report on their particular results. However, the shared framework for teaching this course generally relies on a common set of measures, which generate a substantial amount of quantitative and qualitative feedback over 11 weeks:

- Anecdotal observation
- Eight individual assignments
- 10 anonymous weekly self-reflections
- 16 project status reports (both individual and team)
- Three team project deliverables
- Project evaluation
- Team evaluation
- Development reflection
- Course evaluation

In quantitative terms, on average 88% of the students stated that the architecture permitted them to build interesting and entertaining real-world systems that they thought they would never have been able to do on their own. Furthermore, 90% indicated that the test reports directly contributed to a stronger understanding of what the programmatic solution was actually doing, whereas they otherwise would have had much less confidence in it. Overall, the students rated the projects 4.6 out of 5 (excellent).

7. Future Work

Developing a new project for each of three quarters in an academic year is taxing for the instructor. Although much of this framework is reusable in principle, it is not a simple and straightforward activity in practice. A classroom aspect of future work will be to streamline this process further. With an ever-growing set of complete projects, hybrid projects that combine several, such as the current aircraft accident reenactment simulator, are becoming much more feasible.

A second aspect of future work relates to the breadth and depth of domain coverage in these projects. Students investigate a relatively small subset of the capabilities. The author would not develop such large and complex projects if this limited perspective were the only goal. Rather, the dual-purpose intent is also to use them for research. Although the underlying models tend to be gross simplifications and thus do not adequately capture the fidelity necessary to study the problem domain in intricate detail, they do lend themselves nicely to other research considerations. Sensitivity analysis, for example, is important in determining appropriate or optimal configurations of components. Monte Carlo methodology is a powerful means of exercising the models in ways that reflect real-world uncertainty without undue explicit configuration. Finally, incorporation of machine learning appears especially promising for countless aspects of the problem and solution domains.

8. Conclusion

The eight projects showcased throughout this paper demonstrate a rich breadth and depth of examples of using modeling, simulation, visualization, and analysis in support of teaching software systems engineering. The underlying pedagogical foundation successfully helps students to understand how to approach, carry out, and verify the many confusing and error-prone steps of analysis, design, implementation, testing, and evaluation in a way that is educational, practical, and engaging.

References

[1] autsys.aalto.fi/en/research/mechatronics, last accessed May 11, 2016.

[2] D. Tappan. “Experiencing Real-World Multidisciplinary Software Systems Engineering Through Aircraft Carrier Simulation.” In Proc. of American Association for Engineering Education Conference, New Orleans, LA, June 26–29, 2016.

[3] M. Chemuturi. *Mastering Software Quality Assurance: Best Practices, Tools and Technique for Software Developers*. Page ix, J. Ross: Ft. Lauderdale, FL, 2010.

[4] P. Johnson-Laird. *Mental Models*. Cambridge University Press: Cambridge, 1983.

[5] J. Van Gaasbeek and J. Martin. “Getting to Requirements: The W5H Challenge.” In Proc. of 11th Annual Symposium of INCOSE, Melbourne, Australia, 2001.

[6] Adapted from thomas-robert.fr/en/loganalysis-open-source-web-tool-for-geographic-business-intelligence, last accessed May 11, 2016.

[7] J. Rowley. “The wisdom hierarchy: representations of the DIKW hierarchy.” Journal of Information Science, vol. 33, no. 2, 2007.

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

[9] P. Denning. “The Science in Computer Science.” CACM, vol. 56, no. 5, May 2013.

[10] M. Waldrop. “Why we are teaching science wrong, and how to make it right.” Nature, vol. 523, no. 7560, July 15, 2015.

[11] sciencebuddies.org, last accessed May 11, 2016.

[12] D. Tappan and M. Hempleman. “Toward Introspective Human Versus Machine Learning of Simulated Airplane Flight Dynamics.” In Proc. of 25th Modern Artificial Intelligence and Cognitive Science Conference, Spokane, WA, Apr. 26, 2014.

[13] D. Tappan. “A Holistic Multidisciplinary Approach to Teaching Software Engineering Through Air Traffic Control.” Journal of Computing Sciences in Colleges, vol. 30, no. 1, pp. 199–205, 2014.

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

[15] D. Tappan. “A Quasi-Network-Based Fly-by-Wire Simulation Architecture for Teaching Software Engineering.” In Proc. of 45th IEEE Frontiers in Education Conference, El Paso, TX, Oct. 21–24, 2015.

[16] app.ntsb.gov/news/events/2010/clarence_center_ny/animation.html, last accessed May 11, 2016.

[17] D. Tappan. “Multiagent Test Range: Fostering Disciplined Software Engineering Practices in Students via Modeling, Simulation, Visualization, and Analysis.” In Proc. of Alabama Modeling and Simulation Council International Conference and Exposition, Huntsville, AL, May 6–7, 2014.

[18] A. Hunt. *Pragmatic Thinking and Learning: Refactor Your Wetware*. Pragmatic Bookshelf, 2008.

[19] A. Grosskopf, M. Weske, J. Edelman, M. Steinert, and L. Leifer. “Design thinking implemented in software engineering tools.” In Proc. of 8th Design Thinking Research Symposium, Sydney, Australia, 2010.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis: Addison-Wesley, 1995.

[21] D. Tappan. “Pedagogy-Oriented Software Modeling and Simulation of Component-Based Physical Systems.” 21st Annual Conference on Software Engineering and Knowledge Engineering, Boston, MA, July 1–3, 2009.

Author Biography

DAN TAPPAN is an Associate Professor of Computer Science at Eastern Washington University. He has been a professor of computer science and engineering for 11 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 main research areas are software and hardware systems engineering, especially for aviation and military applications with embedded systems and mechatronics; modeling, simulation, visualization, and analysis; intelligent systems/artificial intelligence (knowledge representation, reasoning, machine learning); and computer science and engineering education.