A Holistic Integrated Development Environment for Interdisciplinary Model-Based Critical Thinking and Doing in Software Systems Engineering

Application for Professional Leave AY19–20

Dan Tappan

Professor, Department of Computer Science

Table of Contents

1 Project Statement

The PI proposes to build a software tool to help integrate his rich breadth and depth of supporting STEM resources and processes into a coherent framework for use in the classroom, research, and outreach. It relies on his many years of experience in industry and academia to provide students with practical exposure to understanding and solving real-world problems in software systems engineering.

2 Purpose and Scope

The primary purpose of this work is to build a software tool that supports the execution and assessment of the principal investigator's teaching philosophy. This philosophy emphasizes STEM-related reasoning and actions to guide a disciplined process of developing real-world software in the classroom in a manageable way. EWU is a primarily undergraduate institution with heavy teaching loads and high-maintenance students. It is difficult to give them a realistic educational experience under these conditions. This work will help make the existing process more manageable, flexible, and effective. In particular, it addresses strengths and weaknesses of EWU students and leverages what they have learned before college and in other courses into a coherent, integrated learning experience.

This tool naturally extends beyond the classroom as a platform for STEM outreach because the real-world projects it showcases are appealing and intriguing. Furthermore, the projects are valuable for research and to foster collaborative academia-industry partnerships. In Fall 2020, the Computer Science and Electrical Engineering departments are moving to downtown Spokane to support the Catalyst Project for design and innovation [33]. Software systems engineering will play a key role. Specifically, the PI's efforts will be heavily focused on the local and regional aerospace industry. Washington is ranked as the best state in the country for aerospace companies [9]. The Spokane region is second in the state for their presence and fifth in the entire nation [17]. However, EWU does not currently make good use of this potential.

2.1 Problem

The current state of software development in industry is disturbing, to say the least. Poor quality is the norm. Horror stories, especially in regard to security, make headlines nearly every day. At the national level, almost every American has been directly or indirectly affected. Locally, the $100 million system under development for the Community Colleges of Spokane is a disaster [10]. In fact, 70% of software projects fail [20], and of the remainder, almost two thirds of the final product is rarely or never used [31]. A classic quote sums up this sad state of affairs [46]:

> *If builders built buildings the way programmers write programs,*
> *then the first woodpecker that came along would destroy civilization.*

At a time when the world is demanding more and more from technology, and the risks and consequences continue to grow at a staggering rate, the software industry is no longer able to do its job acceptably. This situation has reached a tipping point, such as when the entire state of Washington lost 911 service for six hours [34]. Society and the software industry itself admit that this level of performance cannot continue. Another classic statement in Figure 1, which the PI frequently refers to as "The Cartoon" in teaching, sadly captures the sheer absurdity of modern software development.

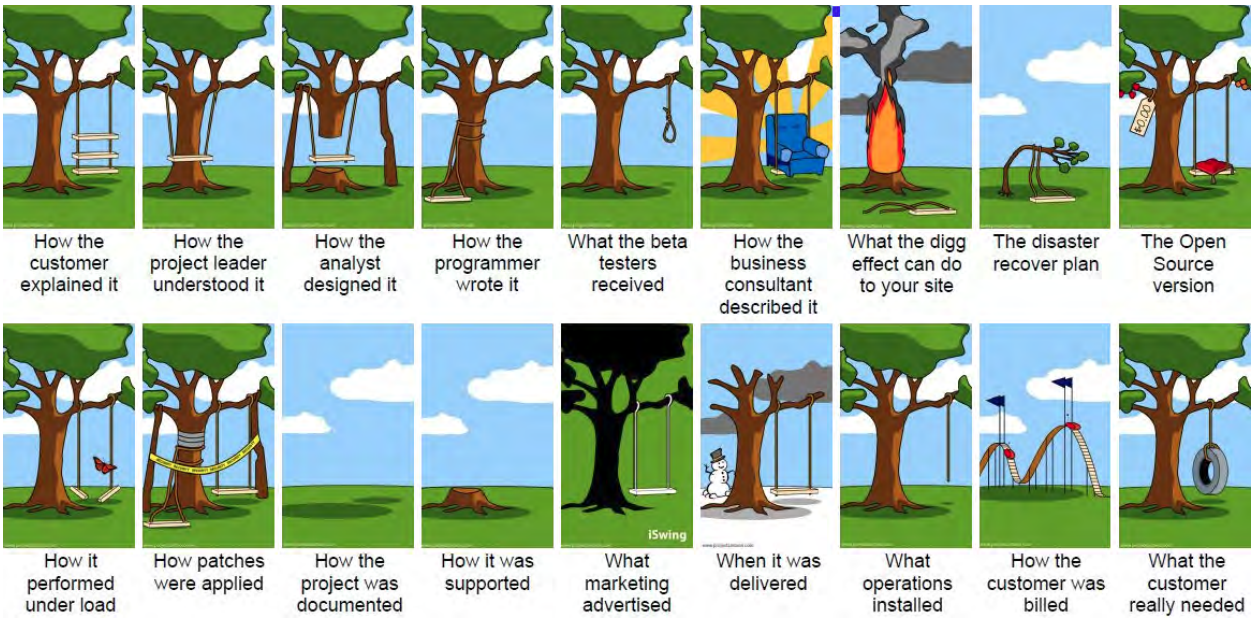


Figure 1: The Cartoon [44]

2.2 Primary Causes

There are many causes of this predicament, but the primary ones are the astronomical complexity of modern software and software engineers' inability to manage it. Even a typical program that students write as a homework assignment, with 100 lines of code (computer instructions), can theoretically translate to hundreds of trillions tests to run to verify its correctness exhaustively [27]. Today's typical car, however, has 100 million lines of code and is expected to double or even triple within a few years [8]. The solutions to today's problems[1] clearly exceed the ability of software engineers to manage them properly.

Paradoxically, as society now demands more from its software systems, software engineers are actually capable of less. Part of the reason is inherent in today's frenetic development environment, which often values expediency over quality, but the PI argues that really the difficulties start at the student level: The vast majority of difficulties in the professional environment stem from computer science students not learning to think and act appropriately in the academic one.

Computer science degree programs are hot because the job market is overflowing with high-paying positions. What used to be a relatively niche subject frequented by stereotypical "geeks" is now a mainstream degree pursued by all kinds. Programs are graduating students at a pace that does not accommodate exposing them to a sufficient breadth and depth of realistic material to develop their skills adequately. Many of the difficulties come from a lack of discipline in how they work. When the problems are smaller, students can produce solutions by any means, no matter how undisciplined (often referred to as brute force or "hacking"[2] for its lack of forethought and elegance). Unfortunately, students do not automatically grow out of such behavior when they

[1] The word "problem" is ironically problematic. From this point on, this proposal defines it as the task to solve. For the interpretation of an impediment or difficulty, it uses other terms as appropriate.

[2] This term can also mean undermining security, but here it means flailing away at a problem in hope of finding a solution.

graduate and transition to larger problems. This work aims to address underlying issues at the university level by fostering disciplined thinking and doing.

Students come into the degree program with many misconceptions about what computer science is and what computer scientists[3] do [13]. Identifying and correcting these is a critical first step. The PI has conducted many classroom surveys and studies over the years. They consistently show that most students do not know what the field is about, why they are in it, or what they want to do with it as a career. In particular, they think it is all about computer programming. Edsger Dijkstra, a pioneer in the field, sums it up well: "Computer science is no more about computers than astronomy is about telescopes" [43].

Similarly, software engineering — the process of developing software — is at least 80% about thinking, and only 20% about doing (i.e., programming). This mismatch in expectations manifests itself clearly in the PI's required junior-level introductory software engineering class, where students immediately object to the non-programming "busy work" that is actually the most important part of the development process. In fact, neglecting this part is precisely what leads to horror stories.

Industry is experiencing a similar mismatch. College is not adequately preparing graduates for the workplace, although students overwhelmingly believe the opposite [6]. Software engineers must be adroit thinkers. They must actively engage in the development and refinement of their thinking skills. Unfortunately, today's students are more likely to be passive participants in their education who blindly jump through hoops as told, check off requirements, and equate generous grades with actual knowledge, skills, and experience. The result, however, is not a software engineer. At best, this behavior produces a computer programmer (or "coder"), who is someone who takes instructions from others and follows them. Software engineers are the ones who write these instructions.

As Einstein said, "Education is not the learning of facts, but training the mind to think" [12]. Unfortunately, study after study warns that today's students are not developing the ability to think critically [4]:

- 45% did not demonstrate any significant improvement during the first two years of college.
- 36% did not after four years.
- Any improvements tended to be modest.

The numbers and methodology in this heavily cited study are still under debate, but the overall conclusion is sound. Nowhere is it more apparent than in this field: "Software engineers don't understand the *problem* they're trying to solve, and don't care to" and "[They] know how to code. The problem is *what* to code" (both original emphasis) [34]. This disconnect comes from a lack of critical thinking about problems, which leads to inappropriate solutions [25]. Further misconceptions involve not seeing — or more commonly, actively fighting against — the science in computer science and the engineering in software engineering [14]. The PI's teaching philosophy calls these weaknesses out, explains them, and helps students make improvements. This project aims to facilitate this process by addressing two broad categories of weaknesses.

[3] The terminology in this field is not consistent. For this proposal, a software engineer is a type of applied computer scientist who designs, builds, and tests software, whereas a computer scientist focuses more on theory. Despite the discipline name, very few graduates ever become true computer scientists.

First, inadequate problem-solving skills naturally result in inadequate solutions. A major contributing factor is a lack of understanding of the problem domain and the inability to make sense of it. Identifying and connecting dots is not standard thinking anymore. Classroom surveys and anecdotal evidence show that many students lack an understanding and curiosity about the world around them. This world is full of elegant solutions to complex problems, both natural and human-made. This missing world view results in a loss of conceptual integrity where solutions do not actually solve the problems, and in fact often actually create more. Even when the solutions do work, they are often inordinately complex, unreliable, and vulnerable. After all, "The purpose of software engineering is to control complexity, not to create it" (Pamela Zave).

Second, human nature and millennial mentality are large impediments to success. Today's students tend to be hesitant learners who are fearful of failure. As a result, they try to stay safely in their comfort zones instead of seeking out challenges that would broaden their knowledge, skills, and experience. Overconfidence also plays a large role [48]. There are no shortcuts to learning and applying this material. The PI's teaching philosophy intentionally disrupts these detrimental behaviors by forcing students to work on projects in unfamiliar problem domains. Without a background or preconceived notions, they are forced to follow his disciplined approach.

2.3 Proposed Mitigating Solution

This work does not expect to solve these challenges or eliminate their causes. Rather, the intent is to develop an environment that facilitates reducing their range, frequency, and severity. To this end, it has the following goals:

- Develop, deploy, and assess a software tool for classroom usage.
- Improve computational critical thinking and doing.
- Promote STEM innovation, creativity, curiosity, problem solving, etc., especially among underprivileged, underrepresented groups and others deprived of critical pre-university formative world knowledge and experiences.
- Pursue student and faculty research and collaborative efforts with industry.

Satisfying the following objectives contributes to achieving the goals. Section 5.1 provides more detail.

- Analyze the existing curricular environment of courses, students, and resources.
- Dismantle the PI's existing projects.
- Reevaluate the projects for their purpose and needs.
- Develop a model-based tool for formally defining the needs of the projects and designing their solutions.
- Rebuild the projects with the tool while documenting the process.
- Revamp relevant courses with the tool while cross-referencing their content to the projects and other resources.
- Deploy and assess the tool in the classroom.
- Use the tool for undergraduate and graduate research, the PI's research, promotional and outreach efforts, and industry collaboration.

3 Background

Background information is necessary to understand the big picture of what this work proposes to do, as well as how and why. The following sections provide an overview of software systems engineering, relevant pedagogy, and model-based solutions.

This work involves the complex intersection of multiple disciplines: computer science, electrical engineering, mechanical engineering, and the problem domain itself. It draws upon two collaborative philosophies. Both are common in industry but rare in an academic setting:

- Multidisciplinary Philosophy

  Different disciplines work together by contributing their parts to the whole; e.g., software and hardware engineers collaborating to build a physical system, where neither is very conversant in the other's field.

- Interdisciplinary Philosophy

  Different disciplines work together by integrating knowledge and considering each other's perspectives; e.g., software engineers thinking about their work as hardware engineers would.

3.1 Discipline Foundation

This section provides an overview of technical aspects. It is based on the PI's academic and industry experience in combination with these and many other resources: Software Engineering Body of Knowledge [32], Systems Engineering Body of Knowledge [35], Modeling and Simulation Body of Knowledge [24], and Project Management Body of Knowledge [28].

Software

A program is a computational representation, or model, of the real world. The term software is often used interchangeably, but it generally refers to larger programs or a collection of programs, both solving many facets of a complex problem.

All programs follow the same IPO computational model:

- **I**nput: Receive data.
- **P**rocessing: Translate data into another form.
- **O**utput: Pass data on.

Everything that is doable in the real world is doable in the virtual world of a computer model. The primary difficulty is in translating from the former to the latter, as The Cartoon demonstrates. Without careful attention, each translation loses necessary details, introduces unnecessary ones, mangles what it retains, and very little survives unscathed. The goal is to translate only what is needed — nothing less, nothing more. As Einstein and da Vinci respectively said, "Make everything as simple as possible but not simpler" and "Simplicity is the ultimate sophistication" [12]. In other words, the computer model ideally represents only the parts that are needed to solve the problem. The overall translation is an iterative, multilayer process, where each layer takes the same basic idea in a more general form and produces an equivalent but more specific form. The chain originates from a person whose description is in a human language and ultimate leads to a machine with its nearly incomprehensible language.

Software Systems Engineering

Software systems engineering is not a standard term on its own (yet). Rather, the PI uses it as a combination of systems and systems thinking, software engineering, and systems engineering. The overlap is substantial because real-world problem solving does not have such arbitrary boundaries in terminology.

Systems

A system is a collection of interconnected, interacting parts that as a whole solve a problem in the input-processing-output sense. Natural systems in the real world are typically large, complex, and convoluted because nature, while phenomenal at finding solutions to problems, does not make long-term proactive decisions for a purpose. Rather, it shortsightedly reacts to current conditions. Over time — a lot of time — its solutions often accumulate a tremendous amount of excess complexity. Understanding such systems is the realm of science. Artificial systems, on the other hand, are created by humans for a purpose. Understanding and building them is the realm of engineering. Section 3.3 below addresses both perspectives as systems thinking, which is the basis for the proposed tool.

Software Engineering

Software engineering is primarily about the development of computational solutions. It takes the chaos of the problem domain and (hopefully) translates it into order in the solution domain. It is not an exclusively linear process, but the following steps typically occur in the following order:

- Problem Analysis: Determine the customer's needs.
- Background Research: Understand the customer's world.
- Requirements Specification: Elicit and stating the features to deliver.
- Design: Translate the requirements into conceptual form.
- Implementation: Translate the conceptual form into a program.
- Testing: Determine whether the program works.
- Verification: Ensure that the program satisfies the requirements.
- Validation: Ensure that the program satisfies the customer's needs.
- Accreditation: Certify that the final product meets official standards.

Systems Engineering

Systems engineering is primarily about the development of physical engineering solutions, which include software solutions [21]. Its process is similar to that of software engineering, but it encompasses more breadth with the bigger picture of the problem and solution domains. The PI is both a software engineer and a systems engineer who comfortably operates at any level from the abstract to the concrete. This ability aligns ideally with modern systems. The virtual computer science (CS) layer is the core decision-making and control element, which communicates with the electrical engineering (EE) layer, which in turn communicates with the mechanical engineering (ME) layer, which finally interacts with the physical world. Just as any software solution is an IPO model, most engineering solutions reflect this holistic CS–EE–ME model (which also closely overlaps with the field of mechatronics). In fact, this *systems of systems* perspective is the norm, including the current internet and mobile environments and the nascent Internet of Things poised to explode into mainstream life. Students will likely spend their careers supporting it.

Modeling and Simulation

The field of modeling and simulation plays a major supporting role across software systems engineering and in science in general. The National Science Foundation states [11]:

> *Science used to be composed of two endeavors: theory and experiment.*
> *Now it has a third component: computer simulation, which links the other two.*

As discussed below, it uses the scientific method and engineering method as a formal, disciplined, executable, verifiable, and justifiable process across all stages of development. The PI has worked in this area for decades, including 10 years for the Department of Defense on huge systems engineering projects.

3.2 Classroom Foundation

The PI's approach in the classroom is to teach software systems engineering the way it works for real outside the classroom while still keeping it reasonable and manageable in a junior-level course. He develops a project in its entirety before the quarter starts. The project is far larger and more complex than the students could ever do on their own. Based on the strengths and weaknesses of the particular group, he removes parts of his solution and has the students redo them. This approach supports the guidance from the EWU Computer Science Professional Advisory Board (see Appendix B.2 #2) to expose students to working with existing solutions instead of building everything from scratch, which is typical in the classroom but rare in industry.

The PI teaches from his own perspective of QMSVA, which is in many respects an implementation of the scientific method. It is also highly flexible in that one approach can apply to solving many types of problems; e.g., class projects, master's theses, and industry projects.

- **Q**uestioning: Pose a question of interest related to the problem domain.
- **M**odeling: Build a software representation of the problem domain to investigate at the appropriate level of detail.
- **S**imulation: Execute the model under controlled conditions.
- **V**isualization: Show the results in meaningful, useful text or graphical form.
- **A**nalysis: Make sense of the results with respect to the question and generate a report.

The nine projects deployed in recent years are [40]:

1. Air traffic control with airplanes operating on the ground and in the air in various scenarios and airspace configurations [36].
2. Aircraft accident reenactment environment for creating, recreating, and analyzing events.
3. Military test range with airplanes, ships, and submarines using sensors and weapons [38].
4. Aircraft carrier operations with fighters taking off, landing, and refueling from tankers [41].
5. Spacecraft systems simulator for designing, launching, manipulating, and recovering rockets and satellites.
6. Unmanned aerial vehicle remote cockpit with instrumentation and flight data recording [37].
7. Fly-by-wire control system with networked control surfaces and external components of an airplane on a test stand [39].
8. Railroad layout manager with tracks, cars, engines, and signaling and safety systems.
9. Heavy construction equipment toolkit with sensors and electrical, mechanical, hydraulic, and pneumatic actuators.

The first seven projects are directly related to aerospace. The last two are quite different, but they demonstrate the extensibility of the PI's approach to other problem domains. Appendix D includes publications on 1, 3, 4, 6, and 7, which go into much greater detail than this proposal can.

Although each project is independent, the aerospace ones could naturally organize into systems of systems. In the global top-down view, 1 controls aircraft in 2, 3, 4. Conversely, in the local bottom-up view, 7 implements actions in 6 and 5. However, no such interoperable, integrated form is currently possible. With this tool, it could be.

The PI's investment in each project is extensive. It is justifiable because the same project serves multiple purposes: It is a classroom project, which often leads to pedagogy research, and it is a sufficient solution in the problem domain for academic research; e.g., Project 6 led to a publication on machine learning [37]. Unfortunately, time pressures have always prevented the PI from carefully documenting his development process in a way that could be integrated meaningfully into teaching. This tool will help.

3.3 Model-Based Systems Thinking Foundation

Problem solving is very much a chaotic, nonlinear process. There are many ways to summarize it. This section provides an overview in terms of what problem-solving thinking is in general, what systems thinking is, how to apply it, and how to learn from it. It relies on well-established, discipline-specific resources: Association for Computing Machinery Computer Science [5] and Software Engineering Curricula [5]; Accreditation Board for Engineering and Technology (ABET) program criteria for Computer Science, Software Engineering, and Systems Engineering [1, 2, and Appendix B.1]; the EWU Computer Science Professional Advisory Board Program Educational Objectives [Appendix B.2]; the Graduate Reference Curriculum for Systems Engineering [16]; and Microsoft Transform Science Computational Education for Scientists report [22].

Critical Thinking

The translation process in Section 3.1 fundamentally involves eliciting, understanding, telling, and retelling stories. Each layer is a story that needs to be established in the form appropriate for that layer, then correctly translated to the next layer into its own different form. Mistakes in translation lead to The Cartoon. They come from mistakes in critical thinking, including not applying critical thinking at all.

The PI's teaching philosophy relies on many well-established pedagogical sources, but the following two play arguably the largest role. The classic Bloom's Taxonomy of Educational Objectives supplies the framework [7]. It describes a bottom-up conceptual process of establishing the initial building blocks and repeatedly combining them through increasingly more complex mental manipulation into a final form of a solution. It is a conceptual framework, not a recipe, but the processes described throughout this proposal map to it, and vice versa.

- Remember: Recognize and recall facts.
- Understand: Understand what the facts mean.
- Apply: Apply the facts, rules, concepts, and ideas.
- Analyze: Break down information into components.
- Evaluate: Judge the value of information or ideas.
- Create: Combine parts to make a whole.

At the same time, but in a different way, the Data, Information, Knowledge, Wisdom (DIKW) Hierarchy in Figure 2 provides a more structured framework of establishing and connecting the dots in a story [19]:

- Data: Dots with no context.
- Information: Dots connected in a single context or relationship.
- Knowledge: Dots connected in multiple contexts or relationships.
- Wisdom: Overall understanding of what the dots are, what they do and do not do, and how they interact, etc.; i.e., a systems view.

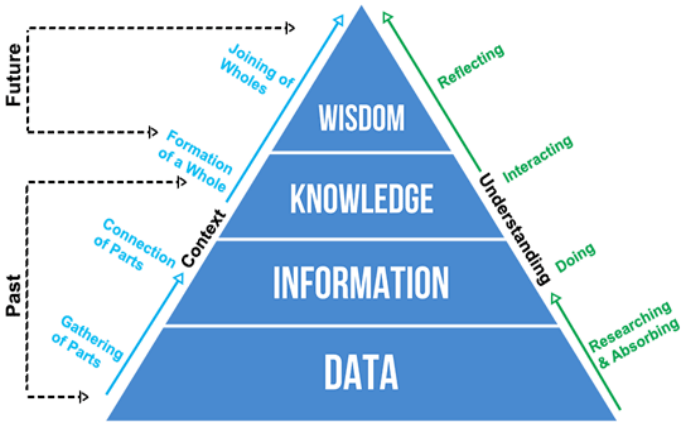


Figure 2: DIKW Hierarchy [47]

Both perspectives overlap in some respects and complement each other in others. Together they form a mental framework for defining and understanding past, present, and future aspects of the problem to solve. This foundation leads directly into multidimensional *what if* systems thinking. Neither perspective provides a recipe for thinking or doing, but together they do provide some structure and guidance for a process flow. Figure 3 shows a meaningful mapping to various kinds of knowledge, and Figure 4 walks through it. These diagrams are too detailed and complex to address here, but many of its elements correspond to parts of this proposal.

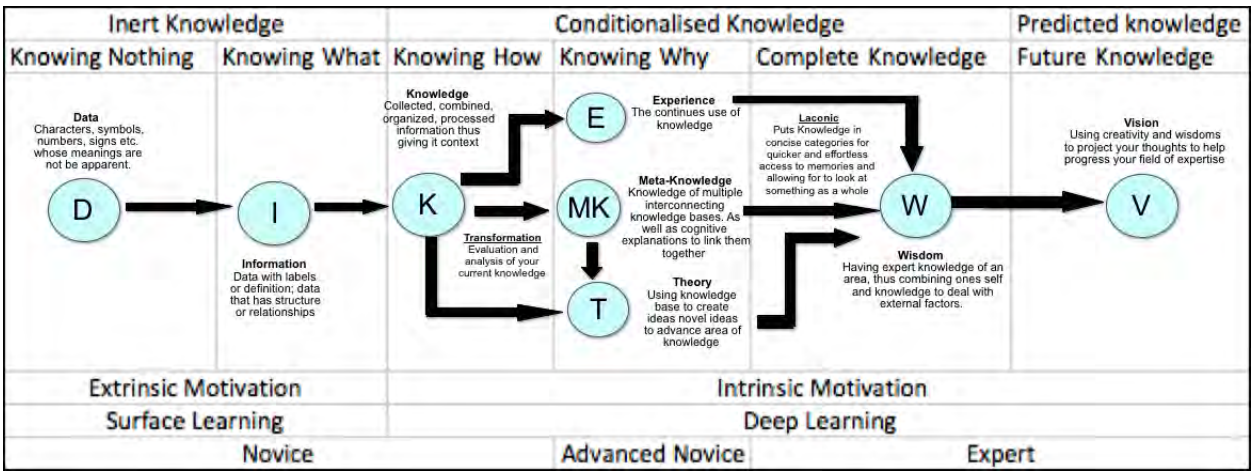


Figure 3: DIKW Mapping [44]

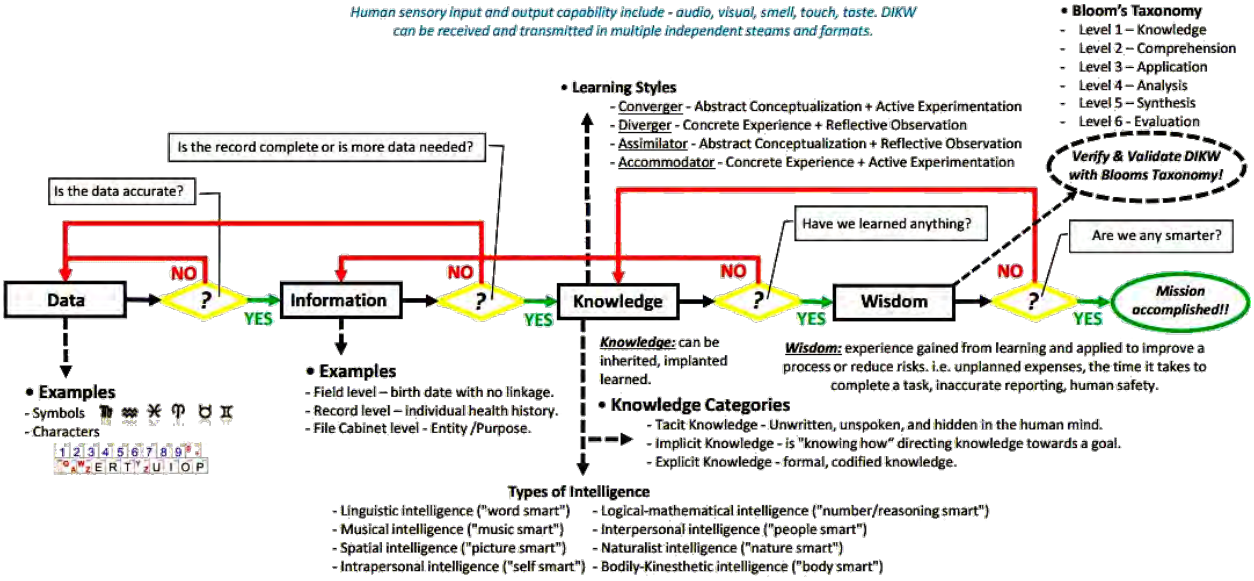


Figure 4: DIKW Flow [44]

Systems Thinking

As with critical thinking, systems thinking has no concrete definition or description. Instead of trying to provide one, the PI treats the following habits of system thinkers as the objectives to be partly addressed in the remainder of this section [3]:

- Seeks to understand the big picture.
- Observes how elements within systems change over time, generating patterns and trends.
- Recognizes that a system's structure generates its behavior.
- Identifies the circular nature of complex cause-and-effect relationships.
- Recognizes the impact of time delays when exploring cause-and-effect relationships.
- Makes meaningful connections within and between systems.
- Changes perspectives to increase understanding.
- Surfaces and tests assumptions.

- Considers an issue fully and resists the urge to come to a quick conclusion.
- Considers how mental models affect current reality and the future.
- Uses understanding of system structure to identify possible leverage actions.
- Considers short-term, long-term, and unintended consequences of actions.
- Pays attention to accumulations and their rates of change.
- Checks results and changes actions if needed through successive approximation.

Applying Systems Thinking

Applying systems thinking to achieve these objectives is also very nonlinear and chaotic. This overview covers an organizational strategy of modeling the different layers, making sense of their correspondences, decomposing them into workable units, establishing a plan to solve them, executing it, evaluating the results, and reflecting on the process. Appendix C provides several classroom examples.

Model layers establish a path from the problem domain to the solution domain. The PI uses the following top-down hierarchy:

- World Model

  The problem domain is the real world, so this model is reality. Students need to actively engage the real world to understand how it works and does not, as well as why.

- Mental Model

  The understanding of the world model is a matter of perception in the mind of those thinking about it. Students need a background in the subject matter, the ability to reason over it, and a way to verify their reasoning.

- Conceptual Model

  The mental model needs to be rich enough to capture the essence of the problem but simple enough to abstract away irrelevant and distracting details. A cartoon perspective, often in the style of Road Runner engineering, helps formulate an appropriate understanding of the system, verify that it is consistent with reality, and communicate about it.

- Formal Model

  The conceptual model is actually informally computable: It contains all the necessary interconnected dots along with the capability to make them perform, albeit in the cartoon sense. The formal model is a corresponding story that uses software engineering design constructs like object-oriented diagrams and design patterns. If the cartoon form does not work, nor will the formal form.

- Computational Model

  The representation in the formal model translates to the final computational model, or program. Here is where most of the programming "doing" should happen, but the least thinking.

Section 3.1 explained how everything that is doable in the real world is doable in the virtual world; only the form and execution differ. This same logic applies between any two model layers, not just the first and last. Each intermediate model derives from the previous and leads to the next. The form changes radically, but the essence should not. Students immediately run into difficulties at the

start by not knowing the subject matter and not putting in the effort to become adequately familiar with it. Their mental model usually reflects feelings, belief, opinion, overconfidence, and least effort instead of what the world model actually needs. Without a mental model, there is no reasonable conceptual model to play with; nor is there a way to evaluate whether the needs would be met before committing to later, more expensive models. Formal models are usually absent because there is not much to formalize. Instead, students generally start and end their development process at the computational model. The result is The Cartoon.

It is therefore necessary to get students under control early and keep them on track in a disciplined, ideally self-policing manner. The same basic questions have to be posed and answered at every layer. Students have remarkable difficulty with this process. The PI partially manages it by making sure the critical questions are stated early and continuously revisited and reevaluated from different perspectives. Students learn this strategy in grade school with *who*, *what*, *when*, *where*, *why*, and *how* questions applied to understanding and telling stories; e.g., book reports and Show and Tell. As obvious as this step seems, students can rarely answer basic questions about *anything* they are thinking or doing. Their mental model is weak or nonexistent. Pushing them to articulate their thoughts this way makes this deficiency clear, where it can then be resolved before moving on.

A critical step is to understand the basics of how dots are referred to:

• World Model:	Things	Properties	Actions	Relationships
• Linguistic Model:	Nouns	Adjectives	Verbs	Prepositions
• Computational Model:	Objects	Data	Functions	Composition

The world is rarely clean and consistent, but in this case, there is a direct correspondence between the world people live in and how they talk about it and how programs can represent it [45]. This correspondence bridges the world and computational models. The world is what it is naturally, and human language passively evolved to reflect it, but people actively designed computational constructs to represent the world in programs. Student miss this incredibly powerful guide to making sense of everything. They often see words that others write as just words without much meaning or intent, and words that they write tend to lack meaning and intent. This deficiency leads to difficulties in communication at all levels. It is not surprising that employers identify poor communication skills as a major concern with today's graduates [6].

Decomposing each layer into its appropriate form is complicated. Students have difficulty making sense of what they are given. They especially struggle with linguistic concepts that other parts of their education have already covered:

- Syntax: The form of words.
- Grammar: The connection of words.
- Semantics: The meaning of words without context.
- Pragmatics: The meaning of words in context.

A good example is the simple concept of angular degrees. Students fixate on the mathematical definition of a circle, where 0 degrees is to the right and values increase counterclockwise. Despite having learned about maps and compasses in grade school, they consistently have difficulty with the second interpretation of 0 degrees being up (north) and values increasing clockwise (east, south, west). This interpretation of the world leads to a disconnect with the math in their

programs, which then behave incorrectly. The typical response is to cover up the symptoms with more complexity instead of treating the cause. A simple Google search about compass directions would be enough reality check to demonstrate this disconnect, but students fixate on their one interpretation as the only possible one, even when explicitly warned not to. Overcoming a faulty mental model and their overconfidence in its correctness is difficult. In fact, this example commonly results in grumbling that the PI "sets them up for failure." It is indeed challenging to teach today's students when they think and act this way so resolutely.

The computational perspective derives from the linguistic one:

- Data: What something is.
- Control: What something can do.
- Behavior: What something actually does in contextual operation.

This framework captures the functional definition of anything at any level. An oversimplified example is an airplane: Its data are position, direction, and speed. Its control is the capability to change direction (turn clockwise or counterclockwise) and speed, which together indirectly change position. Its behavior could be any "mission" involving flying from one position to another. In this way, it is the same common solution to any number of related problems. The data must be appropriate such that the control can operate on it to realize the behavior. Any misalignment is a costly disconnect: Losing something necessary renders the model incomplete and not fully functional, whereas adding something unnecessary increases the cost and risk with no benefit. This kind of basic critical thinking eludes most students, even though their programs use exactly these constructs: Data is variables, control is functions, and behavior is running the program. It is no surprise that their programs generally reflect chaos, and they cannot explain what the programs do or how or why.

Disciplined development requires disciplined planning. This planning, however, does not need to be complicated or onerous. It merely needs to be effective. The PI strives to get the most benefit for the least cost. To this end, he presents this simple framework:

- Read: Look at the materials; this can be any form, but normally is written.
- Understand: Make sense of the materials.
- Plan: Develop a plan to convert the understanding into action.
- Execute: Follow the plan.
- Verify: Confirm that the plan produced the expected results.
- Reflect: Think about what went right and wrong, etc. to learn from the experience.

It is an iterative framework that repeats at every level. If verification fails, do not move on: Identify the disconnect, go back and fix it, then verify the fix. Not doing so leads to The Cartoon. The PI's learning management system[4] directly supports this framework with required periodic status reports on the project and weekly meta-assessments of students' perceptions of their learning. Forcing them to do so is necessary because they would otherwise not verify and reflect on their own. Moving forward no matter what is typical, but it does not equate to making progress. Ironically, what they perceive as busy work for having no value is in fact precisely what they

[4] This system at `shelby.ewu.edu` is another example of an integrated computational tool that the PI designed and implemented in direct support of his teaching philosophy [42].

themselves end up doing by working hard without an actual purpose or a way to determine if they are addressing it.

This framework is hardly revolutionary. In fact, it is fundamentally the scientific method and the engineering method in Figure 5 [18]. Science courses are required of all computer science majors. However, students rarely realize that the true purpose is to understand and apply scientific thinking, which works for any problem solving. Instead, they see it as memorizing facts about rocks.[5] This perspective corresponds to the lowest level of Bloom's Taxonomy, whereas these courses are intended to teach the upper levels.

Figure 5: The Scientific Method and Engineering Method [44]

The design of experiments in science very much serves the same purpose in computer science in two ways. First, the testing, verification, validation, and certification stages of software engineering determine whether the solution satisfies the established performance criteria. These go/go-no decisions must be objective: Does the program do what it is supposed to do or not? Second, when the criteria are satisfied, iterative refinement is often necessary or desirable to improve on an already acceptable result. Modeling and simulation supports both roles. A well-defined model generally results in easier and better testing and refinement [23]. Furthermore, this approach naturally leads into discussion of when to stop refining. The Law of Diminishing Returns, for example, suggests a point where further effort (cost) produces no appreciable benefit. In this way, students can explain and justify their decisions to stop, as opposed to stopping whether they feel like it — usually far too early.

[5] The overwhelming majority takes geology, by their own admission because it is the easiest science option.

The combination of systems thinking and the scientific and engineering methods leads to the culminating part of every project: the test report. It gives students the opportunity to demonstrate competence in software development. Determining correctness and evaluating performance require three critical components: the expected results, the actual results, and a meaningful way to compare them. Students often lack one, two, or even all three, but still think that they are checking their solutions. Even when their solutions do work, they rarely can demonstrate and articulate this outcome in a convincing manner. In order to improve the testing process and the communication skills to explain it, the students have to conduct experiments (provided by the PI) to show representative aspects of system performance. Each experiment has eight parts related to planning, execution, and presenting the results.

Learning from Systems Thinking

This proposal has emphasized many times how critical it is to understand the world. Effective systems thinking demands it. Unfortunately, by the time students are at the university level, they (and indeed adults in general) have typically lost the inquisitive perspective of kids. Pestering parents with endless *why why why* questions may be annoying, but it is also effective in forming the data, information, knowledge, and wisdom used in adulthood. Systems thinking helps resurrect this behavior (minus the annoying part). As Einstein said, "Imagination is more important than knowledge," and "Any fool can know. The point is to understand" [12]. Knowledge is necessary, but alone not sufficient, for success.

A novel aspect of this tool is the rich breadth and depth of cross-referencing between everything discussed here (and more) and external resources. The PI is a voracious reader with endless resources that he has compiled over many years. These include news articles and case studies, pictures and videos of others' work and his own, and so on. The various parts of the processes described here connect to those contextual examples. In particular, he considers historical perspectives — from notable figures like Thomas Edison and Leonardo da Vinci, from the Industrial Revolution, and even as far back as the Roman and Greek Eras — to be invaluable. "Creativity is the secret sauce to science, technology, engineering, and math" [29]. People were arguably more creative, innovative, and resourceful when confronted with pressing problems and few resources. The KISS Principle, *Keep It Simple, Stupid*, goes a long way in systems thinking. Today's students, however, suffer from the opposite mentality: They are overwhelmed and drowning in seemingly endless resources and unwittingly produce inordinately large and complex solutions that do not actually solve the problems. Indeed, they often cause more problems. Simplicity and a kid's view go hand in hand, as Einstein stated: "If you can't explain it to a six year old, you don't understand it yourself" [12].

The ability to decompose and understand others' solutions and the rationale behind them helps students create their own. Haynes Manuals, known commonly for being repair guides that document the process of dismantling and reassembling automobiles, have a lesser known series that does effectively the same with present and past engineering systems like civilian and military aircraft, military equipment and systems, and space technology. The PI has over 100 of these books, among many others, which he plans to use in this work. They are an untapped gold mine.

4 Value and Significance

This section addresses the intrinsic value and significance of this work and its relationship to the PI's teaching and other responsibilities at EWU.

Student Success

This work is all about promoting student success. Engaging students is critical. EWU students tend to be passive participants in their education. This work provides active experiences. Its value in the classroom has already been discussed. This section extends it outside the classroom. At the undergraduate level, students need to distinguish themselves to be marketable. Every EWU computer science graduate has the same ABET-accredited degree with fundamentally the same background. The only things in the program that differentiate one student from another are elective coursework and grade point average. The elective component is comparatively minor, and many employers put little stock in GPAs because of grade inflation [4]. Therefore, successful students need experiences and accomplishments beyond the minimum expectations. This background also provides a basis for promoting themselves in conversation with potential employers at career fairs and interviews. Feedback from employers consistently indicates that students strongly need to work on their professional communication skills. Many struggle because they have little to say and do not know how to say it.

The PI continually encourages students to seek out formative opportunities. He supports a variety of EWU internal programs, as well as external ones (see also Section 6.2); for example:

- Student Research and Creative Works Symposium
- McNair Postbaccalaureate Achievement Program
- CSTEM Undergraduate Research and Creative Activities Fund
- National Conference on Undergraduate Research

Unfortunately, it is difficult to find appropriate projects for students who are earlier in their education and do not yet have the background for substantial effort. This work opens up many opportunities because the breadth and depth of substance in these projects can be aligned with individual student's interests and abilities.

A similar situation exists at the graduate level. Students have a difficult time formulating a research project on their own, and the PI rarely has any in a form that they could work with. Two recent master's theses were indeed related to aerospace, but neither was based on the projects here. As Graduate Program Director, the PI has played an integral role in expanding the options for current and future students. In particular, the program soon will be offering certificate tracks in focused areas. Four are directly relevant to this work. The first three are the PI's, and the fourth overlaps with them:

- Software Engineering
- Modeling and Simulation
- Artificial Intelligence/Intelligent Systems (proposed)
- Data Science and Machine Learning

Similarly, the PI is in discussion with the CSTEM dean to create a degree program in computer engineering supported by the Departments of Computer Science and Electrical Engineering. The PI's courses and resources, as well as this tool, could contribute to it.

This work also serves as a vehicle for promoting STEM within and outside the university. It packs a serious *wow!* punch because of the interesting subject matter presented in a colorful, enticing way. The PI has decades of outreach experience to support this claim.

Finally, the undergraduate degree has a required senior capstone sequence. Over two quarters, students work through developing a real software solution to a real problem that a real customer has, as opposed to the throwaway toy problems in earlier courses.[6] This course is experiencing a variety of logistical difficulties, mainly from exploding enrollments, that are rendering the current approach unmanageable. The department is in the early stages of considering using internal projects with extensive, realistic content as a substitute. The projects showcased here could serve this purpose, as could working on the tool itself.

The Catalyst Project

At the groundbreaking for the Catalyst building in August, US Senator Maria Cantwell listed four pillars of design and innovation to pursue: software development, aviation, security, and biosciences. This work solidly hits the first three. The PI's background and vision (Section 6.2) are a natural fit. Furthermore, he is currently chair of the search committee for a second faculty member in software engineering. This person is expected to support existing efforts and/or complement them. For example, the PI is already working toward multidisciplinary collaborative opportunities with other EWU faculty in small unmanned aircraft systems ("drones"). Part of this effort involves possibly developing a certificate program to prepare students and faculty to attain their FAA Part 107 certification for commercial operations.

Needs Assessment

What is inadequate about existing tools? There are already model-based analysis and design tools for software and systems engineering, commonly referred to as computer-aided software engineering (CASE) tools [15, 30]. This work is not groundbreaking in this respect. However, it also does not reinvent the wheel. Rather, it repackages a wide range of concepts, tools, techniques, etc. into an integrated development environment targeted specifically at the needs and goals addressed throughout this proposal. Professional tools are indeed great for professionals, but they seldom function as well in the classroom. In other words, professionals already know what to do and how (more or less). The tools mostly help expedite known processes and behaviors. For beginners, however, the steep learning curve associated with the endless array of professional features targeted at professional needs and skills can be overwhelming and counterproductive.

Nevertheless, this tool shares many similarities. For example, it supports various analytical and problem-solving methodologies by using industry standards such as software design patterns and object-oriented programming structures. It also employs an architectural modeling language similar to the Universal Markup Language (UML) and Systems Modeling Language (SysML).

More importantly, however, this tool exhibits marked differences. Primarily, it is designed from the ground up to be student-oriented. It is accessible and friendly in a way that students can relate to because it targets their unique strengths and weaknesses. It is also explicitly pedagogy-oriented to support the PI's teaching philosophy directly. Professional products require the instructor to shoehorn their teaching into the constraints of the tool. In this case, the tool is part of the teaching, and vice versa. David Parnas, a pioneer in modern programming, succinctly aptly captures this perspective [26]:

We can write good or bad programs with any tool.
Unless we teach people how to design, the languages matter very little.

[6] A current example is the website for the CSTEM Undergraduate Research and Creative Activities Fund.

The integrated framework is novel. The capability to cross-reference among elements within the development process is common, but to do so with elements outside it is arguably unique. The PI's wealth of resources in various forms provides a rich, immersive, and effective educational experience. The explicit crossover to research and collaborative efforts is similarly advantageous. Numerous graduates have made laudatory comments about their strong preparation for industry from the PI's efforts.

Finally, this tool will be freely available to the educational community. Other universities have similar students and needs. The PI has presented a number of these projects at conferences. Other instructors have asked for access to them for their own use or adaptation in the classroom. Unfortunately, the projects are not currently in a form where this transfer would be practical. The proposed framework will help accommodate such requests.

5 Detailed Plan

5.1 Methodology

1. Problem Analysis
   a. Analyze the existing nine projects.
   b. Analyze the content and delivery of relevant courses.
   c. Analyze hundreds of the PI's books and scanned periodical clippings, thousands of pictures compiled from many dozens of STEM museums in 41 countries, and pictures and videos from flight tests and other experiences, etc.
   d. Identify observations, expectations, etc. from industry.
   e. Identify strengths and especially weaknesses in EWU CS students and in general.
2. Solution Analysis
   a. Analyze the current state of model-based problem solving, software engineering, and systems engineering.
   b. Analyze current model-based problem-solving tools.
3. Solution Implementation

   Develop a model-based tool in the Java programming language.
4. Project Reconstruction
   a. Dismantle existing projects one at a time. The expected number is difficult to predict, but it will be initially at least enough for the PI's course offerings in his first year back.
   b. Rebuild each project using the tool while documenting and cross-referencing.
5. Course Reconstruction

   Rebuild and deploy lectures and assignments around the new projects and approach.
6. Solution Classroom Deployment
   a. Evaluate student performance though pre- and post-tests, surveys, built-in tracking features, etc.
   b. Compare the same project using new approach to the old approach.

7. Dissemination and Distribution
    a. Publish and present with students.
    b. Release freely under the open-source GNU General Public License.
8. Research Follow-on (for the NSF grants; not accountable in the sabbatical effort)
    a. Develop modules based on standalone engineering examples; i.e., not part of existing or planned full-fledged projects.
    b. Develop tutorials for these examples and the tool itself.
    c. Deploy a cloud-based environment for collaborative and competitive individual and team design challenges.

5.2 Timeline

Stages 1 and 2 will begin in Fall quarter 2019. Stage 3 should take the Winter quarter. Stages 4 and 5 will occur in Spring. Stage 6 will be conducted in Fall 2020 in the classroom, which will lead to Stage 7 during the Spring 2021 publication cycle. Stage 8 is aligned with the NSF proposal (see Section 10). It is a natural extension to this sabbatical work, but it is not part of the proposed deliverables or timeline.

6 Feasibility, Qualifications, and Expected Product

6.1 Feasibility

All the components and resources for this work already exist in some form; e.g., pedagogical framework, projects, model-based tools, data from past classes, and qualifications and experience. Time to perform it has been the limiting factor. The PI has never had a sabbatical.

Funding is desirable but not critical. The project will proceed regardless.

6.2 Qualifications

The PI is eminently qualified in breadth and depth to conduct this work in at least the following respects.

Relevant academic experience:

- Associate Professor teaching in computer science and engineering for 14 years, up for promotion now.
- Director of the Computer Science Graduate Program.
- Contributor to over two decades of service to STEM support at all levels of effort.
- Author of at least 16 directly relevant publications. See Appendix D.
- Advisor for three recent master's theses: "Image Processing for Machine Learning of Helicopter Flight Dynamics," "Improving Aerial Package Delivery Through Simulation of Hazard Detection, Mapping and Regulatory Compliance," and a drone-based architecture for military signals intelligence.
- Supervisor for two senior capstone projects on building a winch launcher for full-size gliders for the Spokane Soaring Society (of which the PI is a member).

- Recipient of two EWU Faculty Research and Creative Works grants: "An Exploratory Framework for Introspective Machine Learning of Helicopter Flight Dynamics" and "Adaptive Electromechanical Stability Control for Model Aircraft."
- Author of EWU Faculty Creative Works Symposium posters on "Automated Monitoring, Feedback, and Reporting for an Aviation Performance Study," "Adaptive Electromechanical Stability Control for Model Aircraft," and "Image Processing for Data Acquisition and Machine Learning of Helicopter Flight Dynamics."
- Program evaluator for the Accreditation Board for Engineering and Technology (ABET) in Computer Science and Software Engineering.
- Reviewer multiple times for the NSF Graduate Research Fellowship Program; Department of Defense National Defense Education Program; Department of Defense Science, Mathematics and Research for Transformation Fellowship; and Department of Defense National Defense Science and Engineering Graduate Fellowship; all of which support students for this kind of multi- and interdisciplinary effort.
- Reviewer for many relevant conferences and journals.
- Member of the Alabama Modeling and Simulation Council and active contributor to the yearly international conference and exposition. Huntsville, Alabama is the modeling and simulation capital of the world.

Relevant industry experience:

- Worked for a decade in the defense industry, primarily in software systems engineering and modeling and simulation of flight and weapon systems at White Sands Missile Range and Aberdeen Proving Ground, with many accolades.
- Qualified as a Certified Systems Engineering Professional (CSEP) by the International Council on Systems Engineering (INCOSE) (in review).
- Qualified as a Certified Modeling and Simulation Professional (CMSP) by the Modeling and Simulation Professional Certification Commission and National Training and Simulation Association (in review).
- Earned an Applied Systems Engineering Certificate and a Modeling and Simulation Professional Certificate from the University of Alabama at Huntsville, and expect their Systems Test and Evaluation Certificate next year.
- Earned a Systems Engineering Professional Certificate from Massachusetts Institute of Technology.

Aerospace background:

- Pilot for over 25 years and airplane owner currently FAA qualified for airplanes, seaplanes, gliders, helicopters, and small unmanned aircraft systems, as well as a licensed skydiver and a former balloonist for 10 years; experienced in flying everything from miniature radio-controlled aircraft and drones to full-size military fighter jets and classic warbirds; ferry pilot for Inland Helicopters in Spokane.

- Member of or affiliated with Academy of Model Aeronautics, Aircraft Owners and Pilots Association, Civil Air Patrol, Collings Foundation, Commemorative Air Force, Experimental Aircraft Association, Historic Flight Foundation, Inland Helicopters, Northwest Aviation, Spokane Soaring Society, Washington Pilots Association, Yankee Flying Club, and Young Eagles.
- Subscriber to a dozen aviation print periodicals (see Appendix A).

6.3 Expected Product

Phase I applies to the sabbatical period. It emphasizes software and resource development. It will deliver the tool as defined along with rebuilt courses and a number of rebuilt projects. Dissemination cannot occur until the tool is fielded in the classroom and produces data.

Phase II derives from Phase I, but is not part of the sabbatical term or deliverables. It relies on the last two years of the proposed NSF grants. See Step 8 in Section 5.1.

Phase III also applies after the sabbatical period, hopefully as part of other grants. It emphasizes hardware development. It is a natural complement to the software in the interdisciplinary CS–EE–ME perspective. The intent is to build a human-sized, full-motion immersive virtual reality simulator for airplanes, helicopters, cars, boats, etc. It should be useful for undergraduate and graduate research, and for the PI and colleagues, as well as a promotional vehicle for outreach. The software tool should contribute to and document its development and showcase its operation.

7 Support

Charlie Cleanthous is a full professor of Psychology at EWU and a co-owner of the PI's airplane. They have collaborated on other aviation-related efforts. He will not play a direct role in any of the activities here, but he is a useful resource, especially for the psychological and pedagogical aspects.

8 Dates and Length of Leave Requested

The PI requests professional leave for the entire 2019–2020 academic year.

9 Time to be Devoted to Other Activities

None

10 Remuneration from Other Sources

The PI has submitted a single-PI grant proposal, "A Holistic Integrated Development Environment for Interdisciplinary Model-Based Critical Thinking and Doing in Software Systems Engineering," to parallel NSF programs 18-568 *Computing and Communication Foundations: Software and Hardware Foundations* and 14-579 *Facilitating Research at Primarily Undergraduate Institutions: Research in Undergraduate Institutions (RUI) and Research Opportunity Awards (ROA)*, for $269,952 in total over three years. Both proposals address the same core work derived from this sabbatical proposal, but the purpose of the funding differs. For example, the second one directly supports underrepresented undergraduates.

11 Proof of Program Elimination

Not applicable.

References

[1] "Criteria for Accrediting Computing Programs, 2018–2019." www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-computing-programs-2018-2019. Last accessed Oct. 22, 2018.

[2] "Criteria for Accrediting Engineering Programs, 2018–2019." www.abet.org/accreditation/accreditation-criteria/criteria-for-accrediting-engineering-programs-2018-2019. Last accessed Oct. 22, 2018.

[3] "Habits of a Systems Thinker." Adapted from www.watersfoundation.org/systems-thinking-tools-and-strategies/habits-of-a-systems-thinker. Last accessed Oct. 22, 2018.

[4] Arum, Richard. Academically Adrift: Limited Learning on College Campuses. University of Chicago Press: Chicago, 2011.

[5] Association for Computing Machinery Computer Science and Software Engineering Curricula. www.acm.org/education/curricula-recommendations. Last accessed Oct. 22, 2018.

[6] Bauer-Wolf, Jeremy. "Overconfident Students, Dubious Employers." www.insidehighered.com/news/2018/02/23/study-students-believe-they-are-prepared-workplace-employers-disagree. Feb. 23, 2018. Last accessed Oct. 22, 2018.

[7] Bloom, Benjamin; Krathwohl, David; and Masia, Bertram. The Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain. David McKay: New York, 1956.

[8] Broy, Manfred; Kruger, Ingolf; Pretschner, Alexander; and Salzmann, Christian. "Engineering Automotive Software." In Proc. of the IEEE, Feb. 2007, vol. 95, no. 2, pp. 356–373.

[9] Bruno, Michael. "Where Is the Best Place to Be an Aerospace Manufacturer?" Aviation Week & Space Technology, Oct. 15–28, 2018.

[10] Camden, Jim. "Troubled software for Community Colleges of Spokane on 'pause' as vendor files for bankruptcy." The Spokesman-Review. April 11, 2017.

[11] Colwell, Rita. "Information Technology: Ariadne's Thread Through the Science and Technology Labyrinth." In keynote speech of EDUCASE '99. Oct. 28, 1999.

[12] Commonly referenced quotes without original source.

[13] Denning, Peter; Tedre, Matti; and Yongpradit, Pat. "Misconceptions About Computer Science." Communications of the ACM, March 2017, vol. 60, no. 3, pp. 31–33.

[14] Denning, Peter. "The Science in Computer Science." Communications of the ACM, May 2013, vol. 56, no. 5, pp. 35–38.

[15] Estefan, Jeff. "Survey of Model-Based Systems Engineering (MBSE) Methodologies." www.sebokwiki.org/wiki/A_Survey_of_Model-Based_Systems_Engineering_(MBSE)_Methodologies. Last accessed Oct. 22, 2018.

[16] Graduate Reference Curriculum for Systems Engineering. www.bkcase.org/grcse. Last accessed Oct. 22, 2018.

[17] Greater Spokane, Inc. advantagespokane.com/aerospace. Last accessed Oct. 22, 2018.

[18] Hamming, Richard. The Art of Doing Science and Engineering: Learning to Learn. CRC Press, Oct. 1, 1997.

[19] Henry, Nicholas. "Knowledge Management: A New Concern for Public Administration." Public Administration Review, May–June 1974, vol. 34, no. 3:189.

[20] Krigsman, Michael. "Study: 68 percent of IT projects fail." www.zdnet.com/article/study-68-percent-of-it-projects-fail. Jan. 14, 2009. Last accessed Oct. 22, 2018.

[21] Long, David and Scott, Zane. "A Primer for Model-Based Systems Engineering." www.vitechcorp.com/resources/mbse.shtml. Last accessed Oct. 22, 2018.

[22] Microsoft Transform Science Computational Education for Scientists. research.microsoft.com/CEfS. Last accessed Oct. 22, 2018.

[23] Miller, Steven; Whalen, Michael; and Cofer, Darren. "Software Model Checking Takes Off." Communications of the ACM, Feb. 2010, vol. 53, no. 2, pp. 58–64.

[24] Modeling and Simulation Body of Knowledge. www.scs.org. Last accessed Oct. 22, 2018.

[25] Moore, Thomas; Wick, Michael; and Peden, Blaine. "Assessing Students' Critical Thinking Skills and Attitudes Towards Computer Science." In Proc. of the 25th SIGCSE Symposium on Computer Science Education, Mar. 1994, pp. 263–267.

[26] Parnas, David. In Brooks, Fredrick: The Mythical Man-Month. Addison-Wesley Longman: Crawfordsville, IN, 1995.

[27] Pressman, Roger. Software Engineering: A Practitioner's Approach. 7th ed. McGraw Hill: New York, 2010.

[28] Project Management Body of Knowledge. www.pmi.org. Last accessed Oct. 22, 2018.

[29] Ramirez, Ainissa. "Creativity is the Secret Sauce in STEM." www.edutopia.org/blog/creativity-secret-sauce-in-stem-ainissa-ramirez. Last accessed Oct. 22, 2018.

[30] Ramos, Ana Luísa. "Model-Based Systems Engineering: An Emerging Approach for Modern Systems." IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), Jan. 2012, vol. 42, no. 1.

[31] Rice, David. "Why 45% of all software features in production are NEVER Used." www.linkedin.com/pulse/why-45-all-software-features-production-never-used-david-rice. Last accessed Oct. 22, 2018.

[32] Software Engineering Body of Knowledge. www.computer.org/web/swebok. Last accessed Oct. 22, 2018.

[33] Sokol, Chad. "EWU plans major expansion into Spokane's University District as anchor tenant of Avista's planned 150,000-square-foot Catalyst building." The Spokesman-Review. Feb. 6, 2018.

[34] Sommers, James. "The Coming Software Apocalypse." www.theatlantic.com/technology/archive/2017/09/saving-the-world-from-code/540393. The Atlantic. Sept. 26, 2017. Last accessed Oct. 22, 2018.

[35] Systems Engineering Body of Knowledge. www.sebokwiki.org. Last accessed Oct. 22, 2018.

[36] Tappan, Dan. "A Holistic Multidisciplinary Approach to Teaching Software Engineering Through Air Traffic Control." Journal of Computing Sciences in Colleges, 2014, vol. 30, no. 1, pp. 199–205. shelby.ewu.edu/research/docs/2014_ccsc.pdf. Last accessed Oct. 22, 2018.

[37] Tappan, Dan and Matt Hempleman. "Toward Introspective Human Versus Machine Learning of Simulated Airplane Flight Dynamics." In Proc. of the 25th Modern Artificial Intelligence and Cognitive Science Conference, Spokane, WA, Apr. 26, 2014. shelby.ewu.edu/research/docs/2014_maics.pdf. Last accessed Oct. 22, 2018.

[38] Tappan, Dan. "Multiagent Test Range: Fostering Disciplined Software Engineering Practices in Students via Modeling, Simulation, Visualization, and Analysis." In Proc. of the Alabama Modeling and Simulation Council International Conference and Exposition, Huntsville, AL, May 6–7, 2014. shelby.ewu.edu/research/docs/2014_alasim.pdf. Last accessed Oct. 22, 2018.

[39] Tappan, Dan. "A Quasi-Network-Based Fly-by-Wire Simulation Architecture for Teaching Software Engineering." In Proc. of the 45th IEEE Frontiers in Education Conference, El Paso, TX, Oct. 21–24, 2015. shelby.ewu.edu/research/docs/2015_fie.pdf. Last accessed Oct. 22, 2018.

[40] Tappan, Dan. "A Meta-Case Study of Modeling, Simulation, Visualization, and Analysis for Real-World Software Systems Engineering Education." In Proc. of the Alabama Modeling and Simulation Council International Conference and Exposition, Huntsville, AL, May 3–5, 2016. shelby.ewu.edu/research/docs/2016_alasim.pdf. Last accessed Oct. 22, 2018.

[41] Tappan, Dan. "Experiencing Real-World Multidisciplinary Software Systems Engineering Through Aircraft Carrier Simulation." In Proc. of the American Association for Engineering Education Conference, New Orleans, LA, June 26–29, 2016. shelby.ewu.edu/research/docs/2016_asee.pdf. Last accessed Oct. 22, 2018.

[42] Tappan, Dan. "A Data Analytics Approach to a Computer Science Senior Capstone Project Management Tool." In Proc. of the WorldComp 12th International Conference on Frontiers in Education: Computer Science and Computer Engineering, Las Vegas, NV, July 25–28, 2016. shelby.ewu.edu/research/docs/2016_worldcomp_fec.pdf. Last accessed Oct. 22, 2018.

[43] Usually attributed to Edsger Dijkstra as a direct quote but in fact a paraphrase.

[44] Variations on this figure have been floating around the internet for years. There is no clear attribution anymore.

[45] Videla, Alvaro. "Metaphors We Compute By." Communications of the ACM, Oct. 2017, vol. 60, no. 10.

[46] Weinberg, Gerald. In Chemuturi, Murali: Mastering Software Quality Assurance: Best Practices, Tools and Technique for Software Developers. p. ix, 2010.

[47] www.certguidance.com. Last accessed Oct. 22, 2018.

[48] Zakas, Nicholas. "The care and feeding of software engineers (or, why engineers are grumpy)." www.humanwhocodes.com/blog/2012/06/12/the-care-and-feeding-of-software-engineers-or-why-engineers-are-grumpy. June 12, 2012. Last accessed Oct. 22, 2018.

Appendices

Appendix A: Periodical Resources

Print subscriptions to these periodicals, in addition to general news sources like *Time* and *The Week*, provide current events and case studies related to classroom activities. Many lectures present scanned articles or make them available to students.

STEM

Circuit Cellar, Communications of the ACM (Association for Computing Machinery), Computing Edge (IEEE: Institute for Electrical and Electronics Engineers), Digital Machinist, Discover, Home Shop Machinist, Machinist's Workshop, Make, Nuts and Volts, Popular Mechanics, Popular Science, Prism (ASEE: American Society for Engineering Education), Science Focus, Science News, Scientific American, Servo, Spectrum (IEEE), Transactions on Education (IEEE), Transactions on Techniques in STEM Education (IEEE), Wired

Aerospace

Aerospace and Electronic Systems (IEEE), Air & Space (Smithsonian), AOPA Pilot (Aircraft Owners and Pilots Association), Aviation History, Aviation Week & Space Technology, Flight Journal, Flying, Model Aviation, Parachutist, Plane & Pilot, Soaring, Sport Aviation (EAA: Experimental Aircraft Association)

Historical

Archaeology, Military History, National Geographic, Smithsonian, World War II

Appendix B: Curriculum Assessment Criteria

B.1. ABET Student Learning Outcomes for Computing Programs

The program must enable students to attain, by the time of graduation:

a. An ability to apply knowledge of computing and mathematics appropriate to the program's student outcomes and to the discipline.
b. An ability to analyze a problem, and identify and define the computing requirements appropriate to its solution.
c. An ability to design, implement, and evaluate a computer-based system, process, component, or program to meet desired needs.
d. An ability to function effectively on teams to accomplish a common goal.
e. An understanding of professional, ethical, legal, security and social issues and responsibilities.
f. An ability to communicate effectively with a range of audiences.
g. An ability to analyze the local and global impact of computing on individuals, organizations, and society.
h. Recognition of the need for and an ability to engage in continuing professional development and lifelong learning.
i. An ability to use current techniques, skills, and tools necessary for computing practice.
j. An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems in a way that demonstrates comprehension of the tradeoffs involved in design choices.
k. An ability to apply design and development principles in the construction of software systems of varying complexity.

B.2. EWU Professional Advisory Board Program Educational Objectives

Our students will be prepared to:

1. Grow their roles in the community and the organization that employs them.
2. Pursue and apply lifelong learning, assessing the value of older, established, stable systems in relation to new systems, and working within legacy systems, not just create new solutions.
3. Act on the recognition that all decisions have an impact on the organization, business partners, and customers, being cognizant of the end users—and whether it is improving lives.
4. Contribute with an understanding that there is more to a product than technology, and that product development is a collaborative and ongoing process.
5. Collaborate across disciplines and with nontechnical, as well as technical, people.
6. Discuss customer needs at the customer's level, including through the process of gathering requirement specifications.
7. Expand technical competence beyond the fundamentals in areas such as software and interface development, databases, concurrent systems, refactoring, design patterns, and systems integration.
8. Create robust and testable software, with regard to architectural domain, security considerations, deployment, maintenance, validation, and verification.
9. Act with cultural awareness and ethical integrity in a global community.

Appendix C: CSCD 350 Software Engineering Assignment Examples

This assignment forces individual students to decompose the problem domain of a project in a disciplined manner by using the tools, techniques, and thinking highlighted here. It is the first step of the development process for them.

This assignment cross-references many layers of the development process to formalize a plan for it. Its decisions derive from C.1.

This assignment serves as a culmination of the development process. Teams use this description to build and test parts of the project.

CS 350 Task 1: Project Grounding and Conceptualization

This task establishes a guided introductory framework for the primary elements of the quarter project. In real life, you would have to assemble the equivalent of such a list yourself. It would be far larger, more complex, and messier, and would take much longer to compile. Based on the high-level context being discussed in class at this point, your job is to perform preliminary background research on each element in this outline. At this inception phase in the top-down development process, your answers will necessarily be general. In real life, it would be your responsibility to compile something to the effect of this list based on a subjective breadth and depth of coverage. Nobody would ask to see it, but it would be clear if you had not done it. This list establishes the vocabulary and context for understanding of the problem domain and the initial design thinking for the solution domain to come.

Lecture 6 covered the definitions of the bold words.

1. Briefly discuss what each element means to you so far with respect to your current understanding of the project as introduced in the kickoff "meeting" in lecture. Do not just provide a blanket definition or something factually correct but useless; e.g., dogs bark, or the APU does APU stuff. Your interpretations ultimately may not be relevant to our actual solution later, but they must be arguably within its purview. You must address each element separately as structured here and cite its primary source in the format `[@url]`, where *`url`* is the complete text link to your reference (minus any `http[s]://`). (Do not make it a dynamic embedded link.) Use at least six sources in total; Wikipedia is acceptable, but it serves better as a starting point (a resource) than actual information (a reference). Put the term in bold. Indicate your name at the top of the document and the total word count at the bottom.
2. Indicate the grammatical *category* as **noun**, **adjective**, **preposition**, **verb**, or **other**.

For each element:

3. Briefly address these aspects from a consistent, coherent, practical, computational perspective. If multiple interpretations are possible, choose the most representative. Include the head words:

   data: what it is; properties that describe its existence. For each, indicate whether it is **static** (unchanging) or **dynamic** (changing) and why
   control: what it can do; actions that describe its capabilities
   behavior: what it actually does or is done with it; appropriate actions to satisfy a goal; user stories or use cases

   The entries should be reasonably consistent; e.g., if control acts on something, then corresponding data is likely.
4. Indicate the *role(s)*: **input**, **processing**, or **output** and why.
5. Indicate the paradigm(s) of design *pattern*: **creational**, **structural**, or **behavioral** and why.
6. Indicate the *concern(s)* of an MVC architecture: **model**, **view**, or **controller** and why.
7. Indicate the expected *difficulty* to manipulate it programmatically: **easy**, **moderate**, or **hard** and why.
8 Indicate the expected *risk* of not being able to do it or doing it wrong: **low**, **moderate**, or **high** and why.
9. Describe a plausible (for this course) two- or three-dimensional visual *presentation* of the content.

   For example:

   1. **landing gear, main**: supports and stops an aircraft while on the ground; retracted in flight

      source: [@en.wikipedia.org/wiki/Landing_gear]
      category: noun
      data: wheels (static because they do not need to rotate in our project); strut (dynamic because it needs to extend and retract)
      control: wheels have braking; strut can extend and retract
      behavior: the pilot commands the gear to retract after takeoff and to extend before landing
      role: output because it is a mechanical device that supports and stops a plane
      pattern: creational because it must be defined; structural because it connects to the airplane
      concern: model because it is something manipulated; view because we need to see the state
      difficulty: moderate because it involves motion between two states
      risk: high risk because it involves code for motion; not having this prevents the gear from functioning
      presentation: a down-arrow for extended, up-arrow for retracted

A. Address all the following elements in order:

1. acoustic proximity fuze
2. acoustic sensor
3. acquisition process
4. active radar sensor
5. active sonar sensor
6. actuator
7. attenuation
8. battleship
9. bomb
10. bomber aircraft
11. countermeasure
12. cross-section/reflectivity
13. defensive maneuver
14. depth charge
15. depth fuze
16. destroyer
17. distance fuze
18. engagement process
19. evasive maneuver
20. fighter aircraft
21. lethality process
22. main battery gun
23. missile
24. mobility process
25. munition
26. offensive maneuver
27. passive radar sensor
28. passive sonar sensor
29. power
30. radar proximity fuze
31. sensor
32. sensor fusion
33. submarine
34. thermal proximity fuze
35. thermal sensor
36. timed fuze
37. torpedo
38. triangulation
39. trilateration

B. Group the elements from (A) (by name only) into a higher-level organization that makes sense to you. For example, *head, body, tail, legs, teeth, eyes, claws, paws, ears, nose, bark* for a dog could be organized as:

<u>senses</u>
eyes
ears
nose

<u>movement</u>
legs
paws

<u>defense</u>
claws
teeth

communication

bark
tail
paw

and so on…

Some terms may belong to more than one category. Be sure to account for all elements. There is no order within a group.

The real world is very messy with ill-defined boundaries. Not everything here will fall cleanly into the specified partitioning. Use your best judgment. Team collaboration would iron out the inconsistencies to establish a consensus.

Tasks C and D apply to the spreadsheet posted with this task. Do not otherwise modify the existing structure or contents.

C. Indicate the bold designations from 2 through 8 for each element as a lowercase **x** in the corresponding cells.

D. Add three elements related to this element respectively in the Related Elements columns. Limit each to a word or two, and do not clarify with parenthesis or commas. Do not reuse any of the elements from (A).

Deliverables

Submit Parts A and B in a PDF document. Submit Parts C and D in CSV format. Incorrect formats will incur a stiff penalty.

Start early. Do not underestimate the time it will take to do this work. The analysis stage of a project is critical to do as completely and correctly as possible. Nobody is born knowing this stuff, and there are no shortcuts to learning something about it. This course is not about materiel test and evaluation or video games, etc.; it is about applying software engineering to a concrete example that we can work with. This same process works with any project.

Assessment

This is a software engineering class that is designed so you learn how to analyze problems, design and implement solutions, and verify that you have done so correctly. There is no difference in this respect between establishing and following a process for doing this with code or non-code. Remember, over 80% of software engineering involves non-code, such at this document. Also remember that most errors occur in thinking and understanding, not in execution (or errors in execution stem from the former). Take this process seriously because it is helping you to train your mind to read, understand, plan, execute, verify, and reflect at all levels of software development. It is your responsibility to determine whether you understand this task and to complete it properly. Ask if you are unsure.

Follow our process: read ➜ understand ➜ plan ➜ execute ➜ verify ➜ reflect

(Put Name Here)

	Element	Category					Data		Role			Pattern			Concern			Difficulty			Risk			Related Elements		
		Noun	Adjective	Preposition	Verb	Other	Static	Dynamic	Input	Processing	Output	Creational	Structural	Behavioral	Model	View	Controller	Easy	Moderate	Hard	Low	Moderate	High	1	2	3
1	acoustic proximity fuze																									
2	acoustic sensor																									
3	acquisition process																									
4	active radar sensor																									
5	active sonar sensor																									
6	actuator																									
7	attenuation																									
8	battleship																									
9	bomb																									
10	bomber aircraft																									
11	countermeasure																									
12	cross-section/reflectivity																									
13	defensive maneuver																									
14	depth charge																									
15	depth fuze																									
16	destroyer																									
17	distance fuze																									
18	engagement process																									
19	evasive maneuver																									
20	fighter aircraft																									
21	lethality process																									
22	main battery gun																									
23	missile																									
24	mobility process																									
25	munition																									
26	offensive maneuver																									
27	passive radar sensor																									
28	passive sonar sensor																									
29	power																									
30	radar proximity fuze																									
31	sensor																									
32	sensor fusion																									
33	submarine																									
34	thermal proximity fuze																									
35	thermal sensor																									
36	timed fuze																									
37	torpedo																									
38	triangulation																									
39	trilateration																									

<code>	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
<count>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

CS 350 Task 2: Project Software Requirements Specification Elicitation

This task lays out the framework for a proposed Software Requirements Specification (SRS) related to our project. It relies on the breadth and depth of understanding developed in Task 1 to establish and bidirectionally cross-reference the following hierarchy:

- Scenarios
- User Stories
- Questions
- Requirements
- Specifications
- Requirement Verification
- Specification Verification
- ~~Validation~~

Lectures 14 onward cover them in more detail. Note that the final component, validation, is not in play because we have no good definition of what the customer's needs are. We (intentionally) glossed over this step early in the development process. Now we would be in trouble. Think about the first panel in The Cartoon.

The color coding highlights the cross-references here for clarity, but it is not part of the solution.

Part 1: Scenarios

A scenario is defined for this task as a general end-to-end story narrating one complete action, as done on the board in Lecture 12 for a ship firing a missile at a plane.

Given our scope from Task 1 and lecture, select two of the following scenarios, then title three of your own, all numbered 1 through 5, and finally narrate the reasonably complete process of performing a corresponding story action. Do not use any from lecture as yours.

- Flying an airplane to a location
- Dropping a bomb from an airplane onto a ship
- Detecting another submarine by using an acoustic sensor

Form of Solution

<scenario_num>: <scenario_title>

<narrative>

Example

1: Dropping water balloons from a tower

The holding mechanism at the top of the tower releases the balloon. The balloon falls straight down under gravity. It impacts the ground.

Part 2: User Stories

A user story is a specific part of a scenario from the user's perspective in the form:

As an <actor> I want to do <action> so that <achievement>.

For each of your scenarios from Part 1, title and state a user story, numbered 1.

Form of Solution

<scenario_num>.<user_story_num>: <user_story_title>

<narrative>

Example

1.1: Aiming at friend

As a friend I want to drop a water balloon on another friend so that he gets wet.

Part 3: Questions

A question elicits a targeted response to an aspect that needs further clarification.

For each of the user stories from Part 2, state and justify two representative W5H questions, numbered 1 through 2, that would be important to clarify.

Form of Solution

<scenario_num>.<user_story_num>.<question_num>: <question>

<justification why>

Example

1.1.1: How is the water balloon secured?

Holding onto the water balloon properly will ensure that it is dropped at the right time.

Part 4: Requirements

A requirement states the form of the solution that is expected. It does not answer a question.

For each of the questions from Part 3, state and justify two representative requirements, numbered 1 through 2, that are important to satisfy.

Form of Solution

<scenario_num>.<user_story_num>.<question_num>.<requirement_num>: <requirement>

<justification why>

Example

1.1.1.1: Balloon shall contain water

It is a water balloon.

Part 5: Specifications

A specification constrains the form or scope of the solution for a requirement.

For each requirement from Part 4, state and justify two representative specifications, numbered 1 through 2.

Form of Solution

<scenario_num>.<user_story_num>.<question_num>.<requirement_num>.<specification_num>: <specification>

<justification why>

Example

1.1.1.1.1: Balloon shall contain at least one liter of water.

This amount produces the best effect.

Part 6: Requirement Verification

Requirement verification defines how to determine whether a requirement has been satisfied; i.e., did we build the product right?

For each requirement from Part 4, briefly describe a go/no-go check, numbered 1.

Form of Solution

<scenario_num>.<user_story_num>.<question_num>.<requirement_num>.A.<verification_num>: <description>

Example

1.1.1.1.A.1: Is water present in the balloon?

Part 7: Specification Verification

Specification verification defines how to determine whether a specification has been satisfied; i.e., did we build the product right?

For each specification from Part 5, briefly describe a go/no-go check, numbered 1.

Form of Solution

<scenario_num>.<user_story_num>.<question_num>.<requirement_num>.<specification_num>.<verification_num>: <description>

Example

1.1.1.1.1.1.1: Is at least one liter of water present in the balloon?

Part 8: Validation

Validation defines how to determine whether the solution satisfies the customer's needs; i.e., did we build the right product?

No action is required for this part because we cannot do it without having formally defined the customer's needs. Therefore, we have no disciplined way to determine whether we have satisfied them.

Deliverable

Submit one PDF document with each part in a separately named section in order as indicated below (presented in columns here only to save space). Do not nest the sections or use columns.

Part 1: Scenarios

1 blah blah
2 blah blah
3 blah blah
4 blah blah
5 blah blah

Part 2: User Stories

1.1 blah blah
2.1 blah blah
3.1 blah blah
4.1 blah blah
5.1 blah blah

Part 3: Questions

1.1.1 blah blah
1.1.2 blah blah

2.1.1 blah blah
2.1.2 blah blah

3.1.1 blah blah
3.1.2 blah blah

4.1.1 blah blah
4.1.2 blah blah

5.1.1 blah blah
5.1.2 blah blah

Part 4: Requirements

1.1.1.1 blah blah
1.1.1.2 blah blah
1.1.2.1 blah blah
1.1.2.2 blah blah

2.1.1.1 blah blah
2.1.1.2 blah blah
2.1.2.1 blah blah
2.1.2.2 blah blah

3.1.1.1 blah blah
3.1.1.2 blah blah
3.1.2.1 blah blah
3.1.2.2 blah blah

4.1.1.1 blah blah
4.1.1.2 blah blah
4.1.2.1 blah blah
4.1.2.2 blah blah

5.1.1.1 blah blah
5.1.1.2 blah blah
5.1.2.1 blah blah
5.1.2.2 blah blah

Part 5: Specifications

1.1.1.1.1 blah blah
1.1.1.1.2 blah blah
1.1.1.2.1 blah blah
1.1.1.2.2 blah blah
1.1.2.1.1 blah blah
1.1.2.1.2 blah blah
1.1.2.2.1 blah blah
1.1.2.2.2 blah blah

2.1.1.1.1 blah blah
2.1.1.1.2 blah blah
2.1.1.2.1 blah blah
2.1.1.2.2 blah blah
2.1.2.1.1 blah blah
2.1.2.1.2 blah blah
2.1.2.2.1 blah blah
2.1.2.2.2 blah blah

3.1.1.1.1 blah blah
3.1.1.1.2 blah blah
3.1.1.2.1 blah blah
3.1.1.2.2 blah blah
3.1.2.1.1 blah blah
3.1.2.1.2 blah blah
3.1.2.2.1 blah blah
3.1.2.2.2 blah blah

4.1.1.1.1 blah blah
4.1.1.1.2 blah blah
4.1.1.2.1 blah blah
4.1.1.2.2 blah blah
4.1.2.1.1 blah blah
4.1.2.1.2 blah blah
4.1.2.2.1 blah blah
4.1.2.2.2 blah blah

5.1.1.1.1 blah blah
5.1.1.1.2 blah blah
5.1.1.2.1 blah blah
5.1.1.2.2 blah blah
5.1.2.1.1 blah blah
5.1.2.1.2 blah blah
5.1.2.2.1 blah blah
5.1.2.2.2 blah blah

Part 6: Requirement Verification

1.1.1.1.A.1 blah blah
1.1.1.2.A.1 blah blah
1.1.2.1.A.1 blah blah
1.1.2.2.A.1 blah blah

2.1.1.1.A.1 blah blah
2.1.1.2.A.1 blah blah
2.1.2.1.A.1 blah blah
2.1.2.2.A.1 blah blah

3.1.1.1.A.1 blah blah
3.1.1.2.A.1 blah blah
3.1.2.1.A.1 blah blah
3.1.2.2.A.1 blah blah

4.1.1.1.A.1 blah blah
4.1.1.2.A.1 blah blah
4.1.2.1.A.1 blah blah
4.1.2.2.A.1 blah blah

5.1.1.1.A.1 blah blah
5.1.1.2.A.1 blah blah
5.1.2.1.A.1 blah blah
5.1.2.2.A.1 blah blah

Part 7: Specification Verification

1.1.1.1.1.1 blah blah
1.1.1.1.2.1 blah blah
1.1.1.2.1.1 blah blah
1.1.1.2.2.1 blah blah
1.1.2.1.1.1 blah blah
1.1.2.1.2.1 blah blah
1.1.2.2.1.1 blah blah
1.1.2.2.2.1 blah blah

2.1.1.1.1.1 blah blah
2.1.1.1.2.1 blah blah
2.1.1.2.1.1 blah blah
2.1.1.2.2.1 blah blah
2.1.2.1.1.1 blah blah
2.1.2.1.2.1 blah blah
2.1.2.2.1.1 blah blah
2.1.2.2.2.1 blah blah

3.1.1.1.1.1 blah blah
3.1.1.1.2.1 blah blah
3.1.1.2.1.1 blah blah
3.1.1.2.2.1 blah blah
3.1.2.1.1.1 blah blah
3.1.2.1.2.1 blah blah
3.1.2.2.1.1 blah blah
3.1.2.2.2.1 blah blah

4.1.1.1.1.1 blah blah
4.1.1.1.2.1 blah blah
4.1.1.2.1.1 blah blah
4.1.1.2.2.1 blah blah
4.1.2.1.1.1 blah blah
4.1.2.1.2.1 blah blah
4.1.2.2.1.1 blah blah
4.1.2.2.2.1 blah blah

5.1.1.1.1.1 blah blah
5.1.1.1.2.1 blah blah
5.1.1.2.1.1 blah blah
5.1.1.2.2.1 blah blah
5.1.2.1.1.1 blah blah
5.1.2.1.2.1 blah blah
5.1.2.2.1.1 blah blah
5.1.2.2.2.1 blah blah

If the document were nested (simply by sorting the indices), it would produce this embedded structure. However, we do not normally want this form because it confounds the context of what we are looking at. For example, the Requirements section should be exclusively about requirements: nothing less, nothing more. If we need to understand where a requirement comes from or where it goes, then the cross-referencing leads us to that independent section.

1 blah blah	2 blah blah	3 blah blah	4 blah blah	5 blah blah
1.1 blah blah	2.1 blah blah	3.1 blah blah	4.1 blah blah	5.1 blah blah
1.1.1 blah blah	2.1.1 blah blah	3.1.1 blah blah	4.1.1 blah blah	5.1.1 blah blah
1.1.1.1 blah blah	2.1.1.1 blah blah	3.1.1.1 blah blah	4.1.1.1 blah blah	5.1.1.1 blah blah
1.1.1.1.1 blah blah	2.1.1.1.1 blah blah	3.1.1.1.1 blah blah	4.1.1.1.1 blah blah	5.1.1.1.1 blah blah
1.1.1.1.1.1 blah blah	2.1.1.1.1.1 blah blah	3.1.1.1.1.1 blah blah	4.1.1.1.1.1 blah blah	5.1.1.1.1.1 blah blah
1.1.1.1.2 blah blah	2.1.1.1.2 blah blah	3.1.1.1.2 blah blah	4.1.1.1.2 blah blah	5.1.1.1.2 blah blah
1.1.1.1.2.1 blah blah	2.1.1.1.2.1 blah blah	3.1.1.1.2.1 blah blah	4.1.1.1.2.1 blah blah	5.1.1.1.2.1 blah blah
1.1.1.1.A.1 blah blah	2.1.1.1.A.1 blah blah	3.1.1.1.A.1 blah blah	4.1.1.1.A.1 blah blah	5.1.1.1.A.1 blah blah
1.1.1.2 blah blah	2.1.1.2 blah blah	3.1.1.2 blah blah	4.1.1.2 blah blah	5.1.1.2 blah blah
1.1.1.2.1 blah blah	2.1.1.2.1 blah blah	3.1.1.2.1 blah blah	4.1.1.2.1 blah blah	5.1.1.2.1 blah blah
1.1.1.2.1.1 blah blah	2.1.1.2.1.1 blah blah	3.1.1.2.1.1 blah blah	4.1.1.2.1.1 blah blah	5.1.1.2.1.1 blah blah
1.1.1.2.2 blah blah	2.1.1.2.2 blah blah	3.1.1.2.2 blah blah	4.1.1.2.2 blah blah	5.1.1.2.2 blah blah
1.1.1.2.2.1 blah blah	2.1.1.2.2.1 blah blah	3.1.1.2.2.1 blah blah	4.1.1.2.2.1 blah blah	5.1.1.2.2.1 blah blah
1.1.1.2.A.1 blah blah	2.1.1.2.A.1 blah blah	3.1.1.2.A.1 blah blah	4.1.1.2.A.1 blah blah	5.1.1.2.A.1 blah blah
1.1.2 blah blah	2.1.2 blah blah	3.1.2 blah blah	4.1.2 blah blah	5.1.2 blah blah
1.1.2.1 blah blah	2.1.2.1 blah blah	3.1.2.1 blah blah	4.1.2.1 blah blah	5.1.2.1 blah blah
1.1.2.1.1 blah blah	2.1.2.1.1 blah blah	3.1.2.1.1 blah blah	4.1.2.1.1 blah blah	5.1.2.1.1 blah blah
1.1.2.1.1.1 blah blah	2.1.2.1.1.1 blah blah	3.1.2.1.1.1 blah blah	4.1.2.1.1.1 blah blah	5.1.2.1.1.1 blah blah
1.1.2.1.2 blah blah	2.1.2.1.2 blah blah	3.1.2.1.2 blah blah	4.1.2.1.2 blah blah	5.1.2.1.2 blah blah
1.1.2.1.2.1 blah blah	2.1.2.1.2.1 blah blah	3.1.2.1.2.1 blah blah	4.1.2.1.2.1 blah blah	5.1.2.1.2.1 blah blah
1.1.2.1.A.1 blah blah	2.1.2.1.A.1 blah blah	3.1.2.1.A.1 blah blah	4.1.2.1.A.1 blah blah	5.1.2.1.A.1 blah blah
1.1.2.2 blah blah	2.1.2.2 blah blah	3.1.2.2 blah blah	4.1.2.2 blah blah	5.1.2.2 blah blah
1.1.2.2.1 blah blah	2.1.2.2.1 blah blah	3.1.2.2.1 blah blah	4.1.2.2.1 blah blah	5.1.2.2.1 blah blah
1.1.2.2.1.1 blah blah	2.1.2.2.1.1 blah blah	3.1.2.2.1.1 blah blah	4.1.2.2.1.1 blah blah	5.1.2.2.1.1 blah blah
1.1.2.2.2 blah blah	2.1.2.2.2 blah blah	3.1.2.2.2 blah blah	4.1.2.2.2 blah blah	5.1.2.2.2 blah blah
1.1.2.2.2.1 blah blah	2.1.2.2.2.1 blah blah	3.1.2.2.2.1 blah blah	4.1.2.2.2.1 blah blah	5.1.2.2.2.1 blah blah
1.1.2.2.A.1 blah blah	2.1.2.2.A.1 blah blah	3.1.2.2.A.1 blah blah	4.1.2.2.A.1 blah blah	5.1.2.2.A.1 blah blah

Assessment

Each part and its contents will be evaluated with respect to the rest of the document. Your responses need only be reasonable and consistent, not 100% correct or optimal. You are not qualified in the subject matter to make such decisions. However, keep in mind that in industry, you would also not be qualified, but you would be expected to get everything right. Think about the role of Task 1 in bridging the gap between knowing nothing and knowing something, but not enough. What would you do further?

Follow our process: read ➜ understand ➜ plan ➜ execute ➜ verify ➜ reflect

CS 350 Project Software Requirements Specification

Description

This document serves as the central repository guiding the requirements and specifications for the remainder of the project. Other resources on Shelby and in lecture may also play a role, especially Javadoc. A large part of this software engineering experience is learning how to manage the endless resources. Be sure to apply the iterative process of reading, understanding, approaching, executing, and verifying at every stage. There is only one chance to get these solutions correct. Solutions that do not compile will receive no credit. Ask questions! Think!

All package names in the project assume the prefix `f16cs350` here. Unless otherwise specified, throw a `RuntimeException` for any error.

Part I.a: Terrain Loader

The role of the terrain loader is to read a terrain definition file and translate it into a collection of polygons that supply the terrain in the world model. The logical model is as follows:

- A *node* is point in three-dimensional space defined by a latitude, a longitude, and an altitude.
- A *surface* is an ordered collection of at least three nodes.
- A *terrain* is an unordered collection of surfaces.

The file definition is as follows:

```
RLAT index_l degrees:minutes:seconds
```

Defines a latitude reference with index $\texttt{index}_{\texttt{l}}$ for a latitude point with `degrees` degrees, `minutes` minutes, and `seconds` seconds. It maps to datatype `atc.datatype.Latitude_ATC`.

```
RLON index_o degrees:minutes:seconds
```

Defines a longitude reference with index $\texttt{index}_{\texttt{o}}$ for a longitude point with `degrees` degrees, `minutes` minutes, and `seconds` seconds. It maps to datatype `atc.datatype.Longitude_ATC`.

```
RALT index_a feet
```

Defines an altitude reference with index $\texttt{index}_{\texttt{a}}$ for an altitude point at `feet` feet. It maps to datatype `atc.datatype.Altitude_ATC`.

```
NODE index_n index_l index_o index_a
```

Defines a node with index $\texttt{index}_{\texttt{n}}$ composed of latitude reference $\texttt{index}_{\texttt{l}}$, longitude reference $\texttt{index}_{\texttt{o}}$, and altitude reference $\texttt{index}_{\texttt{a}}$. It maps to datatype `atc.datatype.CoordinatesWorld3D_ATC`.

```
SURFACE index_s index_n_i
```

Defines a surface with index $\texttt{index}_{\texttt{s}}$ composed of at least three node references $\texttt{index}_{\texttt{n}_i}$. It maps to data structure `List<atc.datatype.CoordinatesWorld3D_ATC>`.

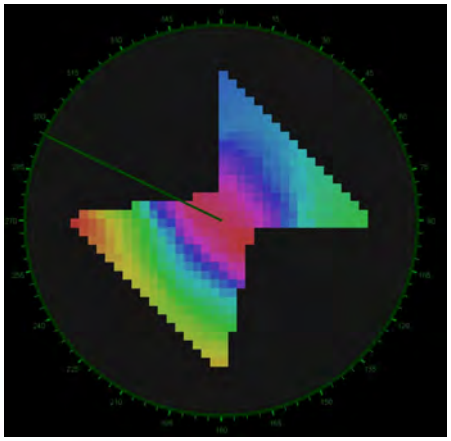
```
TERRAIN index_s_i
```

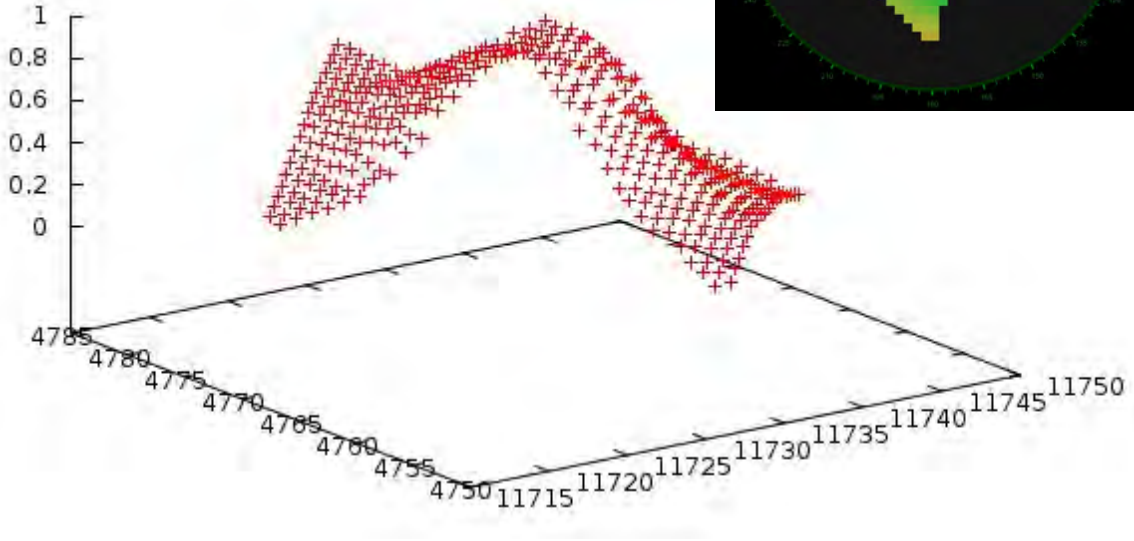
Defines the one and only terrain composed of at least one surface reference $\texttt{index}_{\texttt{s}_i}$. Duplicate surfaces are not allowed. It maps to data structure `List<List<atc.datatype.CoordinatesWorld3D_ATC>>`.

These six statement types may appear in any order. Whitespace does not matter. Everything is case-insensitive. Indexes must be unique within the context of a statement type, but may be reused by other statement types. For example, index 1 may be used for `RLAT` and `RLON` statements, but it must not be used for two `RLAT` statements. Ignore C++-style comments. Indexes are always positive.

The following example appeared in Lecture 32 to demonstrate the development of the terrain engine:

```
RLAT 1 47:40:58.8
RLON 1 117:19:21
RALT 1 1957

RLAT 2 47:50:58.8
RLON 2 117:19:21
RALT 2 2000

RLAT 3 47:40:58.8
RLON 3 117:9:21
RALT 3 1500

RLAT 4 47:30:58.8
RLON 4 117:19:21
RALT 4 1000

RLAT 5 47:40:58.8
RLON 5 117:29:21
RALT 5 750

RLAT 6 47:43:0
RLON 6 117:17:0
RALT 6 4000

NODE 1 1 1 1
NODE 2 2 2 2
NODE 3 3 3 3
NODE 4 4 4 4
NODE 5 5 5 5
NODE 6 6 6 6

SURFACE 1 2 3 1
SURFACE 2 4 6 5

TERRAIN 1 2
```



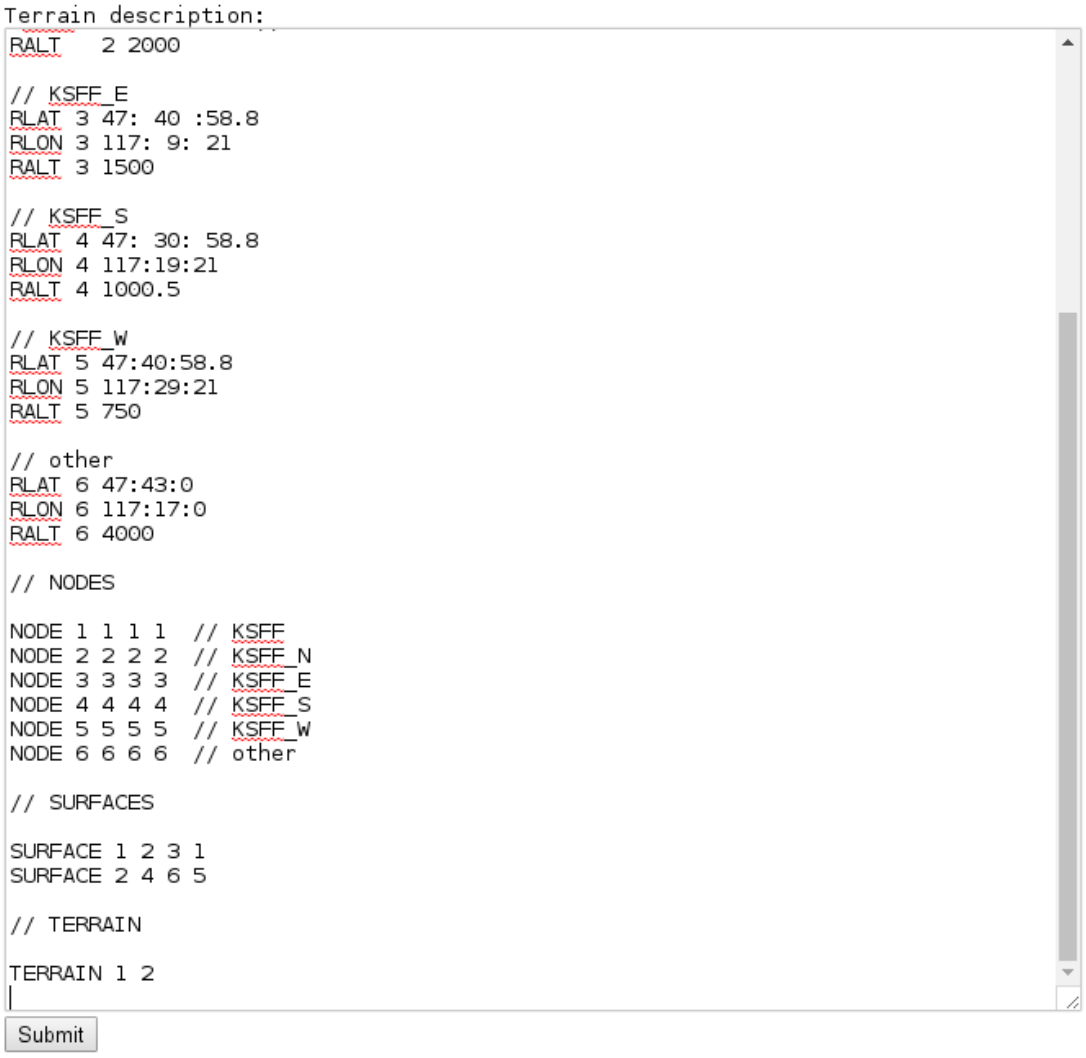


It maps to this Java code:

```
List<List<CoordinatesWorld3D_ATC>> terrain = new ArrayList<>();

List<CoordinatesWorld3D_ATC> surface1 = new ArrayList<>();

surface1.add(new CoordinatesWorld3D_ATC(
               new Latitude_ATC(47, 50, 58.8), new Longitude_ATC(117, 19, 21), new Altitude_ATC(2000)));
surface1.add(new CoordinatesWorld3D_ATC(
               new Latitude_ATC(47, 40, 58.8), new Longitude_ATC(117,  9, 21), new Altitude_ATC(1500)));
surface1.add(new CoordinatesWorld3D_ATC(
               new Latitude_ATC(47, 40, 58.8), new Longitude_ATC(117, 19, 21), new Altitude_ATC(1957)));

terrain.add(surface1);

List<CoordinatesWorld3D_ATC> surface2 = new ArrayList<>();

surface2.add(new CoordinatesWorld3D_ATC(
               new Latitude_ATC(47, 30, 58.8), new Longitude_ATC(117, 19, 21), new Altitude_ATC(1000)));
surface2.add(new CoordinatesWorld3D_ATC(
               new Latitude_ATC(47, 43,  0),   new Longitude_ATC(117, 17,  0), new Altitude_ATC(4000)));
surface2.add(new CoordinatesWorld3D_ATC(
               new Latitude_ATC(47, 40, 58.8), new Longitude_ATC(117, 29, 21), new Altitude_ATC(750)));

terrain.add(surface2);
```

Create class `project.loader.terrain.TerrainLoader` as defined in the Javadoc public API. The constructor takes a file handle to the definition file. The `parse()` method returns the terrain as defined above. No other public methods are permitted.

My solution is available online for comparison at `shelby.ewu.edu/cs350_elicit_terrain`. For example, this input



produces this output (cut off on the right):

```
Description:
// hello

/* ignore
this
stuff */

// REFERENCES

// KSFF
RLAT 1 47:32:58.8
RLON 1 117:19:21
RALT 1 1957

// KSFF_N
RLAT 2 47:50:58.8
 RLON 2 117:19:21  // I am a comment
RALT   2 2000

// KSFF_E
RLAT 3 47: 40 :58.8
RLON 3 117: 9: 21
RALT 3 1500

// KSFF_S
RLAT 4 47: 30: 58.8
RLON 4 117:19:21
RALT 4 1000.5

// KSFF_W
RLAT 5 47:40:58.8
RLON 5 117:29:21
RALT 5 750

// other
RLAT 6 47:43:0
RLON 6 117:17:0
RALT 6 4000

// NODES

NODE 1 1 1 1  // KSFF
NODE 2 2 2 2  // KSFF_N
NODE 3 3 3 3  // KSFF_E
NODE 4 4 4 4  // KSFF_S
NODE 5 5 5 5  // KSFF_W
NODE 6 6 6 6  // other

// SURFACES

SURFACE 1 2 3 1
SURFACE 2 4 6 5

// TERRAIN

TERRAIN 1 2

Structure:
[
[coordinates=[latitude=[degrees=47 N minutes=50 seconds=58.8] longitude=[degrees=117 W minutes=19 seconds=21.0]]
[coordinates=[latitude=[degrees=47 N minutes=30 seconds=58.8] longitude=[degrees=117 W minutes=19 seconds=21.0]]
]
```

Part I.b Communication Loader

The role of the communication loader is to translate bidirectionally between basic flight-related utterances and their encoded representation ostensibly stored on various data recorders in esoteric formats.

The translator uses a dictionary to substitute known words with numerical codes, and vice versa. For example, dictionary d={1=DOG, 2=THE, 3=CAT, 4=CHASED} accommodates encoding the utterance *THE DOG CHASED THE CAT* as 2 1 4 2 3. (For simplicity, all text is uppercase with no punctuation. Numbers must be spelled out; e.g., 13 is ONE THREE) The default dictionary contains the majority of the flight-related words and variants needed to process expected utterances in the project:

> *a*, *above*, *across*, *affirmative*, *after*, *again*, *ah*, *airport*, *alpha*, *altitude*, *american*, *an*, *and*, *approach*, *approved*, *are*, *as*, *at*, *atis*, *before*, *begin*, *behind*, *below*, *bravo*, *center*, *cessna*, *charlie*, *clearance*, *cleared*, *climb*, *contact*, *correct*, *course*, *cross*, *declare*, *decrease*, *degrees*, *delay*, *delivery*, *delta*, *departure*, *descend*, *direct*, *discretion*, *dme*, *do*, *east*, *echo*, *eight*, *emergency*, *enable*, *engine*, *established*, *execute*, *executing*, *expect*, *failure*, *feet*, *filed*, *fire*, *five*, *flight*, *for*, *four*, *foxtrot*, *frequency*, *front*, *fuel*, *gate*, *golf*, *has*, *have*, *heading*, *heavy*, *hold*, *hotel*, *how*, *hundred, ident*, *if*, *ils*, *immediate*, *inbound*, *increase*, *india*, *intentions*, *intercept*, *is*, *juliett*, *kilo*, *land*, *landing*, *leaving*, *left*, *level*, *lima*, *looking*, *maintain*, *mayday*, *mike*, *minute*, *minutes*, *missed*, *navigation*, *ndb*, *negative*, *niner*, *north*, *northeast*, *northwest*, *not*, *november*, *o'clock*, *of*, *one*, *only*, *option*, *or*, *oscar*, *out*, *outbound*, *own*, *pan-pan*, *papa*, *per*, *pilot*, *point*, *problem*, *proceed*, *quebec*, *radar*, *ramp*, *readback*, *ready*, *report*, *request*, *resume*, *right*, *risk*, *roger*, *romeo*, *runway*, *say*, *second*, *seconds*, *service*, *seven*, *sierra*, *sight*, *six*, *south*, *southeast*, *southwest*, *speed*, *squawk*, *standby*, *takeoff*, *tango*, *taxi*, *taxiway*, *terminal*, *terminated*, *terrain*, *the*, *then*, *thousand, three*, *to*, *tower*, *traffic*, *turn*, *two*, *um*, *unable*, *uniform*, *united*, *until*, *vector*, *vfr*, *via*, *victor*, *vor*, *west*, *what*, *when*, *where*, *whether*, *whiskey*, *who*, *why*, *wilco*, *will*, *with*, *x-ray*, *yankee*, *you*, *zero*, *zulu*

Although they are alphabetized here, the order does not matter. The dictionary also allows custom entries; e.g., $d \cup$ {5=SHELBY} accommodates 5 4 2 3.

Each utterance also has a timestamp to indicate when it occurred. For example:

Original	Encoded
10.0 THE DOG CHASED THE CAT	**10.0** 2 1 4 2 3
10.5 SHELBY CHASED THE CAT	**10.5** 5 4 2 3
11.7 THE CAT CHASED THE DOG SHELBY	**11.7** 2 3 4 2 1 5

To ensure that the encoding and decoding entities have the same dictionary, the encoded form includes a checksum, which is an arbitrary hash of the dictionary. The encoded form might appear as follows:

Encoded with Checksum

```
12345
10.0 2 1 4 2 3
10.5 5 4 2 3
11.7 2 3 4 2 1 5
```

With this representation, it should be possible to bidirectionally translate individual utterances and entire sequences of utterances, known here as a log. This approach is simple and effective (it would qualify as a CSCD 211 lab), but the representation is not consistent with what actually happens with realistic data transfers. For exposure to more real-world environments, we use a fixed-size base-encoded representation that resembles a binary file. For example:

Fixed-Size Base-Encoded

```
12345
01000000201040203
01050000504020 3
01170000203040201 05
```

The time values in blue appear in format **iii**rrrr, where `i` and `r` stand for the integer and real parts, respectively. The decimal place disappears because its position is known and never moves. Thus, `10.5` becomes **010**5000. The utterances appear in format `ii`, so `5 4 2 3` appears as 05040203. Fixed fields eliminate the need for delimiters. For simplicity, the checksum does not undergo any transformation.

A problem with this example is that two digits can accommodate only 100 words. One way to change the word space is to change the number of digits; i.e., the field size can be appropriate for the dictionary size (at least within an order of magnitude). Our solution accommodates any field size as a configuration parameter in the constructor.

Another way to make better use of the available space is to change the numerical base. Two digits in base 10 accommodate 100 words (from 99), but in base 16 (hexadecimal), it is 256 (from FF). Our solution accommodates any base between two (binary) and 36 (hexatrigesimal) as a configuration parameter in the constructor. (Base 36 is not remotely as rare as you might expect.)

The following code snippet illustrates a typical execution story for encoding:

```
CommunicationLoader loader1 = new CommunicationLoader(16, 2); // base, field_size

loader1.registerCustomWord(loader1.getNextFreeIndex(), "DOG");

int checksum = loader1.getDictionaryChecksum();

String encoding = loader1.encodeStatement(123.45,
            "dog one two three cleared to land runway three one left");

System.out.println("checksum=" + checksum);

System.out.println("encoding=" + encoding);
```

The output is:

```
checksum=1768562
encoding=12D644C773ACA71DA85B8EA7735E
```

The number of places in the blue time field is determined by $\lceil \log(9999999) / \log(base) \rceil$, where `9999999` derives from the maximum time `999.9999`. `12D644` in hex is `1234500` in decimal, and therefore `123.4500` because the implied decimal point is always three places from the left. Assume we always receive a time that fits into seven base-10 places.

The remainder of the output is based on the particular word substitutions in the dictionary with a field size of two. Your output may differ due to your implementation, which is fine. The critical part is that your decoder function in reverse according to the same scheme.

The following code snippet illustrates a typical execution story for decoding this example:

```
String log = ("1768562\n" +                     // checksum
              "12D644C773ACA71DA85B8EA7735E");  // encoding

CommunicationLoader loader2 = new CommunicationLoader(16, 2);

loader2.registerCustomWord(loader2.getNextFreeIndex(), "DOG");

String decoding = loader2.decodeLog(log);
```

The output is:

```
decoding=
123.4500: DOG ONE TWO THREE CLEARED TO LAND RUNWAY THREE ONE LEFT
```

Note that the entire log, including checksum, can be decoded at once with `decodeLog()`, but there is no way to encode all at once. You must call `getDictionaryChecksum()` once and `encodeStatement()` for each statement. For bidirectional translation to work correctly, the configurations in the constructors must match exactly, as must the dictionaries. The checksum verifies both. In a real system, it would also verify the encoding, but this aspect is not considered because we assume no errors can occur in transmission.

Hints

This task involves math (actually only arithmetic). The math is not complex or difficult, but it offers plenty of opportunity for something to go wrong. Be sure to understand what you are telling your code to do. Do not hack the math! It will not work out well for you. Stop and think. Work it out on paper first. Ask questions. Start early.

Consider using `Integer.toString(int,int)` and `Integer.valueOf(String,int)` for base conversions and `HashMap` for the dictionary.

The larger solution is composed of its parts. If the parts do not work, neither will the whole. Approach your solution bottom-up by getting the parts working before assembling them. The following methods may be helpful.

Use this method as a warm-up exercise to be sure you truly understand how numbers work.

int **calculateMaxValue**(int base, int size)

It returns the maximum base-10 value that can be represented in `size` digits in base `base`. For example, familiar base 10 with size 3 returns 999 because 9 is the largest digit that can fill any place, and there are three places. Similarly, base 16 returns 4095 from `FFF`.

Make sure you can encode a time and get the same time back when you decode it:

String **encodeTime**(double time)

double **decodeTime**(String time)

Make sure you can encode a statement and get the same statement back when you decode it. Since this statement translation uses the time translation, the latter must be working at this point.

String **encodeStatement**(double time, String words)

String **decodeStatement**(String statement)

Decoding the entire log is the process of decoding each statement, plus the checksum aspect.

String **decodeLog**(String log)

Make sure you can add a custom word and that it changes the dictionary checksum:

String **registerCustomWord**(String index, String word)

The typical form of this call is:

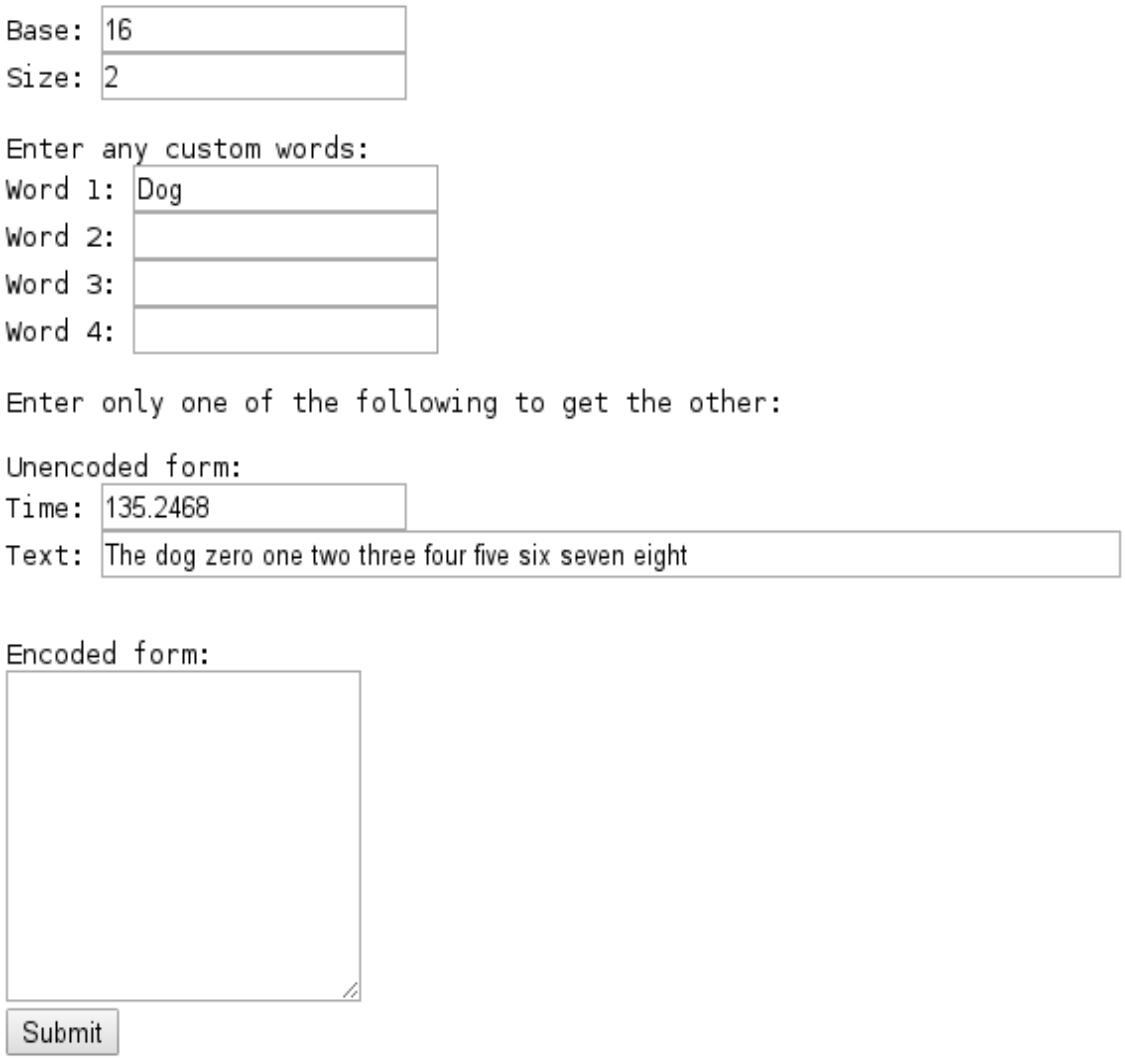
registerCustomWord(**getNextFreeIndex**(), word)

Create class `project.loader.communication.CommunicationLoader` such that it satisfies the description here and in the Javadoc. No other public methods are permitted.

The Javadoc indicates in square brackets the number of lines of functional code (i.e., not related to error checking) my solution took. Use them as guidelines for your solution. Do not make yours more complex than necessary. Think before doing! Verify after doing! No hacking! No magic numbers will work in all cases, so do not even think of using any! Ask yourself what every line of code really does, and do not proceed if you cannot confidently articulate a reasonable answer to yourself and your teammates.

Methods `isValidIndex()` and `isValidCustomIndex()` are no longer required.

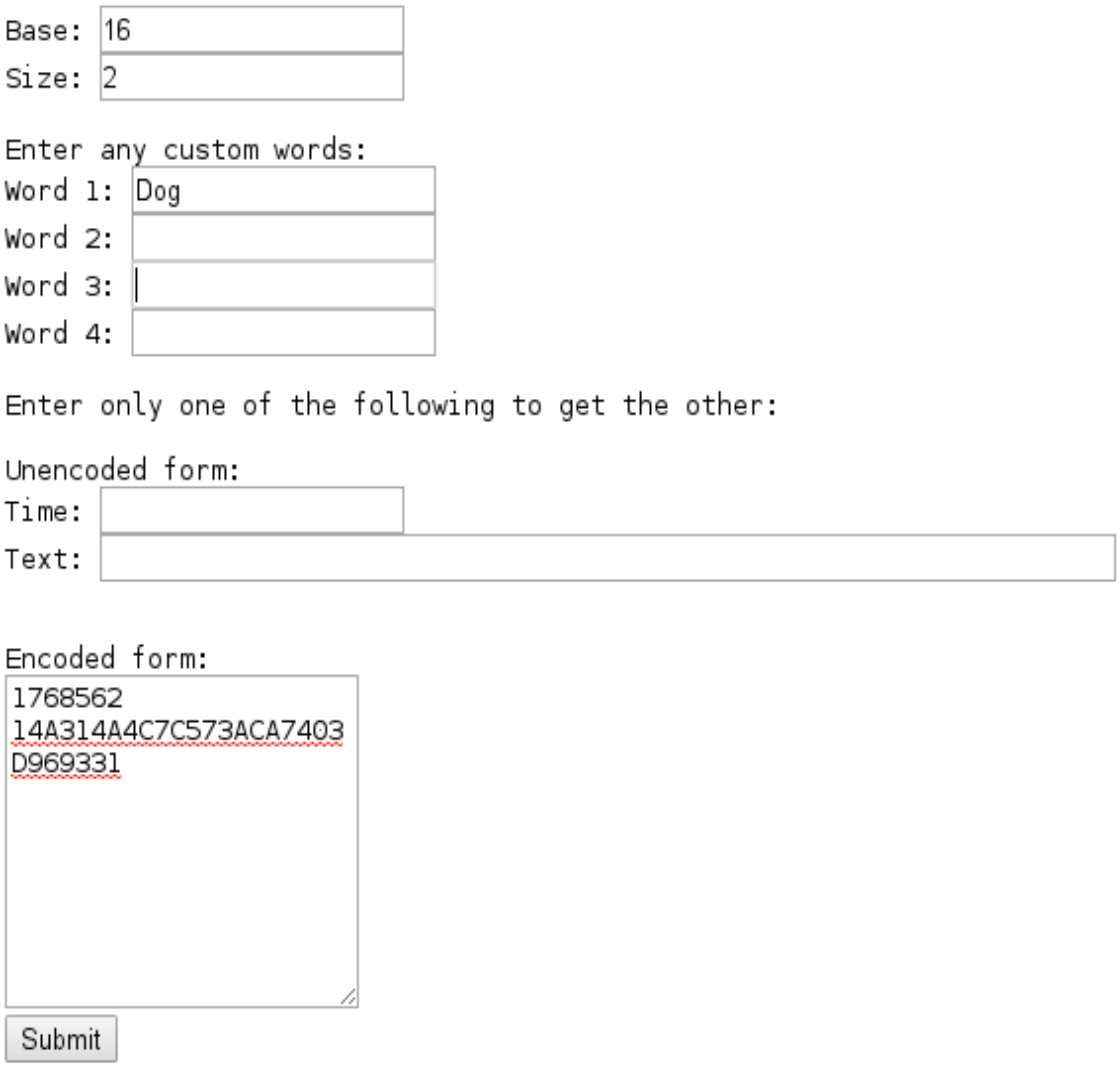
My solution is available online for comparison at `shelby.ewu.edu/cs350_elicit_communication`. For example, this input



produces this output:

```
Base = 16
Size = 2

Custom Word 1: Dog
Custom Word 2:
Custom Word 3:
Custom Word 4:

Unencoded:
135.2468: THE DOG ZERO ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT

Encoded:
1768562
14A314A4C7C573ACA7403D969331
```

In the inverse direction, this input



produces this output:

```
Base = 16
Size = 2

Custom Word 1: Dog
Custom Word 2:
Custom Word 3:
Custom Word 4:

Unencoded:
135.2468: THE DOG ZERO ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT

Encoded:
1768562
14A314A4C7C573ACA7403D969331
```

Examples

The following outputs are from the same example but by calling `CommunicationLoader` with different configurations. The blue portion is the timestamp.

```
(2,8):

0001001011010110010001001100011101110011101011001010011100011101101010000101101110001110101001110111001101011110
```

```
(8,3):

045531043071632542470352501332162471631 36
```

```
10,3):

12345001991151721670291680911421671150 94
```

```
(16,4):

12D64400C7007300AC00A7001D00A8005B008E00A70073005E
```

```
(36,2):

0QGJO5J374S4N0T4O2J3Y4N372M
```

All produce the identical decoding:

```
123.4500: DOG ONE TWO THREE CLEARED TO LAND RUNWAY THREE ONE LEFT
```

Part II: World Loader

The role of the world loader is to read the commands that define the navigation aids and airports in the world. Implement `project.loader.world.WorldLoaderParser` to accept the language as follows:

- The constructor is `public WorldLoaderParser(WorldLoader loader, InputStream stream)`, where `stream` contains the commands. Instantiate your own `WorldLoader` in your tester, which will instantiate your `WorldLoaderParser` as defined here. Call `WorldLoader.load()` to execute your parser on your input and present the results in XML format.
- Method `void parse()` executes the parser, which calls `WorldLoader.addNavaids()` and `WorldLoader.addAirports()` as indicated below.

Be sure to use JAR 0.5 or later.

Punctuation is not part of commands unless it is in blue. Vertical bar indicates logical *or*. Asterisk indicates zero or more instances of the preceding term or parenthetical group; plus indicates one or more. Square brackets indicate an optional group. Singular and plural forms (with an `[S]` suffix) need not correspond grammatically to the number of elements in a list.

Whitespace, except in literals, does not matter. All text except identifiers is case insensitive. All statements appear on a separate line.

Identifiers must be unique within the navaid definitions and the airport-component definitions.

Italicized fields are defined as follows:

Field	Format	Datatype
Altitude	*Number*	Altitude_ATC
Beacon	(*Distance* , *Altitude*)	NavaidILSBeaconDescriptor
Bearing	*Real*	AngleNavigational_ATC
CoordinatesAnchor	< *Number* , *Number* >	CoordinatesAnchor_ATC
CoordinatesCartesian	{ *Number* , *Number* }	CoordinatesCartesianAbsolute_ATC
CoordinatesWorld	*Latitude* : *Longitude*	CoordinatesWorld_ATC
CoordinatesWorld3D	*Latitude* : *Longitude* : *Altitude*	CoordinatesWorld3D_ATC
Distance	*Real*	Distance_ATC
FrequencyUHF	*Integer*	UHFFrequency_ATC
FrequencyVHF	*Real*†	VHFFrequency_ATC
id	any ordinary Java identifier with letters, numbers, and underscores	String
Integer	any ordinary integer, with optional - or +	int
Latitude	*Integer* # *Integer* ' *Number* "	Latitude_ATC
Longitude	*Integer* # *Integer* ' *Number* "	Longitude_ATC
Number	*Integer* | *Real*	double
Real	any ordinary real value, with optional - or +	double
String	any ordinary string delimited by single quotes; no escape characters	String

†Whole values require a `.0` suffix.

Anchor and Cartesian coordinates are in feet relative to a reference point on the component that they belong to.

1. CREATE NAVAID AIRWAY id_1 FROM id_2 TO id_3

Creates airway id_1 from navaid id_2 to navaid id_3. Use `ComponentNavaidAirway`.

2. CREATE NAVAID FIX *id* AT *CoordinatesWorld*

Creates navigation fix *id* at coordinates *CoordinatesWorld*. Use `ComponentNavaidFix`.

3. `CREATE NAVAID NDB id AT CoordinatesWorld3D ON FREQUENCY FrequencyUHF`

Creates nondirectional beacon *id* on frequency *FrequencyUHF* at coordinates *CoordinatesWorld3D*. Use `ComponentNavaidNDB`.

4. `CREATE NAVAID VOR id AT CoordinatesWorld3D ON FREQUENCY FrequencyVHF`

Creates VOR station*id* on frequency *FrequencyVHF* at coordinates *CoordinatesWorld3D*. Use `ComponentNavaidVOR`.

5. `CREATE NAVAID ILS id AT CoordinatesWorld3D BEARING Bearing ON FREQUENCY FrequencyVHF WITH BEACONS` $Beacon_1$ $Beacon_2$ $Beacon_3$

Creates instrument landing system *id* on frequency *FrequencyVHF* at coordinates *CoordinatesWorld3D* with its inner (n=1), middle (2), and outer (3) marker beacons respectively $Beacon_n$ at angle *Bearing*. Use `ComponentNavaidILS`.

6. `CREATE AIRPORT` id_1 `AT CoordinatesWorld3D WITH [BUILDING[S]` id_2`+] [GATE[S]` id_3`+] [TOWER[S]` id_4`+] [SIGN[S]` id_5`+] [RAMP[S]` id_6`+] [TAXIWAY[S]` id_7`+] RUNWAY[S]` id_8`+`

Creates airport id_1 at coordinates *CoordinatesWorld3D* with runways id_8 and buildings id_2, gates id_3, towers id_4, signs id_5, ramps id_6, and taxiways id_7. Use `ComponentAirport`.

7. `CREATE BUILDING` id_1 `FROM` id_2

Creates building id_1 from polyline id_2. Use `ComponentAirportBuilding`.

8. `CREATE GATE id CALLED String AT CoordinatesCartesian`

Creates terminal gate *id* named *String* at coordinates *CoordinatesCartesian*. Use `ComponentAirportGate`.

9. `CREATE RAMP` id_1 `FROM` id_2

Creates ramp id_1 from polyline id_2. Use `ComponentAirportRamp`.

10. `CREATE RUNWAY` id_1 `FROM` id_2

Creates runway id_1 from polyline id_2. Use `ComponentAirportRunway`.

11. `CREATE SIGN id LABELED String AT CoordinatesCartesian`

Creates information sign *id* labeled *String* at coordinates *CoordinatesCartesian*. Use `ComponentAirportSign`.

12. `CREATE TAXIWAY` id_1 `FROM` id_2

Creates taxiway id_1 from polyline id_2. Use `ComponentAirportTaxiway`.

13. `CREATE TOWER id AT CoordinatesCartesian`

Creates tower *id* at coordinates *CoordinatesCartesian*. Use `ComponentAirportTower`.

14. `ADD AIRPORT[S]` id_n`+`

Adds airports id_n to the world through `WorldLoader.addAirports()`.

15. `ADD NAVAID[S]` id_n`+`

Adds navaids id_n to the world through `WorldLoader.addNavaids()`.

16. `DEFINE POLYLINE id AS ({`$CoordinatesAnchor_{n1}$ $CoordinatesAnchor_{n2}$`} | `$CoordinatesAnchor_{n3}$`)+`

Defines polyline *id* as connections of coordinates. The first form specifies a line segment from n_1 to n_2. The second specifies n_3 as a continuation from the previous point. Use `GeometryPolyline.addLine()`.

My solution is available online for comparison at `shelby.ewu.edu/cs350_elicit_world`. For example, this input

World description:

```
CREATE NAVAID FIX myfix1 AT 47#40'58.8" : 117#19'21"
CREATE NAVAID FIX myfix2 AT 47#50'58.8" : 117#20'21"

CREATE NAVAID AIRWAY myairway1 FROM myfix1 TO myfix2

CREATE NAVAID NDB myndb1 AT 47#32'38" : 117#17'25" : 1200 ON FREQUENCY 220

CREATE NAVAID VOR myvor1 AT 47#30'58" : 117#19'20" : 1000 ON FREQUENCY 123.45

CREATE NAVAID ILS myils1 AT 47#30'38" : 117#19'50" : 1500 BEARING 45.8 ON FREQUENCY 112.325
WITH BEACONS (1,1200) (2,1800) (3.5,2400)

ADD NAVAIDS myfix1 myfix2 myairway1 myvor1
ADD NAVAIDs myndb1 myils1

CREATE SIGN mysign1 LABELED 'my sign rules!' AT {123,-45}

CREATE GATE mygate1 CALLED 'Gate 99Z' AT {-123,0}

CREATE TOWER mytower1 AT {-12.3,45.6}

DEFINE POLYLINE mypoly1 AS {<1,2> <3,4>}
DEFINE POLYLINE mypoly2 AS {<10,20> <30,40>} <5,6>
DEFINE POLYLINE mypoly3 AS {<-10,-20> <-30,0>} {<15,-25> <-35,-20>} <50,60>
DEFINE POLYLINE mypoly4 AS {<11,2> <13,4>}
DEFINE POLYLINE mypoly5 AS {<11,12> <13,14>}

CREATE RAMP myramp1 FROM mypoly1

CREATE RUNWAY myrunway1 FROM mypoly2
CREATE RUNWAY myrunway2 FROM mypoly5

CREATE TAXIWAY mytaxiway1 FROM mypoly3

CREATE BUILDING mybuilding1 FROM mypoly4

CREATE AIRPORT myairport1 AT 47#41'5":117#1'2":500 WITH RUNWAYS myrunway1

CREATE AIRPORT myairport2 AT 47#41'5":117#1'2":800 WITH BUILDINGS mybuilding1 GATES mygate1
TOWERS mytower1 SIGNS mysign1 RAMPS myramp1 TAXIWAYS mytaxiway1 RUNWAYS myrunway2

ADD AIRPORTS myairport1 myairport2
```

Submit

produces this raw XML output:

Structure

```
</coordinate-world-3d>
</position>
</navaid-fix>
<navaid-fix>
<id value="myfix2"/>
<position>
<coordinate-world-3d>
<coordinate-world>
<latitude_atc degrees="47" minutes="50" seconds="58.8"/>
<longitude_atc degrees="117" minutes="20" seconds="21.0"/>
</coordinate-world>
<altitude value="0.0"/>
</coordinate-world-3d>
```

The blue dots indicate where Chrome wrapped the text in the input area. Everything is actually on the same line.

Paste the XML contents into a text file with an XML extension and open it back up in the browser to see the pretty-printed form (abridged here):

```
▼<world>
  ▼<navaids>
    ▼<navaid-fix>
       <id value="myfix1"/>
      ▼<position>
        ▼<coordinate-world-3d>
          ▼<coordinate-world>
             <latitude_atc degrees="47" minutes="40" seconds="58.8"/>
             <longitude_atc degrees="117" minutes="19" seconds="21.0"/>
           </coordinate-world>
           <altitude value="0.0"/>
         </coordinate-world-3d>
       </position>
     </navaid-fix>
    ▼<navaid-fix>
       <id value="myfix2"/>
      ▼<position>
        ▼<coordinate-world-3d>
          ▼<coordinate-world>
             <latitude_atc degrees="47" minutes="50" seconds="58.8"/>
             <longitude_atc degrees="117" minutes="20" seconds="21.0"/>
           </coordinate-world>
           <altitude value="0.0"/>
         </coordinate-world-3d>
       </position>
     </navaid-fix>
    ▼<navaid-airway>
       <id value="myairway1"/>
      ▼<position-from>
        ▼<coordinate-world-3d>
          ▼<coordinate-world>
             <latitude_atc degrees="47" minutes="40" seconds="58.8"/>
             <longitude_atc degrees="117" minutes="19" seconds="21.0"/>
           </coordinate-world>
           <altitude value="0.0"/>
         </coordinate-world-3d>
       </position-from>
      ▼<position-to>
        ▼<coordinate-world-3d>
          ▼<coordinate-world>
             <latitude_atc degrees="47" minutes="50" seconds="58.8"/>
             <longitude_atc degrees="117" minutes="20" seconds="21.0"/>
           </coordinate-world>
           <altitude value="0.0"/>
         </coordinate-world-3d>
       </position-to>
     </navaid-airway>
```

Part III: System Test and Evaluation

In this final part, you will evaluate my complete working solution in the role of the intended end user, an NTSB trainee.

Be sure to use JAR 0.7 or later. Execute it from the command line of your operating system:

```
java -jar cs350-project-release-v0_7.jar
```

If you execute it by double-clicking from your GUI, any error or terminal output messages are lost. The error handling in general is not friendly, so be careful. If you get any error, restart the simulator and try again. It is not unusual to have a transient startup failure resulting from a delay in your virtual machine to bring all the necessary services online in time.

The Javadoc is provided to show what the final implementation looks like. You do not need to use it.

The sample controller file `sample.nsc` contains this basic annotated example that produces a variation on the output below. Use the provided files, not the text here.

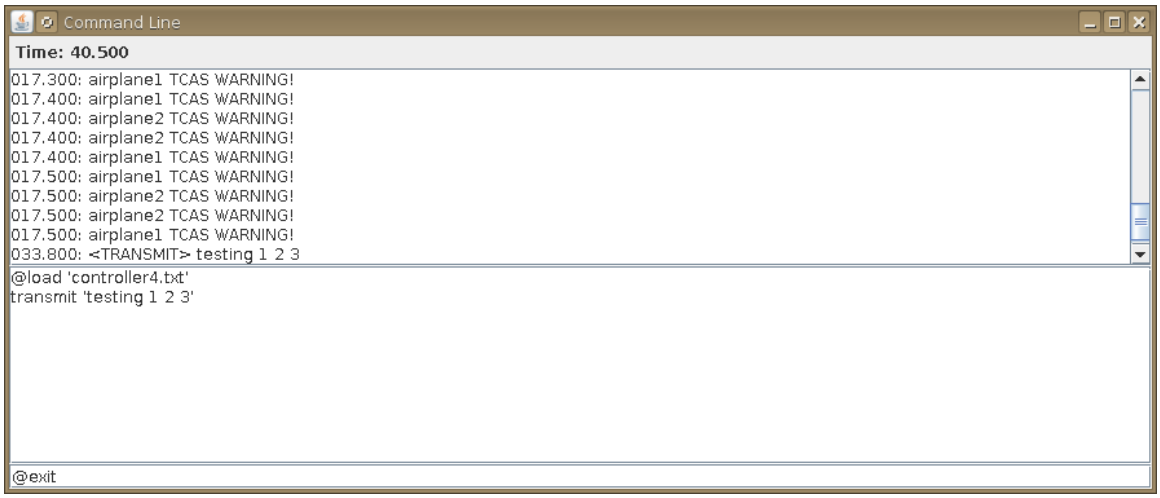
```
// putting the filename in a comment at the top allows for easy copy/paste execution
// @LOAD 'sample.nsc'

TRANSMIT 'starting the sample'

@IMPORT TERRAIN 'sample.trn'
@IMPORT COMMUNICATION 'sample.com' BASE 16 SIZE 2
@IMPORT WORLD 'sample.wld'

// place the airplane at the center of the radar display at 5000 feet traveling west
CREATE AIRPLANE airplane1
  AT 47#40'58":117#19'21":5000
  COURSE 270
  POWER 50
  GEAR DOWN

@COMMIT

@CREATE RADAR radar1 AT 47#40'58":117#19'21" SIZE 0#25'0"

// wait a few seconds for the airplane to fly straight before turning to the right
@WAIT 10

COMMAND airplane1 AILERON 8
COMMAND airplane1 ELEVATOR 12

// exit after 65 seconds of simulation time
@SCHEDULE 60 <TRANSMIT 'ending the sample in five seconds'>
@SCHEDULE 65 <@EXIT>
```

The communication file `sample.com` calls out every 10 seconds for a minute:

```
1762906
0186A01273C591
030D4012ACC591
0493E012A7C591
061A801240C591
07A120123DC591
0927C0127365
```

Command Line Interface

Enter commands into the bottom field. Use the up and down arrows to scroll through the command history in the middle field. Output from the simulation appears in the top field. The current simulation time appears at the top. Execute the sample with `@LOAD 'sample.nsc'`



Instrument Panel

There is one instrument panel per `CREATE AIRPLANE` command (below). All the instruments work, but most are not relevant to the requirements of this project. Probably the only ones you may need are these:

A: Airspeed indicator in knots.

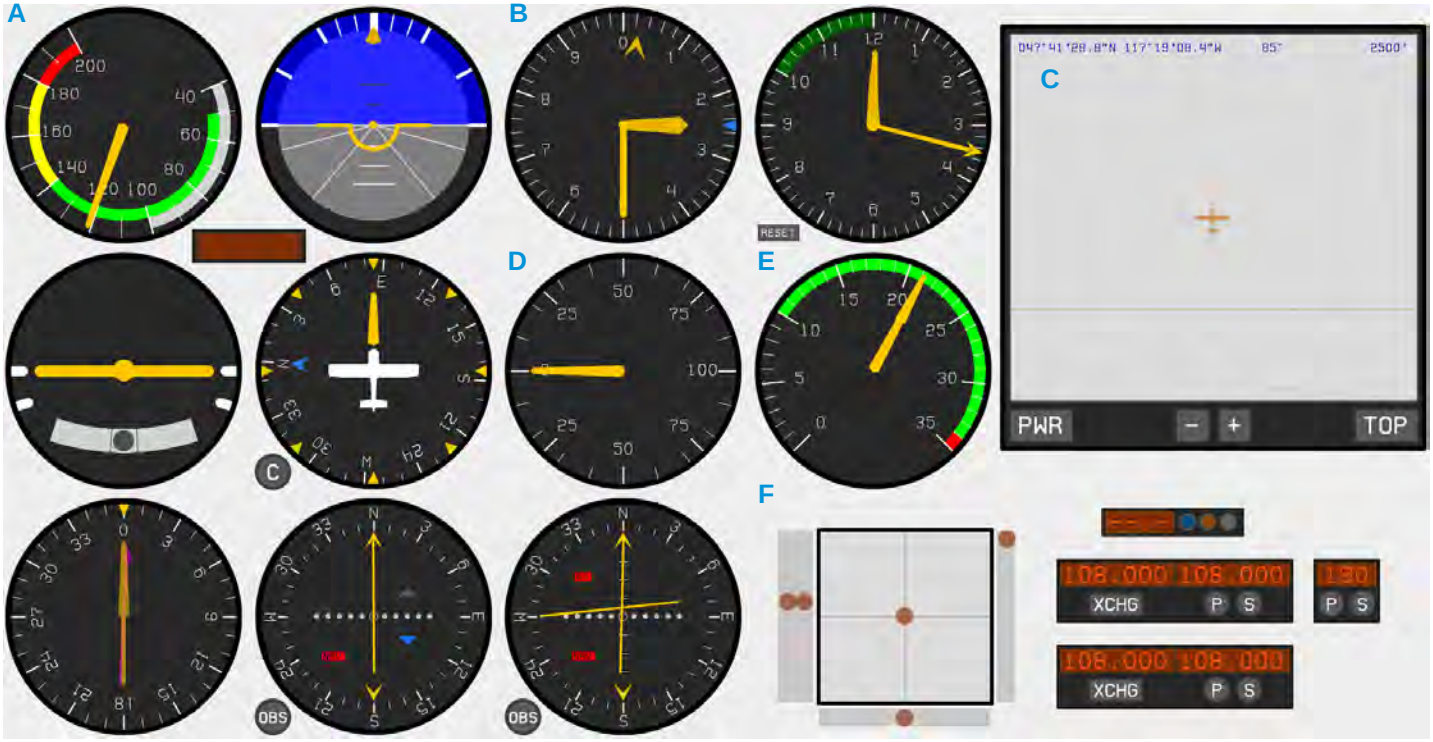
B: Altimeter. The orange caret, thick needle, and thin needle are respectively 10 thousands, one thousands, and hundreds of feet above sea level, which is the case when there is no terrain. The blue bug is altitude in thousands of feet above the ground. It turns red below one thousand feet to indicate a TAWS (Terrain Awareness and Warning System) alert.

C: GPS. The `TOP` button alternates between north up and track up. The + and – buttons zoom.

D: Vertical speed indicator in feet per minute

E: Thrust indicator (sort of). It is actually a tachometer, but we are using it to indicate the power of the two engines. The longer needle refers to the left engine; the shorter one is the right. When both engines are producing the same power, the needles are joined, as here.

F: Control indicator. Each dot corresponds to a control input. A green dot refers to the expected state; red refers to the actual. When they agree, the overlapping color is orange. The left two dots are the left and right engines from minimum power upward to maximum. The bottom center dot is the rudder. The right dot is the flaps. The center dot depicts a control stick with aileron on the horizontal axis and elevator on the vertical. The word `GEAR` appears at the top left when the gear is down.

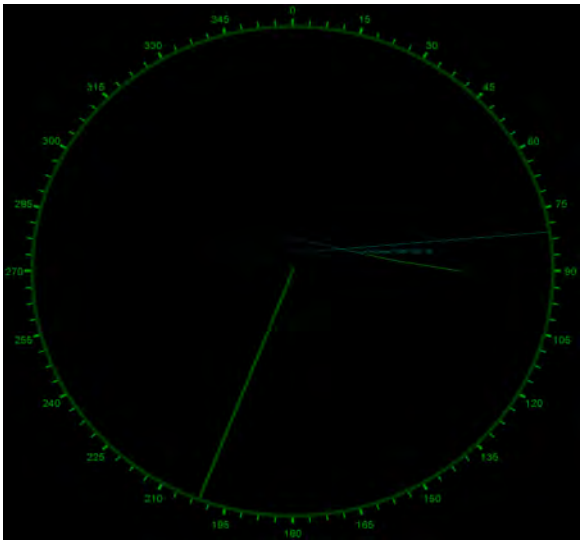


Three-Dimensional State Viewer

There is one state viewer per `CREATE AIRPLANE` command (below). It shows only the airplane itself, not the world in which it is operating. Change the camera perspective with the cursor keys. The box in the middle indicates the axes of the world: purple is the pitch plane, red roll, and yellow yaw. The airplane model does not reflect any other details of the actual state, like the position of the flight control surfaces or landing gear.

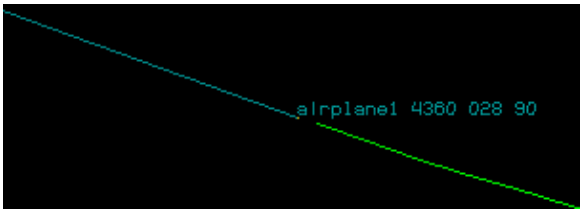
Radar Display

There is one radar display per `CREATE RADAR` command (below). It shows each airplane within the boundaries.



The symbols specify (left to right) the airplane identifier, the transponder squawk code (not used), the altitude in thousands of feet, and the speed in knots. The tags `/TCAS` and `/TAWS` appear when a traffic and terrain warnings are issued, respectively.

The green line is the breadcrumb track depicting where the airplane was over time. The teal line projects where it will be in 30 seconds at the current course and speed.

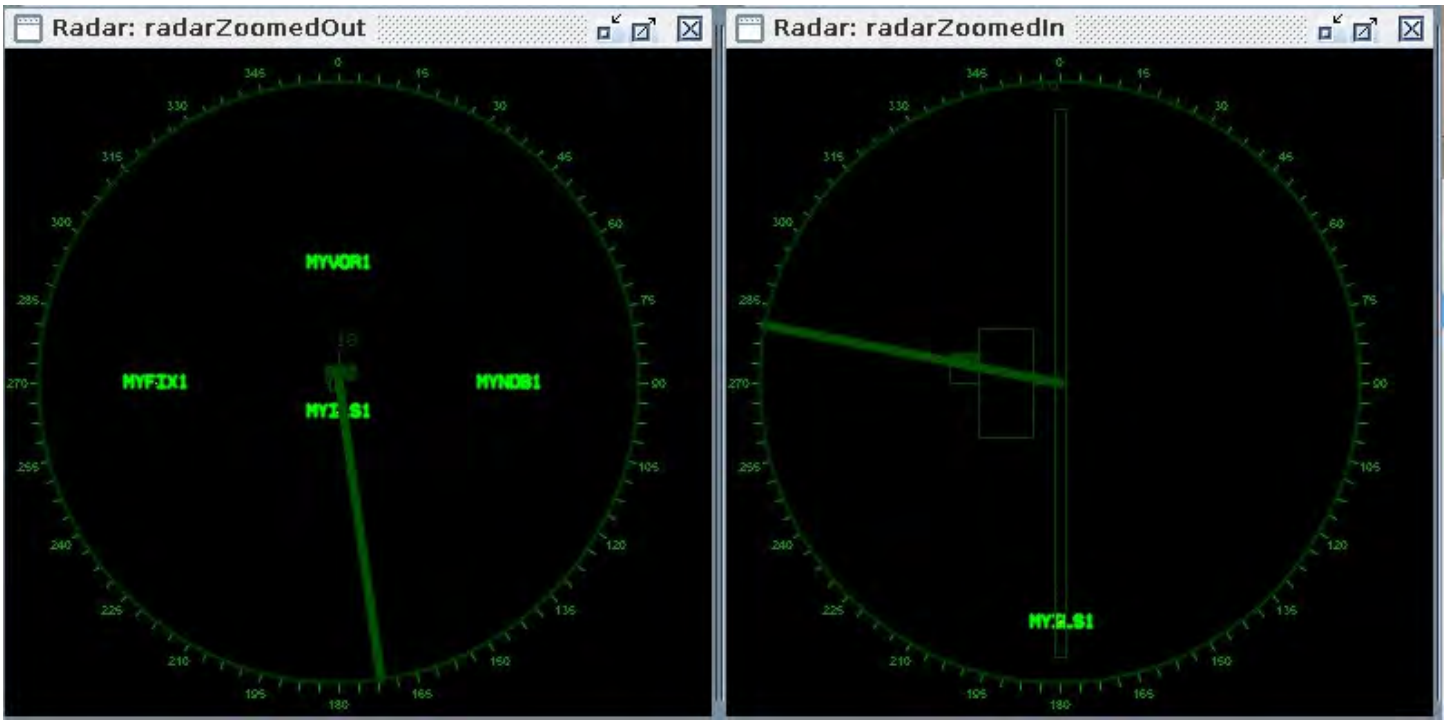


For the sample world `sample.wld`,

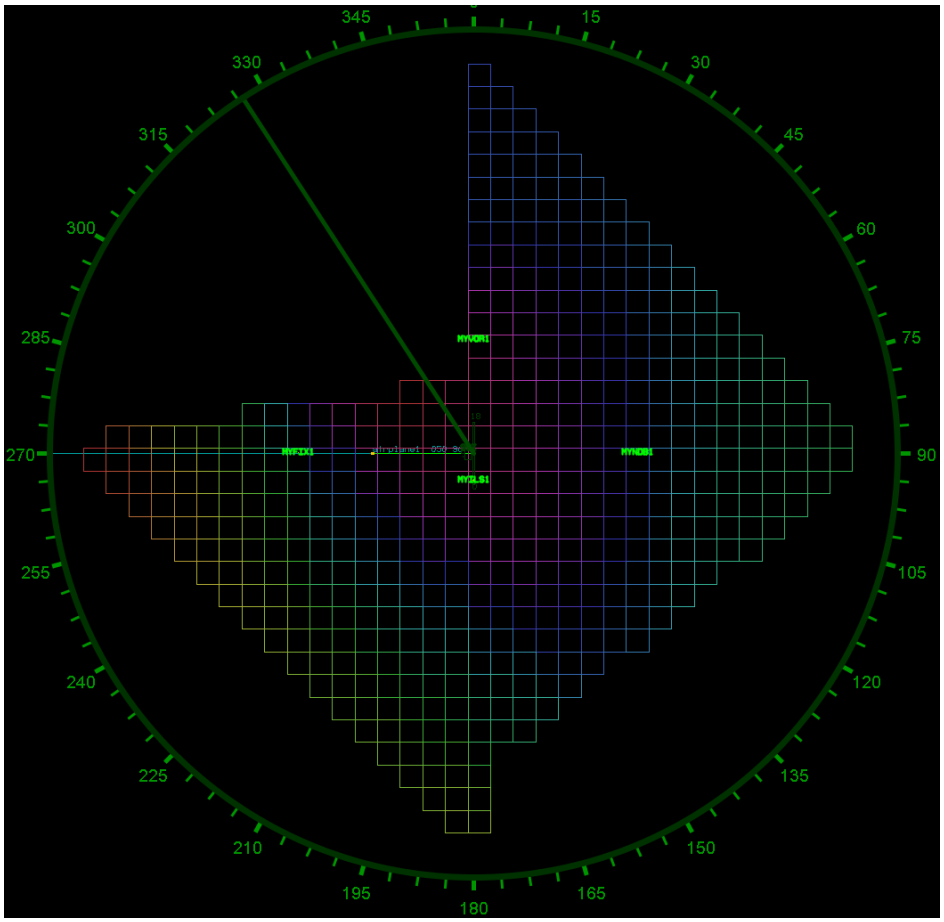
```
CREATE NAVAID FIX myfix1 AT 47#40'58.8" : 117#24'00"
CREATE NAVAID FIX myfix2 AT 47#40'58.8" : 117#15'00"

CREATE NAVAID AIRWAY myairway1 FROM myfix1 TO myfix2

CREATE NAVAID NDB myndb1 AT 47#40'58.8" : 117#15'00" : 1200 ON FREQUENCY 220

CREATE NAVAID VOR myvor1 AT 47#44'00" : 117#19'21" : 1000 ON FREQUENCY 123.45

CREATE NAVAID ILS myils1 AT 47#40'15" : 117#19'21" : 1500 BEARING 45.8 ON FREQUENCY 112.325
 WITH BEACONS (1,1200) (2,1800) (3.5,2400)

ADD NAVAIDS myfix1 myairway1 myvor1 myndb1 myils1

DEFINE POLYLINE mypoly1 AS { <-100,5000> <100,5000> } <100,-5000> <-100,-5000> <-100,5000>
DEFINE POLYLINE mypoly2 AS { <-100,0> <-500,0> } <-500,100> <-100,100>
DEFINE POLYLINE mypoly3 AS { <-500,1000> <-500,-1000> } <-1500,-1000> <-1500,1000> <-500,1000>
DEFINE POLYLINE mypoly4 AS { <-1500,500> <-2000,500> } <-2000,0> <-1500,0>

CREATE RUNWAY myrunway1 FROM mypoly1

CREATE TAXIWAY mytaxiway1 FROM mypoly2

CREATE RAMP myramp1 FROM mypoly3

CREATE BUILDING mybuilding1 FROM mypoly4
```

```
CREATE GATE mygate1 CALLED '99Z' AT {-2000,250}

CREATE TOWER mytower1 AT {-1750,-500}

CREATE SIGN mysign1 LABELED '0'  AT {-125,-6100}
CREATE SIGN mysign2 LABELED '18' AT {-400,+5300}

CREATE AIRPORT myairport1 AT 47#40'58":117#19'21":800 WITH BUILDING mybuilding1 GATE mygate1 TOWER mytower1
 SIGNS mysign1 mysign2 RAMP myramp1 TAXIWAY mytaxiway1 RUNWAY myrunway1

ADD AIRPORT myairport1
```

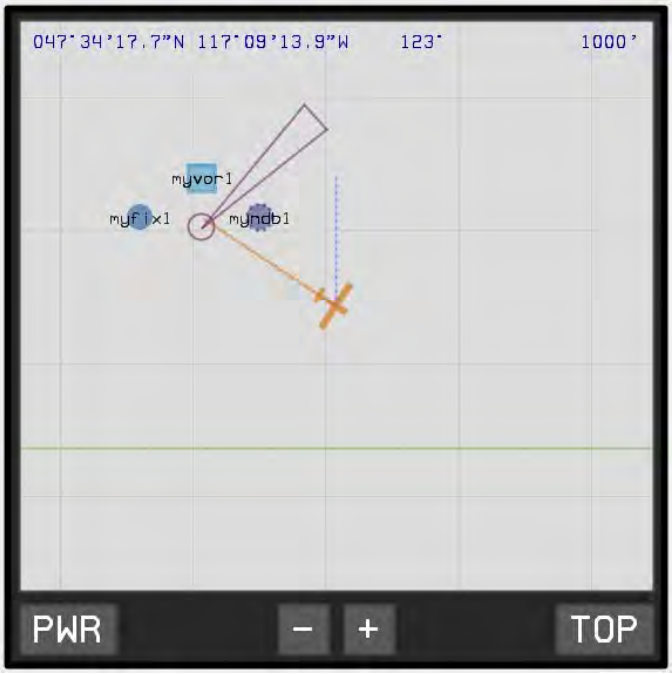
the left display is zoomed out to show the navigational aids. The right display is the airport to be used here.



The terrain appears color-coded from “cool” blue for low elevation to “hot” red for high. The gradient is relative between the lowest to the highest elevations. See the terrain file for the correspondence.



The GPS view is similar. You may have to zoom in before all features appear.



Execution Log

The states of every airplane are recorded by time in file `airplane.log`. See Task 4b for details on the 29 fields.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	# log started Fri Nov 25 13:13:27 PST 2016																
2	time_elapsed	callsign	longitude	latitude	altitude_pressure	altitude_radar	yaw	pitch	roll	course	airspeed	turn_rate	climb_rate	power_current_1	power_target_1	power_current_2	power_target_2
3	0.1	airplane1	117.2504166865	47.6820367101	1306.36255510939	1306.36255510939	0	30	3	278.05	90.9	0.5	4602.57	50.5	50.5	50.5	50.5
4	0.1	airplane2	117.4161086132	47.6828266011	2500	2500	0	0	0	85	121	0	0	60.5	60.5	60.5	60.5
5	0.2	airplane1	117.2508333716	47.682886006	1312.72610218/9	1312.72610218/9	0	30	3	278.1	90.9	0.5	4602.57	50.5	50.5	50.5	50.5
6	0.2	airplane2	117.4155805596	47.6828764745	2500	2500	0	0	0	85	121	0	0	60.5	60.5	60.5	60.5
7	0.3	airplane1	117.2512499046	47.6829556665	1319.0876532818	1319.0876532818	0	30	3	278.15	90.9	0.5	4602.57	50.5	50.5	50.5	50.5
8	0.3	airplane2	117.4149925081	47.6829242478	2500	2500	0	0	0	85	121	0	0	60.5	60.5	60.5	60.5
9	0.4	airplane1	117.2516664354	47.6830156885	1325.4502043757	1325.4502043757	0	30	3	278.2	90.9	0.5	4602.57	50.5	50.5	50.5	50.5
10	0.4	airplane2	117.4144244526	47.6829730712	2500	2500	0	0	0	85	121	0	0	60.5	60.5	60.5	60.5
11	0.5	airplane1	117.2520829137	47.683076075	1331.8127554696	1331.8127554696	0	30	3	278.25	90.9	0.5	4602.57	50.5	50.5	50.5	50.5
12	0.5	airplane2	117.4138763991	47.6830218946	2500	2500	0	0	0	85	121	0	0	60.5	60.5	60.5	60.5
13	0.6	airplane1	117.2524993391	47.6831368249	1338.1763066636	1338.1763066636	0	30	3	278.3	90.9	0.5	4602.57	50.5	50.5	50.5	50.5
14	0.6	airplane2	117.4133183466	47.6830707179	2500	2500	0	0	0	85	121	0	0	60.5	60.5	60.5	60.5
15	0.7	airplane1	117.2529157114	47.6831979382	1344.63785/6575	1344.63785/6575	0	30	3	278.35	90.9	0.5	4602.57	50.5	50.5	50.5	50.5
16	0.7	airplane2	117.4127602921	47.6831195413	2500	2500	0	0	0	85	121	0	0	60.5	60.5	60.5	60.5
17	0.8	airplane1	117.2533320301	47.6832594148	1350.9004087514	1350.9004087514	0	30	3	278.4	90.9	0.5	4602.57	50.5	50.5	50.5	50.5
18	0.8	airplane2	117.4122022306	47.6831683646	2500	2500	0	0	0	85	121	0	0	60.5	60.5	60.5	60.5
19	0.9	airplane1	117.2537482951	47.6833212547	1357.2629598453	1357.2629598453	0	30	3	278.45	90.9	0.5	4602.57	50.5	50.5	50.5	50.5

Control Language

The controller language sets up and executes a simulation.

Punctuation is not part of commands unless it is in blue. Vertical bar indicates logical *or*. Asterisk indicates zero or more instances of the preceding term or parenthetical group; plus indicates one or more. Square brackets indicate an optional group.

Whitespace, except in literals, does not matter. All text except identifiers is case insensitive. C++ comments are supported. Statements may wrap over multiple lines.

Italicized fields are defined as follows:

Field	Format
Altitude	*Number*
CoordinatesWorld	*Latitude* **:** *Longitude*
CoordinatesWorld3D	*Latitude* **:** *Longitude* **:** *Altitude*
Deflection	*Number* [**/** *Number*]
id	any ordinary Java identifier with letters, numbers, and underscores
Integer	any ordinary integer, with optional **-** or **+**
Latitude	*Integer* **#** *Integer* **'** *Number* **"**
Longitude	*Integer* **#** *Integer* **'** *Number* **"**
Number	*Integer* **|** *Real*
Real	any ordinary real value, with optional **-** or **+**
String	any ordinary string delimited by single quotes; no escape characters

The commands are organized here according to their design roles, not their user roles. Although they map reasonably well to the creational, structural, and behavioral categories from the Design Patterns course, the alignment is not exact because some commands play more than one role.

Creational Commands

Creational commands are concerned with creating entities in the simulation environment.

1. CREATE AIRPLANE *id* AT *CoordinatesWorld3D* COURSE $Number_1$ POWER $Number_2$ GEAR (UP | DOWN)

Creates airplane *id* at coordinates *CoordinatesWorld3D* with course $Number_1$ at combined power setting $Number_2$ and the gear state.

2. @CREATE RADAR *id* AT *CoordinatesWorld* SIZE *Latitude*

Creates radar display *id* centered at *CoordinatesWorld* and covering the square area *Latitude*.

Structural Commands

Structural commands are concerned with connecting entities that were created.

3. `@LOAD` *`String`*

Loads controller file *`String`* as defined in Part III. Use these strategically to reduce redundant setup.

4. `@IMPORT WORLD` *`String`*

Imports world definition file *`String`* as defined in Part II.

5. `@IMPORT TERRAIN` *`String`*

Imports terrain definition file *`String`* as defined in Part I.a.

6. `@IMPORT COMMUNICATION` *`String`* `BASE` *`Integer`*$_1$ `SIZE` *`Integer`*$_2$

Imports communication definition file *`String`* with base *`Integer`*$_1$ and size *`Integer`*$_2$ as defined in Part I.b.

7. `@COMMIT`

Commits the initial configuration of the environment. Call this command after all `@IMPORT` and `CREATE AIRPLANE` commands, but before any `@CREATE RADAR` commands.

Behavioral Commands

Behavioral commands are concerned with manipulating entities that were created and connected.

8. `COMMAND` *`id`* `AILERON` *`Deflection`*

Deflects the right aileron of airplane *`id`* up (positive) or down (negative) according to *`Deflection`* for a right or left turn, respectively. (The opposite happens to the left aileron.) If the slash argument is present, the deflection specifies the expected and actual angles, respectively; if not, both are the same.

9. `COMMAND` *`id`* `ELEVATOR` *`Deflection`*

Deflects the elevator of airplane *`id`* up (positive) or down (negative) according to *`Deflection`*. If the slash argument is present, the deflection specifies the expected and actual angles, respectively; if not, both are the same.

10. `COMMAND` *`id`* `RUDDER` *`Deflection`*

Deflects the rudder of airplane *`id`* right (positive) or left (negative) according to *`Deflection`*. If the slash argument is present, the deflection specifies the expected and actual angles, respectively; if not, both are the same. Do not turn the airplane with the rudder — use the ailerons!

11. `COMMAND` *`id`* `GEAR ( UP | DOWN )`

Raises or lowers the landing gear of airplane *`id`*. The airplane cannot stall with the gear down.

12. `COMMAND` *`id`* `FLAPS (` *`Integer`* `| UP ) [ /` *`Number`* `]`

Extends the flaps of airplane *`id`* downward to position *`Integer`* $\in\{1,2,3,4\}$ or completely retracted up. If *`Number`* is present, the extension specifies the expected position and actual deflection angle, respectively; if not, both are the same position.

13. `COMMAND` *`id`* `POWER` *`Number`*$_1$ `[ /` *`Number`*$_2$ `]`

Commands the engines of airplane *`id`* to *`Number`*$_1$ percent power. If *`Number`*$_2$ is present, the power specifies the left and right engines, respectively; if not, both are the same.

14. `TRANSMIT` *`String`*

Transmits statement *`String`*, which appears in the output window of the command-line interface.

15. `@FORCE` *`id`* `[ POSITION` *`CoordinatesWorld`* `] [ ALTITUDE` *`Altitude`* `] [ COURSE` $Number_1$ `] [ POWER` $Number_2$ `]`

Forces airplane *`id`* to position *`CoordinatesWorld`* and/or altitude *`Altitude`* and/or course $Number_1$ and/or power $Number_2$. At least one field must be present.

16. `@CONFIG CLOCK` $Number_1$ $Number_2$

Reconfigures the system clock to update the simulation time by $Number_2$ seconds every $Number_1$ wall-clock seconds. The default at startup is 0.1 for both. Do not mess with this for real tests!

17. `@CONFIG RADAR` id_1 `(` id_n `=` $String_n$ `)+`

Reconfigures radar display id_1 with value $String_n$ for parameter id_n as follows:

Parameter	Values	Description
`backcolor`	hexadecimal RGB color	Sets the background color for better contrast in screenshots for the report
`is_persistent`	boolean	Disables the decay effect of aircraft to accommodate screenshots of the entire display
`has_grid`	boolean	Enables the reference grid with a spacing of one nautical mile
`has_range`	boolean	Enables the range grid at increments of one nautical mile
`has_terrain`	boolean	Enables terrain rendering
`has_breadcrumbs`	boolean	Enables breadcrumb rendering of aircraft movement
`breadcrumb_rate`	integer	Sets the sampling rate for breadcrumb recording

18. `@CONFIG DISPLAY (` id_n `=` $String_n$ `)+`

Reconfigures the display elements with value $String_n$ for parameter id_n as follows:

Parameter	Values	Description
`update_instrument_panels`	boolean	Enables updating instrument panels
`update_3d_views`	boolean	Enables updating airplane three-dimensional state viewers
`update_radar_displays`	boolean	Enables updating radar displays

19. `@WAIT` *`Number`*

Waits *`Number`* simulation seconds before executing the next command.

20. `@SCHEDULE` *`Number`* `<` *`command`* `>`

Schedules command *`command`* to execute at time *`Number`* simulation seconds. If the time is in the past, the command executes immediately.

Notice how this command supports the communication loader in Part I.b by scheduling `TRANSMIT` commands behind the scenes for the specified phrases at the specified times.

21. `@PAUSE`

Pauses the simulation clock until any command is issued. See `@RESUME`.

22. `@RESUME`

Resumes the simulation clock after a pause without causing any other action.

23. `@EXIT`

Exits the simulation gracefully. Closing any of the windows with the mouse may not shut down all services properly and flush the output streams.

Report Requirements

This final part of the project addresses selective testing and evaluation of the provided solution. It focuses predominantly on breadth of coverage, not depth. The goal is to demonstrate that each unit of functionality reasonably works for at least one representative scenario. A real test plan for a project of this relatively small size could easily expand to hundreds or even thousands of times the size of this section.

Deliverable

You need to produce one document with all your tests. Tests are stated in the form of requirements. Unless otherwise specified, you may satisfy each however you want. Each must address the following in exactly this form, including the number, in a separate section:

The test designator and title in bold; e.g., **Test A.1: Constant speed straight and level**

1. The rationale behind the test; i.e., what is it testing and why we care.
2. A general English description of the initial conditions of the test.
3. The commands for (2), which must appear in a standalone form that could be directly copied into a text file to reproduce the test without manual intervention. Do not crossreference other tests.
4. A brief English narrative of the expected results of executing the test. (Proper testing discipline demands that you do this *before* running the test.)
5. At least one graphical representation of the actual results. The form is your choice.
6. A brief discussion on how the actual results differ from the expected results.
7. A suggestion for how to extend this test to cover related aspects not required here.

Your document must be formatted professionally. It must be consistent in all respects across all team members. Code references must be in monospace font. Use any resources available, such as screenshots and log excerpts, to make a reasonably convincing claim. Provide nothing less, but also nothing more.

Tests

Each test is independent. Import terrain `sample.trn` for group G only. Except for loss of control and wild maneuvers, use realistic pitch angles under 10 degrees and bank angles under 20 degrees. There is no guarantee that all tests are possible because we are evaluating an unknown system. If in doubt, ask for clarification.

Teams of three should do 12 tests from these options. Teams of two do eight in some reasonable fashion of your choice.

A. Enroute Operations

Do three from this group.

Import world `sample.wld`. Assume all tests start at 5,000 feet. Tests A.7 through A.12 repeat A.1 though A.6, respectively, but start with power 30 and increase to power 100.

Test A.1: Constant speed straight and level

Fly straight and level without changing speed.

Test A.2: Constant speed straight and climbing

Fly straight without changing speed, but climb at 1,000 feet per minute.

Test A.3: Constant speed straight and descending

Fly straight without changing speed, but descend at 1,000 feet per minute.

Test A.4: Constant speed turning and level

Fly a clockwise circle without changing altitude.

Test A.5: Constant speed turning and climbing

Fly a clockwise circle while climbing at 2,000 feet per minute.

Test A.6: Constant speed turning and descending

Fly a clockwise circle while descending at 2,000 feet per minute.

Test A.7: Increasing speed straight and level
Test A.8: Increasing speed straight and climbing
Test A.9: Increasing speed straight and descending
Test A.10: Increasing speed turning and level
Test A.11: Increasing speed turning and climbing
Test A.12: Increasing speed turning and descending

B. Takeoff Operations

Do one from this group.

Test B.1: Takeoff

Start at the south end of the runway at 800 feet, increase to 50% power, wait for two seconds, increase to 100%, lift off, raise the landing gear, turn east at 3,500 feet, and reduce to 80% power.

C. Landing Operations

Do two from this group.

Test C.1: Landing straight in without flaps

Start south of the airport at 6,000 feet heading north at 100 knots. Touch down on the runway and come to a stop. Assume the runway elevation is 800 feet.

Test C.2: Landing straight in with flaps

Do C.1, but lower the flaps one increment at a time and stabilize during the descent to touchdown.

Test C.3: Landing approach

Start west of the airport at 5,000 feet heading east at 120 knots. Cross over the center of the runway, turn right, and land as in C.1.

D. Aircraft Failure

Do one from this group.

Test D.1: Aileron failure

Fly straight and level, then command a turn to the right that actually turns to the left.

Test D.2: Engine Failure

Fly straight and level at full power in both engines, then cut the left engine to zero power.

E. Pilot Failure (Loss of Control)

Do one from this group.

Test E.1: Stall

Fly straight and level at full power, reduce power to just above stall, hold it, then stall.

Test E.2: Excessive bank

Fly straight and level, then bank excessively to the right.

F. Midair Collisions

Do both from this group.

Test F.1: Collision

Create two airplanes at opposing positions heading toward each other until they collide.

Test F.2: Collision averted (playing chicken)

Create two airplanes at opposing positions heading toward each other until they perform crazy maneuvers at the last moment to avert the collision.

G. Terrain Collisions

Do two from this group. Import terrain `sample.trn`.

Test G.1: Collision

Fly straight and level into the mountain.

Test G.2: Collision averted (playing chicken)

Fly straight and level toward the mountain, but perform a crazy maneuver at the last moment to avert the collision.

Test G.3: Nap of the earth

Fly as closely as possible to the terrain through a variety of turns.

H. Flight Dynamics

Do neither or both from this group.

Test H.1: Discrete turn

From straight and level, turn right with a bank of 20 degrees commanded instantaneously.

Test H.2: Continuous turn

From straight and level, turn right with a bank of 20 degrees commanded stepwise by five degrees per simulation second. Compare the turning performance with H.1.

Appendix D: Most Relevant Publications

These papers represent various perspectives related to the overall plan in this proposal:

- "A Meta-Case Study of Modeling, Simulation, Visualization, and Analysis for Real-World Software Systems Engineering Education" compares eight projects in a way similar to the proposed process of tearing them apart and rebuilding them. [page 63]
- "Experiencing Real-World Multidisciplinary Software Systems Engineering Through Aircraft Carrier Simulation" describes one project in extensive detail. [page 77]
- "Multiagent Test Range: Fostering Disciplined Software Engineering Practices in Students via Modeling, Simulation, Visualization, and Analysis" shows the QMSVA perspective. [page 107]
- "A Quasi-Network-Based Fly-by-Wire Simulation Architecture for Teaching Software Engineering" describes a project with low-level engineering details. [page 116]
- "A Holistic Multidisciplinary Approach to Teaching Software Engineering Through Air Traffic Control" describes a project with high-level managerial details. [page 124]
- "Toward Introspective Human Versus Machine Learning of Simulated Airplane Flight Dynamics" uses student subjects to evaluate a machine learning approach. [page 131]

Peer Reviewed Paper Presented at AlaSim 2016

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

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

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

1. Introduction

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

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

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

2. Software Systems Engineering

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

fosters an understanding and appreciation of such a holistic perspective.

2.1 Software Engineering

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

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

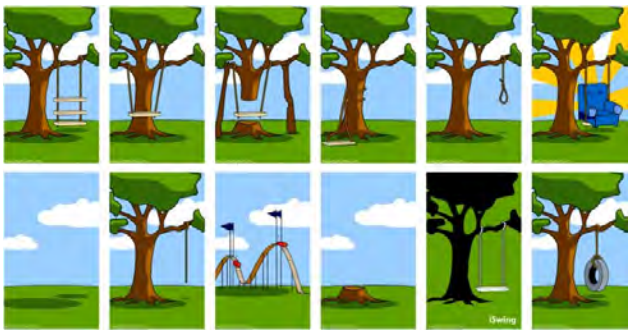


Figure 2.1: Software Engineering in Practice

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

2.2 Systems Engineering

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

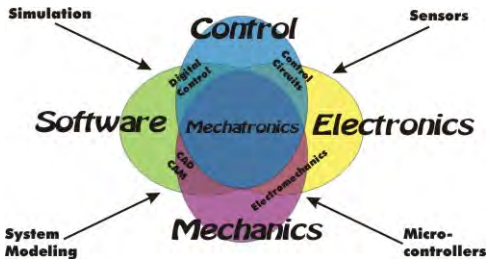


Figure 2.2: Systems Engineering Convergence [1]

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

3. Pedagogical Foundation

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

3.1 Modeling

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

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

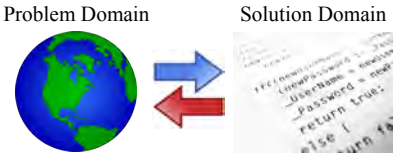


Figure 3.1: Real to Virtual-World Correspondence

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

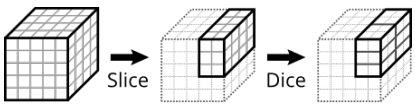


Figure 3.2: Domain Decomposition [6]

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

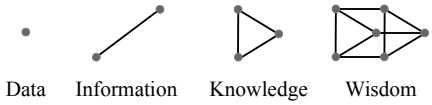


Figure 3.3: DIKW Hierarchy

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

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

3.2 Simulation

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

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

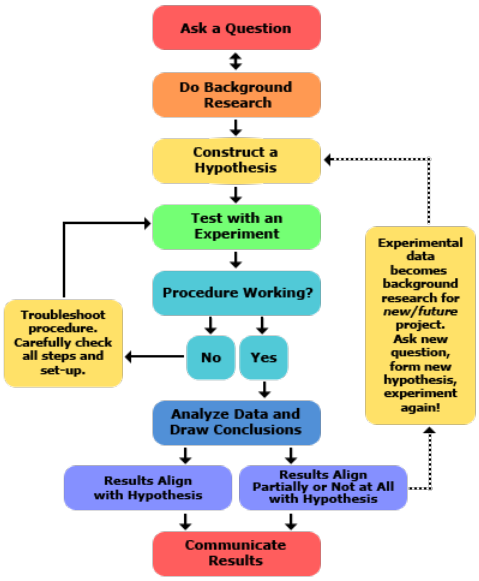


Figure 3.4: Scientific Method [11]

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

4. Project Showcase

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

4.1 Unmanned Aerial Vehicle

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

Figure 4.1: UAV Viewer

4.2 Air Traffic Control

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

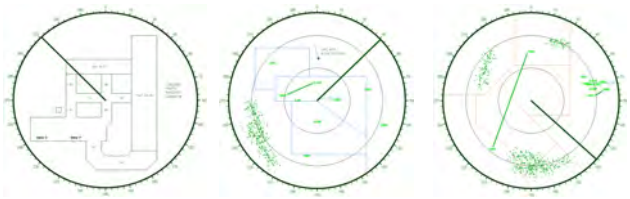


Figure 4.2: ATC Controller Viewers

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

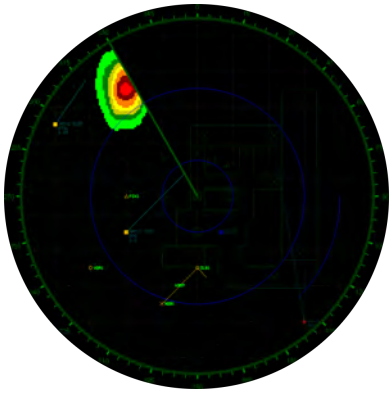


Figure 4.3: ATC Viewer, Composite

4.3 Fly-by-Wire Aircraft Control

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

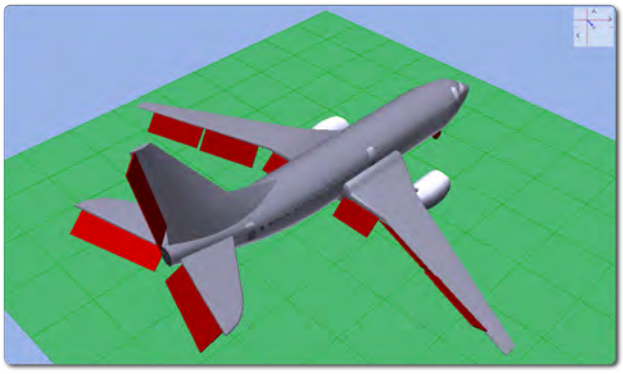


Figure 4.4: FBW Viewer

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

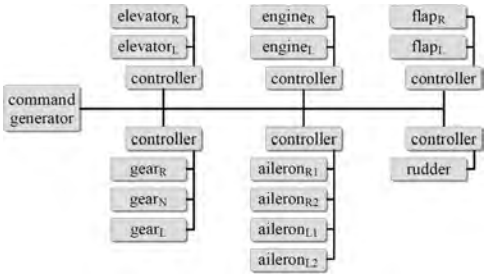


Figure 4.5: Fly-By-Wire Network Architecture

4.4 Aircraft Accident Reenactment

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

Figure 4.6: AAR Viewer Mockup [16]

4.5 Aircraft Carrier Operations

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

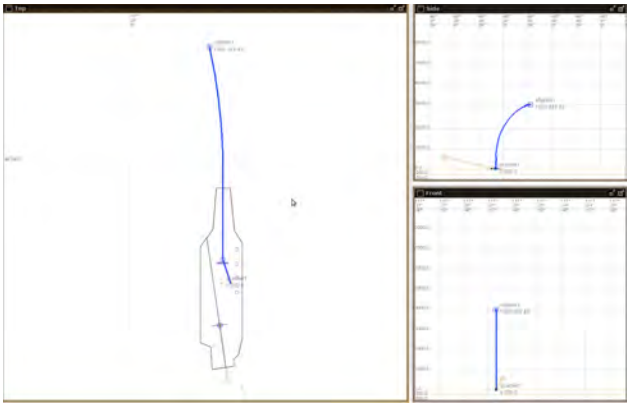


Figure 4.7: ACO Viewer

4.6 Military Test Range

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

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

Table 4.1: Compatibility Matrix

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

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

Table 4.2: Applicability Matrix

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

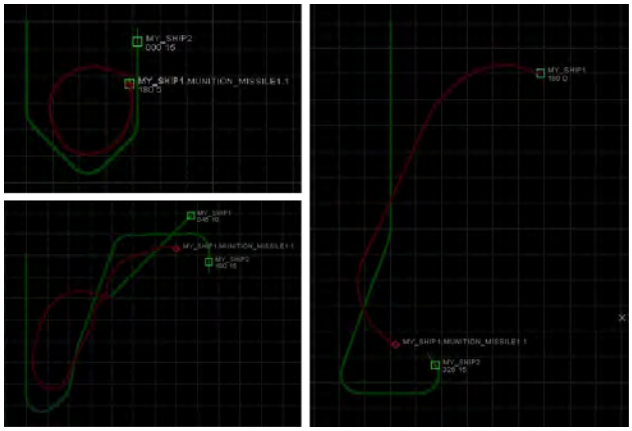


Figure 4.8: MTR Viewer, 2D Perspective

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

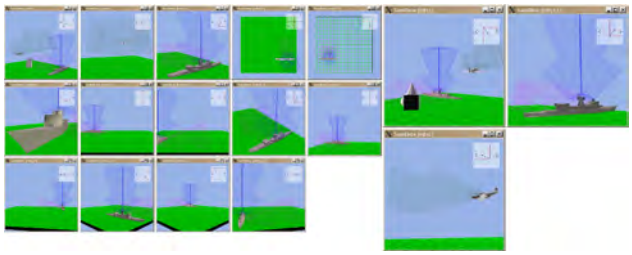


Figure 4.9: MTR Viewer, 3D Perspective

4.7 Heavy Construction Equipment

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

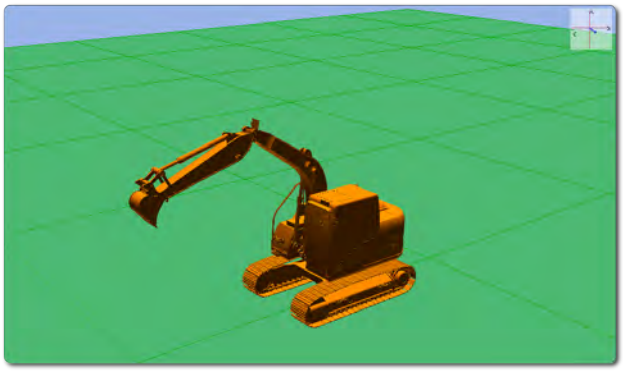


Figure 4.10: HCE Viewer

4.8 Railroad Layout Manager

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

Figure 4.11: RLM Viewer

5. Architectural Framework

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

5.1 Modeling

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

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

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

Table 5.1: Components

5.1.1 Data

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

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

Table 5.2: Data

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

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

Figure 5.1: Datatypes

5.1.2 Control

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

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

Table 5.3: Control

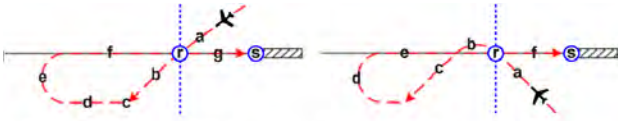
5.1.3 Behavior

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

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

Table 5.4: Behavior

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



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

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

Figure 5.2: Landing Approaches

5.1.4 Decompositional Characteristics

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

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

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

Table 5.5: Independent/Dependent Components

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

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

Table 5.6: Static/Dynamic Components

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

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

Table 5.7: Discrete/Continuous Components

5.2 Simulation

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

5.2.1 Simulation Framework

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

5.2.2 Domain-Specific Languages

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

```
/* load "/home/author/workspace/MultiagentTestbed/tests/test3.mat" */

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

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

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

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

set MY_SHIP1 load munition MUNITION_DEPTHCHARGE1
set MY_SUBMARINE1 load munition MUNITION_TORPEDO1

set MY_SHIP1 deploy munition MY_SHIP1.MUNITION_DEPTHCHARGE1 4
```

Figure 5.3: Script Snippet

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

```
CREATE something WITH arguments
```

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

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

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

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

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

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

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

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

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

5.2.3 Simulation Implementation

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

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

5.3 Visualization

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

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

Figure 5.4: XML Representation

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

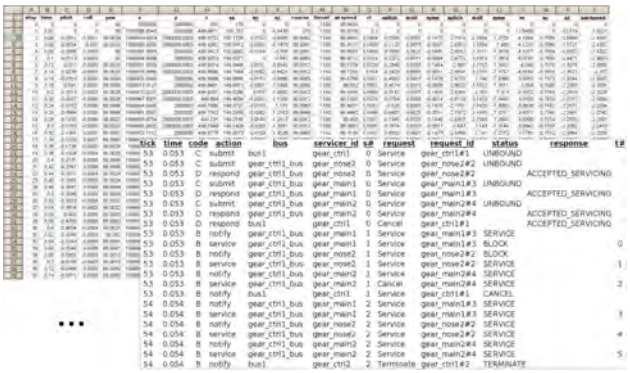


Figure 5.5: Excel Table Representations

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

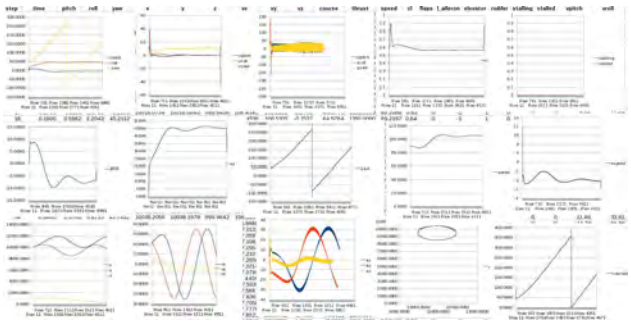


Figure 5.6: Excel Graph Representations

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

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

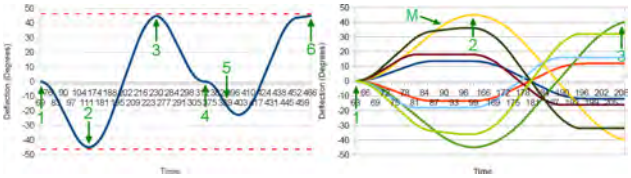


Figure 5.7: Annotated Events

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

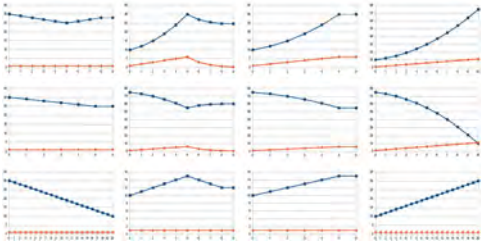


Figure 5.8: Excel Graph Representations

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

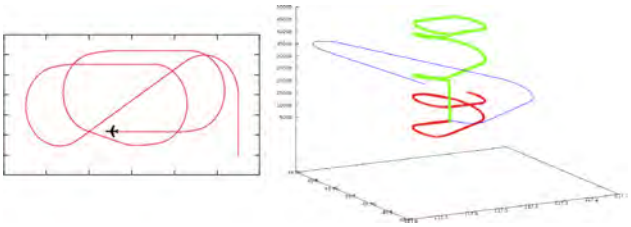


Figure 5.9: Gnuplot 2D Representations

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

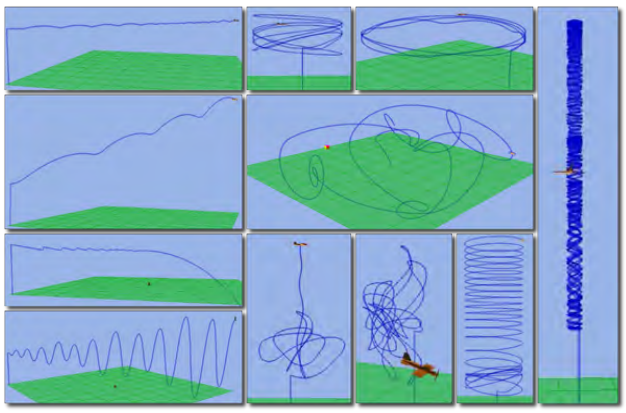


Figure 5.10: 3D Visualizer

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

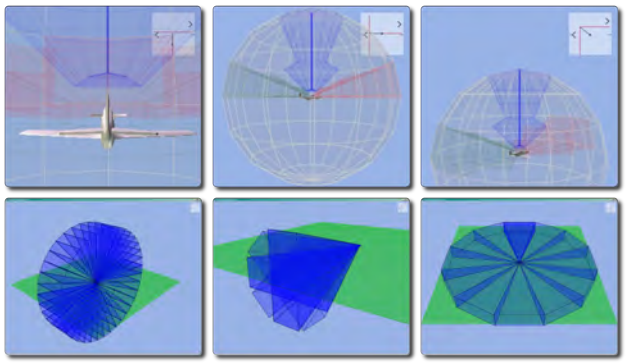


Figure 5.11: 3D Visualizer Augmentation

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

Figure 5.12: Google Earth Visualization

5.4 Analysis

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

To this end, students have to produce a professional-looking test report based on a cross-section of roughly 40 experiments that demonstrate representative aspects of the system. For consistency, since not every team's own solution was correct or functioned identically, they used the author's. Each experiment addressed eight requirements, where 1–4 relate to planning, 5–6 to execution, and 7–8 to presenting the results:

1. The rationale behind the test; i.e., what it was testing and why it mattered.
2. A general English description of the initial conditions.
3. The commands for (2).
4. An English narrative of the expected results.
5. The actual results with at least one graph showing the most representative view of the states.
6. A snippet of the actual results from the log file with a supporting explanation, including statistics, metrics, and graphs, as appropriate.
7. A discussion on how well the actual results agreed with the expected results, or if they disagreed, a hypothesis on why.
8. A suggestion for how to extend this test to address related aspects of potential interest.

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

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

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

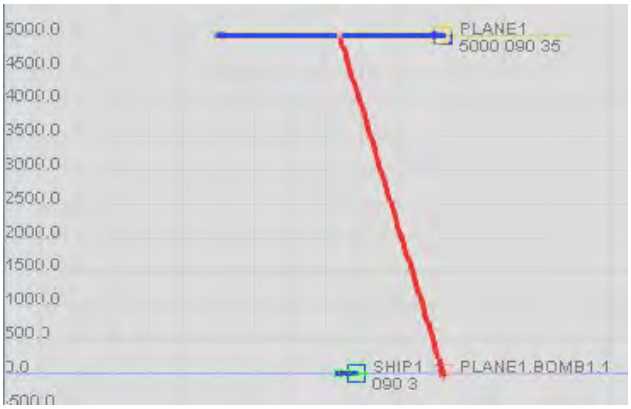


Figure 5.13: Bomb Release, Side View

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

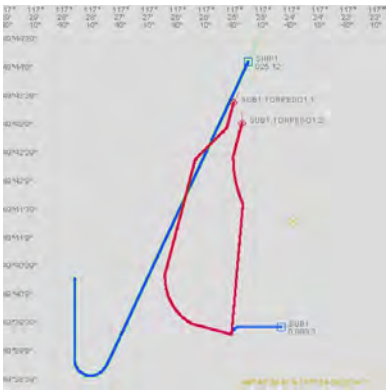


Figure 5.14: Torpedo Engagement, Top View

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

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

6. Results

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

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

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

7. Future Work

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

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

8. Conclusion

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Author Biography

DAN TAPPAN is an Associate Professor of Computer Science at Eastern Washington University. He has been a professor of computer science and engineering for 11 years, before which he spent a decade as a defense contractor, mostly involved in the modeling and simulation of weapon systems at White Sands Missile Range and Aberdeen Proving Ground. His main research areas are software and hardware systems engineering, especially for aviation and military applications with embedded systems and mechatronics; modeling, simulation, visualization, and analysis; intelligent systems/artificial intelligence (knowledge representation, reasoning, machine learning); and computer science and engineering education.

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.

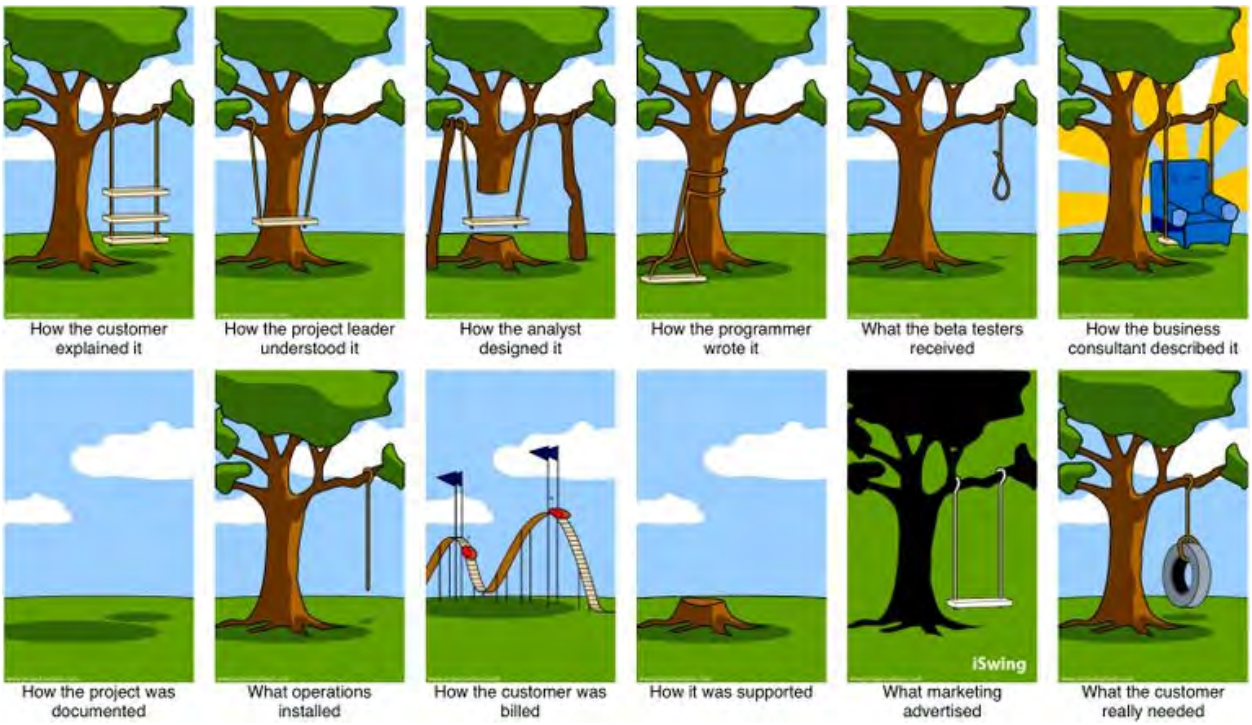
This flexible Java-based environment allows students both to analyze existing solutions (which would be impractical to build themselves) and to synthesize their own. At all stages, it fosters critical thinking because the subject matter, pedagogical approach, and environment force students outside of 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

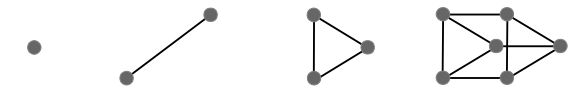


Figure 2: DIKW Hierarchy

Background research and low-level W5H 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.

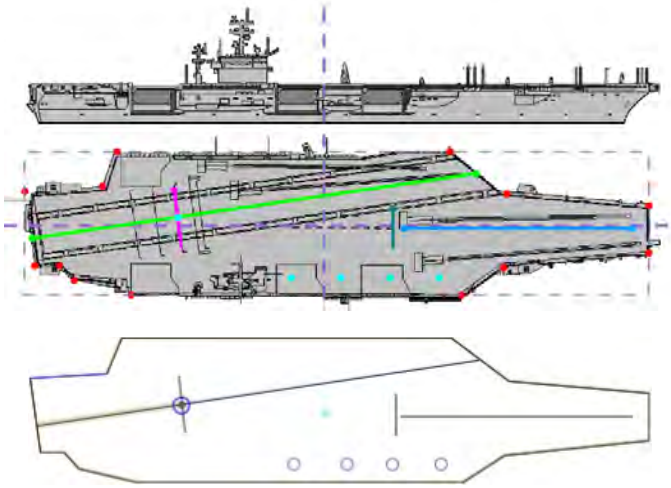


Figure 4: Carrier Definition[11]

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 aid1 WITH FIGHTER[S] aidn+
```

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

```
POPULATE WORLD WITH aidn+
```

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.

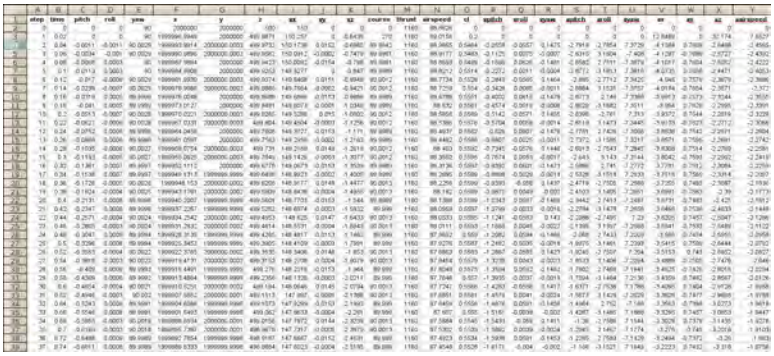


Figure 6: Excel State Log View

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.

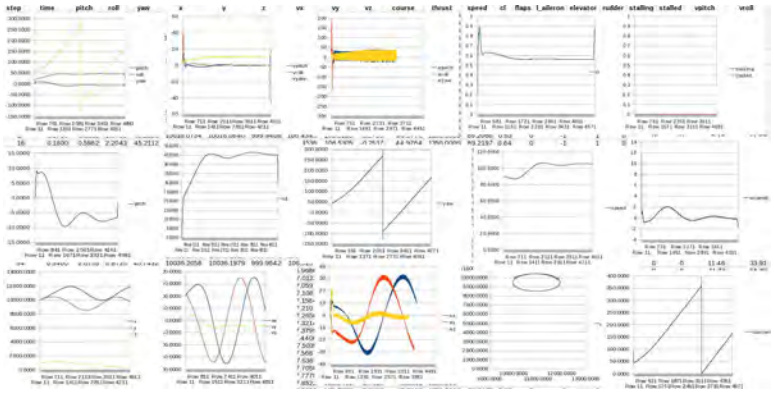


Figure 7: Excel Graph View

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.

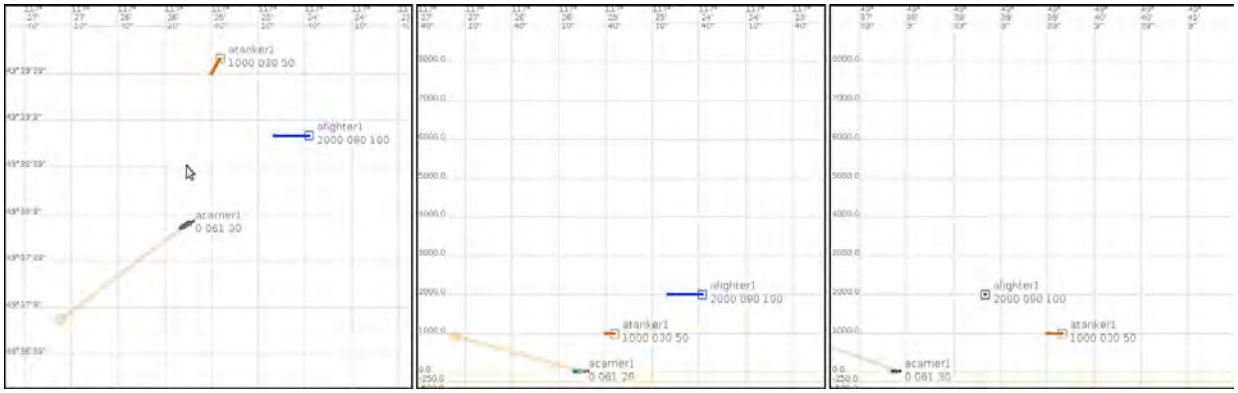


Figure 8: Two-Dimensional Views

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.

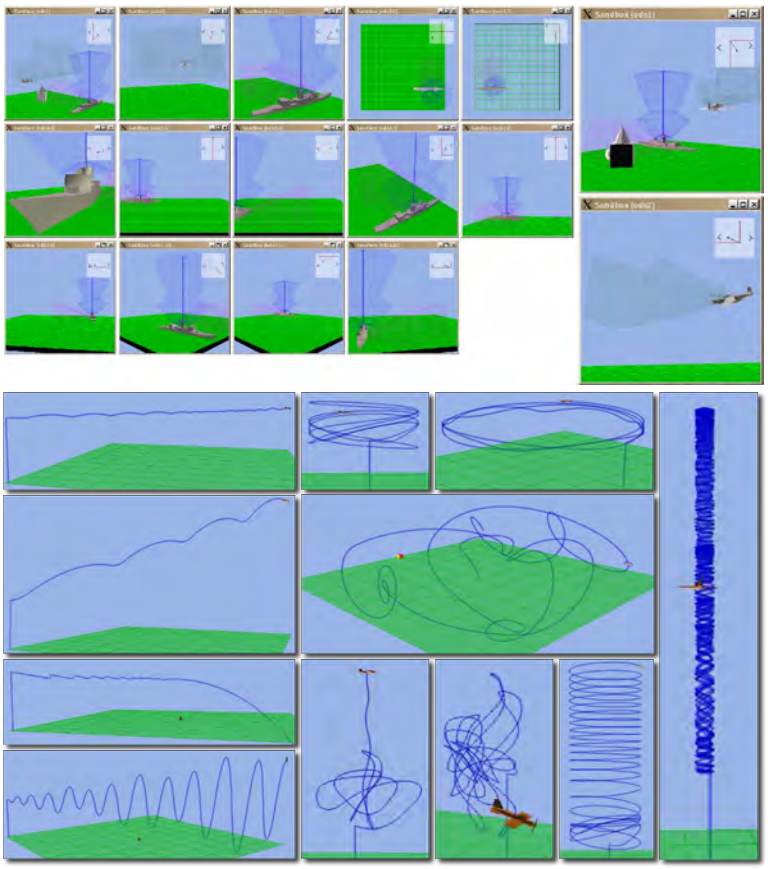


Figure 9: Three-Dimensional Views

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
```

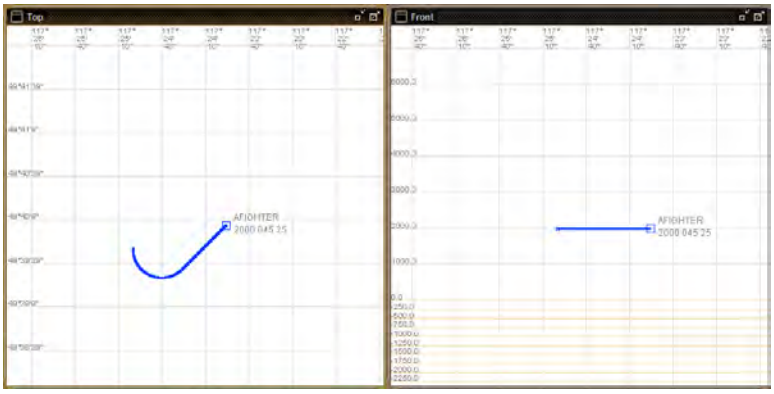
Figure 10: Test script

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.

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

Dan Tappan
Department of Computer Science, Eastern Washington University, Cheney, WA, USA
dtappan@ewu.edu
Keywords: software engineering, experiment-based testing and evaluation, pedagogy

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

1 Introduction

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

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

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

2 Pedagogical Foundation

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

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

2.1 Critical Thinking

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

Problem-solving for any domain has been studied in endless detail. The approach of multidisciplinary critical systems thinking has long been effective in traditional engineering, and has more recently become an emphasis in software engineering [4,5]. This work considers a multidimensional approach of forcing students to decompose the pieces of a problem into what they are (data), what they can do (control), and what they actually do or have done to them (behavior) by critical analysis with the W5H question words of *who*, *what*, *when*, *where*, *why*, and *how*. This low-level analysis then combines to form associative structures that connect the dots in a DIKW network, as in Figure 1.1 [6]:

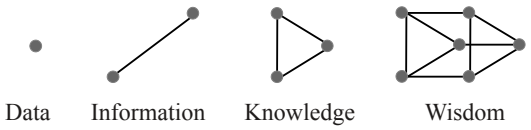


Figure 1.1: DIKW Learning Associativity

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

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

2.2 Software as Surrogate

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

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

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

2.3.1 Code Considerations

The low-level goal of testing that students' code works as specified is the realm of traditional object-oriented programming and software engineering. This architecture provides two complementary opportunities: analysis for investigating how existing components function and interact, and synthesis for adding new ones. The emphasis in both cases is on clean, orthogonal solutions with well-designed, inherently defensive structure. In other words, with minimal effort, the architecture permits acceptable actions and prohibits unacceptable ones. This goal is critical in such a large, highly compositional, dynamic, plug-and-play system like this one, with its 320 classes. If

done well, students do not have to expend time coding and testing for illegal combinations if they can show that these cases cannot happen. Unfortunately, most students work quite differently and love to hardcode uniquely for every little perceived special case, which leads to code that is bloated, brittle, and difficult to maintain. For example, one student practically bragged that in his development process, he "kept throwing more code at the compiler until it shut up." Section 4 discusses how the pedagogical approach here helps instill better discipline.

2.3.2 Simulation Considerations

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

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

3 Architecture

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

3.1 Model

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

3.1.1 Agents

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

3.1.1.1 Actors

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

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

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

3.1.1.2 Munitions

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

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

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

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

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

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

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

3.1.1.3 Sensors and Fuzes

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

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

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

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

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

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

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

3.1.2 Datatypes

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

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

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

3.2 View

The view module of the architecture manages how the user sees the output. While not considered agents in the traditional simulation sense, view windows are actually treated as such because they can be created, manipulated,

and deleted dynamically through six commands. They play an integral role in testing and evaluation, so they are part of the world. For simplicity, it is a flat-earth model. The plug-and-play nature of the view allows any visualizer to connect to the architecture, provided that it follows the specified protocols.

3.2.1 Two-Dimensional Visualization

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

Figure 3.1: Two-Dimensional Top Perspective

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

3.2.2 Three-Dimensional Visualization

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

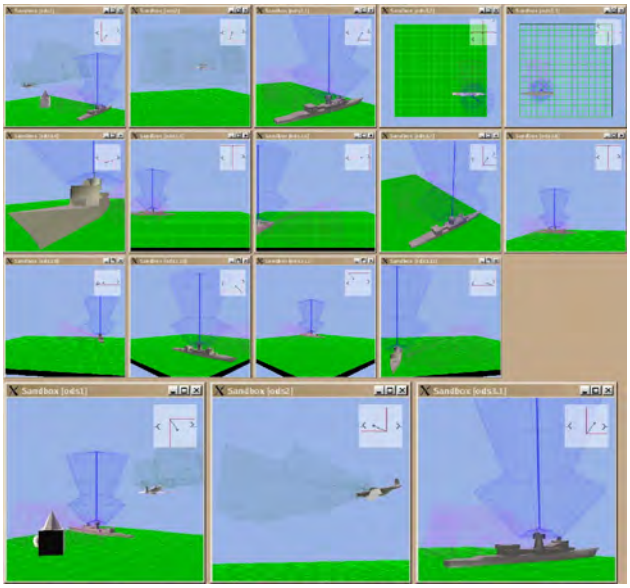


Figure 3.2: Three-Dimensional Perspectives

3.2.3 Logging

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

3.3 Controller

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

3.3.1 Input

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

3.3.1.1 Creational and Structural Commands

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

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

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

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

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

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

Finally, the actors are defined with munitions:

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

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

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

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

```
load munition id1 from id2
```

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

The second action deploys the munition accordingly:

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

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

3.3.1.2 Behavioral Commands

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

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

3.3.1.3 Miscellaneous Commands

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

```
execute maneuver climb_left on my_fighter
```

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

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

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

4 Preliminary Results

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

Although the project was already complete in advance, the students had to go through the analysis and design stages without access to it, as if they themselves were in charge of its outcome. Background research got everyone up to speed on the subject matter. It also provided valuable insight into their generally limited critical-thinking skills. For example, the entire class misinterpreted which horizontal direction on a map increasing longitude corresponds to in North America, despite several obvious opportunities to crosscheck their understanding. Many students expressed that it was a shocking reality check demonstrating the importance of

connecting the dots and actually understanding what the connections mean. Such cases are so common that a related exercise with potential "gotchas" has become part of every project.

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

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

Table 4.1: Compatibility Matrix

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

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

Table 4.2: Applicability Matrix

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

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

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

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

4.1 Test Report

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

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

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

4.2 Examples

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

Figure 4.1: Bomb Release

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

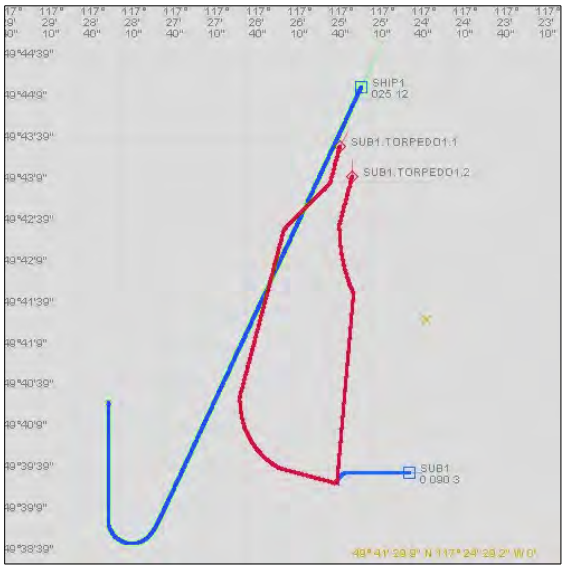


Figure 4.2: Torpedo Tracking

5 Future Work

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

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

6 Conclusion

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Author Biography

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

A Quasi-Network-Based Fly-by-Wire Simulation Architecture for Teaching Software Engineering

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

***Abstract*—Undisciplined cohesion and coupling undermine countless aspects of quality software. Students, however, unfortunately tend to gravitate toward such approaches. This system mitigates this problem by forcing them to communicate with their components through a well-constrained hierarchical virtual network of networks. The application is a dynamic, plug-and-play aircraft fly-by-wire system that processes a wide variety of commands to design, construct, and manipulate sensors, actuators, controllers, and communication buses concurrently in a flexible model-view-controller architecture. It successfully employs many systems-engineering concepts of modeling, simulation, visualization, and analysis toward the goal of instilling disciplined design, implementation, testing, and evaluation practices in students. In particular, it provides the pedagogical and programmatic frameworks for creating, executing, presenting, and analyzing meaningful test cases as part of formal test plans carried out in controlled experiments via scientific method. The system can be adapted relatively easily to countless other real-world multidisciplinary domains for reuse in other projects. Extensive results from classroom deployments show that students overwhelmingly benefit from this approach.**

Keywords—software engineering; control system; simulation

I. Introduction

Software, by its flexible virtual nature, allows programmers to make quick and easy changes, at least compared to related physical disciplines like engineering. Unfortunately, without appropriate self-discipline, it is all too easy to produce messy spaghetti code with endless nasty interconnections. The equivalent rat's nest of wires in an electrical system would be immediately apparent to anyone, but in a software system, no such obvious red flags exist without first learning to recognize them. The goal of this system is to instill disciplined design, implementation, and testing practices in software-engineering students by forcing them to work within a Java model-view-controller architecture that behaves like a well-defined and protected physical plug-and-play network.

The overarching philosophy is to use a systems-engineering approach of modeling, simulation, visualization, and analysis respectively to build a solution, execute it in a controlled way, present the results visually, and analyze what they mean as part of testing and evaluation. The model, in particular, is a simplified aircraft fly-by-wire system that controls a variety of flight components in complex real-time ways. This context, supported by background research, provided students in an undergraduate software-engineering course with a holistic understanding of the problem space such that they could create a corresponding clean implementation in the solution space. The objectives are typical of any software system: to separate the concerns, maximize cohesion, minimize coupling, and delegate responsibility appropriately. These considerations are further complicated by the need for uniform, safe, and repeatable concurrency among the components. Building and testing single-threaded code is difficult enough for most students; multithreaded (or the appearance of such here) absolutely requires a disciplined approach.

II. Background

All control systems take inputs of some sort and produce corresponding outputs of some sort. In traditional (non-electronic) systems, the connections from the former to the latter are mechanical linkages like cables, pushrods, and shafts. For example, a car steering wheel directly drives the gearbox to deflect the front wheels. There is little possible dynamic variation in the operation, like changing the steering sensitivity based on vehicle speed, because the configuration of the static system is fixed. A "by-wire" system, on the other hand, translates the input from a sensor into an electronic signal that travels via a network to an actuator, which acts upon it as output. In this form, any amount of computer processing is now possible for dynamic real-time reconfiguration.

In a fly-by-wire system, the primary control sensors are located in the cockpit in the form of a stick or yoke, as well as pedals, switches, and levers. The actuators are located around the airplane. Fig. 1 shows a basic configuration, which also includes the engines and landing gear.

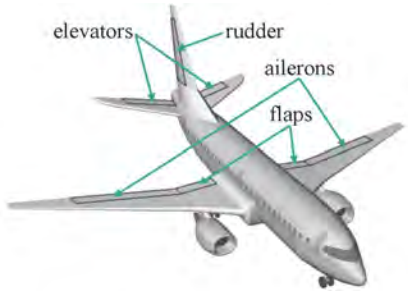


Fig. 1. Basic airplane actuators [1]

This work focuses on the behavior of the control system only, not on its effect on the airplane in flight. In other words, the airplane is basically stationary on a test stand. As a result, its degrees of freedom of motion and aerodynamics are ignored. For background, however, the elevators affect pitch (nose up or down) to change altitude; the ailerons affect wing roll to turn; and the rudder affects yaw to coordinate turns, much like the front wheel of bicycle does.

III. Model

This work uses a model-view-controller architecture. The model is the module that defines the machine being manipulated. The plug-and-play nature of the architecture allows it to accommodate other by-wire systems with relatively little difficulty. Further supporting this goal is the extensive use of well-established, reusable software design patterns [2]. For example, a subsequent offering of the same course used it as the basis of a toolkit for building and controlling heavy construction equipment. Follow-on work is planned for modeling railroads and railway equipment.

A. Datatypes

Significant, consistent anecdotal classroom evidence shows that students have a major problem with abstracting, maintaining, and manipulating data properly. Java primitives are appropriate in earlier low-level courses, but at higher project-based levels like software engineering, they lead to a proliferation of problems. For example, units, magnitudes, and limits are not applied consistently, error handling is inconsistent or almost nonexistent, and code bloats from haphazard attempts at reimplementing similar solutions in multiple places.

To mitigate this situation, the architecture provides a rich set of self-contained concrete datatypes for every kind of relevant data; e.g., Acceleration, AngleHorizontal, AngleVertical, FlapPosition, Identifier, Percent, Power, Rate, Speed, and many dozens more not directly in play in this paper. Each maintains its own error checking and helper methods for manipulating and converting it appropriately. This approach lends itself to convenient unit testing in isolation. It also reduces the burden of documentation; e.g., avoiding having to state and enforce everywhere that horizontal angles are in mathematical degrees (as opposed to navigational degrees or radians) because AngleHorizontal always uses this form.

Another consistent problem students have is with indiscriminate coupling and undisciplined, unprotected sharing of objects. Changing a mutable object in one place may have countless unexpected consequences throughout an entire system. To mitigate this problem, datatypes employ a functional paradigm, which makes them immutable. Any mutable action on them produces a new object via copy-on-write semantics [3]. It is therefore exceedingly difficult for students to interact with the system outside the prescribed network infrastructure, intentionally or not.

Finally, datatypes extensively use Java generics to constrain their application to appropriate contexts. Students are prone to hard-coding dangerous runtime casts and making decisions based on querying objects for their type with the `instanceof` operator instead of properly utilizing the object-oriented principles of inheritance and polymorphism. Explicit casting should be avoided as much as possible in a dynamic, plug-and-play system.

B. Intervals

An interval is the data-structure equivalent of the presumed motive force that moves an actuator from one state to another. The basis is kinematics, or geometry in motion without regard to its causes [4]. Actuators play different roles and therefore have different state types, which the datatypes define, and intervals directly map onto and control.

An interval has implicit or explicit limits; e.g., a Percent interval always ranges inclusively from 0 to 100, whereas a Speed interval ranges inclusively from its specified minimum to maximum values. In all cases, the current state must always reside on the interval. It is impossible for a student to cause an inconsistent state without detection and notification.

Change in state as a delta is also configurable to account for slower or faster movement. In the linear variant, the delta never changes. In the nonlinear, it does so to account for acceleration or deceleration. Again, it is impossible to cause an inconsistent state. The interval always reflects a continuous function; e.g., it cannot achieve maximum delta without accelerating to that value by the rules. Likewise, it cannot change direction without decelerating to zero first. This behavior reflects the reality of the mechanical system that the interval represents.

An interval is like an operating-system process in several ways. A nonpreempting request submitted to it is queued for execution. For example, requesting it to go to maximum value and then immediately to minimum value would entail increasing to completion (with initial acceleration and final deceleration) and then similarly decreasing to completion. A preempting request, on the other hand, would cause the currently executing request to complete gracefully with deceleration to zero as soon as possible, followed by servicing the new request. A terminate request functions in the same way, except that it schedules no subsequent action. A cancel request kills the currently executing request immediately with no graceful shutdown. It is not an option for normal interaction because it violates the kinematics; i.e., infinitely fast deceleration from the equivalent of dividing by zero in

$$rate = distance / time \tag{1}$$

Fig. 2 notionally depicts the state and delta (top and bottom lines of each graph, respectively) for a variety of combinations of linear and nonlinear movement in increasing and decreasing directions with intervening terminate and cancel actions. To demonstrate the value of using the architecture-supported intervals over their own ad hoc implementations, the students had to solve the basic elements of this problem as a standalone proof-of-concept Java program. The results were telling: their solutions were overwhelmingly large, unmanageable hacks, not one worked completely, and the average grade was 11%. They said it was an extremely eye-opening experience and acknowledged the value of the orthogonal approach in this work: one solution applicable to many problems [5].

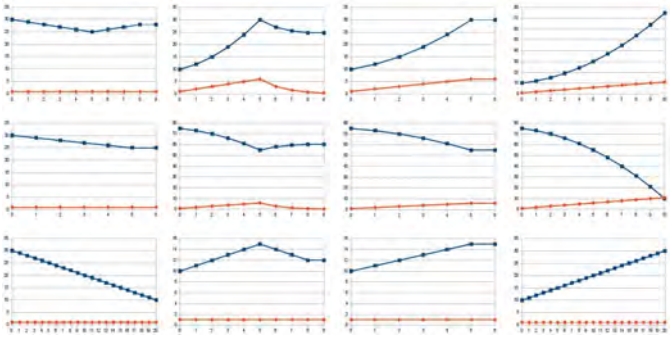


Fig. 2. Sample kinematics intervals

C. Buses

Every interval (as part of an actuator) resides on a communication bus that transfers requests. Three types of interval servicing are possible:

- *Oneshot*: service a request once, then expire automatically; e.g., instantaneous on/off actions.
- *Definite*: service a request until a specified end condition, then expire automatically; e.g., movement.
- *Indefinite*: service a request continuously, ending only upon a terminate or cancel request; e.g., engine rotation.

In addition, requests may specify in detail the timing of the interval servicing:

- *Lead time*: the amount of time that an interval should initially perform no action. This models initialization.
- *Duration*: the total amount of time that the request will be serviced. This applies to definite intervals only.
- *Frequency*: the rate at which actions are performed while being serviced. This allows them to operate at a fraction of the clock speed.

Requests are exactly that: *requests*. They are not imperative commands that must be honored (even though "command" is the conventional term used in flight control) [6]. In fact, the servicer decides how to process the request and responds to the submitter as follows:

- `IGNORED`: ignored and discarded the request because it was not considered applicable.
- `REJECTED_INVALID`: rejected the request, which normally would be serviced, but cannot be now because it is somehow invalid or inappropriate.
- `REJECTED_UNABLE`: rejected the request, which normally would be serviced, but cannot be now for some reason on the servicer's side. The servicer may inform the requester when it is available again.
- `ACCEPTED_BLOCKED`: accepted the request, but it will be queued for later servicing because the servicer is busy.
- `ACCEPTED_SERVICING`: accepted the request, and it will be serviced immediately.

This handshaking approach was difficult for the students to embrace because they are used to shouting at their code imperatively to do what they want, and if it does not, then shouting louder. For example, one student admitted that his development process was to "[keep] throwing more code at the compiler until it shut up."

D. Actuators

Actuators are the interval-based virtual mechanisms that cause the aircraft components in Fig. 1, as well as others, to change state appropriately. The presumed underlying motive force (electrical, hydraulic, pneumatic, thermodynamic, etc.) plays no role, only the resulting action. Section III.F addresses specific actuator behaviors in detail, especially in combination. In general, however, their data (what they are) and control (what they can do) adhere to the following constraints:

- *Rudder*: deflects left or right to an angle.
- *Elevator*: deflects up or down to an angle.
- *Engine*: changes speed as a percentage of maximum revolutions per minute.
- *Aileron*: deflects up or down to an angle; in roll mode, they are always paired antisymmetrically on the wings, so when one deflects, its counterpart deflects by the same amount in the opposite direction.
- *Speed brake*: deflects upward to an angle; on a real aircraft, separate dedicated actuators typically play this role, but for pedagogical reasons, ailerons do so here in speed-brake mode. There is no antisymmetry: all ailerons deflect upward to cause increased drag.
- *Main gear*: extends or retracts as a percentage of downward deployment.
- *Nose gear*: extends or retracts as a percentage of downward deployment, but also simultaneously rotates 90 degrees to stow sideways in the fuselage.
- *Flap*: deflects downward to an angle. As Fig. 3 shows, plain flaps rotate about a fixed point; for double-slotted flaps, this point moves backward, and the flap separates, requiring two coordinated intervals that are both simultaneous and sequential.

Fig. 3. Plain and double-slotted flap deployment [7]

E. Sensors

Whereas an actuator is an output device that changes state on command, a sensor is its complement as an input device to indicate this state by querying the actuator on command. Sensors do not play a significant role in the current manifestation of this work because the logger discussed in the next section already captures state data for analysis. Nevertheless, they do introduce interesting advanced safety

considerations into the flight-control system for future work on model reliability, for example.

For any number of reasons beyond the scope of this paper, a real-world mechanical system may erroneously exceed its specified design limits. An active preventative solution, in the form of a watchdog device, would immediately report any deviation as a fault [8]. A passive solution is also commonly present as mechanical stops that would physically prevent an actuator from exceeding its hard limits. However, it would still be possible to hit a stop and continue to try to move, likely resulting in wear or damage in the drive mechanism. A common hybrid approach is to monitor the actuator itself to see how hard it is working with respect to how much work it is performing: working hard with no effect likely means that it is at the limit, jammed, or otherwise interrupted. A garage-door opener is a good example.

F. Controllers

A fly-by-wire system is actually a hierarchical system of systems [9]. In this model, there is one master bus that runs from the cockpit throughout the entire aircraft, but no actuators are connected directly to it. Rather, it consists of controllers, which themselves have slave subbuses containing relevant actuators. This extra level of indirection allows for arbitrarily complex coordination at the receiving end (the actuators), where otherwise it would be the transmitting end (the cockpit) that would have to assume this responsibility. It also reduces bus traffic by sending consolidated requests to be interpreted, decomposed, and redistributed by the controllers as appropriate. The controllers are:

- *Rudder controller*: always contains a single rudder actuator on its subbus. A request to it (i.e., change deflection angle) passes untouched to the actuator.
- *Elevator controller*: always contains two elevator actuators. A request to it passes untouched to each.
- *Gear controller*: always contains two main-gear actuators and a nose-gear actuator. A request to it (extend or retract) passes untouched to each.
- *Flap controller*: contains an even number of symmetrically configured flap actuators, evenly distributed across the wings from left to right from the cockpit perspective. A request to it (downward deflection angle) passes untouched to each.
- *Engine controller*: contains any number of symmetrically configured engine actuators, distributed the same as flaps. A request to it (power percentage) has two interpretations. In gang mode, it passes untouched to each actuator; in isolation mode, it passes untouched to only the specified one.
- *Aileron controller*: contains an even number of symmetrically configured aileron actuators, also distributed the same as flaps. In standard roll mode, a request to it (upward or downward deflection angle) passes untouched to all actuators on the left wing, but the direction is inverted for the right wing. In mixed roll mode, the request addresses only the specified left actuator, and the other actuators on the left wing deflect as a ratio of it, and likewise those on the right. (Section VII discusses this process in more detail.) Finally, in speed-brake mode, the request (deploy or retract) passes to all actuators to deflect upward to their maximum angle or downward to neutral, respectively.

Fig. 4 is an architecture for a typical two-engine passenger airplane, where L, R, and N respectively indicate left, right, and nose. Section V addresses the command generator.

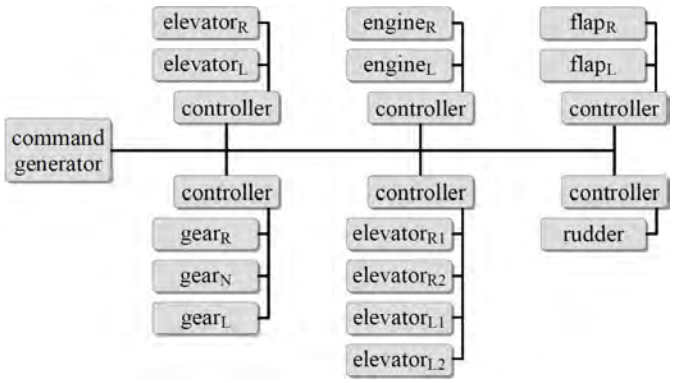


Fig. 4. Typical bus architecture

Additional levels of indirection (i.e., controllers containing controllers) could also be added for complicated decision-making. For example, in flight the appropriate amount of yaw from the rudder depends on the amount of roll from the ailerons with respect to airspeed. An aileron request could issue a secondary rudder request [10]. Similarly, roll reduces lift, which may be compensated for by the pilot or the control system. In general, Boeing's approach, for example, requires the pilot to introduce more pitch as a separate intentional action, whereas Airbus's does so automatically [11].

Although not part of this work, controllers in conjunction with sensors could monitor the behavior of actuators in concert. Symmetric configurations must be in identical (or identically inverted) states at all times; otherwise, the aerodynamic effects could be catastrophic. Similarly, advanced coordination of an autothrottle is possible [12].

Other complex monitoring relationships are also possible, but are deferred to future work. For example, in combination with flight data, the flight-control system could determine the operating limits that the pilot is permitted to reach [4]. Under normal circumstances, operating in so-called Normal Law, the system does not allow the aircraft to enter a region of unusual or diminished flight control. For rare cases, Alternate Law relaxes these restrictions, but it still would not allow the aircraft to exceed its maximum operating limitations. Direct Law basically disables automated oversight altogether for exceptional cases and allows the actuators to be driven as if the control system were purely mechanical.

IV. View

The view module of the model-view-controller architecture depicts the state of the system in multiple forms. Its plug-and-play nature allows other implementations to be added or substituted relatively easily.

A. *Log*

The most basic — but also most informative — view is text output to a log file. This view supports arbitrary logging of any aspects of interest, but the built-in output is generally sufficient for most analysis. The relevant details of each state at each clock tick go to a text file that directly exports to Excel. Fig. 5 shows an abridged form, which actually contains many more columns, as well as typically thousands of rows of events.

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		

Fig. 5. Abridged bus log

Not only does this log represent the states of the controllers and actuators, but it also shows the bus traffic. The communication process involves submitting a request and getting one or more immediate responses from its servicer. If the request is subsequently serviced, notifications are sent back to the requester for specified conditions like decelerating for arrival at the final state, arrival at the final state, preemption, and so on. This information could be exported to other applications to generate timing and UML sequence diagrams.

B. *Graph Visualization*

The textual form of the log file is richly informative, but it is not intuitively understandable. However, its structure not only exports natively to Excel for a tabular representation; its fields are also strategically organized to allow event chains to be plotted as line graphs. Fig. 6, for example, depicts the actions of a rudder actuator at the following key time points:

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

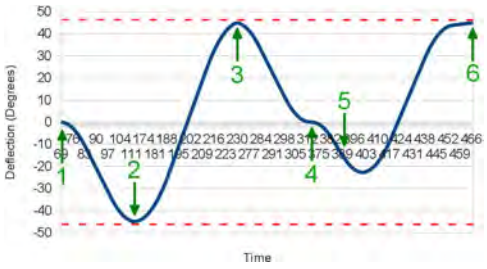


Fig. 6. Preemption test

Despite multiple requests, it is clear that at no time does the actuator find itself in an inconsistent state. i.e., above or below the dashed lines depicting the physical limits. Moreover, the line represents a smooth, continuous function with no jerks or breaks through various acceleration, deceleration, and preemption actions. Such intuitive visual inspection is invaluable for testing and evaluation. Furthermore, mathematical analysis on the slope of the function would demonstrate that the performance remained within the specifications at all times.

C. *Three-Dimensional Visualization*

Visual representation in graph form is useful with respect to a localized part of the system like a single actuator or a group of related actuators. However, for a global systems-level view, three-dimensional visualization, as in Fig. 7, is far better.

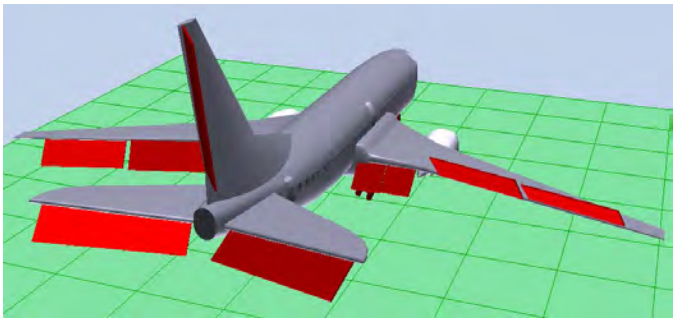


Fig. 7. Actuator visualization [1]

As the next section discusses, actuator configuration is highly dynamic and need not correspond to any particular aircraft. As a result, the visualizer is stylized with oversize control surfaces that represent the appropriate actions, not necessarily the appropriate appearance. The visualizer can also depict metadata like physical limits and breadcrumb tracks showing a history of state changes.

This OpenGL-based Java tool has seen extensive use in the author's artificial-intelligence and software-engineering courses, related pedagogical research, and industry work as a general-purpose world viewer [13,14]. It is freely available at shelby.ewu.edu.

V. Controller

The controller is the user interface for building the model and running the simulation. It is based on a regular grammar that employs the Interpreter and Command design patterns [2]. The instructor's solution defines the parser with JavaCC, but the students had to design and implement their own with standard Java. This effort entailed thoroughly understanding the problem domain of the commands and their proper usage, as well as the solution domain of the API for the provided architecture. It strongly discourages head-first, brute-force coding by making such an undisciplined approach obvious and unpleasantly difficult.

A. *Creational Commands*

Creational commands define and build the actuators via the Builder and Factory design patterns [2]. Each contains a unique identifier, the interval limits, a delta value for changing state on the interval, and an acceleration for changing the delta value.

All eight of these commands (for aileron, elevator, and rudder actuators, etc.) have the following form:

```
CREATE RUDDER id WITH LIMIT angle
  SPEED speed ACCELERATION acceleration
```

B. Structural Commands

Structural commands define and build the controllers from the actuators created above. Each contains a unique identifier and the actuators. The controllers with a fixed number of actuators are:

```
DECLARE RUDDER CONTROLLER id1
 WITH RUDDER id2

DECLARE ELEVATOR CONTROLLER id1
 WITH ELEVATORS id2 id3

DECLARE GEAR CONTROLLER id1
 WITH GEAR NOSE id2 MAIN id3 id4
```

The controllers with a variable number of actuators are:

```
DECLARE FLAP CONTROLLER id WITH FLAPS idn+

DECLARE ENGINE CONTROLLER id
 WITH ENGINE[S] idn+

DECLARE AILERON CONTROLLER id
 WITH AILERONS idn+ PRIMARY idx
 (SLAVE idslave TO idmaster BY
 percent PERCENT)*
```

All controllers must be added to the master bus:

```
DECLARE BUS id WITH CONTROLLER[S] idn+
```

And finally the configuration is locked to prohibit further creational or structural commands and to authorize most behavior commands:

```
COMMIT
```

It is at this point that students perform any late consistency checks to verify that the system is configured properly before any manipulation of it can occur. For example, the number of engines and their properties must mirror each other on the wings. From a practical design standpoint, such a check cannot be done earlier while the engines are still being added because the process is sequential. Determining what can be done immediately versus deferred for later, as well as how, is an important part of software design thinking [15].

C. Behavioral Commands

The behavioral commands send requests across the master bus to be interpreted by the appropriate controller(s):

```
DO id DEFLECT RUDDER angle LEFT | RIGHT
DO id DEFLECT ELEVATOR angle UP | DOWN
DO id DEFLECT AILERONS angle UP | DOWN
DO id SPEED BRAKE ON | OFF
DO id DEFLECT FLAP position
DO id SET POWER power
DO id SET POWER power ENGINE id
DO id GEAR UP | DOWN
HALT id
```

D. Miscellaneous Commands

The miscellaneous commands manipulate the execution of the simulation. The first set affects the system clock:

```
@CLOCK rate
@CLOCK PAUSE | RESUME | UPDATE
```

Especially important for testing is the capability to wait a fixed amount of time. The preemption tests, in particular, need to be timed exactly. Manually issuing a command would result in inconsistent results over multiple runs. The command is:

```
@WAIT time
```

Finally, commands may be supplied in text files for execution as scripts, which greatly simplifies repeatable testing:

```
@RUN "filename"
```

VI. Controlled Experiments

The primary goal of this work is to instill disciplined software-development behavior in students. However, it also naturally serves as an effective platform for systematically evaluating the performance of a model by using scientific method in controlled experiments as follows:

1. Design and carry out an experiment (a test) to investigate something of interest.
2. Visualize and analyze the results.
3. If the results are unsatisfactory, perturb one and only one parameter in the experiment and rerun it.
4. If the new results are more promising, continue down this line of investigation; otherwise, either perturb the parameter in a different way or reset it and perturb a different parameter.
5. Continue to refine the model until the results are satisfactory.

Consistent with the primary goal, students gain valuable experience with discovering patterns and building mental models that connect causes to effects [16]. This approach reduces the amount of random, uninformed generate-and-test cycles, which generally allows students to be more productive by getting better results with less effort.

VII. Results

The students' project consisted of three parts: implement the parser, implement the controllers, and execute a detailed test plan. The results of the first two parts were straightforward because this system does not allow them to deviate much from proper coding principles. The third part — testing and evaluation of the entire system — is the emphasis here. The deliverable required a formal report describing the test plan and its results. Each of 27 experiments addressed eight points, where 1–4 related to planning, 5–6 to execution, and 7–8 to presenting the results:

1. The rationale behind the test; i.e., what it was testing and why it mattered.

2. A general English description of the initial conditions.
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 results themselves are not as relevant here as what the students learned from them, but the following examples help convey how they learned it. Fig. 8 depicts the behavior of a three-engine configuration (where the dashed lines overlap). At time point (1), all three engines were commanded to increase power from 0 to 70%. Upon reaching this target at (2), the center engine was commanded to reduce power back to 0%. At (3), while the center engine was still decreasing, all three were commanded to go to 100%, which took until (4) to achieve.

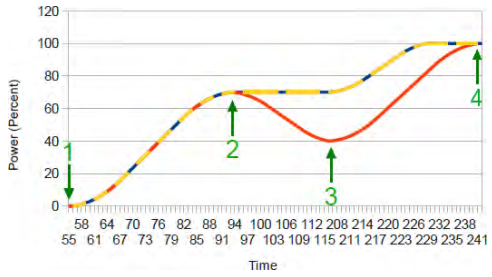


Fig. 8. Engine manipulation

Determining correctness (testing) and evaluating performance (optimization) require three critical components: the expected results, the actual results, and a meaningful way to compare the two. All too often students lack one, two, or even all three of these. While the technical act of acquiring such data is necessary, it alone 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 in this work provides endless opportunities to ground the programmatic exercises to reality in order to help students develop and improve their critical-thinking skills in computer science.

For example, experiments in engine manipulation could be better understood by knowing that it is normal procedure for pilots of many commercial airliners to spool the engines initially to 50% power on takeoff and then wait until they all reach this point before going to full power. The natural variability in engine performance does not guarantee exactly the same behavior simultaneously, which could have adverse effects on controllability during the takeoff roll. With this knowledge, students may realize that they are not just generating graphs; they are generating graphs that mean something, and if that something is not what it should be, then they have considerable insight into what may need fixing.

Another example in Fig. 9 shows a complex example of mixing eight ailerons with different performance properties in roll mode. At (1), with all ailerons at their neutral position of 0 degrees, the master aileron (M) is commanded to deflect upward to 45 degrees. The slave ailerons on the same wing move upward as a ratio of its movement, while those on the opposite wing correspondingly do so downward. Upon reaching 45 degrees at (2), the master is commanded downward to –40 degrees, which is achieved at (3). The graph convincingly shows that all ailerons remain antisymmetrically synchronized at all times.

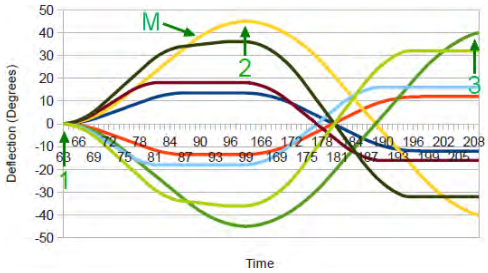


Fig. 9. Aileron manipulation

Fig. 10 shows the same ailerons acting in speed-brake mode. Starting again from neutral 0 degrees at (1), the eight ailerons deflect upward to their maximum limit of 90 degrees, which is achieved by all by (2). The master is then commanded to –20 degrees in roll mode, which results in behavior analogous to that in Fig. 9. The details of the action are beyond the scope of this paper, but again, the graph clearly demonstrates them to the students, who have the theoretical and practical foundation to know how to interpret it.

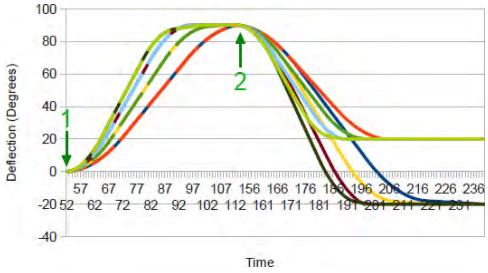


Fig. 10. Speed-brake manipulation

Finally, Fig. 11 shows a four-flap configuration, where two types of flaps differ in performance slightly. At all times through a variety of actions, the flaps of the same type remain synchronized, and the two types act appropriately with respect to one another. Such insight into the behavior of a complex system is invaluable in testing and evaluation.

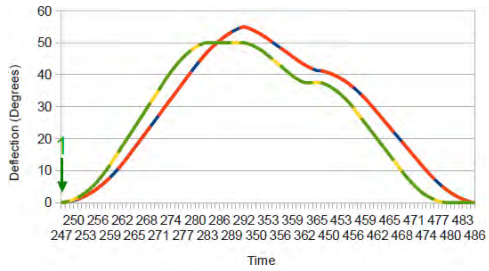


Fig. 11. Flap manipulation

Reporting and evaluating the results of human subjects in work of this scope must be summarized due to limited space. However, it is based on a significant breadth and depth of objective and subjective measures including anecdotal observation, individual contributions from a background survey, 11 assignments, and 10 anonymous weekly assessments, as well as individual and team contributions from 18 project status reports, a project reflection, a team evaluation, a self-evaluation, and a course evaluation.

Most telling, 90% (28 of 31) of the participating students indicated that the graphical form of the test report directly contributed to a better understanding of what their code was actually doing, where they otherwise would have had less confidence in their results. Overall, the students rated the project 4.6 out of 5 (excellent).

VIII. Future Work

Almost any complex physical system is characterized by input, processing, and output that could map to this architecture. For reuse in future projects, it should accommodate more complex interval behaviors, as well as other sensors, actuators, and controllers. Of particular interest within this application are detecting and handling faults and managing operating laws. For a graduate-level course, especially for software quality assurance, the evaluation framework based on controlled experiments could be significantly expanded for richer analysis. In particular, the introduction of probability would contribute to a powerful stochastic Monte Carlo methodology.

IX. Conclusion

This work mitigates the common problem-solving strategy of many students who undermine the design of a system by indiscriminately throwing more code at every issue they encounter. The hierarchical quasi-network-based architecture effectively prevents them from communicating with their components by any means except the prescribed ones. Its datatype-oriented interval framework safely and elegantly manages complex physical behaviors concurrently. This unified approach, combined with an overarching framework of modeling, simulation, visualization, and analysis, further provides a disciplined strategy for creating, executing, presenting, and analyzing meaningful test cases as part of formal test plans based on a sound methodology of controlled experiments via scientific method. Extensive results from a classroom deployment support the conclusion that students overwhelmingly benefit from this approach.

References

[1] Adapted from Google Sketchup Warehouse, 3dwarehouse.sketchup.com, last accessed 21 Apr. 2015.

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

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

[4] W. Phillips. Mechanics of Flight. Hoboken: Wiley, 2009.

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

[6] M. Cook. Flight Dynamics Principles. Elsevier, Waltham, MA, 2013.

[7] Adapted from Wikipedia, wikipedia.org/wiki/Flap_(aircraft), last accessed 21 Apr. 2015.

[8] R. Isermann, R. Schwartz, and S. Stolzl. “Fault-tolerant drive-by-wire systems.” IEEE Control Systems 22(5), 2002.

[9] S. Pope. “Fly by wire: Fact versus science fiction.” Flying, pp. 53–59, May 2014.

[10] M. Napolitano. Aircraft Dynamics: From Modeling to Simulation. Hoboken: Wiley, 2011.

[11] R. Collinson. Introduction to Avionics Systems. Springer, Netherlands, 2011.

[12] S. Pope. “Autothrottle advances.” Flying, pp. 66–69, April 2015.

[13] D. Tappan. “A pedagogical framework for modeling and simulating intelligent agents and control systems.” Technical Report WS-08-02, AAAI Press, 2008.

[14] D. Tappan. “Student-friendly Java-based multiagent event handling.” In Proc. of Association for the Advancement of Artificial Intelligence, Bellevue, WA, 2013.

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

[16] M. Salmon. Introduction to Logic and Critical Thinking. Cengage Learning, Boston, MA, 2013.

A HOLISTIC MULTIDISCIPLINARY APPROACH TO TEACHING SOFTWARE ENGINEERING THROUGH AIR TRAFFIC CONTROL

Dan Tappan
Department of Computer Science
Eastern Washington University
Cheney, WA 99004
509 359-7093
dtappan@ewu.edu

ABSTRACT

Software engineering is a highly multidisciplinary effort that plays a core role in today's complex systems of systems. Students need breadth and depth exposure to classroom projects based on substantial real-world problems, but in a way that is manageable for them and the instructor. This work showcases a holistic approach to an extensive, student-friendly Java simulation architecture for an air-traffic-control system. It addresses weaknesses and preconceived notions among students to help them understand, define, connect, manipulate, and evaluate the endless dots among vast, complex resources in an intentionally unfamiliar problem domain.

1. INTRODUCTION

Contemporary software engineering is a multidisciplinary fusion of many domains far beyond just the core programming that students often believe it to be. Learning to develop software for complex real-world systems of systems is essential preparation for the field [4]. However, adequately exposing students to the breadth and depth of a representative project within the classroom environment is logistically difficult. This paper discusses an approach applied to the vast domain of air traffic control. The objectives were conventional: to understand and translate a complex problem domain into an appropriate solution domain, then to evaluate its performance from end to end. The novel aspect is the 372-class Java model-view-controller simulation architecture made available to the students in support of their tasks. Unlike most software that addresses non-toy problems, this architecture was designed from the ground up to be understandable and accessible to students. Furthermore, its modularized approach intentionally aligns with the teaching philosophy for the third-year software-engineering course that used it, as well as to specifics of the curriculum and the student population.

A pedagogical emphasis is to push students outside of their comfort zone, where it becomes unavoidable to apply research and critical-thinking skills to make holistic sense of an overwhelmingly unfamiliar problem. They must understand not only how the real-world system is constructed and operates, but also how those elements map onto the software-development process and the subsequent 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. When left to their own devices, students tend to gravitate toward bloated and brittle ad hoc solutions made up as they go, whereas this approach required demonstrably unified, orthogonal, reusable, scalable, and extensible solutions. The final product was a multiagent, continuous time-stepped simulation in which students played the computer-science roles of analyst,

designer, implementer, and tester, as well as multiple end-user roles as air traffic controllers.

2. BACKGROUND

The National Airspace System is a vast subject with complex technology and endless rules and regulations [2, 6]. For practicality, the problem space for this work is reduced as much as possible, while still retaining its essential elements.

Aircraft are generalizations of airplanes, helicopters, and unmanned aerial vehicles. The only requirement is that they can be controlled in the air and on the ground in terms of which direction to travel in, where to go, how fast, and how high. All aircraft are automated; there is no user role.

Navigational aids (navaids) help aircraft fly between fixed points in the world. The operational aspects are generalized into two categories. *Global navaids* define points between airports with nondirectional beacons (NDB) and very-high-frequency omnidirectional range (VOR) stations. *Local navaids* define points within airport environments. Their components (i.e., marker beacons for distance, and localizer and glideslope for horizontal and vertical guidance, respectively) comprise an instrument landing system (ILS) to guide an aircraft to a runway. All navaids are automated.

Airports consist of interconnecting runways and taxiways, as well as ramp areas between them and the terminals, which contain gates. Runways have an ILS at each end.

Airspace consists of various three-dimensional geometric partitions that delineate different control regions and procedures. The students were using this system to learn about applied real-world software engineering, not about how to manage air traffic realistically, safely, and efficiently, so there was considerable freedom here.

Air traffic controllers are the roles the user plays with different radar displays. While multiple users could play different roles simultaneously, the intent was for a single user to transition a single aircraft through all the gate-to-gate stages of a flight by changing roles at appropriate times. The *ground controller's* role is to manage aircraft at all points on the airport grounds except on the runways. The normal flow is to instruct a departing aircraft to move from its gate onto the ramp, and then via any number of taxiways up to a runway for takeoff. The process is reversed for arrival. The *tower controller's* role is to manage aircraft on the runways and in the air immediately departing or approaching them. For departure, the ground controller hands the aircraft off to the tower controller, who instructs it to take off. The process is reversed for arrival. The *departure controller's* role is to manage airborne aircraft from beyond the immediate runway environment out to roughly 30 nautical miles. For departure, the tower controller hands the aircraft off to the departure controller, who guides it outbound. The process is reversed for arrival with the *approach controller*. Finally, the *enroute controller's* role is to manage airborne aircraft between the departure and approach regions of airports. The departure controller hands the aircraft off to the enroute controller, whose control zone extends hundreds of miles. The process is reversed for arrival with the approach controller. An aircraft can take off and land within the same zone, or it may be handed off to adjacent zones and enroute controllers.

3. ARCHITECTURE

The architecture combines traditional model-view-controller modules that clearly

separate the main concerns of the system [5]. While they are still interconnected, the dependencies are kept to a minimum such that different versions of the modules may be implemented and tested independently and then substituted seamlessly.

3.1 Model

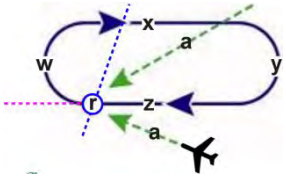
The model defines the components in the world from various software-engineering perspectives. All maintain a state with their identifier, position, orientation, etc. *Static* components like navaids and airports do not change state, whereas *dynamic* ones like aircraft do.

Component *data* and *control*—what they are and are capable of doing, respectively—map directly to class member data and methods in object-oriented programming. Extensive classroom evidence shows that students have a major problem with abstracting, maintaining, and manipulating real-world data properly. Java primitives are appropriate in earlier low-level courses, but at higher project-based levels, they lead to a proliferation of problems. For example, units, magnitudes, and limits are not applied consistently, error handling is almost nonexistent, and code bloats from haphazard reimplementation of similar solutions in multiple places. To mitigate this situation, the architecture provides a rich set of self-contained concrete datatypes for every kind of relevant data; e.g., airspeed, altitude, coordinates, course, distance, heading, latitude, longitude, and dozens more. Each maintains its own error checking and helper methods for manipulating and converting it appropriately. This approach lends itself to convenient unit testing in isolation. It also reduces the burden of documentation.

Component *behavior* is what is done with dynamic components, or what they do on their own. In particular, it provides the context for making aircraft act appropriately with respect to their real-world counterparts they represent. There are two categories. *Primitive instructions* turn an aircraft to a course, fly to a navaid, fly for a certain time, assume an altitude, or change speed, all within its defined performance limitations. They are independent, and the controller must issue them individually, which takes significant time and communication bandwidth. *Composite instructions*, on the other hand, are predefined maneuvers that the controller issues once and then delegates their interpretation and autonomous execution to the aircraft. They are the emphasis here because they required students to represent the behavior of maneuvers in terms of contextually dependent primitive instructions mapped onto the data and control. The logic must operate solely by issuing the primitives to the architecture through the API. Customary control statements like conditionals and loops are not an option. This approach forces students to understand how to communicate with the architecture.

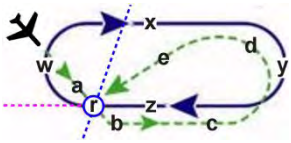
One significant task was to implement variations on a holding pattern, which is any orientation of the racetrack shape in Figure 1 that keeps an aircraft continuously flying within a protected region until it can be handled further. The controller would simply instruct the aircraft to hold at navaid ***r***. Depending on where it is initially with respect to ***r***, it must reorient itself to fly clockwise around the pattern. The sequence of corresponding primitives is below each. The last step of *Parallel* and *Teardrop* corresponds to the last five in *Direct*.

Direct



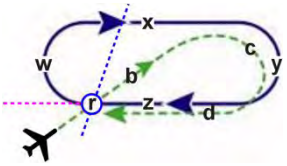
a fly to fix ***r***
w turn right to reciprocal course
x fly for one minute
y turn right to pattern course
z fly to fix ***r***
r repeat from ***w***

Parallel



a fly to fix ***r***
b turn left to reciprocal course
c fly for one minute
d turn left 180 degrees
e fly to fix ***r***
w continue around ***x***, ***y***, ***z***, ***w***, ...

Teardrop

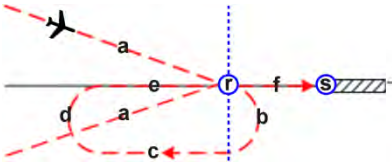


a fly to fix ***r***
b fly for 45 seconds
c turn right to pattern course
d fly to fix ***r***
w continue around ***x***, ***y***, ***z***, ***w***, ...

Figure 1: Holding Patterns [1]

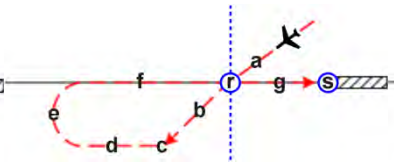
Another representative task was similar. To land (generally after holding), an aircraft must be aligned with the assigned runway. However, it does not fly directly there. Rather, the controller instructs it to intercept a navaid twice before following the instrument landing system down to the runway. As with holds, the initial location and direction of the aircraft with respect to the navaid and runway dictate what it must do to align itself autonomously. Figure 2 shows three variants of an aircraft executing this maneuver, called a procedure turn, along with their underlying primitives. A fourth variant, where the aircraft is already aligned, would fly directly to ***r*** then ***s***.

Racetrack



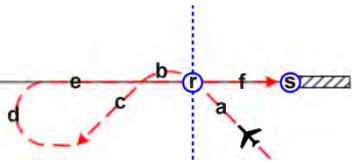
a fly to initial approach fix ***r***
b turn right to runway reciprocal
c fly for one minute
d turn right to landing course
e fly to final approach fix ***r***
f fly to ILS ***s***

Modified Racetrack



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

S-Turn



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

Figure 2: Procedure Turns

3.2 View

The role of the view module is to depict what is happening in the world. The user's view consists of any number of radar displays, as in Figure 3. The only technical difference between them is their scale and the details they render. The primary architectural aspect of interest for the students was the layered compositionality that turns details on or off. Any number of layers, like weather or background clutter, can be superimposed for unlimited scalability and extensibility.

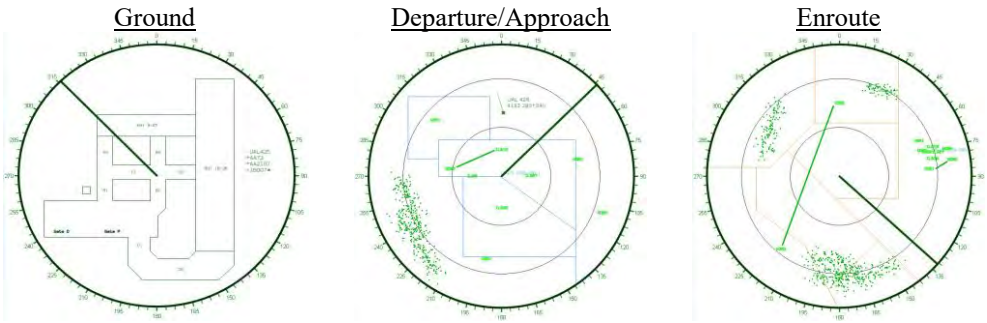


Figure 3: Radar Displays

For testing and evaluation, the state of the model also exports to external visualization tools. Figure 4 respectively shows two and three-dimensional views from Gnuplot and from a Java 3D visualizer used in many of the author's projects [7].

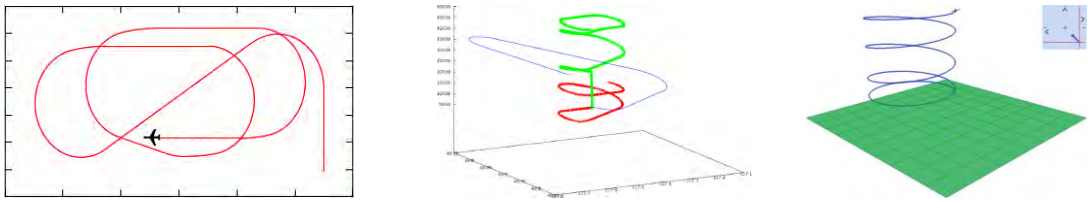


Figure 4: External Visualization

3.3 Controller

The role of the controller module is threefold: to accept instructions from the user to build the world; to control the aircraft in the world; and to operate the metalevel aspects of the simulation. The basis of these text commands is well-established software design patterns that partition them into four categories [3]. The students had to build the parser for them. The 16 *creational* commands define the available components; e.g., `CREATE VOR MyVOR AT LAT 47*33'53.805" LON 117*37'36.789" ALT 2756.3 ON FREQ 115.50`. The 11 *structural* commands connect composite components, such as airports with taxiways and runways, as well as add all components in use to the world. The 25 *behavioral* commands allow the user to communicate with the aircraft. Finally, the 10 *miscellaneous* commands allow for control over the simulation, such as setting up and running tests, and logging their results.

4. RESULTS AND DISCUSSION

The objectives were to understand and translate a large, complex problem domain into an appropriate solution domain, then to evaluate the performance of the final product. Reporting and evaluating results in work of this scope is limited by space, so this part is heavily generalized. However, it is based on a significant breadth and depth of objective and subjective measures including anecdotal observation, individual contributions from a background survey, 11 assignments, and 10 anonymous weekly assessments, as well as individual and team contributions from 18 project status reports, a project reflection, a team evaluation, and a course evaluation.

The development process could be characterized through a narrative as mildly oppositional. Students entered the course with limited skills but an overabundance of confidence. They wanted to code, which is what they considered software engineering to

be solely about. Given the early opportunity to demonstrate their perceived coding skills on a proof-of-concept task, the results were disastrous because they did not recognize the value of critical thinking in decomposing, analyzing, and understanding the problem domain. In fact, they objected to these activities as "busy work." They considered themselves already conversant in the subject matter from media portrayals of aviation and air traffic control, as well as from "common sense." Most of this background was irrelevant, misleading, or completely wrong. They did not appreciate the value of designing a solution that demonstrably corresponded to the problem it addressed because they lacked an understanding of both the problem and how to use code appropriately for real-world solutions. They did not make effective use of the documentation for the API and many other extensive resources provided for the architecture, and instead opted to try to do it their own way by brute force, with little success. Gradually, however, they did come to appreciate the tenets of software engineering that they were being forced to apply. In the project reflection, 84% said that they recognized the purpose of each step, and 86% admitted that they would have been unlikely to achieve a solution of similar quality if they had done it their own way.

Evaluating performance of a system is a critical part of testing, but often in the classroom environment, it does not get adequate coverage for logistical reasons. Here it was integrated throughout as a major part of the project. The final deliverable was a formal report describing the test plan and its results. Each of 47 experiments addressed eight points related to the planning, execution, and presentation of results. The students had only a general description of each test to satisfy, from which they had to determine and execute the appropriate instructions, collect the data, compare and contrast the actual with the expected results, present the findings, and draw conclusions. It was here that they definitely appreciated the holistic activities that led them to this point: 89% said they understood how everything connected and why.

5. CONCLUSION

This paper showcased part of an approach to helping students establish and connect the dots among numerous complex, unfamiliar resources within a large but manageable real-world project. It walked them through a development process of accepting what software engineering is really about, and then helping them make the best use of their existing technical skills to perform it. The final product was an extensive and highly flexible multiagent simulator that allowed the students to play many roles in its development and usage. A rich set of measurements showed how the students matured from start to end.

6. REFERENCES

[1] Adapted from wikipedia.org/wiki/Holding_(aeronautics), retrieved April 10, 2014.
[2] Federal Aviation Administration. *Federal Aviation Regulations Aeronautical Information Manual*. Newcastle, WA: Aviation Supplies and Academics, 2014.
[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis, IN: Addison-Wesley, 1995.
[4] Hunt, A. *Pragmatic Thinking and Learning: Refactor Your Wetware*. USA: Pragmatic Bookshelf, 2008.
[5] McConnell, S. *Code Complete: A Practical Handbook of Software Construction*.

Redmond, WA: Microsoft Press, 2004.
[6] Mills, T. and Archibald, J. *The Pilot's Reference to ATC Procedures and Phraseology*. Van Nuys, CA: Reavco, 2000.
[7] Tappan, D. A Pedagogy-Oriented Modeling-and-Simulation Environment for AI Scenarios. *Proceedings of WorldComp International Conference on Artificial Intelligence*, 2009.

Toward Introspective Human Versus Machine Learning of Simulated Airplane Flight Dynamics

Dan Tappan and Matt Hempleman
Department of Computer Science
Eastern Washington University
{dtappan,mhemple}@ewu.edu

Abstract

This paper presents the preliminary results of an extensible Java architecture for modeling, simulating, visualizing, and analyzing modularized, plug-and-play machine-learning strategies applied to instrument-based airplane flight control. A set of basic flight maneuvers challenged the machine to learn how to fly unsupervised by trial and error, from which the learning module attempted to introspectively determine interdependencies among the many inputs and outputs. For baseline comparison, this work also included a pilot study on human subjects who conducted the same experiments. The overarching goal was to determine how, and how well, both groups learned to solve the same flight-related problems on their own, which could be useful to refine and expand the learning strategies.

Introduction

Flying an airplane by reference to its cockpit instruments alone—no external visual cues—is a complex, multi-dimensional, real-time task that maps a small set of inputs to a large set of dynamically changing outputs in a continuous feedback loop. Formally learning to understand and manipulate such a system is mostly a top-down directed process, whereby a teacher explains problems and how to solve them, and then the learner repeatedly practices variations on the solution process under different conditions until achieving consistent, satisfactory performance (Guralnick and Levy 2009). A problem with this approach for machine learning is that the teacher's investment and oversight may become so extensive that they are almost explicitly programming the solution (Poli, Langdon, and McPhee 2004).

Although impractical in real life, learning to fly in a predominantly unsupervised bottom-up manner by trial and error may also be effective. In a simulated environment with no real consequences for failure, the unsupervised learner may be able to develop their own model of how the system operates with far less hands-on involvement from the teacher. Not only may it be possible for this reinforcement approach to achieve the same goals, but if done strategically, it could also introspectively show how it learned to do so for insight into the process of both flying and learning to fly (Haykin 1994; Harrington 2012).

This work focuses on an extensible architecture for the modeling, simulation, visualization, and analysis of instrument-based airplane flight control, with a plug-and-play module for the learning strategy. The long-term application is to investigate and compare various machine-learning strategies. This paper describes the architecture, a straightforward proof-of-concept learning strategy, and a pilot study of human subjects for comparison. The primary goal is to determine how, and how well, both groups learn to solve the same flight-related problems on their own.

Pedagogical Foundation

Any nontrivial system has complex interrelationships among its components. The continuous mapping of inputs to processing to outputs is based on countless direct and indirect dependencies, correlations, causes and effects, stimuli and actions, and so on (Haykin 1994; Jones 2008). The framework for learning here is based on first decomposing the problem space of flight data into its constituent W5H question words (i.e., *who*, *what*, *when*, *where*, *why*, and *how*), and then trying to establish a richly interconnected associative DIKW structure for it hierarchically from superficial to deep understanding as follows (Bloom 1956; Dorn 1989; Irish 1999; Rowley 2007):

- ***D**ata*: raw values with no associativity or context; *what* questions.
- ***I**nformation*: values in one context; *how* questions.
- ***K**nowledge*: values in multiple contexts; *when*, *where*, and *why* relationships.
- ***W**isdom*: creation of generalized principles by connecting a network of contexts from different sources for predictive, anticipatory, proactive understanding.

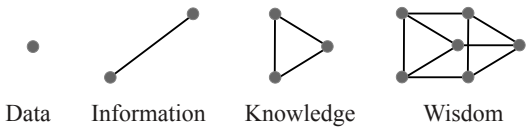


Figure 1: Learning Associativity

An accomplished learner (the *who*) can generally indicate *what* happens *when* and *where*, and *how* it happened or *how* to make it happen, but they do not necessarily understand *why*. The introspective aspect of this work allows for postanalysis by a subject-matter expert to glean insight into the rationale behind decisions. Such insight could be used to refine teaching and learning processes.

System Architecture

The system consists of 327 Java classes, with Swing and Java 3D for the graphics. The human test subjects were using this code base primarily for developing an unmanned aerial vehicle simulator as the project in their undergraduate software-engineering course, so much of this code is not directly related to this work yet. The main components of interest here are the flight-dynamics model, machine-learning engine, instrumentation, and data logger.

Flight Dynamics

The flight dynamics reflect a Cessna 172, which is the world's most popular airplane thanks to its docile handling characteristics and forgiving nature (Cessna 2014). The underlying flight-dynamics model, while a necessary abstraction and simplification of reality, still captures the main elements of any traditional fixed-wing aircraft (FAA 2011). Its six degrees of freedom represent where the airplane is positioned in three-dimensional space, and where it is facing. Specifically, it uses a right-hand coordinate system for x, y, and z, as indicated in Figure 2, where rotation about each axis is respectively roll, pitch, and yaw.

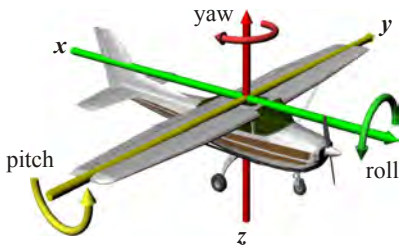


Figure 2: Coordinate System (Sketchup 2014)

In addition, two axes correspond to the main forces of flight. Thrust moves the airplane forward along the x axis, which drag opposes. Lift is always perpendicular to the xy plane, while weight (gravity) is always straight down. The x, y, z and weight components are in the global (world) frame of reference and are independent of the airplane, whereas roll, pitch, yaw, thrust, drag, and lift are in the local frame of reference.

Input

The flight control surfaces in Figure 3 redirect airflow over the airplane to change the roll, pitch, and yaw, which in turn contribute to changes in the (x,y,z) position. The elevator on both sides of the horizontal stabilizer deflects up or down in unison to change pitch. The ailerons outboard on each main wing deflect up or down in opposition to induce roll. The rudder on the vertical stabilizer deflects left or right to coordinate changes in yaw. The flaps inboard on the wings deflect down in unison to increase the wing lift and drag, generally only for landing. Finally, the propeller generates thrust. The *Flight Dynamics Processing* section describes these relationships in detail.

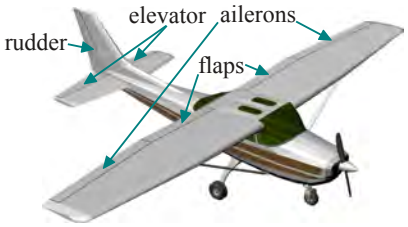


Figure 3: Flight Control Surfaces (Sketchup 2014)

The primary real-world control interface usually involves a wheel, yoke, or stick, as well as pedals. For logistical reasons, the human interface was limited to the keyboard. There were three modes of operation connecting a key press to an action:

- *Instantaneous* changes go to the maximum limit immediately and return to neutral upon release.
- *Incremental auto* changes occur stepwise until reaching the maximum limit or the key is released, then return stepwise to neutral.
- *Incremental manual* changes occur stepwise until reaching the maximum limit or the key is released, then remain there. Opposite action is necessary to neutralize the effect.

The throttle was always in incremental manual mode. Otherwise, this paper consider only instantaneous and incremental auto. The modes remained separate in the experiments for independent analysis. The rationale is that instantaneous inputs are likely tied to determining only *what* the appropriate action is and *when*, whereas incremental inputs also factor in *how much* to apply in terms of time, as well as how to cancel the action.

Output

To fly—and especially to learn to fly—the pilot needs constant awareness of the state of the airplane with respect to the world, known as situational awareness (FAA 2011). The underlying mathematical model, with its 32 variables, is a major simplification of the real world with perhaps several times this number (Napolitano 2011). However, most of these data are not directly accessible to the pilot, who is limited to observing only what is depicted by the instruments. (Visual and kinesthetic [motion] senses play a role in visual flight, but not in instrument flight; in fact,

ignoring kinesthetic inputs, which are dangerously deceiving, is a major challenge.)

Excel

Instruments depict data or information either by directly presenting it (e.g., altitude determined by air pressure) or indirectly computing it from multiple fused sources (e.g., vertical speed as a change in altitude over time). While the focus on learning here by both human and machine is limited to the instrument depiction, it is valuable (from a DIKW standpoint) to see the underlying raw source. An extensive log file conveniently exports directly to Excel, as in Figure 4.

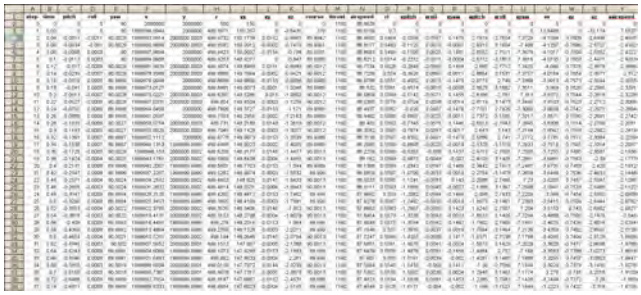


Figure 4: Excel Log Data

While these values represent the discrete states of the simulation in every pertinent detail, no human—even a subject-matter expert—could make intuitive sense of them in this form, which continues for thousands of entries for most maneuvers. Basic visualization as line plots, however, as in Figure 5, can be very revealing. While this representation is beyond the scope of this paper, it is relevant and worthwhile to mention because the key aspect in their value is in deciding *which data* to plot: meaningful relationships are only apparent when presented as appropriate combinations of independent and dependent variables.

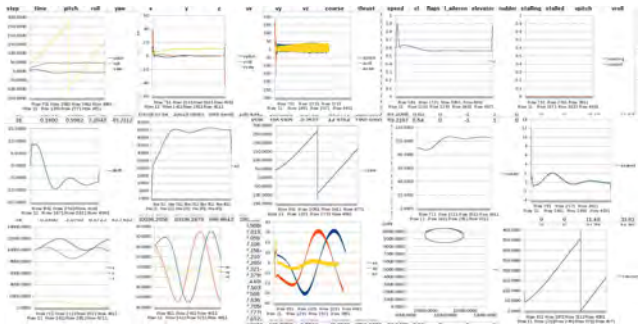


Figure 5: Excel Log Plots

Humans, lacking any insight into the raw data at all, would not be able to decide wisely which plots to generate. Most combinations would be meaningless, although a human would likely find many baseless correlations. Indeed, in an earlier assignment, students were seriously confused by extraneous data and drew wildly incorrect conclusions. A similar situation commonly occurs with machine learning by overfitting the data, among other causes (Conway 2012). Although a machine can easily consider countless combinations, very few of them would truly reflect meaningful correlative and causative behaviors of the unknown system. Therefore, any brute-force approach on the raw data would need to be selective. This foresight played an important role in deciding how to set up the machine learning to operate on the instrumentation data, as discussed in the *Machine Learning* section.

Instrumentation

The nine instruments in Figure 6 depict the refined state of the airplane derived from the raw data. Students in another earlier assignment had already researched their basic form and function, but until this assignment had never seen them in operation. The only difference between the student and machine perspectives was that the students saw this visual representation, whereas the machine saw the equivalent variable representation (e.g., needle position).

A. *Airspeed Indicator* (ASI): shows airspeed in knots.
B. *Attitude Indicator* (AI): shows pitch and roll via an artificial horizon.
C. *Altimeter*: shows altitude in feet above sea level (which is the ground here); the caret, thick needle and thin needle are 10,000, 1,000, and 100 feet, respectively.
D. *Turn Coordinator* (TC): shows rate of turn in degrees per second via the bar, as well as nose-to-tail alignment in a turn via the ball; the *Preliminary Results and Discussion* section elaborates on this relationship
E. *Directional Gyro* (DG): serves as a compass, where the numbers rotate around the stationary airplane.
F. *Vertical-Speed Indicator* (VSI): shows change in altitude in positive or negative feet per minute.
G. *Clock*: serves as an ordinary clock; the caret and reset button were not in play.
H. *Tachometer*: shows propeller revolutions per minute.
I. *Stall Warning*: shows when the wings have ceased to provide lift, resulting in imminent loss of control.

This set of primary instruments, minus G, H, and I, is often called the "six pack" because together they minimally depict the state of the airplane. Loss of one or more, known as a partial panel, may be accommodated with significantly more difficulty by interpreting the others in combination, but such a condition was not part of this work. Nevertheless, the general approach should still apply, although likely with degraded results.

Figure 6: Instrument Panel

The architecture also supports six navigational instruments, but the panel omitted them for these experiments. None of the tests addressed a global frame of reference that required the pilot to know where the airplane was with respect to the world (except in altitude).

3D Viewer

Although the scope of this work was limited to the internal cockpit view of the instruments, for reference after tests, an external view was available. Not only was it entertaining to review both the successful and spectacularly disastrous results, but the discussion proved to be very informative to both students and instructor on why students made their decisions. Such rich reflective and introspective interaction with the machine-learning aspect would be an ideal goal for future work beyond this limited approach .

Figure 7 shows three-dimensional visualizations for two attempts at a counterclockwise turn. This visualizer has seen extensive use in the first author's artificial intelligence courses, related pedagogical research, and industry work as a general-purpose world viewer (Tappan 2008, 2009, 2012).

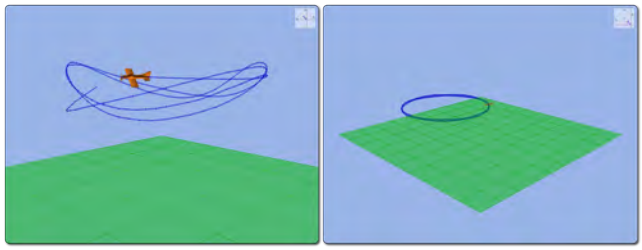


Figure 7: Turn Visualizations

Flight Dynamics Processing

The flight-dynamics model is a Java port of the C++ code by Bourg (2002). The main differences are in the input mechanism to account for the instantaneous and incremental modes, the extensive logging capability, and changes to the flight characteristics to model a Cessna 172. Higher-fidelity models are available, but the internals of this one are especially accessible for inspection and logging (Allerton 2009; Napolitano 2011).

While the complex differential equations of flight involve countless intricate interactions, the main objectives of this study were to elicit an understanding of at least the following representative cause-and-effect relationships, which are generalized here for aerodynamic reasons beyond the scope of discussion (FAA 2011):

- An increase in elevator deflection (up) causes an increase in pitch (depicted in the AI), which causes an increase in lift (in the VSI and altimeter) and a decrease in speed (in the ASI) until a stall occurs (in the stall warning); the opposite holds for a decrease in elevator deflection, except for the stall, and the propeller speed increases (in the tachometer).
- An increase in left aileron deflection (up), and therefore down on the right, causes a roll to the left (in the AI), which causes a turn to the left (in the DG and TC bar and ball), as well as a loss of lift (in the VSI and altimeter); the opposite holds for a decrease in left aileron.
- An increase in rudder (right) causes a yaw to the right (in the TC ball), which causes a roll to the right (in the AI), which causes a turn to the right (in the DG and TC bar), as well as a loss of lift (in the VSI and altimeter); the opposite holds for a decrease in rudder. The *Preliminary Results and Discussion* section discusses this relationship further.
- An increase in flap deflection (down) causes a decrease in pitch (in the AI) and speed (in the ASI), but an increase in lift (in the VSI and altimeter); the opposite is dependent on the initial state.
- An increase in throttle causes an increase in propeller speed (in the tachometer), which increases thrust (not depicted on any instrument), which results in an increase in speed (in the ASI) and therefore an increase in lift (in the VSI and altimeter); the opposite holds for a decrease in throttle.

Machine Learning

The long-term purpose of this plug-and-play architecture is to investigate various machine-learning strategies applied to this problem space. At this preliminary stage, only a proof-of-concept module is in play.

Evaluation of learning (machine and human) was not through the traditional crossvalidation approach of learning on a training set, then performing on a withheld test set.

Rather, the goal was simply to reach the objectives however possible reactively, and then for a subject-matter expert to analyze these steps qualitatively to gain insight into how the subjects presumably learned. For now, there is no way to repeat the actions proactively based on this experience, but this capability will be added eventually for rigorous quantitative analysis. Specifically, the steps are:

1. *Acquisition*: receive data from sensors
2. *Transformation*: convert data into usable form
3. *Fusion*: combine data into coherent, unified views
4. *Inference*: derive unstated data
5. *Reasoning*: make sense of data
6. *Prediction*: anticipate trajectory of data

It is fair to characterize the provisional approach here as pure brute force and very restrictive, but it does reasonably reflect the students' approach of developing their own generalized principles through trial and error without understanding the underlying aerodynamic principles. It is an enumerative approach of trying an input, seeing its effects, and continuing if the trajectory toward the objective appears promising, or discontinuing otherwise and trying something else.

The objectives are declarative statements defining the form of an acceptable solution (with some freedom). For humans, English sufficed (e.g., climb at 80 knots); for the machine, it was equivalent hardcoded conditional statements. A priori knowledge was necessary to constrain the solutions to reasonable flight characteristics and avoid undesirable states like flying upside down (Mitchell 1997). Students had acquired this background from earlier research; the machine required additional logic.

The reinforcement signal for evaluating trajectory was crude: converging, diverging, or no effect. It functioned somewhat like a myopic feed-forward neural network with no or few hidden layers and a three-state linear activation function (Haykin 1999; Bourg and Seemann 2004). Each of the four inputs (elevator, aileron, rudder, and throttle) mapped to the 11 accessible values in the instruments (roll, pitch, yaw, speed, etc.). Flaps were initially considered but quickly discarded due to their overwhelmingly destructive effect on the other inputs. The direct mapping considered 44 combinations (4×11); the indirect mapping had a second layer with 440 (4×11×10), and a third layer with 3,960 (4×11×10×9), for a grand total of 4,444 combinations. This network captures relationships of input→output, input→(output$_1$∧output$_2$), and input→(output$_1$∧output$_2$∧ output$_3$), respectively. The decreasing count reflects no need to map to the same instrument output twice. This approach addresses steps 1 through 3 above.

Experiments

A suite of rudimentary experiments provided a rich basis for discovering relationships. Each experiment consisted of a task to perform, which could be attempted any number of times. The logger kept track of the performance data.

Tasks

The 14 tasks considered are basic flight maneuvers that demonstrate a recognition of the current state of the airplane and some understanding of what needs to be done to achieve the desired next state repeatedly toward the final objective (FAA 2012). Each attempt at satisfying a task started in the air with the same initial conditions and was independent of any others. The attempt ended upon reaching the objective or significantly exceeding the specifications. The tasks could be performed in any order.

- *Straight and level*: fly in a straight line with no change in course (0 degrees), altitude (3,000 feet), or speed (80 knots), which are the initial conditions.
- *Indefinite climb*: increase altitude indefinitely at any sustainable vertical rate, where sustainable means stall or loss of control is not imminent.
- *Definite climb*: increase altitude to 4,000 feet at any sustainable vertical rate, then level off.
- *Indefinite constant-rate climb*: increase altitude indefinitely at 500 feet per minute (FPM).
- *Indefinite constant-speed climb*: increase altitude indefinitely while holding speed at 80 knots.
- *Indefinite descent*: decrease altitude indefinitely at any sustainable vertical rate.
- *Definite descent*: decrease altitude to 2,000 feet at any sustainable vertical rate, then level off.
- *Indefinite constant-rate descent*: decrease altitude indefinitely at 500 feet per minute.
- *Indefinite constant-speed descent*: decrease altitude indefinitely while holding speed at 80 knots.
- *Left turn*: perform a 360-degree left turn while holding altitude at 3,000 feet.
- *Climbing constant-rate left turn*: perform a 360-degree left turn while climbing at 500 FPM.
- *Descending constant-rate left turn*: perform a 360-degree left turn while descending at 500 FPM.
- *Descending constant-speed left turn*: perform a 360-degree left turn while descending at 80 knots.
- *Landing*: synchronize a descent with flaps with no change in course (0 degrees) such that altitude is 0 feet when rate of descent is 0 FPM and airspeed is 40 knots (stall). There was no actual runway to target.

Right turns were not considered because in this simplified flight model, they would be mirror images of the left turns. In real airplanes, the characteristics would often be different for reasons beyond the scope of this discussion (Phillips 2009).

All attempts started from straight and level. The first maneuver therefore was to transition to the intended flight maneuver, then to hold it. Tasks with definite targets then

transitioned back to straight and level, whereas indefinite ones simply terminated. For the machine, there is no planning of any sort to carry out tasks. Students were not asked about how they carried them out.

Data Acquisition

The protocol for performing each task was the same for human and machine. The task was indicated, and the state data through each attempt were recorded from start to end. Any number of attempts was possible; only the best was considered here.

The human subjects consisted of three groups. Two were students in different offerings of fundamentally the same upper-division undergraduate software-engineering course, 41 subjects in total. According to a preassignment survey, none had any background in aviation, although some had relevant gaming experience. It was not a goal of this work to compare these groups to each other, so they were considered together as the student subjects.

The third group consisted of a single person, the instructor and principal investigator, with over 20 years of relevant real-world flight experience in both airplanes and helicopters. These results served as a control to verify that the tasks could be performed to the specifications. They also provided some indication of the maximum variation to expect on each task. Even a subject-matter expert exhibits some learning curve and performance inconsistencies, especially due to the unorthodox keyboard input mechanism. The results of the control group were not part of the analysis due to obvious biases. A better control group would consist of real pilots with no role in the development of the project, but for this pilot study, such objective baseline performance was not critical.

Humans subjects had the option of discarding the data from an attempt if they deemed it too unrepresentative of a valid attempt. For example, mistakes in keyboard commands were common. Without this option, the data would subsequently record the process of regaining control, which was not under study.

Data acquisition from the machine-learning process was identical, except that it could not opt to discard its results itself. For both groups, there was selective manual postprocessing for consistency. A common example was removing data from a protracted initial straight-and-level configuration to the start of the attempt, and then after achieving the objective, if the attempt did not terminate on its own.

Preliminary Results and Discussion

Despite working on a graded assignment requiring substantial effort, students by and large enjoyed the exercise, even going so far as to write in the postassignment analysis that they had “serious fun” with it. Moreover, their results were quite consistent with the relationships expected in the *Flight Dynamics Processing* section.

A typical subset of students did not take all or parts of the assignment seriously and submitted unusable results, but these were easily culled by inspection of the three-dimensional visualizations. The remaining results of primary interest are characterized here, but due to space limitations, this discussion addresses only the highlights. Unless otherwise indicated, the student and machine actions were fundamentally the same, although the performance of the former group was collectively always much better.

Instantaneous input mode (i.e., neutral and full control-surface deflection only) was surprisingly much better than incremental auto mode (i.e., smooth stepwise actions) for both groups for all tasks. While instantaneous mode produced choppy (vomit-inducing) results, on average they were more consistent with the expected trajectory. Unfortunately, there was no postassignment survey question that addressed this aspect, so the reason cannot be substantiated. Anecdotally, it appears related to uncertainty in how much force was being applied to the controls, which a real pilot is usually aware of by feel (Napolitano 2011). No instrument depicts this feedback.

Elevator operation was partially intuitive: push forward to go down, and pull back to go up. However, it was not immediately clear that the pitch remains set even when the elevators return to neutral (i.e., input changes elevator, which changes pitch), so the climb and descent continue for some time until aerodynamic effects level the pitch. As a result, the definite tasks often overshot their altitude targets. The machine approach never began this transition early because it is purely a reactive process.

Maintaining a constant speed or rate in climb or descent requires coordination between the elevator and the throttle. The climb and descent, once established, were acceptable, but the transitions usually deviated and required substantial corrections to converge on the appropriate trajectory. Landing was an outright disaster because the flaps and minimal airspeed radically changed the flight characteristics, reducing the margin for error. The target conditions were also the most complex. Flaps were not under machine control as an input, so they were already deflected as part of the initial conditions.

Turning via ailerons was not intuitive. In a car, the driver turns the steering wheel to the desired angle and holds it, returning to neutral to cancel the turn at the end. This relationship is therefore direct between the input and output. In an airplane, it is indirect: the ailerons change the bank, which causes the turn. If the wheel is held as in a car, the bank continues to increase and rolls the plane over. Having to neutralize the ailerons after establishing the bank surprised the students. The machine never figured it out consistently, usually due to an inadequate or excessive

bank angle. Thirty degrees is typical in a Cessna 172, with 45 degrees considered steep.

The bank diverts some of the lift perpendicularly away from gravity in order to force the turn, which results in a loss of altitude. Students realized that they required some additional elevator up for pitch to account for this loss. The machine tried, but it could not coordinate the amount well and generally increased in altitude or entered an unrecoverable spiral descent (known as a "graveyard spiral" when done by human pilots) (FAA 2011). A few students attempted to increase speed (which is aerodynamically valid because it also increases lift), but the lag in acceleration is too difficult to manage. The machine never came close to figuring out this relationship, although it tried.

Rudder usage was an utter failure. Initially both groups tried to turn the airplane with it, which appears deceptively intuitive because it indeed affects the vertical axis and initially appears to have the expected result. However, this approach is completely wrong. Its true purpose is to coordinate the nose-to-tail angle through a turn, in the same way the front wheel on a bicycle maintains the appropriate arc of travel for the amount of lean (bank), where critically the lean/bank *comes first*. Attempting to steer with the handle bars first would result in an upset at any appreciable speed. The only difference in mechanics between these two systems in where the vertical axis is located. On a bike, it is over the rear wheel, whereas on an airplane, it is usually over the main wings, as in Figure 8 (Phillips 2009).

Figure 8: Bicycle Versus Airplane Yaw Axes (Sketchup 2014)

The ball in the turn coordinator is the only instrument reflecting this coordination. It is based on centrifugal force, which is actually not even in the flight-dynamics model. Rather, the virtual instrument uses an ad hoc approach to derive a good approximation by calculating the turning arc based on the bank angle and appropriate subarc that corresponds to the nose-to-tail yaw angle based on the rudder deflection. This information was not accessible to the machine.

Worse is that neither group was even aware that the rudder played a role once they discarded it as an option for directly turning the airplane. The airplane appears to turn with or without rudder input, leaving both groups to disregard its value. Even real pilots are often sloppy with the rudder for the same reasons (Langewiesche 1990). Its aerodynamic effects, while subtle, are still substantial. Figure 9 demonstrates the difference between a coordinated turn with appropriate rudder (A) and ones where there is respectively not enough (B), called slipping, and too much (C), called skidding. On a bicycle, the awkward sideways force would be immediately noticeable and corrected, but in this type of airplane, it mostly affects the passengers in the back, not the pilot in the front, due to the position of the vertical axis, and can easily be ignored with no apparent consequence. This discovery was unexpected and warrants separate investigation.

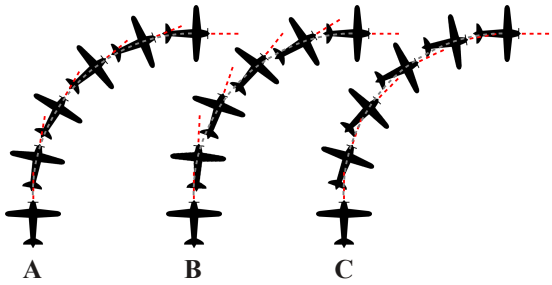


Figure 9: Normal, Slipping, and Skidding Turns

Finally, the bar in the turn coordinator registers rate of turn (normally not to exceed three degrees per second), which is the amount of arc covered in a fixed amount of time. Neither group associated the change in heading with the change in time. Rather, both groups treated the bar as a roll indicator apparently providing the same information as the attitude indicator, despite the depictions rarely agreeing.

Future Work

This plug-and-play architecture was designed for investigating machine-learning strategies, so immediate follow-on work will integrate others beyond the current simplistic one. Moreover, so far the system has considered only the lowest three AI processing levels (acquisition, transformation, and some fusion). Inference, reasoning, and prediction are where higher-level understanding and action occur (Russell and Norvig 2009). Experiments with navigation (both wide-area and local airport approach/departure operations), which the architecture already supports in great detail, offer ample opportunities (FAA 2007). Finally, at all levels, the expressiveness and objectivity of the introspection needs improvement.

Rudder coordination can stand as its own independent investigation. The fact that neither human nor machine could even recognize the situation adequately suggests that it involves many or all of these AI processing levels.

The flight-dynamics model needs to be more flexible in accommodating other test configurations. The current implementation involves significant trial and error to tune. Baseline performance is also difficult to establish, so it could benefit from calibration with real-world airplanes. It

also needs to accept input from a proportional joystick and pedals instead of the keyboard.

Conclusion

As a work in progress, this system has only begun to demonstrate its usefulness. Nevertheless, the flexibility of the plug-and-play modularization of the learning strategy clearly shows promise. The baseline strategy successful captured actions and learning processes of the student group. The de facto machine strategy, while hardly elegant in its application of sheer brute force, showed that it can indeed process many aspects of flight simulation with some semblance to reality. Replacing it with more advanced learning strategies should produce far better results. Finally, the introspective nature of the learning process demonstrated that it can provide valuable insight into how it operates, which was the primary goal of this work.

References

Allerton, D. 2009. *Principles of Flight Simulation*. Chippenham: Wiley.

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

Bourg, D. 2002. *Physics for Game Developers*. Sebastopol: O'Reilly.

Bourg, D. and Seemann, G. 2004. *AI for Game Developers*. Sebastopol: O'Reilly.

Cessna. www.cessna.com/single-engine/skyhawk. Last accessed 12 Feb. 2014.

Conway, D. 2012. *Machine Learning for Hackers*. Sebastopol: O'Reilly.

Dorn, D. 1989. Simulation Games: One More Tool on the Pedagogical Shelf. *Teaching Sociology* 17:1–18.

FAA. 2007. *Instrument Procedures Handbook, FAA-H-8261-1A*. San Bernadino: Skyhorse.

FAA. 2011. *Airplane Flying Handbook, FAA-H-8083-3A*. New York: Skyhorse.

FAA. 2012. *Instrument Flying Handbook, FAA-H-8083-15B*. Washington: ASA.

Guralnick, D. and Levy, C. Putting the Education into Educational Simulations: Pedagogical Structures, Guidance and Feedback. *International Journal of Advanced Corporate Learning* 2(1).

Harrington, P. 2012. *Machine Learning in Action*. Shelter Island: Manning.

Haykin, S. 1994. *Neural Networks: A Comprehensive Foundation*. Englewood Cliffs: Macmillan.

Haykin, S. 1999. *Neural Networks and Learning Machines*. Upper Saddle River: Pearson.

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

Jones, M. 2008. *Artificial Intelligence: A Systems Approach*. Hingham: Infinity Science.

Langewiesche, W. 1990. *Stick and Rudder: An Explanation of the Art of Flying.* McGraw-Hill.

Mitchell, T. 1997. *Machine Learning*. Boston: McGraw-Hill.

Napolitano, M. 2011. *Aircraft Dynamics: From Modeling to Simulation*. Hoboken: Wiley.

Phillips, W. 2009. *Mechanics of Flight*. Hoboken: Wiley.

Poli, R, Langdon, W., and McPhee, N. 2004. *A Field Guide to Genetic Programming*. Creative Commons.

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

Russell, S. and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. Upper Saddle River: Prentice Hall.

Sketchup, adapted from public-domain models on Google Sketchup: www.sketchup.com. Last accessed 12 Feb. 2014.

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

Tappan, D. 2009. A Pedagogy-Oriented Modeling-and-Simulation Environment for AI Scenarios. In Proceedings of WorldComp International Conference on Artificial Intelligence, Las Vegas, NV.

Tappan, D. 2013. Student-Friendly Java-Based Multiagent Event Handling. In Proceedings of Association for the Advancement of Artificial Intelligence, Bellevue, WA.

Appendix E: Curriculum Vitae