Extending Training Capability Through The Use of Embedded Domain-Specific Languages in Training Devices

John Aughey
The Boeing Company
St. Louis, MO
john.h.aughey@boeing.com

ABSTRACT

Modern training devices are becoming increasingly multifaceted as training domains grow larger and the equipment within these domains increases in complexity. Instructors need the specific capabilities complex training devices provide, yet they need the flexibility to adapt these training devices to meet their ever evolving training curriculum. Through the use of domain-specific scripting languages, instructors can quickly and easily adapt their training system to meet changing requirements rather than modifying their training to cope with the limitations of the devices provided to them.

It is nearly impossible to determine all of the capabilities that a training system will need when the system is initially defined. Often, the customer specifies numerous requirements in an attempt to cover every conceivable training scenario. When requirements are defined to this level of specificity, the software is generally written to meet these specific training *requirements* rather than to provide an over-arching training *capability*. This paper will explore the use of scripting languages such as Ruby to embed domain-specific language capabilities into key areas of training systems in order to extend their training capabilities.

Domain-specific languages are also useful in defining requirements specifications. A domain-specific language (DSL) is a language designed to be used for a specific task or set of tasks in a particular field or domain, rather than for general-purpose programming. Requirements defined by a domain-specific language are more concise, more expressive, and easier to test saving both time and money.

This paper will explore where domain-specific scripting languages can be embedded into training solutions, how these domain-specific scripting languages can be leveraged by end users to adapt their training devices to focus on their mission, and where domain-specific scripting languages can be applied across multiple product platforms to form a more complete training solution.

ABOUT THE AUTHOR

John Aughey has worked as a software engineer at the Boeing Company for 10 years. John has developed software for Integrated Defense Systems, including visual systems, F/A-18 and F-15 simulator projects, and Internal Research and Development projects. John is a co-inventor on two visual-systems related patents. John received his Bachelors degree in Computer Science from Purdue University in 1996 and a Masters degree in Computer Science from Washington University in St. Louis in 2001.

Extending Training Capability Through The Use of Embedded Domain-Specific Languages in Training Devices

John Aughey
The Boeing Company
St. Louis, MO
john.h.aughey@boeing.com

INTRODUCTION

Development of complex training systems with hard to define requirements and short schedules can be significantly improved with the use of domain-specific languages and embedded scripting. The current requirements process used in software development is cumbersome, time consuming and prone to error. These issues are exacerbated in the defense industry where systems are huge and complex, deadlines are short, and the incorporated technologies are often new and unproven. This process could be streamlined and the quality and flexibility of the products greatly enhanced by changing both the products and the processes in a few simple but key ways. First, as a matter of design philosophy, including scripting interfaces in programs should be an integral part of product development. They allow for automated testing, dynamic user interfaces, and rapid prototyping. These "windows" would allow short programs to be added later, greatly extending the capabilities of the training devices. Second, software engineers would work with the customer from the inception of the contract, generating the requirements together, and writing them in a domain-specific language that is both understandable by the customer and able to be automatically tested at every stage of development. This language can be used by the customer throughout the development process as well as after delivery to alter or expand the capabilities of the device as the customer wishes.

DEFINITION OF THE PROBLEM

Military training devices can be very complex. Often the customer needs training capability immediately, resulting in a very short development to production schedule. The technology to be trained on may be new, which means both the customer and the contractor may have little experience on how to best train. Additionally, the training system as a whole may be comprised of many pieces provided by different sub-contractors.

Given this complexity, the process of defining requirements for these systems is difficult at best. The requirements document is a part of a legal contractual document. Each requirement must be written specifically enough to be tested and verified, and completely enough to be interpreted and built by the contractor. The test for a requirement must be rendered in pass/fail format in order to determine whether or not the requirement was fulfilled. Traditionally, all requirements for a large project are defined at the beginning of the project.

For the customer, defining the requirements is a difficult process because the systems they need to define may be too complex to completely grasp, even with a large number of subject-matter experts contributing to the process. Assuming the experts have a firm grasp on the requirements, it still can be difficult to communicate those requirements effectively in a comprehensive requirements document that must be generated quickly. Even if the customer is not directly involved with the initial requirements definition process, implied requirements must be formed into a test procedure by the contractor and then approved and verified by the customer.

THE CURRENT REQUIREMENTS PROCESS

Writing the Requirements

At some point in the requirements gathering process, someone is tasked with putting the requirement down on paper. We will define two types of requirements. The first type of requirement is one we will call a "performance requirement." A performance requirement is a requirement that defines a verifiable behavior of the system. An example is, "At 30,000 feet level-flight, the maximum speed of the aircraft is 900 knots" or "The gear shall not extend if the aircraft is traveling over 300 knots." For a given contract, there could be thousands of these requirements. The second type of requirement we will call "conceptual requirements." Generally these types of requirements revolve around the user interaction with the device. Conceptual requirements often read, "When the instructor clicks..... the user interface shall show....." or "the instructor shall be able to command the threat aircraft to....."

Performance requirements are relatively easy to gather and not too difficult to write. Often they are copied from other documents that define the behavior of the device being simulated. There can be a large number of requirements to write however, and English sentences are not often the best way to communicate the specification. Using a domain-specific language when generating the requirements could help eliminate some of these ambiguities.

Conceptual requirements are extremely difficult to write. The writer needs to conceptualize how users of a non-existent system might interact with it. Past experience with other products can aid in defining how a similar product should look and behave, but it can be highly imprecise. A typical training device generally has many different types of users including software engineers, test engineers, maintenance personnel, operators, instructors, and students. Each of these users has their own needs, resulting in potentially conflicting requirements. Additionally, a subject matter expert may conceptualize a need for a feature that in practice turns out to not be useful. On the flip-side, there could be a entire set of use-cases that are never discovered until the device becomes operational.

The Contractor Interprets the Requirements

The contractor is then tasked with interpreting the complex requirements as specified. The performance requirements can be relatively easy to interpret and implement, but several complications are possible. Given the large number of these types of requirements, errors may be included and a misplaced comma or the transposition of and/or can change the entire meaning of the specification. Additionally, the performance requirements might be written either ambiguously or so specifically that they have the complexity of a legal document.

The Test Engineer Reinterprets the Requirements

Faced with verifying proper operation of the device, the test engineer must first re-interpret the requirement as written. Then the test engineer must write a step-by-step procedure for each requirement that properly validates the correct behavior. A test and results document generated from the requirements specifications can be many times the size of the original requirements document because it must include the requirement, a procedure to verify the requirement, and a list of expected results. Once written, the tests then must be verified one-by-one on the actual device. The verification process alone can take weeks or months given a moderately complex trainer.

The End User Re-Tests and Accepts the Device

Once delivered, the customer is now faced with the task of accepting the device. The test process is run again with the customer involved. Eventually, after some re-work, the contractor proves that the requirements are met to the customer's satisfaction and the device is put into use. After a short period of time problems often arise that may have not been conceived of by those who designed the original set of tests. Additionally, the delivered system may meet the "training requirements", but the interfaces provided may not give the users of the system all of the capabilities that might helpful. Non-optimal software is an interesting dilemma because while the system is capable of training a student to perform a given task, additional or modified capabilities, not easy foreseeable before the device is in use, may give the instructors an edge that allows them to provide better or more complete training. Obviously, better training is not an insignificant benefit.

Problem Summary

Requirements are hard to define. Requirements are hard to write clearly. Requirements are hard to interpret accurately. Requirements may not convey intent well. Requirements, as currently used in the development process, can produce a less than optimal product.

PROPOSED SOLUTION

The proposed solution is to define requirements and write a verification of those requirements in a domain-specific language. Creating the domain-specific language that will be used is a collaborative process between the customer, software engineers, and test engineers. Over time, the language in which the requirements are defined will evolve and take shape, providing a syntax that the customer can easily write, the engineers can easily interpret, and the test engineers can easily verify. This language naturally becomes an integral part of the training system that utilizes a built-in scripting environment designed to interpret this language.

DEFINITION OF DOMAIN-SPECIFIC LANGUAGE

A domain-specific language (DSL) is a programming language designed to be useful for a specific set of

tasks. A DSL is a language that is created to concisely solve problems in a particular domain. DSLs and scripting languages often work in harmony with each other. Many times, a domain-specific language is translated to, or written specifically in a scripting language designed to be run in a larger application.

Domain-specific languages are often less comprehensive than traditional programming languages, and are more expressive in the domain they are describing. The point of creating a DSL in this context is to provide a language that the experts writing requirements and tests can understand and validate. If a domain-specific language is an integral part of the software product, then it extends well into a language that the end users can use to create their own program extensions.

Example of domain-specific language

The idea of using an embedded domain-specific scripting language in software programs is not new. The Flash animations prevalent in internet advertising use the ActionScript language to control their behavior. Internet browsers such as Internet Explorer, FireFox, Opera, and others use the Javascript language to provide some of the dynamic behavior used by GMail™, Google™ Maps, Expedia.com™, and uncountable other web sites. Maya, the 3-D graphics tool used to create the digital effects in most action movies, uses a language called MEL. All of these embedded languages enable the end-user to write extensions that allow the software applications to perform tasks that were not part of their originally specified task. Internet browsers were never specifically designed to be email programs, or satellite map viewers, or airline reservation agents; but the scripting language that is an integral part of the software enables this capability. New applications are created every day that allow these browsers to do what they have never done before.

Excel is a perfect example of a software program that uses a domain-specific language that allows it to perform tasks the original designers did not foresee. The domain is very simple, comprising a rectangular grid of cells. The language is equally simple: each cell can contain a formula that can use results of other cells. This simple concept allows a person with very little programming experience to work with data ranging from sorting a list of people to generating budget reports to computing Fourier transformations of digital signal data. It is truly remarkable that a single application can be used throughout such a wide range of use-domains.

All uses of Excel could be performed using special purpose software programs, and some organizations use these. To produce a budget report, for example, all of the data could be entered into a custom written program that computes the necessary values and generates a final report. Any change in the input data, business logic, or the required output, however, would require a very lengthy software change process. New requirements would have to be written and given to the software developer. The developer would have to implement and test the changes and deploy the newly created program to the end users. By the time the new version is in place, it may have new bugs, the developer may not have interpreted the requested change correctly, there might be additional requirements, or the new data might not be useful anymore.

Extendable programs like Excel are much more widely used than custom programs, and it is easy to see why. A competent manager can easily manipulate an Excel document to generate the results and reports they require. Because this capability is provided to the end user through an easy to understand domain-specific language, changes can be made in a matter of minutes rather than requiring the involvement of the original software developer.

When you open up Excel, you are presented with a blank palette of cells that give you the capability to do almost anything. Similarly, modern training devices need to provide the users of those devices with a training *palette* that allows them to use and extend that device in ways the original designers had not conceived. Embedding domain-specific scripting languages into training devices that allows the program to be extended through the use of scripts is one way of empowering end users.

Scripting interfaces

A scripting interface is a bridge between a compiled program written in a language such as C, FORTRAN, or Ada, and a dynamic scripting language such as Perl, Ruby, or Python. The application programmer, working in a language such as C++, includes “hooks” where the scripting language can control the application. This combination is very powerful because you get the operational speed of a compiled language with the flexibility of a scripting language.

For example, most of Excel is written in a compiled language such as C++. When evaluating a spreadsheet, the compiled software defers to a scripting language to evaluate the small formula written in each cell.

The scripting languages used to write programs for these interfaces allow the end users to easily modify the "little programs" and see the change immediately without the time, commitment, and complexity of implementing changes with compiled languages such as C or C++. Using these scripting languages adds flexibility, facilitating making changes in the field quickly and simply.

HOW DOMAIN-SPECIFIC LANGUAGES IMPROVE THE PROCESS

Utilizing domain-specific languages gives the customer flexibility and control. A domain-specific language can be used to streamline the requirements writing process.

Performance Requirements

As mentioned above, some performance requirements are very easy to define. But a typical aircraft simulation could have hundreds or thousands of these basic performance requirements. Throughout the development cycle, the specifications may change to reflect newly discovered data. Testing all of these requirements is a long and tedious process. We will explore how using a domain-specific language can aid in both defining these requirements as well as testing these requirements.

Performance Example

A requirement such as "The maximum airspeed of level-flight at 30,000 feet at military power is 550 knots" is a relatively concise and easily verifiable requirement. A more concise way of stating this requirement might be:

```
initial conditions altitude = 30000 feet,
                   air speed = 540 knots,
                   throttle = full
after 20 seconds
validate airspeed < 555 knots
validate airspeed > 545 knots
```

Figure 1. DSL restatement of requirement

While the syntax may not be a pure English sentence, it is easy to see how the six lines above say the same thing as the sentence version. Statements such as these begin to form a domain-specific language that can be used and extended to define other requirements. As more requirements are defined, patterns will begin emerge in the language that can be reused.

The statements written in a domain-specific language actually say more than the original requirement. The initial conditions in figure 1 set the aircraft's speed slightly below the maximum for this altitude. This allows the aircraft to accelerate to the maximum speed. After 20 seconds, the aircraft should have accelerated to within 5 knots of the specified maximum speed. This is a more accurate representation of how the simulation should behave.

Writing a requirement in this domain-specific language serves several purposes. A requirement written this way can be read by a computer and verified. Now that the requirements are defined in this language that is easy to read and easy to write, requirements specified in this language can be used directly in unit and functional testing. With just a little bit of work, an automated test program could take an entire requirements document written in this form and validate that the simulation performs as specified. Furthermore, requirements written in some type of domain-specific language can be changed and the changes will drop directly into the automated test program and can be re-validated automatically.

We could also re-factor this test to facilitate the writing of requirements that have a similar form. An aircraft simulation will likely have many requirements like this for different conditions such as altitude, weapons load, and weather. In the language seen in figure 2, we specified a generic maximum airspeed test and reused that testing framework to test a wide range of similar requirements.

```
Maximum airspeed test
  initial conditions
     altitude = ALTITUDE feet
     air speed = MAXIMUM - 10 knots
     throttle = full
  after 20 seconds
  validate airspeed is within 5 knots of maximum

At 30000 feet validate maximum airspeed is 550 knots
At 20000 feet validate maximum airspeed is 500 knots
```

Figure 2. Re-factored airspeed test

Related Requirements Research

The Software Cost Reduction (SCR) requirement model is a method of defining requirements using tables. The SCR model was formulated in the late 1970s to specify the requirements of the Operation Flight Program of the A-7 aircraft. The SCR model is a formal model that represents a state-machine where the initial state of the device is defined, an action is performed, and the expected resulting state tested.

The requirements definition approach described in this paper, in contrast to the SCR model, defines

requirements in the form of a test written in a domain-specific language and continues the use of that language into an extendable scripting environment.

Conceptual Requirements

On the other side of the spectrum, there are some requirements that are extremely difficult to define. Unfortunately, these requirements often concern the most visible part of the device, the user interface. There are many reasons why these requirements are difficult to specify.

The end customer may not have a complete understanding of how they will need to train using a given training device. This may be because it is a new technology that has never been trained on before. The device may be so complex that one person or a group of people sitting in a room may not be able to grasp all of the potential use-cases of the device. The result is that requirements might be either grossly over-defined or grossly under-defined, or even a combination of the two. One person might spend 50 pages specifying the login verification requirements while another person spends one page defining the entire mission planning requirements. The truth is that requirements are often not written by the actual person who will end up using the system, so capabilities that might be beneficial to the actual end-user of the system may never become known until after delivery.

Even if the actual end-user is writing the requirements, they may not know precisely how they will use the device. This causes two problems. 1. Requirements may be written for capabilities that are never actually used and 2. Unknown usage cases are not specified. Since it is difficult to accurately define a system that has never been built, users need to hypothesize how they might use a system. Sometimes these hypothesizes will be wrong. For example, while brainstorming, it might seem like a good idea to have the capability to store training performance records in a relational database that is cross-referenced to field data in order to determine weak-points in training. But once deployed, the customer may discover that the data captured is not really useful in practice. As a result, the customer may have spent an extra million dollars on development costs, complexity, testing, training and computational hardware, for a capability they never use.

The unknown usage cases that are not specified may be even worse because it is generally quite costly to go back to the contractor and add functionality to an existing system that was not part of the original requirements. Often this is done after delivery, requiring a new contract to be setup, new requirements to be gathered and written, and a new product to be delivered and tested. The original developers of the system may not be available anymore and the new set of requirements may also be over or under-specified.

While supplying the end user with a domain-specific scripting language may not solve all of these problems, by building flexibility into the final product, it is much more likely that the customer will end up with a device that meets their current and future needs.

EXTENDABLE SOFTWARE

As noted earlier, user interface requirements are particularly difficult to write clearly and interpret accurately. Descriptions of user interface use-cases are often vague, requiring a long narrative description with diagrams and circles and arrows and a paragraph on the back of each one explaining what each one is. A test engineer looking at these types of requirements will often be unsure how to verify correctness. When developers get these requirements they will have to interpret them in their own way.

Product Lifecycle Example

As an example, we will go through the process of writing a basic requirement and see how it might evolve through the product life-cycle. Suppose the general requirement we are trying to define is that the trainer needs to be able to simulate failures. For brevity, we will look at an aircrew trainer with the requirements that the trainer needs to be able to simulate an engine fire, loss of rudder control, COMM failure, and HUD failure. These requirements might be written as:

- The operator software must provide a way to induce the following failure conditions:
    - Engine fire
    - Loss of rudder control
    - COMM failure
    - HUD failure

Let's look at how these requirements were determined. Someone on the customer side likely looked at their current curriculum and saw the need to train pilots for what to do in different failure conditions. For example, the loss of rudder control might have been needed for a lesson that trains the pilot what to do when they lose rudder control during takeoff.

One implementation of this requirement might provide the operator a menu with a list of these failures, and a

button to induce the selected failure. With this interface, the basic requirement is fulfilled. It does not make for a very extendable trainer, however. A failure must be instigated by the operator. For the above lesson, the operator needs to watch the student fly, and hit the failure button at the appropriate point during the takeoff. This requires the operator to pay attention to the *device* during that point in the lesson instead of focusing on the *student*. For takeoff, this may not be a big deal, but suppose an engine fire needs to be simulated once the aircraft reaches 20,000 feet. Now a highly decorated operator is spending his or her time watching the altitude indicator for several minutes in order to press a fail button when they reach 20,000 feet.

When the time comes to update the trainer with the latest operational flight program, the customer adds on a requirement to define a conditional value into the failure menu to avoid needing to monitor the altitude indicator manually. Other failures might need different conditionals for altitude, airspeed, orientation, flight time, and so on. All of these different possible conditions are added to the requirement in lengthy detail.

Now, in order to select a failure, the trainer will provide the operator with conditional controls that will let him/her specify when these conditions should occur. The conditional controls are numerous and complex since the requirement needs many conditional possibilities for many different failures. But again, it meets the specified requirement, and reduces the operator's workload.

In practice, however, the operator soon discovers the failure event triggers too fast, because the instant the altitude hits 20,000 feet, the engine fire failure triggers. The training need is really engine failure at level flight, so the failure really needs to happen as a result of two conditions being met, a given altitude and a given rate of climb.

So this new problem gets tossed back to the contractor as a contract modification and they go to work on the new interface. Now the already complex conditional window becomes even more complex with multiple levels of conditionals and a very difficult to use interface. It is really too late in the game to re-factor the interface, so this new requirement is just layered on top.

Even worse is the difficulty of defining this new requirement. To cover all the bases, the new requirement might now say, "All failure conditions must have an operator defined conditional statement. Once the entire conditional statement is satisfied, the failure will occur. At least 10 different conditional statements must be definable by the operator for each failure condition with the option to chain the statements together with AND or OR logic operators. At least 16 failure conditions can be ready to be triggered at any point in the training scenario." The difficulty of testing a requirement like this – in pass/fail format no less – is apparent.

The result of this latest round of software changes is a bloated and complex user interface that is difficult to use and even more difficult to test. There are likely errors due to the sheer complexity of the interface and the difficulty of adequately testing it. Furthermore, a full test of this new interface is time-consuming and expensive.

DSL TO THE RESCUE

So in this hypothetical situation, we went through three software revisions to come up with a difficult to use interface that probably does not work right some of the time. Now look at how implementing an end-user modifiable domain-specific scripting language could have improved our situation.

The domain-specific scripting language will be an inherent feature of the device. Without going into implementation details, we will assume that the end user can write small scripts and select which scripts will execute during a training lesson. We will look at example scripts throughout this explanation.

With the knowledge that a scripting interface will be provided in the user interface, our failure conditions requirement can be specified very concisely:

- Expose the following failure triggers to the scripting language
    - Engine fire
    - Loss of rudder control
    - COMM failure
    - HUD failure

We can even use our comfortable domain-specific language to specific and test this requirement seen in Figure 3.

```
Validate engine_fire failure exists
Validate loss_of_rudder_control failure exists
Validate comm_failure exists
Validate hud_failure exists
```

Figure 3: DSL to verify existance of triggers

The operator user interface might still provide a basic menu for inducing a failure condition manually. But the menu action can also use the exact same scripting interface to cause the failure to happen. In addition to being very easy to implement, it has the added benefit of being easy to perform a functional test since the menu action would cause the same thing to happen as if a script triggered the failure. The C++ code that is called as a result selecting a failure from the list might look as simple as Figure 4.

```
void onEngineFireSelected() {
  DSL.evaluate("Failures.trigger :engine_fire");
}
```

Figure 4: Usage of DSL inside C++

So if we follow our same sequence of events in the previous scenario, the operator initially uses the software containing the supplied list of failures and triggers the failures manually. Quickly, the operator grows tired of triggering the failures manually, so he/she writes a small script to make his/her life easier seen in Figure 5.

```
description "Fail the rudder 10 seconds after
wheels-up"
  trigger :action => { Failures.trigger :rudder },
          :when => { wheels_up_time > 10 }
```

Figure 5: Failure DSL

This script is self documenting and concise. As a result of writing this script, the item "Fail rudder on takeoff" appears as an option on the list of possible scriptable actions and uses the description defined in the script to provide more information to the operator. During the lesson, the conditional statement defined in the :when clause is continuously evaluated. Once the conditional evaluates to true, the action is performed. In this case, a rudder failure is triggered 10 seconds after wheels up.

The engine fire script might be as simple in Figure 6.

```
description "Trigger an engine fire once the
aircraft reaches 20000 feet"
  trigger :action => { Failures.trigger
:engine_fire_left },
          :when => { altitude > 20000.feet
and rate_of_climb < 10.feet.per.second }
```

Figure 6: Modified DSL procedure

The last line of text was added once they realized that the engine fire was being triggered too soon. Again, this item "Engine fire at 20000 feet" appears as an option on the list of possible scriptable actions.

Functionality provided by scripting environment

All of this functionality was provided by the very basic requirement that the failures be exposed to the DSL. The user interface would have been designed from the beginning with the knowledge that the scripting language would provide a large portion of the decision-making functionality. The customer could plan on extending the base simulator by writing specific use-case scripts, providing functionality not originally conceived.

RAPID PROTOTYPING

Having the capability to do rapid prototyping is another side benefit to having a scripting language embedded in software programs. In software, rapid prototyping means being able to quickly piece together software programs to form a working demonstration. This might be done to illustrate a design concept or gather user feedback.

For one application, Boeing needed to prototype what a flight trainer might look like with a commercial game simulator engine. The commercial simulator provided all of the flight functions and visuals. We needed to quickly integrate this system with our external hardware switches and controls. The commercial simulator provided a simple scripting interface using the LUA language. Using this language, the author was able to quickly write a script to connect to an external computer over TCP/IP. The external computer supplied the switch positions and control stick positions to the simulator over this connection. In just a day, our hardware controls were "flying" the commercial simulator in a way that was not originally conceived by the developers of the simulator.

COST SAVINGS

The savings, in both financial terms and time, from theses techniques described can be huge. The initial length of the requirements definition phase may be the same, but it is extremely beneficial to both the customer and the contractor to have the test defined as part of the requirement in a domain-specific language. These tests apply directly to development and integration testing, and can be used for final acceptance testing too, dramatically reducing time usually spent restructuring the requirements into pass/fail tests and manually evaluating each test.

One of the biggest time saving benefits comes from having an automated test suite from the inception of the project. If the requirements are defined as unit or functional tests, then development methodologies such as Test-Driven Development can be used, if practical. With an automated test suite, especially one that is generated directly from the requirements, the entire system can be tested for correctness at any point in the development cycle. Automated testing also provides a good metric for the overall health of the product. Quite often, many parts of the software are dependent on each other so fixing one bug may create or expose a bug in another part of the software. With automated testing, every test can be run in a relatively short period of time by a script. Automated testing allows the entire system to be validated after every change, greatly reducing "surprises" at the end of the development process.

Leveraging automated testing can be especially beneficial to customers and developers in the defense industry. Quite often, developers have limited access to the system as a whole during development. Sometimes this is because software is developed for hardware that does not yet exist because of manufacturing schedules. Sometimes there are a limited number of devices to test on so time on the simulator may be limited. Time, budget, schedule, and resources all play a role in how the system is tested. For example, without automated testing, a developer might make software fixes that changes the way the aircraft behaves at high-altitude. The developer signs up for trainer time to test the change to make sure it performs correctly at high altitudes. Unbeknown to the developer, the altitude change also affected how the flaps perform at low altitudes. The developer uses precious trainer time to test the altitude change, but never discovers the side effect of how the change affected the flaps. This effect may not be discovered until final testing, at which point the context in which the problem arose is gone. With automated testing in place, the low altitude test would have been run along with the high altitude test and would have likely caught the error. Furthermore, the automated test may not have needed the actual physical trainer at all.

The cost savings of having an extendable user interface enabled by a scripting language is immeasurable. Think about an internet browser. One application enables a vast range of possible uses. Designed correctly, a user interface for a training device with an embedded scripting language could provide very similar extendable functionality.

An extendable system reduces over-specification and puts the future of the training device into the hands of the user, dramatically increasing the likelihood that the device will meet the customer's ultimate needs. If, at some point in the future, the end-user needs to extend their device to do something that it was not originally designed to do, a scripting interface can give them that capability. Using this technology enables companies to focus on new development rather than maintenance of older products. And customers who are satisfied with the capabilities and flexibility of their products are more likely to be interested in taking a chance on the next generation of devices.

REFERENCES

Andrew Hunt & David Thomas (2000). *The Pragmatic Programmer*, Addison Wesley.

Dave Thomas & David Heinemeier Hansson (2005). *Agile Web Development with Rails*, The Pragmatic Bookshelf.

Venkat Subramaniam & Andy Hunt (2006). *Practices of an Agile Developer*, The Pragmatic Bookshelf.

Jared Richardson & William Gwaltney, Jr. (2005). *Ship It!*, The Pragmatic Bookshelf.

Wikipedia, (2006). *Topics, Software_Prototyping, Automated_Testing, Test_driven_development.* Retrieved June 1, 2006, from http://en.wikipedia.org

IBM, (2006). Crossing borders: *Domain-specific languages in Active Record and Java programming*. Retrieved June 1, 2006, from http://www-128.ibm.com/developerworks/java/library/j-cb04046.html

Constance Heitmeyer NRL. (1998). Using the SCR Toolset to Specify Software Requirements. *Proceedings, Second IEEE Workshop on Industrial Strength Formal Specifications Techniques, (WIFT'98).*

David A Black. (2006). *Ruby For Rails*, Manning.