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.

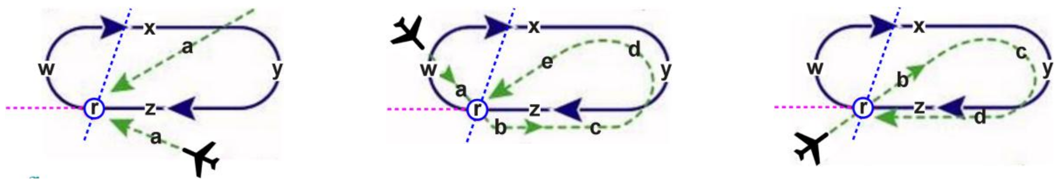
3.1 Model

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

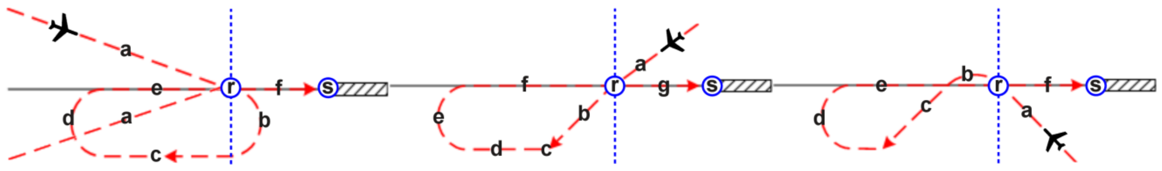
a fly to fix ***r***
b turn left to reciprocal course
c fly for one minute
d turn left 180 degrees
e fly to fix ***r***
w continue around ***x***, ***y***, ***z***, ***w***, ...

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.

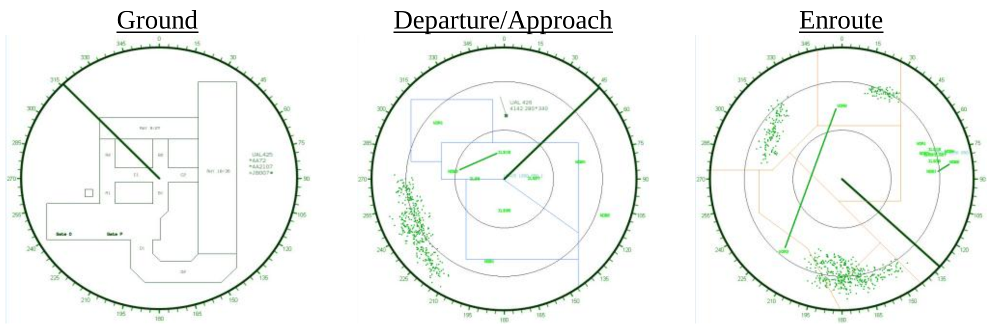


Figure 3: Radar Displays

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].

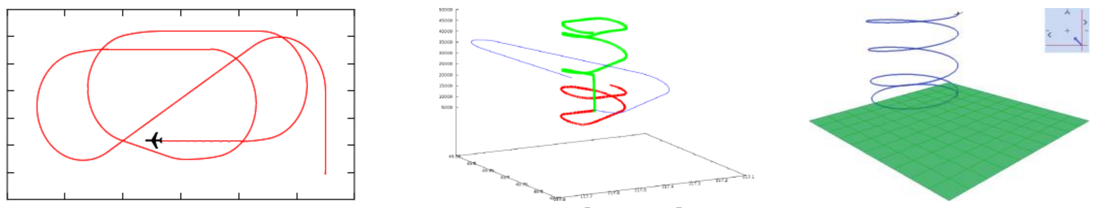


Figure 4: External Visualization

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.