Experiencing Real-World Multidisciplinary Software Systems Engineering Through Aircraft Carrier Simulation

1 Introduction

Modern technology is a complex combination of mechanical systems controlled by electrical systems ultimately controlled by software systems. Mechanical and electrical engineering students generally receive multidisciplinary hands-on exposure to such real-world applications, but those in computer science rarely see or appreciate this perspective. This work provides an engaging virtual environment for investigating an extensive breadth and depth of practical aspects related to the analysis, design, implementation, testing, verification, validation, refinement, and accreditation of software-based systems of systems.

The overarching theme is the operational environment of an aircraft carrier containing a wide variety of complex static and dynamic components. The primary ones are the carrier, its aircraft, and the refueling tankers, all interacting through secondary ones such as catapults, landing arresting wires, optical landing systems, refueling booms, tailhooks, etc. By posing and getting resolution on *who*, *what*, *when*, *where*, *why*, and *how* (W^5H) questions, students thoroughly decompose each component into its data, control, and behavior elements, which respectively correspond to what it is, what it can do, and what it actually does in all relevant contexts. This organization then maps onto well-established creational, structural, and behavioral design patterns within an architectural framework for respectively building, connecting, and using the components in real time. It also establishes a representation that helps students understand the problem domain in terms of requirements and specifications.

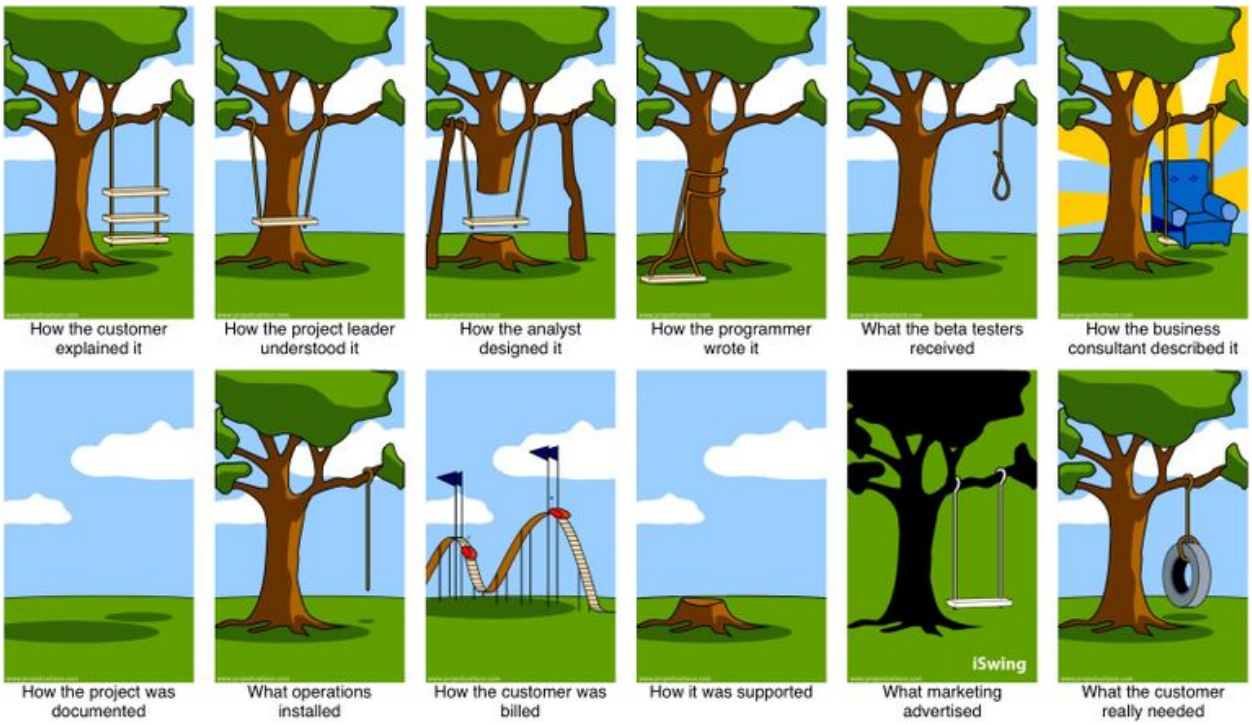
This flexible Java-based environment allows students both to analyze existing solutions (which would be impractical to build themselves) and to synthesize their own. At all stages, it fosters critical thinking because the subject matter, pedagogical approach, and environment force students outside of their comfort zone, where they cannot rely on their generally inexperienced, brute-force problem-solving strategies to construct a solution. In particular, it contributes to understanding how to develop a mental model of the unfamiliar real-world problem space, which ultimately maps through many transformations to the virtual-world solution space in software.[12,18] The final product was a multiagent continuous time-stepped simulation in which students in a junior-level software engineering course played the computer science roles of analyst, designer, implementer, and tester, as well as multiple user roles. The structure of this paper establishes the foundation, describes the project, and connects these concepts to student learning and a summary of the outcomes.

2 Software engineering foundation

Software engineering is a vast collection of theory and practice with the goal of producing the highest-quality software at the lowest cost. It shares many characteristics with traditional engineering design processes, but for the purposes of this work, the following elements are the emphasis. In particular, this course promotes the Agile methodology, which is supposed to achieve the same results without imposing onerous, administration-heavy overhead.[1] Agile is not a substitute for proper planning and execution, however, so this freedom demands discipline, which is generally lacking in students at this stage of their education.

- *analysis*: understand the real-world problem space, especially what the customer most likely truly wants (although rarely realizes it), by eliciting requirements (what to do) and specifications (constraints on how to do it).
- *design*: establish the virtual-world pieces that correspond to those in the analysis and create a conceptual framework in which they reside and interact, as well as a plan to construct it.
- *implementation*: write software based on the design.
- *testing*: assess whether the pieces function individually and collectively.
- *verification*: demonstrate that the software satisfies the requirements and specifications.
- *validation*: demonstrate that the requirements and specifications appropriately address the problem space.
- *refinement*: improve the software until it meets the evaluation criteria in the testing, verification, and validation stages.
- *accreditation*: demonstrate that all criteria imposed by certifying organizations (e.g., FDA, FAA) are satisfied; typically only mission and life-critical software undergo this rigorous and expensive process.

This process of mapping a problem to a solution is deceptively simple and linear in this form. However, despite the similarities between traditional and software engineering, designing software is much more difficult and prone to error.[21] Its virtual nature promotes trial-and-error development, which in itself is a great thing because it unleashes creativity that is not bound by physical constraints. Without self-discipline, however, this freedom becomes a crutch to avoid critical thinking and truly understanding the problem and solution. It is normal for students to start at this level; the danger is that they never grow out of it and continue these poor practices into their careers, where the consequences are real and significant. Figure 1 is a long-standing cartoon from the public domain that captures the universally acknowledged dysfunctional nature of software development in reality. While none of these disconnects are entirely avoidable, many of the problems that could be resolved early unfortunately propagate to the later stages, where the cost to correct them rises exponentially. (The term “disconnect” is appropriate because these decisions indeed seem like the proverbial “good idea at the time”; only later do they manifest themselves as costly errors.) The course starts with a warning to the students that they will experience “The Cartoon”; embracing the philosophy of this course does not avoid this problem, but not embracing it guarantees a much bigger one. Many student comments at the end reflected the sentiment that they should have taken this warning more seriously from the start. In this respect, trial by (harmless) fire was a good experience for the students, who had become understandably accustomed in the introductory courses to easy problems with easy solutions. With enough brute force, anything could be solved. This case no longer applied, and it never will again in their careers. The next major step in their curriculum was Senior Project and Senior Capstone (also taught by the author), where they were left to their own devices to solve their real customer’s real-world problems. This paper does not address those results, but they do consistently show that students who experienced this approach perform better on average than those who had not.



Part of the philosophy to this approach is grounding the virtual model in reality. Too many problems in software stem from making something happen that has no common-sense counterpart in reality (e.g., dividing by zero in Section 5.2.2). But without the critical ability to recognize such disconnects, programmers at all levels successfully manage to build the unbuildable, so to speak. Weinberg's classic quote captures this sentiment perfectly: "If builders built houses the way programmers buil[d] programs, the first woodpecker to come along would destroy civilization."[3]

3 Pedagogical foundation

The pedagogical approach is to push students outside of their comfort zone, where it becomes nearly unavoidable to apply research and critical-thinking skills to make holistic sense of a problem that is intentionally unfamiliar. They must understand not only the construction and operation of the real-world system, but also how these elements map onto the software-development process and the intended solution. In particular, they had to establish the underlying building-block primitives and the operations for combining them into more complex structures and actions within the architecture.[21] When left to their own devices, students tend to gravitate toward bloated and brittle ad hoc solutions made up on the fly, whereas this approach required solutions that demonstrated at least the following characteristics:

- compositional: larger parts hierarchically consist of smaller parts
- modularized: parts are integrated into well-defined, cleanly organized and justifiable units with distinct roles and no gaps or overlaps
- integrated: the different parts work together as a system
- unified: the system appears to the user as a single entity, not as discrete parts
- reusable: parts can be transplanted into other projects without undue effort

- orthogonal: one solution applies to many related problems
- scalable: the solution can manage a larger number of the same kinds of components
- extensible: the solution can manage different kinds of new components

The pedagogical foundation relies on the familiar (original) Bloom's Taxonomy of Educational Objectives.[8] It describes an ordered structure of learning activities from low-level *remembering*, *understanding*, and *applying* to high-level *analyzing*, *creating*, and *evaluating*. This representation maps closely to the process of real-world software development. The education community debates the order of the last two, but for software development, this one is the norm because evaluating strongly corresponds to testing.[24] While creating (writing programs) is the most effective part of active learning and indeed produces the end product, students unfortunately tend to consider it the *only* part; i.e., coding is software engineering. To mitigate this problem, this work emphasizes the earlier levels (often maligned as "busy work") to force students to experience and ultimately appreciate them and the positive results that they bring to the process. Consistent with the Pareto Principle, software engineering is 80% thinking first so that the 20% doing is done right later.[15] This approach aligns well with the author's teaching philosophy, the curriculum, and the specific population at an open-enrollment regional comprehensive university with overwhelmingly unprepared and underprepared students. In particular, it is designed from the ground up to be understandable, accessible, meaningful, and relevant. It addresses weaknesses and misleading or incorrect preconceived notions to help understand, define, connect, manipulate, and evaluate endless dots among vast complex resources in an unfamiliar problem domain.[26]

3.1 Critical thinking and analysis

Disciplined software development defines and then follows requirements and specifications as a road map of what to deliver and how it should operate, respectively. Requirements elicitation is a very messy process of initially collecting a massive amount of supporting materials in different forms and then making sense of them. Bloom's Taxonomy is helpful as an overview, but the classroom environment needs something more tangible. Students do not tend to see how abstract concepts apply without concrete examples.[17] The data-information-knowledge-wisdom (DIKW) hierarchy in Figure 2 plays this role here:[25]

- data: raw values with no associativity or context
- information: values in one context
- knowledge: values in multiple contexts
- wisdom: generalized principles created by connecting a network of contexts from different sources for predictive, anticipatory, proactive understanding

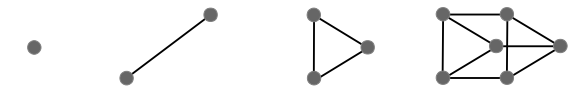


Figure 2: DIKW Hierarchy

Background research and low-level W^{5}H questions about the problem establish data. Repeated investigation from different perspectives builds associations and interrelations like cause and effect to establish information. A sufficiently interconnected network serves as the basis of relevant knowledge and helps form a mental map that can be applied to this problem.[15,16] This process repeated many times over a long period leads to wisdom, or the ability to apply complex problem solving to any problem. The process of becoming an expert is unique for everyone, but rarely is it quick and easy for anyone. In fact, the commonly accepted figure is 10,000 hours of practice to achieve true mastery of a subject.[14] The students at the junior level in this course had had perhaps a few dozen hours of truly productive hands-on programming time. Even by graduation, their experience hardly reaches into the hundreds. While this work does not give students an appreciable number of hours, it does arguably make better use of those hours.

3.2 Scientific method

The basis here of modeling, simulation, visualization, and analysis for software testing and evaluation is the scientific method. Students theoretically have a relatively broad background in science already because they are required to take several introductory science courses. However, far too many fail to grasp the underlying philosophy of science because they fixate on memorizing endless esoteric details of rocks, for example.[30] They acquire low-level data, not higher-level information, knowledge, or wisdom. Consistent, long-term anecdotal evidence by the author strongly suggests that almost none realize that this philosophy extends to computer science (despite its name), and how to utilize it to their advantage.[9] To this end, this system, and its use as a pedagogical tool, employs the scientific method at its core. It is an environment for posing questions, setting up and running meaningful experiments to collect supporting data, combining and putting them into an intuitive form of information, analyzing this form with respect to the questions, reporting the results, and acquiring knowledge and experience throughout the process. It is an iterative methodology. If the results are not good (e.g., a test fails), then it provides a mechanism for assessing where the problems lie, attempting to correct them appropriately, and rerunning the same experiments until they are eliminated. If the results are good, the same mechanism allows them to be refined and improved.[28]

This methodology requires two important pieces: (1) critical-thinking ability to pose informed, probing questions, conduct revealing experiments, and make sense of the results, and (2) an environment for managing these experiments in a reproducible, controlled, and disciplined manner. The basis of a controlled experiment is to run an initial simulation to collect baseline results for subsequent comparison. Based on the actual results compared to the expected (desired) results, perturb one and only one parameter of the simulation, rerun the same experiment, and assess any differences in the results, which can be directly attributed to the change in the parameter. This process establishes cause-and-effect relationships that force students to understand what they are doing and why.[27] Too often their testing "strategy" is a haphazard ad hoc approach of trying whatever comes to mind — especially if it is easy — and believing it would be somehow representative of the overall performance of the entire system. The high frequency of incorrect solutions throughout all their programming courses demonstrates a deficiency in testing skills, which can be attributed arguably to their lack of a true testing methodology.

4 Problem domain

The problem domain defines the real-world counterpart that this system models and then simulates. Although the complete, functional solution that the students would ultimately work on already existed (but was not available to them), they still had to go through the inception-phase exercise of developing requirements for what they would want in a solution. Section 6 goes further into these details, which reflect an overall lack of critical thinking.[13] In their storyboards, which described how the user would carry out various actions, there were many gaps that would have prevented meaningful use. For example, with no capability to move an airplane off the runway after landing, subsequent landings would be impossible.

In an agent-based simulation, the agents are the most important element to model. They are defined primarily in terms of three aspects: *data* (what they are), *control* (what they can do), and *behavior* (what they actually do or have done to them in an operational context). The students' decomposition was riddled with inconsistencies, such as control operating on incompatible data and behavior relying on nonexistent control. It also reflected an overall lack of understanding of the problem domain itself because many did not take the background research seriously.[17] In many cases, students supplied their own (incorrect) knowledge from movies and video games. The research component of this project was explicitly meant to prevent such shortcuts, but some students nevertheless thought otherwise. Trying to manage the behavior of students who think they know better or have found shortcuts is challenging. Even showing them how their inappropriate choices would fail was often met with a "whatever, who cares, it doesn't really matter anyway" attitude.

4.1 Primary agents

The primary agents are the ones that move around in the world and interact with each other with respect to a goal. They are standalone and usually under direct control of the user through behavioral commands (see Section 5.3.4). (User and student are actually the same person here, but they play two different roles as the customer and developer, respectively.) Any number of primary agents can be managed in a simulation, but for logistical reasons, the students usually focused on one of each:

- A *carrier* contains one or more fighters. It is a highly configurable component that allows the user to define how many primary and secondary agents it contains, where they are, and how they behave.
- A *fighter* is a notional aircraft that is based on a carrier, can take off, fly around, refuel, and land. It can start on a carrier or in the air. It is unarmed and (much to some students' disappointment) does not engage in any combat actions.
- A *tanker* is a notional aircraft of unknown origin. It starts in the air and remains there throughout the entire simulation. Its role is to refuel fighters. It cannot land on a carrier.

4.2 Secondary agents

The secondary agents facilitate the primary ones in performing their actions. They are always compositionally part of a primary agent, never standalone. Some parts of the storyboard are automatic, but most require the user to issue multiple commands for configuration and execution.

The code-level solutions to the parts that the students had to implement needed to recognize and enforce the dependencies. Cleanly managing such dynamic coupling where agents connect and disconnect under different conditions in different contexts was challenging because the students were not allowed to hardcode specific combinations. Their solutions were supposed to employ disciplined object-oriented programming (OOP) to accommodate scalability and extensibility by working with both current and future implementations of agents, like other kinds of fighters. The `instanceof` operator and `getClass()` comparisons were strictly prohibited. Casting was strongly discouraged and had to be explained and justified. Far too many students force their programs to function in a brittle way instead of letting the OOP do its job for them.[20] In other words, they invest more effort into a larger solution that actually performs worse.

The typical carrier operations for takeoff are as follows. The secondary agents are initially in italics. Steps with an asterisk are automatic; otherwise, the user must explicitly enter a command.

1. The fighter starts in its parking spot.*
2. It taxis to the start of the *catapult* via a *taxiway*.
3. It connects to the catapult.
4. The *blast barrier* raises behind it.
5. It throttles up to maximum power.
6. The catapult rapidly drags it to the end of the runway.
7. The barrier lowers.
8. The catapult returns to the start position.*

Once airborne the fighter then automatically veers to the left 30 degrees to avoid being hit by the carrier should it crash. After this point, it is under user control.

The typical refueling operations are:

1. The fighter rendezvous with the tanker from behind.
2. The tanker extends the *female refueling boom*.
3. The fighter extends the *male refueling boom*.
4. If the booms are reasonably close, fuel transfer starts. The resolution of the viewer and the precision of the flight commands are inadequate for fine adjustments, so this part is flexible.
5. The fuel transfer proceeds at the specified rate. If either boom retracts or moves too far from the other, the transfer aborts.*
6. The tanker boom retracts.
7. The fighter boom retracts.

The typical landing-approach operations are:

1. The fighter flies into position to follow the carrier.
2. The *optical landing system (OLS) transmitter* on the carrier projects a landing path.
3. If the *OLS receiver* on the fighter aligns with the path, it is ready to land.*
4. The *tailhook* extends.
5. The fighter flies the approach.*

The typical touchdown operations are:

1. The *arresting wire* captures the tailhook and brings the fighter to a stop. The fighter will occasionally miss the wire at random and automatically retract the tailhook and take off.*
2. After a successful landing, the fighter disconnects from the wire and taxis to its parking spot.
3. The wire returns to its original position.*

Alternatively, this sequence can happen:

1. The arresting wire is disabled.
2. The *arresting net* raises.
3. The fighter stops at the net.*
4. Further landing operations cease because there is no way to unfoul the net.

5 Solution domain

The solution domain defines the mapping from the problem domain to the code-level implementation details that make it happen. Mapping top-down from the abstract to the concrete is a multistage process involving many parts.

5.1 Architecture

The architecture provides the unified framework in which all levels of the implementation details reside in an organized, disciplined manner. Students are not accustomed to operating within an architecture or reading its documentation in the form of a Javadoc HTML application programming interface (API) because the toy problems in their earlier courses are too small and standalone to justify one.[22] This course specifically chooses a large, complex project with a rich set of independent and interdependent facets for a strong breadth and depth of exposure to real-world thinking and doing.

The model-view-controller (MVC) architecture consists of 334 classes in Java 7. All directly relevant code is based on what students already know from their earlier courses. The three-dimensional visualizer (see Section 5.4.4) is a separate plug-in application that uses JOGL (Java OpenGL). Graphics programming is neither a prerequisite nor an emphasis in this course, so this part remains a magic black box, which is indeed the intent of MVC: it manages the separation of concerns to delegate responsibility appropriately and keep the discrete pieces from becoming hopelessly coupled and interdependent. It also accommodates dynamic plug-and-play addition, removal, or swapping of components. The view is an especially flexible example.

5.2 Model

As Section 4 introduced, agents dynamically model the real-world components that the user builds and manipulates by proxy in the virtual world of a simulation. The primary and secondary agents consist of the following unified approaches to implementing their many different actions with the minimum amount of different solutions. The intent is to discourage students from perceiving every problem as unique and then creating a unique solution. Such an undisciplined approach results in a large, unmanageable program.[15,16,20]

5.2.1 Datatypes

Extensive anecdotal evidence has consistently shown that students do not manage their data well. From their low-level assignments in earlier courses, they are familiar with using Java primitives like `int` and `double` and corresponding arithmetic operators. From the higher-level project perspective here, however, this approach leads to unsafe code, or at best, unnecessarily complex code. Primitives maintain only the numerical component to data. The context and units and appropriate operations, etc. must be understood and enforced by the programmer. For example, an addition operation on arguments in meters and kilometers or even meters and kilograms still produces the correct sum, but the resultant unit is meaningless. Enforcing type safety and usage rules is left to the students, who are notorious for assuming everything is always correct and doing nothing defensive. To mitigate this problem, this system provides a wealth of predefined concrete datatypes in Figure 3 for almost every low-level representation. Each contains its own error checking, appropriate operations, conversions, convenience methods, etc.

`Acceleration`, `Altitude`, `AngleMath`, `AngleNavigational`, `Attitude`, `AttitudePitch`, `AttitudeRoll`, `AttitudeYaw`, `Azimuth`, `Bearing`, `Callsign`, `CoordCartesianAbsolute`, `CoordCartesianRelative`, `CoordPolarMathematical`, `CoordPolarNavigational`, `CoordPolarNavigational3D`, `CoordWorld`, `CoordWorld3D`, `Course`, `Distance`, `Drag`, `Elevation`, `Flow`, `Heading`, `Identifier`, `Interval`, `Latitude`, `Lift`, `Longitude`, `Percent`, `Power`, `Range`, `Rate`, `Speed`, `Time`, `Thrust`, `Track`, `Vector`, `Velocity`, `Weight`

Figure 3: Datatypes

One odd hurdle that many students consistently encounter is not truly understanding how to formulate a mathematical or logical statement to solve a small problem. Although they have all taken many math courses, by and large they do not know how to *use* math; i.e., data without information or knowledge. Similarly, even after all the earlier programming courses, they do not really know how to *use* programming to solve problems. The combination of the two — a program doing math — clearly demonstrates a lack of ability to make use of existing skills in a new context. For example, *distance* defined as *rate* times *time* is an algebraic expression: given any two knowns, the unknown can be determined. For some reason, this basic mathematical thought eludes many of them and results in unbelievably complex open-form solutions with nested conditionals, loops, and static global variables. It seems that the freedom to throw more code at a problem interferes with their ability to focus on it.[7] These datatypes, while still not preventing wasted time from randomly trying potential solutions, at least enforces that meaningless ones do not compile or run. Despite this course being at the advanced junior level, many students still struggle with the compiler. For some, once the code compiles, they move on because this achievement evidently implies correctness. One student summed up his approach as "I kept throwing more code at the compiler until it shut up."

The second advantage to these datatypes is their functional nature: any operation on one produces a new copy of it; e.g., $time_1$ plus $time_2$ produces $time_3$.[23] Datatypes are immutable, which eliminates any possible issues with improper coupling. Students commonly fail to check whether their data are within acceptable bounds when passed into a method. Almost without exception, they do nothing to prevent the data from being changed as a side effect. Such a case is a design disconnect where data can be changed without corresponding control. For example,

passing an `ArrayList` into an object that is to maintain it means that the owner could still change it without the object's knowledge or approval later. Likewise, the object could change it on the owner. Such unforeseen interactions at a distance are incredibly difficult to diagnose. Immutable datatypes are immune.

The use of objects instead of primitives does result in more intensive memory management from the Java Virtual Machine, which negatively affects performance. However, safety is hierarchically more important than efficiency (correctness being the highest priority). While performance is not a major consideration here, the architecture actually does manage the copy-on-write semantics through the Flyweight, Prototype, and Factory design patterns, which serve as a practical example of these concepts from the prerequisite course.[10] It also shows how the architecture quietly plays a role behind the scenes in managing the system so that the students can focus on their own parts. Explicit memory management, as in C and C++, consumes an inordinate amount of their time and invariably leads to obscure and frustrating errors.

5.2.2 Effectors

Similar to datatypes, but up a level of abstraction, are effectors, which maintain a dynamic state between two static limits. All secondary agents that perform movement use them. They serve as yet another example of orthogonal design because one implementation manages any type of movement. An effector is defined (through Java generics) in terms of two static datatypes for the limits and a dynamic interpolated state between them as a percent. For example, this approach captures the linear motion of the catapult from the start to the end position and the rotational motion of the blast barrier as it tilts from its stowed horizontal angle to its upright vertical angle. Effectors in combination can easily produce complex realistic movement. For example, as the heading effector (for direction of flight) of an airplane changes from the current to the desired angle and the movement effector changes the position, an airplane follows a smooth curve instead of exhibiting an instantaneous sharp turn as would normally occur if the heading were changed at once with a `setHeading(Angle)` method. Students are used to designing classes with setter methods that act instantaneously. Most have no concept of continuous change from one state to another over time.

Effectors also accommodate acceleration and deceleration for very realistic movement. Especially important for fidelity at the physical level is consistent mechanics. For example, changing direction requires the movement to decelerate to zero speed before accelerating in the opposite direction. A typical student solution would negate the speed and instantaneously change direction. In terms of *rate* = *distance* / *time*, this change corresponds to undefined acceleration (effectively infinite) from dividing by zero. Neither math nor engineering nor the universe itself permits such a solution, but in code students find it trivially easy and do not see the underlying problem. Their typical workaround to a divide-by-zero exception is to set the result to some magic number and keep going. This action serves merely as a band-aid that masks the symptoms of the undesirable behavior, but it does nothing to address the underlying causes, which are likely more in the thinking than in the doing. Moreover, it instills a belief in students that correcting problems can be a quick and even easy process that requires little thought. It is like blindly selecting the (correct) suggestion from a spell checker without considering whether the new word fits syntactically, grammatically, and stylistically within the larger context of the existing sentence. This behavior quickly leads to a vicious circle of brittle, ad hoc corrections that lead to

more problems that subsequently lead to more such corrections. A tenet of debugging is that a correction should both correct the known problem *and* not introduce any new ones. Students invariably neglect the latter part because it involves significant effort to rerun existing tests to verify that nothing previously working is now broken. Without the self-discipline to maintain such regression tests and actually execute them, debugging turns into herding cats.

5.2.3 Communication buses

Students are accustomed to explicitly telling their programs what to do. Indeed, the *imperative* programming paradigm by name implies commanding. They tend to take this approach too far, however, by throwing excessive code at a problem because this option is readily available. To counter this undisciplined behavior, agents and the components that compose them communicate over virtual networks of communication buses based on the Observer, Command, and Interpreter design patterns.[10] The protocols are well defined and not subject to indiscriminate hacking. Students need to understand how to behave responsibly within a framework because they will not have the option of circumventing it in today's typical client-server architectures or distributed systems, for example. In fact, they encounter similar problems in their course on operating systems, but most fail to see the connection that the architecture here is indeed playing the same role for this project as an operating system does for the entire computer. This type of project provides a valuable opportunity to make explicit connections (information and knowledge) between dots (data) that they already know.

Communication is based on requests and responses. Only components that legitimately belong on the network and agree to behave can participate. Requests are not demands, and students need to realize that they are no longer fully in control. The typical handshaking process involves submitting a request, getting an immediate confirmation, and then later getting a notification that the request was serviced. The bus protocols support multiple asynchronous message types, which map onto any action that effectors can perform:

- `ACCEPTED_SERVICING_IMMEDIATE`: the receiving component serviced the request immediately and returned the result (if any) with this response.
- `ACCEPTED_SERVICING_CALLBACK`: the component is servicing the request and will return the result when it is completed.
- `ACCEPTED_PENDING`: the component is busy and will queue the request for servicing.
- `REJECTED_INVALID`: the component cannot service the request because it is invalid.
- `REJECTED_UNABLE`: the component would ordinarily be able to service the request but cannot for some reason.
- `IGNORED`: the component silently ignored the request because it is irrelevant.

Requests can be sent to individual agents (by identifier), groups of agents (like all fighters), or all agents. An added benefit of this unified communication system is that the logging (see Section 5.4.1) keeps track of all traffic, which makes debugging, testing, evaluation, analysis, and reporting much easier.

5.2.4 Glyphs

Primary agents consist of secondary agents. Fighters and tankers have a fixed set, but the carrier is very flexible in its definition. Indeed, this flexibility extends far beyond what the creational and structural commands in Sections 5.3.2 and 5.3.3 can reasonably specify. Instead, this process involves loading and interpreting a file that defines the shape of the carrier and where the catapults and arresting wires, etc. are. The students had to build this component. They were not accustomed to using persistent external data structures for such a purpose because normally they hardcode definitions. This experience, which reflects the way real-world programs typically operate, was enlightening.

This experience was also a good example of using wisdom to achieve a quality result with less effort. Specifically, the author showed the example in Figure 4. The top two perspectives are a clipart aircraft carrier scaled in such a way that the pixel coordinates correspond to the coordinate system in feet in the project. As such, any graphics editor serves as a quasi-"carrier editor" that allows students to place and connect reference dots. The process of translating these coordinates into the file representation is manual by reading them off the screen and typing them in to produce the bottom perspective, so it is not a convenient long-term solution or appropriate for the end user. But as a quick-and-dirty development tool, such tricks of the trade go a long way. Indeed, in their initial requirements, many students had indicated the need to build their own editor.

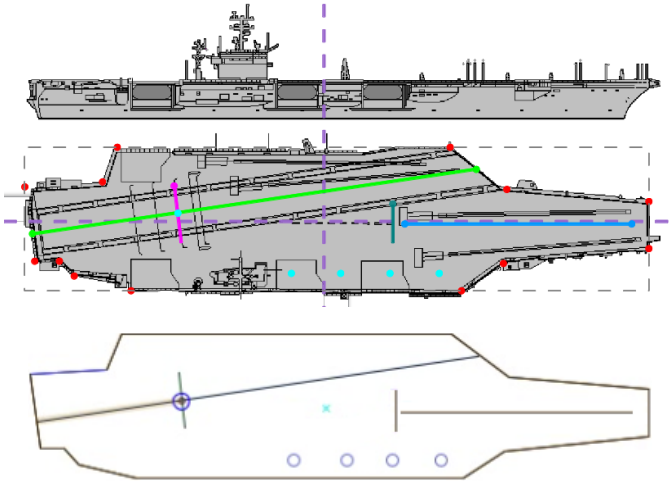


Figure 4: Carrier Definition[11]

5.3 Controller

In this model-view-controller architecture, the controller plays two roles: (1) interpret commands from the user, and (2) manage execution of the simulation. The following subsections summarize the commands. The management aspect is similar to that in any large software system: the architecture is basically a problem-specific operating system. In this case, it plays the especially important role of ensuring repeatable execution for controlled experiments. Contrary to expectation, Java threading is not a viable option because of its nondeterministic behavior. This example was yet another eye-opening experience for the students because in their initial

decomposition of the project, every single one of them had chosen threading for the simultaneous multiagent aspects. This misconception reflects a general understanding of concepts (like concurrency) without a deeper understanding of their implications (like how concurrency needs to function with respect to repeatability). This experience serves as the opposite example from Section 5.2.4: instead of failing to make connections, students overgeneralized to make otherwise correct connections in contexts where they were not appropriate. In the DIKW hierarchy, this error falls under wisdom, which people acquire through both good and bad experiences. The teaching philosophy coupled with this project provides many opportunities that naturally lead to valuable discussions such as this one. Despite some students believing that the instructor "sets them up for failure" this way, the consequences of their inappropriate decisions and actions are minor — just enough to be memorable without actually being detrimental to their grades. To reiterate from the introduction, the overarching philosophy is to push students outside of their comfort zone by challenging them to go one or more steps further from what (they believe) they already know and can do. Isolating themselves within their safe zone may feel comforting, but it is not where true learning actually happens.[2]

5.3.1 Commands

For simplicity and flexibility, all input (except mouse actions in the views) is through text commands at a command line. In the model-view-controller architecture, this component could be replaced with something more convenient in the future, but it aligned well with the students' skills here. Each command is a single case-insensitive statement based on a regular grammar. The author's solution used JavaCC for the parser for personal convenience, but the students had to build theirs using standard Java to exercise their own skills. It is based on the Interpreter and Command design patterns to map user input to the datatypes and the API of the architecture.[10] All commands use the fields in Table 1 for their specifics.

Field	Definition	Example	Datatype
`acceleration`	unsigned integer (feet per second)	`100`	`Acceleration`
`altitude`	unsigned integer (feet)	`1000, 9500`	`Altitude`
`azimuth`	unsigned real (nav. degrees)	`10, 45`	`AngleNav`
`coordinates`	`latitude/longitude`	`45*30'15"/110*30'10"`	`CoordWorld`
`course`	unsigned 3-digit integer (nav. degrees)	`090, 270`	`AngleNav`
`distance`	unsigned real (feet)	`10, 25.3`	`Distance`
`elevation`	unsigned real (math degrees)	`10, 25`	`AttitudePitch`
`flow`	unsigned real (pounds per second)	`10, 25`	`Flow`
`id`	alphanumeric string, plus underscore	`dog, cat32`	`Identifier`
`latitude`	*degrees*'*minutes*'*seconds*"	`45*30'15"`	`Latitude`
`longitude`	*degrees*'*minutes*'*seconds*"	`110*30'10.3"`	`Longitude`
`origin`	signed double:signed double (feet)	`5:8, -5:+3.5`	`CoordCartRel`
`percent`	unsigned integer [0,100] (percent)	`0, 100`	`Percent`
`rate`	unsigned integer (milliseconds)	`50, 1000`	`Rate`
`speed`	unsigned integer (knots)	`25`	`Speed`
`string`	'any character string'	`'myfilename.txt'`	`String`
`time`	unsigned time (seconds)	`5, 10.8`	`Time`
`weight`	unsigned integer (pounds)	`50, 200`	`Weight`

Table 1: Field Definitions

A command-based approach has significant advantages over the direct approach that students know at this point in the curriculum. The first part of the process, parsing, is identical in both cases. For example, for the command `DO id SET SPEED speed`, the parser needs to identify the command (see Section 5.3.4) and then extract the contents of the fields for the identifier of the agent and the speed to assign. It is the execution part that differs. Instead of calling a method like `setSpeed(id,speed)`, the students must instantiate a command object like `CommandSetSpeed` with these values. The final step is to submit the command to the architecture to schedule its execution. At this point, the process is out of the student's hands, which many indicated was awkward (even "weird") because they had no control after this point. However, this handoff is precisely the intent of an architecture-based system because it forces them to make their choices wisely. They must truly understand what they are asking the system to do and how because there is no recourse to hack the effects if they are not what the students expected.

Besides this direct pedagogical value, the students had the opportunity to see a much richer use of the Command pattern than in the Design Patterns course itself. In particular, with this one relatively small design decision, the solution went from addressing a single problem to accommodating many of the common features expected in today's software; e.g., undo and redo functionality, and macro and scripting support. It also connected nicely with the required course on graphical user interfaces, which could provide a more attractive, user-friendly environment for this system via the Facade pattern.[10] The intended purpose here, however, was to have a convenient mechanism for running tests, which Section 6.4 covers in detail.

5.3.2 Creational commands

Creational commands build the primary and secondary agents, and to some degree connect them. The process is similar to object-oriented programming, where a class (a template) defines the blueprint for an object (an agent). Many students do not have a solid understanding of the programmatic difference between declaring something (`int i`) and defining it (`i=1`), or when these two operations occur together (`int i=1`). This approach presented the opportunity to explain basic compiler theory and object management, which they would otherwise never experience in the curriculum.

5.3.2.1 Define

The 15 define commands build the templates, which then serve as the available stock for subsequently creating any number of agents. These commands vary widely depending on the nature of the component. For example, the following defines template *tid* for a catapult with its origin at *origin*, azimuth *azimuth*, length *distance*, acceleration rate *acceleration*, weight limit *weight* for any fighter connected to it, terminal launch speed *speed*, and reset time *time*, which specifies how long it takes to become available again for the next launch:

```
DEFINE CATAPULT tid ORIGIN origin AZIMUTH azimuth LENGTH distance
ACCELERATION acceleration LIMIT WEIGHT weight SPEED speed RESET time
```

Similarly, the commands for primary agents specify basically a class of real-world entities, like a Nimitz-class carrier or a long-range refueling tanker:

```
DEFINE CARRIER tid SPEED MAX speed1 DELTA INCREASE speed2 DECREASE speed3
TURN azimuth LAYOUT string
```

Defines template *tid* for a carrier with maximum speed *speed*$_1$, acceleration rate *speed*$_2$, deceleration rate *speed*$_3$, turning rate *azimuth*, and layout filename *string*.

```
DEFINE TANKER tid SPEED MIN speed1 MAX speed2 DELTA INCREASE speed3
DECREASE speed4 TURN azimuth CLIMB altitude1 DESCENT altitude2 TANK weight
```

Defines template *tid* for a tanker with minimum speed *speed*$_1$, maximum speed *speed*$_2$, acceleration rate *speed*$_3$, deceleration rate *speed*$_4$, turning rate *azimuth*, climb rate *altitude*$_1$, descent rate *altitude*$_2$, and fuel-tank quantity *weight*.

The fighter adds another level of compiler theory by accommodating formal parameters (the prefixes with colons) that the CREATE FIGHTER command can override later:

```
DEFINE FIGHTER tid SPEED MIN speedmin:speed1 MAX speedmax:speed2 DELTA
INCREASE dspeedinc:speed3 DECREASE dspeeddec:speed4 TURN dturn:azimuth
CLIMB dclimb:altitude1 DESCENT ddescent:altitude2 EMPTY WEIGHT
weight:weight1 FUEL INITIAL fuelinit:weight2 DELTA dfuel:weight3
```

Defines template *tid* for a fighter with minimum speed *speed*$_1$, maximum speed *speed*$_2$, acceleration rate *speed*$_3$, deceleration rate *speed*$_4$, turning rate *azimuth*, climb rate *altitude*$_1$, descent rate *altitude*$_2$, empty aircraft weight *weight*$_1$, fuel-tank quantity *weight*$_2$, and fuel burn rate *weight*$_3$ per knot of speed.

5.3.2.2 Create

The create commands combine the templates and specify further details to build the agents. The primary agents are the most complex because they contain secondary agents:

```
CREATE CARRIER aid1 FROM tid WITH CATAPULT aid2 BARRIER aid3 TRAP aid4 OLS
aid5 AT COORDINATES coordinates HEADING course SPEED speed
```

Creates carrier *aid*$_1$ from carrier template *tid* with catapult *aid*$_2$, barrier *aid*$_3$, trap *aid*$_4$, and optical-landing-system transmitter *aid*$_5$ at coordinates *coordinates* with heading *course* and speed *speed*.

```
CREATE TANKER aid1 FROM tid WITH BOOM aid2 AT COORDINATES coordinates
ALTITUDE altitude HEADING course SPEED speed
```

Creates tanker *aid*$_1$ from tanker template *tid* and female boom *aid*$_2$ in the air at coordinates *coordinates* and altitude *altitude* with heading *course* and speed *speed*.

```
CREATE FIGHTER aid1 FROM tid WITH OLS aid2 BOOM aid3 TAILHOOK aid4 [TANKS
aidn+] [OVERRIDING (aidm.argname WITH string)+] [AT COORDINATES coordinates
ALTITUDE altitude HEADING course SPEED speed]
```

Creates fighter *aid*$_1$ from fighter template *tid*, optical-landing-system receiver *aid*$_2$, male boom *aid*$_3$, tailhook *aid*$_4$, and optional auxiliary fuel tanks *aid*$_n$. The optional initial airborne state dictates that the fighter must start in the air at coordinates *coordinates* and

altitude *altitude* with heading *course* and speed *speed*. The value of any named argument *argname* in aid_m can be overridden with *string*.

The nine commands for secondary agents are simpler because they have no configuration arguments. For example, the following creates catapult agent *aid* from catapult template *tid*.

`CREATE CATAPULT` *aid* `FROM` *tid*

5.3.3 Structural commands

The define and create commands are hybrid creational and structural commands for building agents. The only dedicated structural commands are to add these agents to the world and prepare it for usage.

`POPULATE CARRIER` aid_1 `WITH FIGHTER[S]` aid_n+

Populates carrier agent aid_1 with fighter agents aid_n. Only fighters created without an initial airborne state may be added.

`POPULATE WORLD WITH` aid_n+

Populates the world with fighter, tanker, and carrier agents aid_n.

`COMMIT`

Locks the membership in the world. No further creational or structural commands are allowed. This step permits late validation checks and optimizations.

5.3.4 Behavioral commands

The 23 behavioral commands operate on the agents to perform meaningful actions, for example:

`DO` *aid* `BARRIER UP | DOWN`

Instructs carrier *aid* to raise or lower its blast barrier.

`DO` *aid* `CATAPULT LAUNCH WITH SPEED` *speed*

Instructs carrier *aid* to launch its catapult at speed *speed* with the fighter attached earlier.

`DO` *aid* `SET SPEED` *speed*

Instructs fighter, tanker, or carrier *aid* to assume *speed* knots.

`DO` *aid* `SET ALTITUDE` *altitude*

Instructs fighter or tanker *aid* to assume *altitude* feet.

`DO` *aid* `SET HEADING` *course* `[LEFT | RIGHT]`

Instructs fighter, tanker, or carrier *aid* to assume heading *course* degrees in the direction indicated. If no direction is indicated, choose the shortest turning distance.

The world is itself considered an agent and accepts commands for wind conditions (direction and speed) that add some interesting aspects to the simulation for more complex analysis.

Seven behavioral commands interact with agents in ways that are not acceptable in the real world. Their role is to set up the initial conditions in an experiment before it starts generating real data. For example, instead of taking off and flying to a location for a particular test, this command allows a fighter to start there exactly as specified:

`@DO` *`aid`* `FORCE COORDINATES` *`coordinates`* `[ALTITUDE` *`altitude`*`] HEADING` *`course`* `SPEED` *`speed`*

Forces fighter, tanker, or carrier *`aid`* to instantaneously assume coordinates *`coordinates`*, heading *`course`*, speed *`speed`*, and optionally altitude *`altitude`*.

5.3.5 Metacommands

The 10 metacommands interact with the architecture and the views to manage their execution, for example:

`@CLOCK` *`rate`*

Sets the system clock speed to *`rate`* ticks per second.

`@CLOCK PAUSE | RESUME | UPDATE`

Instructs the simulation to pause or resume the system clock, respectively, or to force it to advance a tick when the clock is paused.

`@RUN` *`string`*

Loads a text file with commands, one per line, and executes them in order.

`@WAIT` *`rate`*

Waits *`rate`* ticks before executing the next behavioral command.

`@CREATE VIEW` *`wid`* `ASPECT FRONT | SIDE | TOP AT COORDINATES` *`coordinates`*

Creates a new window *`wid`* with a front, side, or top perspective centered on coordinates *`coordinates`*.

`@SYNC VIEW` *`wid`* `ON` *`aid`*

Locks window *`wid`* onto agent *`aid`* and keeps the agent centered at all times.

5.4 View

In this model-view-controller architecture, the view plays the output role in many ways beyond just graphics. It presents results in terms of data and information, which in combination with students' understanding of their experiments leads to knowledge.

5.4.1 Log views

As the architecture manages all activities on communication buses, it maintains a rich record of what happened in a simulation. The lowest level provides the most details by exporting directly to an Excel spreadsheet, as in Figure 5.

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: Excel Bus Log View

Likewise, all components export their copious state data in a uniform format, as in Figure 6.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
1	step	time	pitch	roll	yaw	x	y	z	vx	vy	vz	course	thrust	airspeed	ci	vpitch	vroll	vyaw	apitch	aroll	ayaw	ax	ay	az	airspeed
2	0	0	0	0	90	2000000	2000000	500	150	0	0	0	1160	86.8626	0	0	0	0	0	0	0	0	0	0	0
3	1	0.02	0	0	90	1999996.9949	2000000	499.9871	150.257	0	-0.6435	270	1160	89.0156	0.2	0	0	0	0	0	0	12.9489	0	-32.1774	7.6527
4	2	0.04	-0.0011	-0.0011	90.0029	1999993.5914	2000000.0003	499.9732	150.1738	0.0152	-0.6965	89.9942	1160	88.9665	0.5464	-0.0658	-0.0557	0.1475	-2.7919	-2.7854	7.3729	-4.1584	0.7609	-2.6498	-2.4565
5	3	0.06	-0.0034	-0.001	90.0029	1999990.9896	2000000.0003	499.9582	150.0912	-0.0002	-0.7479	89.9961	1160	88.9177	0.5483	-0.1125	0.0075	-0.0007	-2.8915	3.1604	-7.408	-4.1297	-0.7686	-2.5727	-2.4392
6	4	0.08	-0.0068	0.0003	90	1999987.9894	2000000	499.9423	150.0092	-0.0154	-0.798	89.9981	1160	88.8603	0.5499	-0.1696	0.0625	-0.1481	-2.8582	2.7511	-7.3679	-4.1017	-0.7604	-2.5052	-2.4222
7	5	0.1	-0.0113	0.0003	90	1999984.9908	2000000	499.9253	149.9277	0	-0.847	89.9999	1160	88.8212	0.5514	-0.2272	-0.0011	-0.0004	-2.8772	-3.1913	7.3816	-4.0735	0.7659	-2.4471	-2.4053
8	6	0.12	-0.017	-0.0009	90.0029	1999981.9939	2000000.0003	499.9074	149.8468	0.0151	-0.8949	90.0012	1160	88.7734	0.5528	-0.2849	-0.0565	0.1464	-2.885	-2.7712	7.3425	-4.046	0.7576	-2.3978	-2.3886
9	7	0.14	-0.0239	-0.0007	90.0029	1999978.9998	2000000.0003	499.8886	149.7664	-0.0002	-0.9421	90.0012	1160	88.7259	0.554	-0.3426	0.0065	-0.0011	-2.8864	3.1531	-7.3757	-4.0184	-0.7654	-2.3571	-2.372
10	8	0.16	-0.0319	0.0005	89.9999	1999976.0048	2000000	499.8688	149.6866	-0.0153	-0.9886	89.9989	1160	88.6788	0.5551	-0.4002	0.0615	-0.1478	-2.8773	2.746	-7.3368	-3.9913	-0.7573	-2.3244	-2.3655
11	9	0.18	-0.041	0.0005	89.9999	1999973.0127	2000000	499.8481	149.6073	-0.0001	-1.0346	89.9989	1160	88.632	0.5561	-0.4574	-0.0019	-0.0008	-2.8629	-3.1682	7.3511	-3.964	0.7626	-2.2995	-2.3391
12	10	0.2	-0.0513	-0.0007	90.0028	1999970.0221	2000000.0003	499.8265	149.5286	0.015	-1.0802	90.0012	1160	88.5856	0.5569	-0.5142	-0.0571	0.1455	-2.8396	-2.761	7.313	-3.9372	0.7544	-2.2819	-2.3226
13	11	0.22	-0.0627	-0.0006	90.0028	1999967.0331	2000000.0003	499.804	149.4504	-0.0003	-1.1256	90.0012	1160	88.5395	0.5576	-0.5704	0.0058	-0.0014	-2.8119	3.1473	-7.3445	-3.9103	-0.7623	-2.2712	-2.3066
14	12	0.24	-0.0752	0.0006	89.9998	1999964.0456	2000000	499.7806	149.3727	-0.0153	-1.171	89.9989	1160	88.4937	0.5582	-0.626	0.0607	-0.1476	-2.7761	2.7426	-7.3066	-3.8838	-0.7542	-2.2671	-2.2904
15	13	0.26	-0.0888	0.0006	89.9998	1999961.0597	2000000	499.7563	149.2956	-0.0002	-1.2163	89.9989	1160	88.4482	0.5588	-0.6807	0.0025	-0.0011	-2.7372	-3.1585	7.3217	-3.8571	0.7595	-2.2691	-2.2742
16	14	0.28	-0.1035	-0.0006	90.0027	1999958.0754	2000000.0003	499.731	149.2189	0.0149	-1.2619	90.0012	1160	88.403	0.5592	-0.7345	-0.0576	0.1446	-2.6913	-2.7543	7.2845	-3.8308	0.7514	-2.2769	-2.2581
17	15	0.3	-0.1193	-0.0005	90.0027	1999955.0925	2000000.0003	499.7049	149.1429	-0.0003	-1.3077	90.0012	1160	88.3582	0.5595	-0.7974	0.0053	-0.0017	-2.643	3.143	-7.3144	-3.8042	-0.7593	-2.2902	-2.2419
18	16	0.32	-0.1361	0.0007	89.9997	1999952.1112	2000000	499.6778	149.0673	-0.0153	-1.3539	89.9989	1160	88.3136	0.5597	-0.8392	0.0601	-0.1473	-2.5886	2.741	-7.2772	-3.7781	-0.7512	-2.3084	-2.2259
19	17	0.34	-0.1538	0.0007	89.9997	1999949.1313	1999999.9999	499.6498	148.9923	-0.0002	-1.4005	89.9989	1160	88.2695	0.5599	-0.8898	-0.0029	-0.0014	-2.5328	-3.1518	7.2933	-3.7516	0.7565	-2.3314	-2.2097
20	18	0.36	-0.1726	-0.0005	90.0026	1999946.153	2000000.0002	499.6208	148.9177	0.0148	-1.4477	90.0013	1160	88.2256	0.5599	-0.9393	-0.058	0.1437	-2.4719	-2.7505	7.2568	-3.7255	0.7485	-2.3587	-2.1936
21	19	0.38	-0.1924	-0.0004	90.0025	1999943.1761	2000000.0002	499.5909	148.8438	-0.0004	-1.4955	90.0013	1160	88.182	0.5599	-0.9875	0.0049	-0.002	-2.4103	3.1405	-7.2851	-3.6991	-0.7563	-2.39	-2.1773
22	20	0.4	-0.2131	0.0008	89.9996	1999940.2007	1999999.9999	499.5601	148.7703	-0.0153	-1.544	89.9989	1160	88.1388	0.5599	-1.0343	0.0597	-0.1469	-2.3442	2.7413	-7.2487	-3.6731	-0.7483	-2.425	-2.1612
23	21	0.42	-0.2347	0.0008	89.9996	1999937.2267	1999999.9999	499.5282	148.6974	-0.0003	-1.5932	89.999	1160	88.0959	0.5597	-1.0799	-0.0033	-0.0016	-2.2784	-3.1479	7.2658	-3.6466	0.7536	-2.4633	-2.1448
24	22	0.44	-0.2571	-0.0004	90.0024	1999934.2542	2000000.0002	499.4953	148.625	0.0147	-1.6433	90.0013	1160	88.0533	0.5595	-1.1241	-0.0583	0.143	-2.2088	-2.7495	7.23	-3.6205	0.7457	-2.5047	-2.1286
25	23	0.46	-0.2805	-0.0003	90.0024	1999931.2832	2000000.0002	499.4614	148.5531	-0.0004	-1.6943	90.0013	1160	88.0111	0.5593	-1.1669	0.0045	-0.0022	-2.1399	3.1397	-7.2568	-3.5941	-0.7533	-2.5469	-2.1122
26	24	0.48	-0.3047	0.0009	89.9994	1999928.3135	1999999.9999	499.4265	148.4817	-0.0153	-1.7462	89.999	1160	87.9692	0.559	-1.2082	0.0594	-0.1466	-2.068	2.7433	-7.2209	-3.568	-0.7454	-2.5955	-2.0958
27	25	0.5	-0.3296	0.0008	89.9994	1999925.3453	1999999.9999	499.3905	148.4109	-0.0003	-1.7991	89.999	1160	87.9276	0.5587	-1.2482	-0.0035	-0.0018	-1.9975	-3.1461	7.2393	-3.5415	0.7509	-2.6444	-2.0792
28	26	0.52	-0.3553	-0.0004	90.0022	1999922.3785	2000000.0002	499.3535	148.3406	0.0146	-1.853	90.0013	1160	87.8863	0.5583	-1.2867	-0.0585	0.1423	-1.9245	-2.7507	7.204	-3.5153	0.743	-2.6952	-2.0627
29	27	0.54	-0.3818	-0.0003	90.0022	1999919.4131	2000000.0002	499.3153	148.2708	-0.0004	-1.9079	90.0013	1160	87.8454	0.5579	-1.3238	0.0043	-0.0023	-1.8533	3.1406	-7.2294	-3.4888	-0.7505	-2.7476	-2.046
30	28	0.56	-0.409	0.0009	89.9993	1999916.4491	1999999.9999	499.276	148.2016	-0.0153	-1.964	89.999	1160	87.8048	0.5575	-1.3594	0.0592	-0.1462	-1.7802	2.7469	-7.1941	-3.4625	-0.7426	-2.8016	-2.0294
31	29	0.58	-0.4369	0.0008	89.9992	1999913.4864	1999999.9999	499.2356	148.1328	-0.0003	-2.0211	89.999	1160	87.7646	0.557	-1.3935	-0.0037	-0.0019	-1.7094	-3.1464	7.2136	-3.4359	0.7482	-2.8567	-2.0126
32	30	0.6	-0.4654	-0.0004	90.0021	1999910.5251	2000000.0002	499.194	148.0646	0.0145	-2.0794	90.0013	1160	87.7247	0.5566	-1.4263	-0.0588	0.1417	-1.6371	-2.7538	7.1788	-3.4095	0.7404	-2.9128	-1.9958
33	31	0.62	-0.4946	-0.0003	90.002	1999907.5652	2000000.0001	499.1513	147.997	-0.0005	-2.1386	90.0013	1160	87.6851	0.5561	-1.4576	0.0041	-0.0024	-1.5673	3.1429	-7.2029	-3.3828	-0.7477	-2.9698	-1.9788
34	32	0.64	-0.5243	0.0009	89.9991	1999904.6068	1999999.9998	499.1073	147.9299	-0.0153	-2.1993	89.999	1160	87.6459	0.5556	-1.4876	0.0591	-0.1458	-1.4964	2.752	-7.168	-3.3563	-0.7398	-3.0273	-1.9618
35	33	0.66	-0.5546	0.0008	89.9991	1999901.6493	1999999.9998	499.062	147.8633	-0.0004	-2.261	89.999	1160	87.607	0.555	-1.5161	-0.0039	-0.002	-1.4287	-3.1485	7.1888	-3.3295	0.7457	-3.0853	-1.9447
36	34	0.68	-0.5855	-0.0003	90.0019	1999898.6934	2000000.0001	499.0156	147.7972	0.0144	-2.3239	90.0013	1160	87.5694	0.5545	-1.5433	-0.059	0.1411	-1.36	-2.7586	7.1544	-3.3029	0.7379	-3.1435	-1.9276
37	35	0.7	-0.6169	-0.0003	90.0018	1999895.7387	2000000.0001	498.9678	147.7317	-0.0005	-2.3879	90.0013	1160	87.5302	0.5539	-1.5692	0.0039	-0.0024	-1.2945	3.1467	-7.1774	-3.276	-0.745	-3.2018	-1.9103
38	36	0.72	-0.6488	0.0009	89.9989	1999892.7854	1999999.9998	498.9187	147.6667	-0.0152	-2.4531	89.999	1160	87.4923	0.5534	-1.5938	0.0591	-0.1453	-1.2285	2.7583	-7.1429	-3.2494	-0.7372	-3.26	-1.893
39	37	0.74	-0.6811	0.0008	89.9989	1999889.8333	1999999.9998	498.8684	147.6023	-0.0004	-2.5195	89.999	1160	87.4548	0.5528	-1.6171	-0.004	-0.002	-1.168	-3.1521	7.1649	-3.2223	0.7432	-3.318	-1.8756

Figure 6: Excel State Log View

5.4.2 Graph view

The text form is highly informative, but not remotely intuitive. However, with strategic selection of its fields, much of its data easily transforms into information through basic Excel graphs. Figure 7 shows an example of evaluating the performance characteristics of a fighter through a variety of maneuvers.

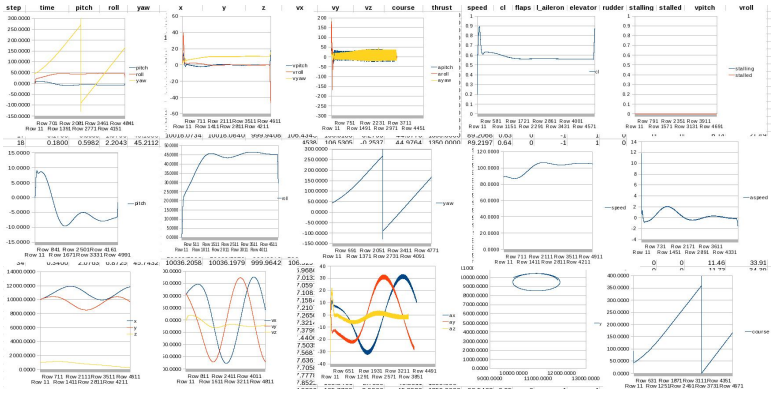


Figure 7: Excel Graph View

5.4.3 Two-dimensional view

The Excel views are static because they present the results after a simulation is complete. The dynamic view is built into the architecture with basic Java Swing two-dimensional graphics. It provides a real-time resizable view with pan and zoom capabilities, among other useful features like metadata on the glyphs for call sign and speed. Figure 8 shows an example of top, side, and front perspectives.

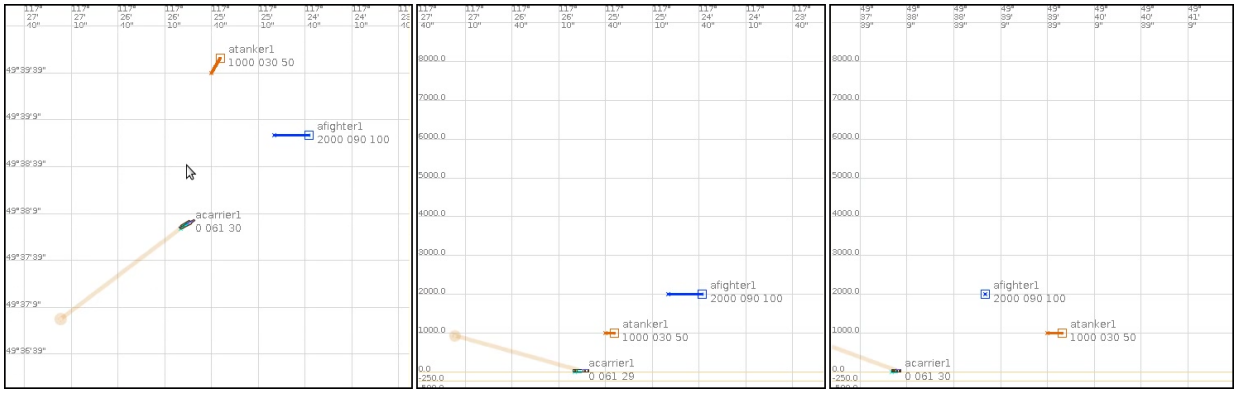


Figure 8: Two-Dimensional Views

5.4.4 Three-dimensional view

The three-dimensional view is also a dynamic, real-time perspective. It is an external tool that has been used in various forms in many of the author's other pedagogical applications, research projects, and industry work.[29] It is not specifically designed for this work, so it does not represent many of the nuances like tailhooks and refueling booms. However, for interactive three-dimensional visualization of the big picture of a simulation, it provides a wealth of intuitive information, as in Figure 9. Gnuplot is also supported as an export format for three-dimensional static mathematical analysis.

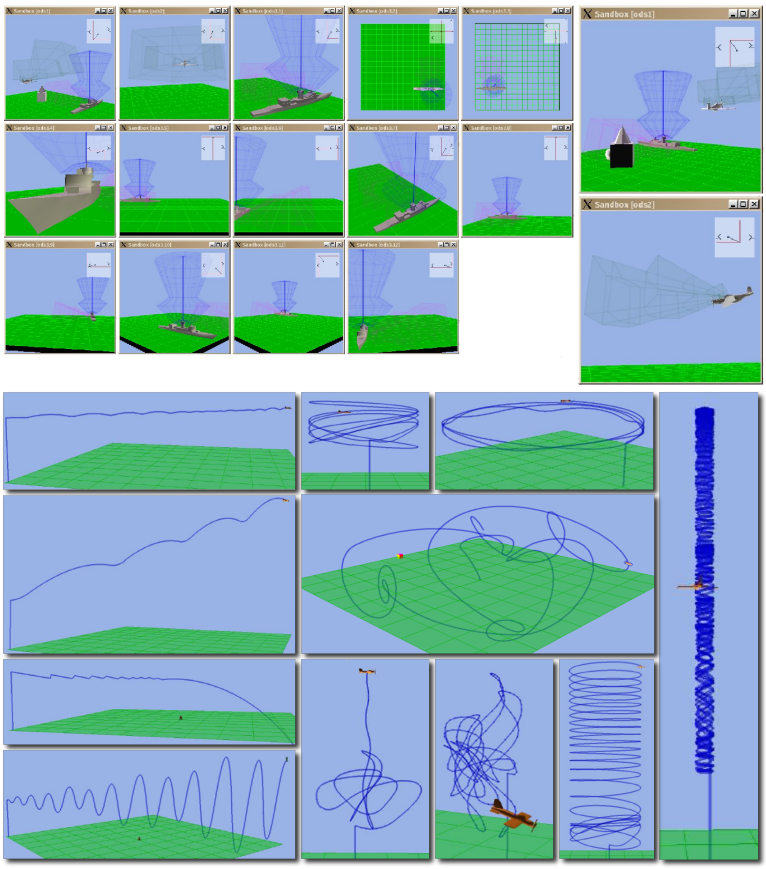


Figure 9: Three-Dimensional Views

6 Methodology

The discussion so far has included numerous anecdotal examples of the methodology in particular contexts. This section threads the entire process into a single, coherent summary. Although each task had an associated point value, quantitative analysis would be less meaningful than qualitative observations because this systems-thinking approach to software engineering is very holistic. The following organization mirrors the software development process in general, not necessarily the chronological order of the tasks. Some had overlapping parts and followups depending on the circumstances, for example.

6.1 Analysis

The intent of the analysis tasks was to establish a basis for decisions in the later tasks. The students did them individually because in the typical team environment, stronger students tend to do the majority of the work and consequently learn more, while weaker students hide and get even weaker, but all generally earn the same grade. While these tasks did not prevent this distribution, they at least exposed the performance differences.

- *Background Survey*: 27 questions about themselves, their career interests, perceived strengths and weaknesses, familiarity with the problem domain, etc. (One student had served on an aircraft carrier and clearly had a huge advantage in understanding how one operates. Others had seen *Top Gun*, which was arguably a liability.)
- *Notion of Systems*: 10 questions about the general features of systems of wildly diverse kinds (e.g., computer, weather, geological) to establish a basis for discussing systems thinking.
- *Project Grounding and Conceptualization*: 66 project-related terms with an example of their data, control, and behavior elements to make sure everyone had a common understanding.
- *Project SRS Elicitation*: a hypothetical software requirements specification (SRS) based on representative examples of what the students would like to see in the project if they were the customer. It organized user stories, use cases, W^{5}H questions, requirements, and specifications into a cross-referenced document resembling an outline.

The outcomes were overwhelmingly positive, with the most deductions coming from incomplete or nonsubmitted work. Negative commentary revolved around the perception of analysis being "busy work" unrelated to the project. (It would be interesting to assess why students think the instructor would feel a need to assign something of no value just to waste time.) Unsurprisingly, some teams with these students later commented to the effect that "they didn't seem to know what was going on with even the most basics parts of the project."

6.2 Design

The design tasks could arguably qualify as analysis because the students did not actually make anything themselves; instead, they considered the instructor's solution as the foundation. In other words, this process involved making sense of an existing design with the understanding that they would have been tasked to produce something similar themselves if time and workload had permitted it. This backwards approach accommodated exposing students to a problem larger than they could had accomplished themselves. For the same reason as in the analysis tasks, they were an individual effort. All three were related to a critical subset of the architecture that the students would be extending themselves or working with next.

- *Project Static Architectural Analysis*: from the provided Javadoc documentation, draw the class diagrams, including inheritance and intercommunication.
- *Project Dynamic Architectural Analysis*: from the provided source code, one-step through a single representative operation to understand how the architecture does its job, and how good object-oriented programming functions.
- *Project Behavioral Architectural Analysis*: propose how to add new functionality to the code, but do not actually do it. This thought exercise forced the students to think about problem solving without having their familiar trial-and-error coding environment.

6.3 Implementation

The implementation tasks involved two facets. The first was an individual task as a proof of concept early in the development process. After the *Project Grounding and Conceptualization* task, the students were getting antsy to write code. The *Project Proof-of-Concept Support Library* provided the opportunity to write a prototype solution for a simple top-perspective map viewer that displayed a grid for latitude and longitude and a circle (representing any agent) at a

particular position. The programmatic details were for the most part a basic *Hello World* graphics program available anywhere online. The true challenge was in reading, understanding, and properly acting on the simple requirements and specifications in English. The pretask gave them the opportunity to pose any questions and have them answered or clarified. Most questions turned out to be procedural, like how many points was it worth. A significant number of responses indicated that "everything was clear, this should be easy." The results were very telling: every solution had longitude going in the wrong direction on the horizontal axis because x normally increases to the right in the familiar math world. In the western hemisphere of the real world, however, the opposite is true. The first Google Images hit on "latitude and longitude in the US" would have prevented this huge error, but nobody took the initiative because they did not even think that their perception of reality could differ from actual reality. Despite this eye-opening experience, a majority considered the exercise "unfair" because the instructor "should have told them how to do it correctly." Somehow the point of the exercise — that they are responsible for their own decisions, and never assume anything — was lost. Nevertheless, most did pay more attention from that point onward.

The second facet was team-based. First, the students had to build the glyph loader in Section 5.2.4. The solution entailed a single class that used basic file input and output from the introductory programming courses, so again, the technical details were not the challenge. This time, however, they generally used the opportunity to pose questions wisely and consequently produced good solutions.

Finally, they had to build the parser for a subset of the commands in Section 5.3.1. Again, the solution entailed basic string processing from the earlier courses. The intent of this course is not to introduce anything substantially new, but to make better use of what the students already know. Unlike the loader, however, there was significantly more code involved, which required better planning on how to distribute the effort among the three team members and integrate their contributions. The use of GitHub to manage this process is outside the scope of this paper, but it is worth mentioning as part of the overall software development process that the students experienced. As with almost everything else here, each new aspect introduced its own learning curve, which forced students to learn to cope. Some managed fine on their own, some required assistance, and some simply complained. This distribution is typical throughout all courses. No approach satisfies everyone, but this one tries to strike a practical balance.

6.4 Testing and evaluation

Determining correctness (testing) and evaluating performance (optimization) require three critical components: (1) the expected results, (2) the actual results, and (3) a meaningful way to compare them. Students often lack one, two, or even all three, but still think that they are testing. While the act of acquiring such data is necessary, alone it is not sufficient to make sense of the data. Students must have a firm understanding of the subject matter and its context within the problem domain. The pedagogical approach here provides endless opportunities to ground the programmatic exercises to reality in order to help students develop and improve their critical-thinking skills in multidisciplinary computer science.

To this end, the students had to conduct 42 experiments to demonstrate representative aspects of the system; e.g., taking off, refueling, and landing. 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 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.
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 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 text-based input mode was very convenient because students could store each test in a separate script file, as in Figure 10, and paste it into the report for requirement (3). Furthermore, strategic use of the `@RUN` command allowed scripts to call other scripts, which drastically reduced the amount of repeated code to set up the parts in common. Cross-references were permitted in the report to reduce duplication. This exercise demonstrated that extensive, meaningful testing and reporting need not be onerous. In fact, a few found it to be a lot of fun.

```
// @run '/home/dtappan/carrier/script6.txt'

// CARRIER
define carrier tcarrier1 speed max 10 delta increase 20 decrease 30 turn 40 layout
  '/home/dtappan/carrier/coordinates.csv'
define catapult tcatapult1 origin +4.4:-141 azimuth 0 length 400 acceleration 20
  limit weight 50 speed 45 reset 20
define barrier tbarrier1 origin +2.4:-119.5 azimuth 0 width 60 time 50
define trap ttrap1 origin -14.9:+256.3 azimuth -8.5 width 400 limit weight 50
  speed 35 miss 30
define ols_xmt tolsxmt1 origin -14.9:+261.3 azimuth 172 elevation 5 range 10000
diameter 500

create catapult acatapult1 from tcatapult1
create barrier abarrier1 from tbarrier1
create trap atrap1 from ttrap1
create ols_xmt aols_xmt1 from tolsxmt1
create carrier acarrier1 from tcarrier1 with catapult acatapult1 barrier
  abarrier1 trap atrap1 ols aols_xmt1 at coordinates 49*39'00"/117*26'00"
  heading 235 speed 0

populate world with acarrier1

do acarrier1 barrier up
do acarrier1 barrier down
```

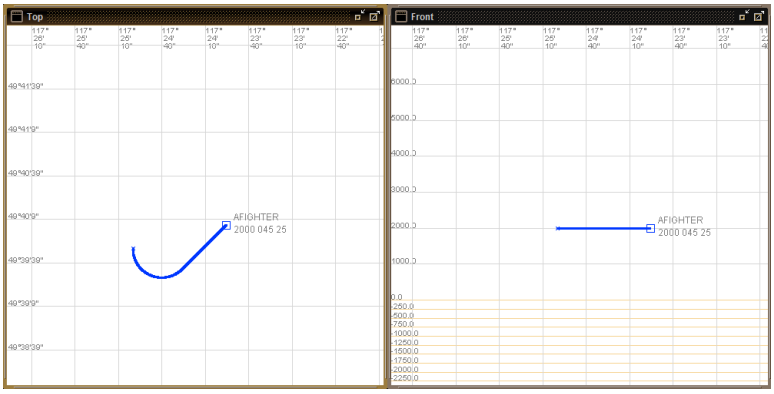
Figure 10: Test script

The majority of the tests evaluated correctness — that individual parts of the solution functioned as specified — which corresponds to software verification and validation. This process appeared to be straightforward, but the author intentionally injected several obscure bugs into the solution. Many students, accustomed to expecting the solution to be correct, failed to notice when it was not. Such complacency is dangerous in testing.[4] The following is a team's complete example:

Test: Heading Change Left

1. This test is to verify the fighter's ability to gradually adjust its heading, to see how quickly the fighter changes heading and its ability to carry out certain directions of change (i.e., counterclockwise).
2. The fighter begins at the state described in the shared initial conditions with an initial heading of 180.
3. (after commands from test A.1 up to, but not including, "create fighter")

```
create fighter AFIGHTER from TFIGHTER with ols AOLS boom ABOOM
tailhook ATAILHOOK tanks ATANK at coordinates 49*39'50"/117*25'00"
altitude 2000 heading 180 speed 25
populate world with AFIGHTER
commit
do AFIGHTER set heading 045 LEFT
```

4. The expected result is that the fighter will begin turning counterclockwise at a turn rate of 15 and speed of 25 until it reaches a heading of 45 degrees, after which it will continue moving at a speed of 25 with a heading of 45 degrees.
5. 

6. The results were as expected.
7. Other tests could include changing heading at different speeds and altitudes, different start and end headings, testing both counterclockwise and clockwise motion, as well as testing no turn direction given to see the program calculate the shortest turn direction.

A downside to this reporting process is that several teams clearly wrote test descriptions for requirement (4) *after* running the tests. While this approach is definitely easier, it defeats the purpose of testing by moving the goalposts to wherever the ball ended up. Despite forcing the students through a disciplined development process, many still found creative ways to reduce their effort. Lectures on ethics and professional responsibility generally fell on deaf ears because students know there are no real consequences to such behavior. In other offerings of this class, the author required test descriptions to be submitted *before* the tests were actually run, but due to the timing, it was not practical here.

All 23 tests of this type required a context of a single agent — fighter, tanker, or carrier — in increasingly complex scenarios. This approach taught the students about partitioning tests into representative combinations and permutations for appropriate breadth and depth of coverage. For example, the following tests apply to a fighter and assume the same initial conditions:

- *Acceleration*: Accelerate to speed 100.
- *Climb*: Climb to altitude 5000.
- *Heading*: From an initial heading of 180, change heading to 45 left.
- *Climb and heading*: Climb to altitude 5000 while changing heading 135 degrees left.
- *Acceleration and climb*: Accelerate to speed 50 while climbing to altitude 10000.
- *Acceleration and heading*: Accelerate to speed 50 while changing heading 135 degrees left.
- *Acceleration, climb, and heading*: Accelerate to speed 50 while climbing to altitude 6000 and changing heading 180 degrees right.

A set of 21 tests evaluated performance — that the solution modeled the real world appropriately — which corresponds to model verification and loosely to software certification and accreditation. These tests all required a context of two paired agents: fighter–tanker, fighter–fighter, fighter–carrier, tanker–tanker, tanker–carrier, and carrier–carrier. For example:

- *Refueling 1*: Refuel a fighter from a tanker. The fighter needs to couple with the tanker and maintain altitude, speed, and heading.
- *Refueling 2*: Refuel as in (1), but have the tanker pull ahead slowly until the coupling breaks.
- *Refueling 3*: Refuel as in (1), but have the tanker turn away until the coupling breaks.
- *Refueling 4*: Refuel as in (1), but have the fighter fall back slowly until the coupling breaks.
- *Refueling 5*: Refuel as in (1), but have the fighter turn away until the coupling breaks.
- *Launch stationary*: Create a carrier (with speed 0) with a fighter on it. Launch the fighter.
- *Launch moving*: Create a carrier (with speed 10) with a fighter on it. Launch the fighter.
- *Recovery stationary trap*: Create a carrier and an airborne fighter. Land the fighter.
- *Recovery stationary miss*: Create a carrier and an airborne fighter. Land the fighter but have it fail to trap.

The final and smallest set of just two tests nominally evaluated higher-level thinking skills. Here the system was used as intended as a training tool to investigate the problem domain. Students (within reason, given their limited background) had to use it as a subject-matter expert would to develop appropriate strategic and tactical actions for certain goals; e.g., the fastest way to refuel. They found this part very entertaining:

- *Once around the pattern*: Launch a fighter from a carrier, have it briefly refuel with a tanker, land back on the carrier, and launch again.
- *Kamikaze*: Add two carriers (one named Godzilla), two fighters (one initially aboard the other carrier, the other airborne), and two tankers. Cause all airplanes to crash into Godzilla at the same time.

7 Results and discussion

Ironically, this extensive framework for testing and evaluating the problem domain does not lend itself to straightforward evaluation. As is typical for pedagogical studies in the classroom, it was very difficult to control for the environment. Grades alone also do not necessarily correlate with learning, and for practical reasons, there is also no comparison with a baseline measurement from a control group who solved the same problem differently. As a result, a major component of the one-quarter course (over 11 weeks and 50 contact hours) with 34 students (on 12 teams)

involved collecting continual feedback and insightful metainformation. This process itself mirrors Agile software engineering in action because at the end of each step (or "sprint") every four calendar days, the development team repeated the same iterative evaluation:[1]

1. Review the objectives that the team expected to have completed at this point.
2. Determine whether they were met. If so, then make note of the conditions and actions that achieved this positive outcome so as to apply this knowledge and experience analogously to similar situations in the future. If not, then evaluate what happened and why, propose and evaluate corrective action, and schedule it for completion.
3. Establish the objectives for the next sprint.

This process closes the loop of the assessment of progress. In particular, it reduces the prevalence of endless disconnects between the stages of software development back in Figure 1. It helps to reduce gaps, overlaps, and inconsistencies between the plan, its execution, and the results. For example, every requirement must have a corresponding solution, and every solution must have a corresponding requirement. There should be no mystery about the origin or purpose of any part of the solution that was under the students' control. (Subsequent offerings on this course introduced a web-based tracking tool that automated much of the review and evaluation process for the instructor.)

Continual assessment resulted in a significant breadth and depth of objective and subjective measures including anecdotal observation, individual contributions from a background knowledge survey, 12 assignments, and 10 weekly assessments with self-reflections, as well as individual and team contributions from 18 project status reports, a project reflection and self-evaluation (a final combined assessment, self-reflection, and status report), a team-member evaluation, and a course evaluation. While these items were required and contributed to the final grade, they were not graded on content per se because there were no right or wrong answers. Rather, they supplied a vast amount of insight into the beliefs, motivations, actions, etc. of students, which would otherwise have remained inaccessible to both the instructor and the students themselves. Many of these observations and conclusions have already been mentioned in context throughout this paper. Most telling in quantitative terms is that 86% of the students stated that the architecture permitted them to do something they would never have been able to do on their own. (It would be very interesting to discover how the other 14% thought they could do it their own way, but there was no relevant assessment measure to make even a guess.) Furthermore, 90% indicated that the test report directly contributed to a better understanding of what the code was actually doing, whereas they otherwise would have had far less confidence in it. Overall, the students rated the project 4.6 out of 5 (excellent).

The weekly assessments required each student to respond to the following questions in a web form. The anonymized aggregated results were available to everyone as a resource for reflecting on how perceptions agree or disagree.

- Enter a brief description (roughly eight lines) of your interpretation of what you learned this week. Consider why it is likely to be important in our field and how you might use it throughout your other courses and career. If appropriate, connect it with things you already know. Do not just repeat the lecture contents.
- For the lecture topics, what was the easiest to understand and why?

- What was the hardest to understand and why?
- For the current task, what is the easiest to understand and why?
- What is the hardest to understand and why?

These questions required selecting one option from *very weak*, *weak*, *ok*, *strong*, and *very strong* to indicate the perception of the following:

- Overall comprehension of the lecture material from this week.
- Overall level of difficulty with the lecture material from this week.
- Overall level of difficulty with the task material from this week.
- Pace of the course (lecture and tasks) this week.

The individual reports required students to indicate the status of their activities in terms of Agile criteria 1–3 above. Sprints were short, so there should not be much activity.

The team reports required them collectively to articulate the following:

- Consider the following four pairs of questions hierarchically. They are not the same question. If you think they are, then you are likely not using an appropriate breadth and depth of software engineering thought. This course is a practical application of the aspects of product, process, and people. We are trying to account for everything: not just to create a good product, but also to learn from the process to improve the people. Reflect on the experience of the entire team collectively over this sprint. You do not need to account for all work, just two examples that are most representative of easiest and hardest. For reference, *understand* relates to the comprehension of what needs to be done; *approach* to how you think it should be solved; *solve* to implementing the actual solution; and *evaluate* to demonstrating to yourself and your team (if applicable) that the performance of your solution is consistent with everything else in the project. Remember The Cartoon. Which aspects of the current work are the {easiest, hardest} to {understand, approach, solve, evaluate}? (The eight combinations are collapsed here to save space.)
- How far along (as a percent) do you feel you are toward the final goal? Does this pace seem likely to succeed?
- Are there any issues, concerns, or comments about the project?

Teaching software engineering involves managing expectations. Many students come to this course believing that they already know software engineering because they believe they already know programming. A basic skills test up front demonstrated that their self-perception and self-confidence are not remotely in line with their actual abilities. Nevertheless, brutal reality does not phase many of them, and they continue to resist the notion that they have anything further to learn, especially about the learning process itself; e.g., “We’re juniors in computer science. We know how to code. Just tell us what to do.” Overcoming this oppositional nature is difficult. Experience has shown that many rationalize away their poor performance by blaming the instructor or anyone or anything else but themselves. For example, one stated that “[the instructor] made us use a weird coordinate system and advanced calculus” for a standard high school algebra problem. There is an attitude prevalent among today’s students that expecting them to connect the dots themselves in the learning process is unreasonable and unfair.[19] They want the instructor to provide the dots and to connect them. Likewise, forcing them to learn

about something that they personally think is unrelated and unimportant sometimes evokes vitriol: “I thought I was taking a software engineering class. Why am I wasting my money on airplanes?” The hugely complex and amorphous multidisciplinary nature of modern software engineering and the software industry does not accommodate this attitude. Surprisingly, despite the sometimes confrontational nature of the experience, the numbers show that most students ultimately realized the value in accepting the unfamiliar approach presented here. This pattern is so consistent across offerings of this course that the author uses various quotes and data from past classes to manage expectations in the current one.

There are also two loosely longitudinal aspects to this study. First, as the many examples peppering this paper show, the author internally coordinates with instructors in at least seven other courses (including himself) to connect the material back and forth. For example, many concepts in operating systems, like scheduling, process, and memory management, directly play a role in the multiagent aspects here. Likewise, the many design patterns throughout the architecture are practical applications in action. Part of the weekly assessments and self-reflections asks the students to connect the material to their own lives and other courses. In this way, it forces them to find dots and associate them themselves. The second aspect is external and relates to the difficult-to-measure ABET program outcome (h): “Recognition of the need for, and an ability to engage in, continuing professional development.”[5] The author keeps track of comments from graduates who appreciated the experience that this course and its approach had given them. While admittedly anecdotal and self-selecting, the supply of laudatory quotes is impressive. Many former students have made comments to the effect that “you know a project is awesome when interviewers want you to tell them all about it.” Others have stated to the effect that “my new coworkers were amazed I had no work experience because your class prepared me so well.” One quote used early in the course to set the tone is: “The next time your students whine about having to do something unfamiliar and challenging, tell them this is exactly what I do now every day.”

8 Future work

As a vehicle for undergraduates to experience real-world software systems engineering, the emphasis of this work is on building the system and using it to test and evaluate itself. A second facet — using the system as intended to learn about the environment that it models — would be a great perspective in a graduate class or ones specifically dedicated to software quality assurance and modeling and simulation. To this end, incorporating a mechanism to manage sensitivity analysis would be very helpful. For example, if students wanted to determine the best speed for turning a fighter under specific conditions, they currently would need to write a script and run it many times by hand with different values to achieve an appropriate breadth and depth of coverage for statistical significance. Adding loops and conventional parameter passing to the scripts could automate this tedious process. The critical-thinking aspects of the scientific method would still be the students’ responsibility, but the execution could be automated. Similarly, incorporating probability to account for real-would variation and unpredictability could lead to a powerful stochastic simulation environment with a Monte Carlo methodology. This line of investigation could lead naturally into machine learning with tie-ins to big data by having a model simulate and optimize itself.[6]

9 Conclusion

The primary goal of this work was to provide students with multidisciplinary hands-on experience to real-world applications of software-based systems of systems. It exposed them to a much larger problem than they would otherwise have been able to investigate, and it did so in a way that was manageable for both the students and the instructor. In particular, it walked the students through the process of analysis, design, implementation, testing, evaluation, and refinement. The pedagogical foundation emphasized critical thinking and the scientific method as a formal, disciplined approach to problem solving. In combination with an extensive, student-friendly integrated framework for modeling, simulation, visualization, and analysis, it provided all the tools that students needed to translate a complex, unfamiliar problem domain to a solution domain, to show that this translation worked correctly, and how well. The overwhelmingly positive results demonstrate that this approach is effective in managing a relatively large class with varied skills, attitudes, and maturity levels.

References

1 Agile Manifesto, http://agilemanifesto.org, last accessed Mar. 14, 2016.

2 Anderson, N. and T. Gegg-Harrison. "Learning computer science in the 'comfort zone of proximal development.'" In Proc. of 44th ACM technical symposium on Computer science education, pp. 495–500, 2013.

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

4 Laplante, P. *What Every Engineer Should Know about Software Engineering*. CRC Press, 2007.

5 ABET Criteria for Accrediting Computing Programs, 2016–2017. www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-computing-programs-2016-2017, last accessed Jan. 27, 2016.

6 Alexander, R. and T. Kelly. "Combining Simulation with Machine Learning to Build Accident Models." www-users.cs.york.ac.uk/~rda/sasemas_06_paper.pdf, 2006, last accessed Jan. 28, 2016.

7 Bing, T. and E. Redish. "Symbolic Manipulators Affect Mathematical Mindsets." Am. J. Phys. Vol. 76, Nos. 4–5, April/May 2008.

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

9 Denning, P. "The Science in Computer Science." CACM, Vol. 56, No. 5, May 2013.

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

11 Google Sketchup Warehouse, 3dwarehouse.sketchup.com, last accessed Jan. 15, 2016.

12 Grosskopf, A., 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.

13 Harel, D. "Statecharts in the Making: A Personal Account." CACM, Vol. 52, No. 3, Mar. 2009.

14 Hayes, J. *The Complete Problem Solver*. 2nd Edition, Lawrence Erlbaum Associates, 1989.

15 Hunt, A. and D. Thomas. *The Pragmatic Programmer*. Addison-Wesley: Reading, MA, 2000.

16 Hunt, A. *Pragmatic Thinking and Learning: Refactor Your Wetware*. Pragmatic Bookshelf, 2008.

17 Irish, R. "Engineering Thinking: Using Benjamin Bloom and William Perry to Design Assignments." Language and Learning Across the Disciplines, Vol. 3, No. 2, 1999.

18 Johnson-Laird, P. *Mental Models*. Cambridge University Press: Cambridge, 1983.

19 Knowlton, D. and K. Hagopian (eds). "From Entitlement to Engagement: Affirming Millennial Students' Egos in the Higher Education Classroom." New Directions for Teaching and Learning, No. 135, Oct. 7, 2013.

20 McConnell, S. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press: Redmond, 2004.

21 Montagne, K. “Tackling Architectural Complexity with Modeling.” CACM, Vol. 8, No. 9, Sept. 17, 2010.

22 Neville-Neil, G. “Code Abuse.” Communications of the ACM, Vol. 10, No. 12, Dec. 5, 2012.

23 Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press: New York, 1999.

24 Pressman, R. *Software Engineering: A Practitioner’s Approach*. McGraw-Hill, 2009.

25 Rowley, J. “The wisdom hierarchy: representations of the DIKW hierarchy.” Journal of Information Science, Vol. 33, No. 2, 2007.

26 Rushkoff, D. “Why Johnny Can’t Program: A New Medium Requires A New Literacy.” www.huffingtonpost.com/douglas-rushkoff/programming-literacy_b_745126.html, last accessed Jan. 29, 2016.

27 Salmon, M. *Introduction to Logic and Critical Thinking*. Cengage Learning: Boston, 2013.

28 Sokoloski, J. and C. Banks. *Principles of Modeling and Simulation*. Wiley: Hoboken, 2009.

29 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.

30 Waldrop, M. “Why we are teaching science wrong, and how to make it right.” Nature, Vol. 523, No. 7560, July 15, 2015.