46th AIAA Aerospace Sciences Meeting and Exhibit
7 - 10 January 2008, Reno, Nevada

AIAA 2008-496

Introduction of Software Development Practices into Aerospace Engineering Curriculum

David D. Marshall[1] and Eric A. Mehiel[2]
California Polytechnic State University, San Luis Obispo, CA, 93407-0352

This paper will discuss the attempts to incorporate software development practices into the aerospace engineering curriculum in order to improve the computer programming capabilities of the students. The main focus is on techniques to integrate functional decomposition, unit level testing, system integration and testing, and verification and validation processes without significantly increasing the workload on the students. The approach taken is an integrated approach where the required information needed for the testing and validation are integrated into the course content via in-class examples and homework problems. This same approach was taken for the other software development skills. This has been integrated into sophomore, junior, senior and graduate level classes. The results of this effort have been an overall improvement in the quality of software that the average student submits and an increase in the complexity of the software that the students write.

I. Introduction

The use of software development practices is not a common topic in most aerospace engineering curricula. As computers become a more important educational tool, students are being asked to write more complex software applications, and the need is increasing for the students to use some software development practices in order to effectively write, debug, test and validate the software that they are producing. In particular the curriculum at California Polytechnic Sate University, Cal Poly, has significantly increased the usage of computer programming in a number of academic areas to educate the students on some of the more advanced topics that are more easily presented to the students via computer programs. The programming language used by the Aerospace Engineering Department is MATLAB.

As with most academic programs there is very little room in the Aerospace Engineering Department's curriculum to add a specific class on software development practices. In addition this class would need to be introduced at the beginning of the student's undergraduate career so that these practices could be put in place and refined as the students progressed through their schooling. However, a class devoted to software development practices without the students knowing any specific aerospace engineering concepts would require them to learn the software development practices in a very abstract context. Thus, it was important to introduce the students to these topics early in their undergraduate classes, but within the existing curriculum. This also provides the added benefit of having the students repeatedly exposed to the same software development topics throughout their academic career further reinforcing the concepts as they apply to aerospace engineering.

There were four software development skills that formed the primary focus of this effort: functional decomposition, unit level testing, system integration and testing, and software verification and validation. While there are other important software development concepts that appear to be growing in popularity such as object oriented methodologies, configuration management of software development artifacts, and real-time software systems; the four skills were determined to be the most universal and accessible for the students. As more feedback is obtained from industry and fellow educators, the skill set may evolve to include other concepts. The most probable candidate for inclusion is object oriented methodologies since the increased use of object oriented programming languages such as C++, Java and Python in scientific applications means that current aerospace

[1] Assistant Professor, Aerospace Engineering Department, Senior Member.
[2] Assistant Professor, Aerospace Engineering Department, Member.

engineering students will have a higher likelihood of using/modifying codes written using object oriented methodologies.

As previously mentioned, this effort was performed by integrating the four software development skills into the existing aerospace engineering curriculum throughout the years. Starting from their sophomore year, Cal Poly aerospace engineering students will take a number of classes that introduce and/or use the software development skills in order to present the class material. The sophomore year tasks are more of an introduction to functional decomposition and unit level testing with a number of opportunities to practice these skills on very simple tasks. The junior year tasks introduce more sophisticated and domain specific examples of functional decomposition and unit level testing as well as introduce verification and validation processes. The senior year tasks further develop the three previously introduced tasks as well as introduce system integration and testing. The graduate classes utilize all four of these concepts to some degree and introduces a new challenge because graduate students from other universities typically do not have the depth of background as the students from Cal Poly's undergraduate program. This issue is usually a short-lived problem as these external graduate students quickly acquire a working knowledge of these concepts and use them adequately.

II. Software Development Skills

A. Functional Decomposition

Functional decomposition, also referred to as procedural decomposition,[1] is the process of breaking-down a complex task in terms of smaller less complex tasks. This allows a complicated task to be programmed by using smaller more manageable tasks. At first this might result in very broad statements of a task, such as “Need to get location,” but with iterations of this process can be refined to clarify what this means in a functional sense, e.g., query object for Cartesian coordinate position vector.

As a simple example, the task of determining how much time a constant velocity object has left before it reaches it's final destination can be decomposed into the following steps:

1. Determine the current position and speed of object
2. Determine the final destination location
3. Calculate the distance from the current position to final position
4. Calculate the speed of the object
5. Use the object speed and distance to destination to determine the amount of time left
6. Return the time left to reach destination

What is expected is that each of these individual steps are easier to solve. More importantly some of these steps might already exist in either a standard library that is being used or as a step in another system task that has already been developed.

With the introduction of functional decomposition there is a natural introduction of software reuse and the discussion about how to develop and maintain the student's own library of useful functions. These are extremely powerful concepts that will make the student's life easier with only a slight increase in effort to develop the functions that will exist in their library. From the start of their learning of functional decomposition as sophomores to their senior (and graduate) level classes, the students can develop a sophisticated library of functions that the can use in their future classes. As an example, most of the senior and graduate level assignments that require significant coding can be composed of a significant amount of functions from their personal software library.

For most programming tasks the functional decomposition process is iterative. The first level of decomposition is typically the coarsest of steps that describe the function. Using the above example, that task might be part of a GPS system that is providing feedback to the user about their trip. One step in that process would be to determine how long until the trip is over. Also, the distance calculation step could be further decomposed into the following sub-steps:

3.1. Subtract the final destination location coordinates (assuming they are Cartesian) from the current location coordinates
3.2. Find the magnitude of the difference vector from previous step
3.3. Return the magnitude as the distance between the current location and final destination

Each of these steps are at such a level of complexity that a new function does not need to be created for any of the steps, and the programming can be done within this function. Note that finding the magnitude of the difference

vector will require a square root function which typically exists in a standard library. Figure 1 shows a graphical representation of the functional decomposition for this example.

It is surprising to observe the students lack of familiarity with the concept of domain decomposition. Attempting to get the students to logically proceed from a starting set of givens (such as a destination with position and an object with position and velocity) and proceed to an end condition (such as the time to reach destination) is a difficult task. Many students do not possess the skill of logically serializing even moderately complicated tasks. It takes a surprising amount of time to introduce this concept to the students with a number of simple examples and performing this task with them in class. It would be reasonable to think that with as much science and mathematics that the students see before entering the aerospace engineering curriculum that they would have become accustomed to this logic process, but for the most part, the students do not possess this critical skill.

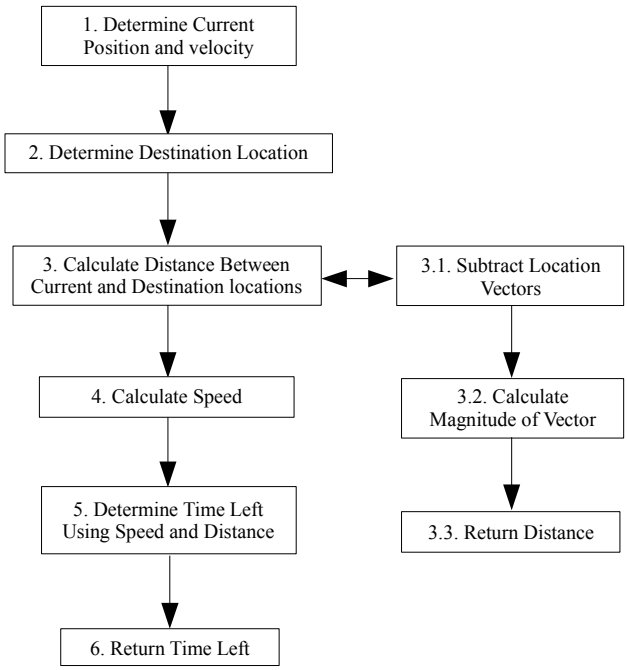

Figure 1: Functional Decomposition for Example. *This shows the functional decomposition for the task of determining how long until a constant velocity object reaches a specified destination.*

B. Unit Level Testing

Unit level testing[2] is the process of testing the most basic parts of the software system, the software unit. The software units typically come from the end result of the functional decomposition. Such that each function that has been identified in the functional decomposition phase has an associated unit level test to go with it. These tests should be retained throughout the lifetime of the software unit so that regression testing can be performed during the development life-cycle. Regression testing is the testing of a software component after the initial software component has been written and has been in use. It typically occurs throughout the software development process and in some well run software development operations it occurs in a nearly continuously.[3] It is used to catch changes in the software system that unexpectedly alter the behavior of the system or its components.

From the above example, shown in Figure 1, it is clear that Step 3 – Calculate Distance Between Current and Destination Locations – warrants a unit level test since its task is complicated enough to have another level of decomposition and therefore needs to be a separate function. To perform the unit level testing of this software unit, a number of current and destination locations can be calculated using the software unit. These results can be compared to independent calculations of the distances (perhaps hand calculated or another computer program, that does not share code with this unit).

Even this simple example of a unit level test provides some surprise complexities. One is that since this calculation is most likely being performed using finite precision mathematical operations, round-off errors are going to be introduced into the final result (especially since there is a square root operation in the calculations). This makes comparisons of the results a non-trivial task (especially if the students are trying to automate their unit level tests). Another added complexity is that different operating systems and hardware architectures can yield slightly different calculations for the same inputs and operations. This mainly occurs because of either a floating point format difference is the display of the data or the hardware does not adhering to the IEEE specification for floating point calculations.[4] These are issues that are not specific to scientific computing but are more common in scientific computing, and addressing these issues can complicate the process of developing a unit level test.

A few topics can evolve from the presentation of unit level testing. One is the idea of testing automation so that a test can be without user intervention and can be run periodically with either a successful or unsuccessful result

automatically recorded. This leads to a discussion of the issues mentioned in the previous paragraph related to computer representation of floating point numbers, approximations associated with these representations, and round-off errors that can propagate through the calculations. While all of these topics are very important to a complete understanding of how computations are performed in the computer, this topic is not suitable for most students as this level of detail goes beyond their interest in the subject.

The other topic that naturally evolve from this presentation of unit level testing is the concept of regression testing. As mentioned above regression testing is the testing of software components that have been written and tested. The objective of regression testing is to detect changes in the software components behavior due to a change in the software system. These are typically unexpected changes, and the regression testing of software components helps to identify these changes soon after the offending change has occurred. This idea is presented to the students as a discussion of the library of functions that they are accumulating. The students are asked to justify why their functions written last year should still work today. Potential issues such as changes to other parts of their software library and changes in functionality associated with an upgrade to their language environment are raised and discussed. This naturally leads to the topic of performing regression tests to ensure their existing code works. When they mention that they might have a lot of functions that need to be tested, then the discussion can turn to testing automation, mentioned above.

C. System Integration and Testing

System integration and testing[2] is the process of combining software units into more complicated software systems. The software systems typically come form the higher level domain decompositions that were developed in the software development process. This usually results from the integration of a number of software components that the student has written along with some existing functionality from some software libraries. This concept is beyond the software unit where the is a relatively simple function with a relatively well-defined result. It involves the development and testing of a complicated software system.

Returning to the simple example of calculating the time left to reach a destination location, system integration and testing would most naturally occur during the use of this function in a larger software system. As an example, a GPS system that displayed the time to destination as one of its functionalities would need to perform an system level test on the integration of the function with the larger software system.

In this presentation of software development practices, system integration and testing is one of the last topics presented since it is the assembly of simpler software units that the students have written. This is opposite to the traditional mode where the students are given the specifications (either explicitly or implicitly) of a software system that they need to write, and they are left to their own devices to develop and test it. Little or no attention is paid to the steps that are involved in increasing the likelihood of success such as functional decomposition, unit level testing, and system integration and testing. When the students get to this point in their education, the hope is that they have much higher success rates in writing and testing complicating software programs that they are being asked to write.

D. Software Verification and Validation

Software verification and validation is the process of verifying that the software system performs the specified actions as expected (validation) and that the software system calculates the correct results (verification).[5] These two topics are typically confused by the students, but their definitions are unique and perform distinctly separate tasks. The process of validation is the process of ensuring that for a given input an expected output is obtained. The process of verification ensures that the output of the system is reasonable and accurate.

Returning to the simple example once more, it has been an implicit assumption in the model specifications that the object trajectory is on a flat surface, hence the use of Cartesian coordinates and the norm of the position difference vector to obtain the distance. Thus a validation of the software is ensuring that within these assumptions a given object position and velocity and destination position will yield the desired time. However, if the actual trajectory is over a curved surface, such as the earth, then the actual distance and resulting time to destination will be different than the calculated on from this program. This comparison between the actual results and the computed results is the verification process. Perhaps for small distances between the object and destination the differences are acceptable, but the verification process should document when the differences become unacceptable and why.

This is clearly an important process of the software development process. It requires the students to use their

understanding of the physics of the system being modeled as well as the assumptions that led to the software being tested. As such, it is a difficult process for the students to comprehend, and a number of opportunities are provided to the students in order for them to refine this skill.

For the students that put the effort into refining this skill, they payoff is a deeper understanding of the topics involved in the software system. To perform the verification thoroughly they need to have an excellent understanding of the physics, and they must be able to apply that understanding to the system being modeled. In general, these students have developed a level of understanding of the physics well beyond those students that do not work at this skill. As much emphasis as possible should be focused on this part of the software development process, but the skills leading up to this point cannot be neglected otherwise the students are left trying to debug a huge, complicated computer program without any justification about what parts work and what parts do not work. Striking this balance is difficult and varies from class to class as well as year to year.

III. Implementations of These Concepts

A. Sophomore Year

The sophomore year is the start of the main aerospace curriculum in the Cal Poly aerospace engineering undergraduate program. As such this is one of the first time that the students take classes from the department, and the software development practices are introduced in this year. The main objectives of this year is to introduce functional decomposition and unit level testing to the students. *Aerospace Engineering Analysis*, AERO 300, is the class that provides the primary introduction to these software development techniques.

1. AERO 300 – Aerospace Engineering Analysis

The first place that the students are introduced to these concepts is in AERO 300, *Aerospace Engineering Analysis*. This class is a culminating experience for the students freshman and sophomore classes in mathematics and science. It shows the students how these topics are applied to aerospace engineering. The class has a heavy focus on applied mathematical and science techniques and utilizes MATLAB for a number of projects and assignments. The AERO 300 content has been significantly altered in the past few years in order to introduce software development practices and some basic software development skills. Most of the significant alterations included the creation of 4 concept modules: ODEs, Matrices and Linear Algebra, multi-variable integral and differential calculus, and PDEs. Along with these 4 modules a number of aerospace engineering related projects were created to be assigned during the modules. These projects tied the pure mathematics with their physics background into an aerospace engineering context. The projects are team based projects to facilitate peer learning and develop team management skills and are programming related.

Every week homework problems are assigned that, among other things, include simple programming tasks that are actually software units from a functional decomposition of the project that is associated with the current module. Since the students are assigned the project at the beginning of the module they see what the larger task to be accomplished before they see the resulting functional decomposition assignments. By the time that the homework is assigned for a module, the students have been assigned all, or nearly all, of the individual programming tasks that are needed to accomplish the project. Their actual group task is to ensure that their unit level tests are successful and to integrate the individual components into a functioning piece of software.

As the weeks progress in this class, the students amass a significant collection of MATLAB functions that should be of great use to them in their other classes. They have quadrature functions, ODE propagators, iterative and direct matrix solvers, and other functions. In addition they have the homework problems that provided a unit level test for each of these functions. As the students get more familiar with the class, the concept of unit level testing and preserving their unit level test artifacts is discussed along with the benefits and problems associated with their retention.

In this class the students struggle with the concept of functional decomposition. On top of that, they are also fairly new to the MATLAB programming environment and are not too familiar with the language. In order to facilitate the comprehension of functional decomposition without being burdened by the lack of understanding of the MATLAB syntax, the students are encouraged to use pseudo-code to express their function decomposition. A formal pseudo-code is not enforced, rather the students are allowed to express the functional decomposition in brief action word phrases, such as “Calculate dot product” or “Integrate for 1 second,” and then they can iterate on the functional decomposition by adding more MATLAB-like statements as they refine their understanding of the task and the

MATLAB language. This has proven to be a successful process of introducing both a software development skill, functional decomposition, and a programming language, most typically the student's first programming language.

One assigned project for this class was an atmospheric entry problem using the two body gravitational model with an added drag coefficient term for the atmospheric drag from a ballistic trajectory. This problem is essentially an ODE solver, 6 coupled first order ODEs need to be propagated, with an atmospheric model to get a more realistic drag force variation with altitude. All calculations were done in ECI coordinates and the propagation was to start in a suitably chosen low earth orbit and end when the trajectory reached the mean earth radius. The students were not responsible for converting between ECI and ECF coordinates, or any other coordinates, nor were they responsible for calculating the drag coefficient variation with Knudsen number, Reynolds number, Mach number, or any other variation.

While this presented a rather simple model it did provide the students with an actual aerospace engineering application of the concept of numerical integration of ODEs. During this module, one homework assignment included a problem asking them to write a function to return the atmospheric density for a given altitude. They were given a reference to a table that produces similar values, most aerodynamics textbooks have one, and the equation to implement which was simply a set of functional relations associated with various altitudes. This task appears to be a task associated with getting them familiar with "if" statements, but it also provides a valuable piece to their project. Their next homework assignment asks the students to develop a fourth order Runge-Kutta ODE solver that can be applied to a system of ODEs. They are given one first order ODE and one system of 2 ODEs both of which have analytic solutions, that they must derive themselves, and they must validate their ODE solver on these two analytic problems. With these two problems completed, the students have the lion's share of the programming accomplished before they have done anything specifically for the project. This has been an extremely successful project with the resulting code of a higher quality and what appears to be a higher success rate than before these changes were introduced.

B. Junior Year

While the students have been exposed to some of the software development techniques, they still require significant work applying it to aerospace specific problems. Up to this point the introduction of these concepts have been more focused on applied mathematics problems that happen to be aerospace related. In their junior year, the students are getting significant exposure to aerospace engineering topics and can begin to combine their software development knowledge with aerospace engineering topics. In addition, Verification and Validation are introduced now since they have developed enough of an understanding of the physics being modeled to develop and comprehend validation cases. Two of the classes that this has occurred are *Aerothermodynamics III*, AERO 303, and *Aerodynamics and Flight Performance*, AERO 306.

1. AERO 303 – Aerothermodynamics III

This class provided some of the earlier results of integration of software development techniques, however an interesting and rather successful project resulted from this. This class is split into two parts, compressible flows and heat transfer with 50% of the quarter spent on each topic. In an attempt to merge these rather separate topics, a project was constructed that by the end of the quarter integrated both topics into a single program. The overall project concept was an atmospheric entry problem into the Martian atmosphere with the physics modeling very similar to the AERO 300 project discussed above. For students that successfully accomplished that project and have their software library from the class, this project was that much easier!

The first project associated with the compressible flow portion asked them to calculate the stagnation pressure and stagnation temperature at the forward stagnation point. The project statement stated that the stagnation pressure was needed to determine the strength of the material needed to protect the vehicle from pressure differences between the exterior and interior pressures (a bit of a red herring, but provided some vaguely reasonable reason to calculate the stagnation pressure). The stagnation temperature was needed to determine the thermal properties (a task to be accomplished in the second project). The program assumed calorically perfect gas throughout the entire trajectory, and the trajectory could be obtained *a priori* from the orbital dynamics of the problem. The main tasks associated with this effort were the calculation of the stagnation pressure and temperature, and each of these were assigned as homework problems. The students had their textbook gas tables[6], or any other text or online source[7], for which to validate their functions against. The added twist is that the ratio of specific heats for Mars is not 1.4 (which is what the tables in their book contained) but was assumed to be 1.28 (a value few, if any, books have in their tables).

The second project utilized the stagnation temperature results and used that as one boundary condition to a one-dimensional, unsteady heat conduction problem. The other boundary condition, the interior of the vehicle had a convective boundary condition. The thermal protection system properties were very similar to the space shuttle's TPS properties. The radiative heating was ignored as well as the rarefied regime heat transfer characteristics. The students task was to design the vehicle interior cooling system by determining what the interior temperature would be if the convective cooling system failed, hint pretty hot, and how much forced convection was needed to keep the interior temperature at a survivable 100°F. For this, the students had seen analytic methods of solving one-dimensional, unsteady heat conduction problems, so they could use that information to gauge the accuracy of their results.

Once again the students are introduced to the idea of functional decomposition and unit level testing with the majority of the effort being associated with homework problems and verbal and written descriptions of the codes to be written. While a number of assumptions that greatly simplified the problem made the physics of this analysis more than a bit suspect, this led to a discussion of these limitations and how much more effort would be associated with correcting for these errors as well as the anticipated impact of the changes on the results. In addition, the quality of the physics is accurate, and some students even researched more details solutions to this problem and discovered that their answers were not as far off as they expected. While this project was more difficult than the AERO 300 projects, the level of integration in the project is not near the level of a typical system integration and testing, but the students begin to see that there are unique issues associated with the integration of various software components, such as unit conversions.

2. AERO 306 – Aerodynamics and Flight Performance

This class has become the main junior level class that develops the software development skills discussed here. The students have been exposed to simple functional decomposition and unit level testing and at this point are capable of performing these tasks with some guidance. In addition, the concept of verification and validation are introduced via homeworks and a project, which constitutes the significant portion of the verification and validation introduction. One project from this class was asking the students to compare the aerodynamic prediction capabilities of thin airfoil theory with flat plat boundary layer approximations,[8] a lumped vortex model[9] that they have to program, and XFOIL,[10] a common Open Source airfoil analysis tool that uses potential flow theory and integral boundary layer methods. They are asked to compare the results from these methods with a standard experimental aerodynamics reference[11]. Finally, they have to discuss the relative accuracy of each method and how each method compares with each other.

This task introduces the concept of verification, they have to compare how well each technique matches the experimental reference data, as well as validation, the have to compare their code predictions to lift and drag to hand calculations that they were assigned as homework. This project consisted of several tasks within the domain decomposition. These include:

1. Developing code to get the mean camber line equation of the airfoil
2. Writing code to calculate the thin airfoil theory with flat plat boundary layer approximations
3. Writing a lumped vortex model code
   - 3.1. Calculate the induced velocity of a point vortex
   - 3.2. Accumulate all of the influence coefficients from all of the interactions between point vortices and collocation points
   - 3.3. Enforce the no flow through the surface boundary condition at each collocation point

Each of these tasks were assigned as homework assignments, with either hand calculations (items 2 and 3) or a standard reference (item 1) to unit level test their software units against. Part of the presentation of the project was a discussion of the overall objective (determine the strengths and weaknesses of each aerodynamic prediction capability) as well as the functional decomposition of this task. It then becomes clear why they were asked to write functions that performed specific tasks within the functional decomposition. This project is an excellent introduction to the concept of verification and validation and provides the students the opportunity to explore the aerodynamics that they have been studying.

With this project, the students are given an in-depth introduction to functional decomposition and unit level testing, and they are introduced to verification and validation in a domain specific context. This has proven to be an exciting project for the students to tackle and has provided them with a greater appreciation of the class content and the limitations and advantages of the various modeling techniques that they have studied. In the future, it is expected

that a constant strength vortex panel method will be introduced into this project in order to provide the students with a more challenging software development task.

C. Senior Year

The senior year is when the students can start to integrate a number of software development skills into one project. At this stage they have started to develop the capability of conceiving and constructing their own unit level tests as well as performing their own functional decomposition. Verification and validation is still a subject that requires discussion to refine their thought process, but they are beginning to make the connections between the physics and the programming. This is where system integration and testing are introduced in order for them to be able to construct moderately complex software systems to solve aerospace problems. One class that utilizes these software development practices is *Supersonic and Hypersonic Aerodynamics*, AERO 405.

1. AERO 405 – Supersonic and Hypersonic Aerodynamics

This class provides the students with another opportunity to develop their software development skills. In this class all four skills discussed above are utilized to varying degrees of complexity. One project from this class is a two-dimensional, supersonic polygonal airfoil design project. The students are given design specifications (such as a some constraint on how thin the airfoil can be) for a supersonic wing and have to use oblique shocks and expansion wave relations[6–8] to model the flow over the airfoil, they can ignore the viscous effects in this project. Some years they are asked to perform an optimization given a cost function and various constraints, other years they are asked to show the range of the design space that can meet the specifications. In any event, they are required to assemble a moderately complex software application which has a number of subtle problems that are not obvious to the casual observer, such as the fact that if the deflection angle is too large, the oblique shocks might not form. A domain decomposition of this task results in the following:

1. Produce a geometry that satisfies the geometric constraints
2. Calculate the flow field over the geometry
   - 2.1. Solve for the properties over a compression turn using oblique shock relations
   - 2.2. Solve for the properties over an expansion turn using expansion wave relations
   - 2.3. Catch any errors associated with invalid flows (either physically invalid or constrained by limits on flow properties such as maximum Mach number or minimum density)
3. Obtain lift, drag, and moment values
4. Process the results either by using an optimization routine in MATLAB, `fmincon`, or checking if the configuration satisfies the design constraints

Notice that there are significant number of non-trivial tasks in this project that the students will have developed throughout the quarter and will need to integrate into their final software application.

This projects introduces a software system of sufficient complexity that concepts related to system integration and testing can be discussed. While they have been given assignments that perform unit level testing of the most complicated parts of the code (items 2.1 and 2.2), they have a number of integration issues to address (including the use of the `fmincon` function in MATLAB). During the discussions about the project, topics related to validation and verification are covered. Ideas are solicited from the class about validation cases (e.g., comparing a design point calculation to a hand calculated result) and verification cases (e.g., finding an experimental result that matches a design point that their code can handle). More freedom is provided to the students to develop their testing strategies, however guidance is provided about what strategies might be more or less successful.

This project provides the students with a challenging software system to develop that has a number of subtle programming issues related to the flow physics. Their understanding of the concepts associated with this project are greatly improved as they have to use their knowledge of the flow physics to perform the verification and validation. At the same time this project provides a nice culminating experience for the students to see the techniques needed to develop software applications of this complexity.

D. Graduate Level

In the graduate level classes, these software development skills are utilized in some of the graduate classes. The challenge here is that there is a large difference between students that went through the Cal Poly's undergraduate aerospace curriculum and those who did not. Those who did not typically have no background in any of these topics, so care must be taken to ensure that these students understand the reasoning behind the unfamiliar looking

assignments and projects. In some classes this means that some advanced software development skills are not utilized and in other classes time must be spent to go over the necessary background information on the software development skills in order to familiarize the students to this process. Two graduate classes that utilize these techniques to different degrees are *Boundary Layer Theory*, AERO 522, and *Computational Fluid Dynamics*, AERO 525.

1. AERO 522 – Boundary Layer Theory

This class utilizes functional decomposition and unit level testing through out the quarter as the students are assigned various differential and integral boundary layer programs to develop. They are typically asked to continually test their codes on a common set of analytical solutions (or experimental data if no comparable analytical solutions exist). The project for this class asks the students to take an inviscid velocity profile around a body (typically a two-dimensional airfoil flow field generated from a potential flow solver) and produce a skin friction profile and drag prediction. They are also asked to find data to verify the accuracy of their code. The validation of the code has been performed in the homework problems. This project provides the students with a general presentation of the testing process that they should perform but leaves the development of the testing up to the students. Some discussion is needed in this class to bring the external graduate students, i.e. students who did not attend Cal Poly's aerospace engineering program as an undergraduate, up to speed on the software development skills that they will be utilizing. This usually takes a few discussions that also serve as a reminder to the internal graduate students with the students most knowledgeable about the process leading the discussion. Overall this has resulted in a higher success rate in the students developing working programs for some of the more complicated boundary layer models.

2. AERO 525 – Computational Fluid Dynamics

This class heavily relies on these software development skills to facilitate the student's development of rather sophisticated CFD software. One project asks the students to write their own two-dimensional, inviscid CFD code using van Leer's Flux Vector Splitting Scheme,[12] van Leer's MUSCL extrapolation scheme,[13] and a variety of flux limiters. All of these pieces can cause significant problems for the students which will most likely result in a non-functioning final code. In an attempt to increase the likelihood of success in this project, each of the above functionalities are assigned as homework problems leading up to the project assignment with associated unit level tests. Thus when the students are given the project statement, they have performed a significant amount of the complicated coding, and their focus can turn towards the system integration and testing task. A lot of guidance is provided the students such as suggesting simple geometries for testing where analytic solutions are available, and an insightful discussion usually results from this where the students brainstorm on a number of valuable tests that could be performed. This has shown a significant increase in the quality of the software that the students produce, and have resulted in a higher likelihood of success on this project. The overall impression from the students after this project is that they found these software development skills invaluable to the success of their project.

IV. Conclusions

Starting with students that have little to know software development skills, a strategy has been developed to integrate four key software development strategies into their aerospace engineering curriculum. The integration is performed across years and happens in a number of classes from their sophomore year to their senior year and even in some graduate classes. The overall motivation of this effort was to provide the students with better software development skills so that they could write more sophisticated software applications in order to have them study aerospace phenomena that go beyond the analytic solutions that have been the traditional concepts presented. In addition, the students are learning a set of skills that will serve them well in their engineering career as more and more emphasis is placed on software programming skills in most engineering disciplines. These engineers are more likely to be presented with tasks of developing engineering level software that will exist for the lifetime of the project on which they are working. More importantly the code that they write might outlast their existence on the project and will therefore need to be maintained by other engineers. Thus it is vitally important that they write code that is better structured and tested that simply the “quick and dirty” solutions that they are accustom to writing.

Overall, this effort has been very successful in improving the students software development skills and increasing the student's likelihood of success. By the end the students are able of performing functional decomposition on more than just the simplest problems and can identify unit level testing strategies that are likely to produce well constructed tests of their software units. They can also appreciate the complexities associated with

system level software integration and have a good idea of the steps involved in developing verification and validation procedures for their code. One of the biggest adjustments to this effort was realizing that while the students are computer savvy, their generation has grown up with the PC, they are not necessarily computer programming savvy. There was a generation of computer savvy students that were also computer programming savvy, but that was in an era where anything that you wanted the computer to do you had to program it to do it, but that does not appear to be the case now. It is worth noting that the number of computer programming savvy students have most likely increased, just not as rapidly as the number of computer savvy students. What resulted from this observation was an increase in the presentation of the functional decomposition in the earlier classes and a greater emphasis on the use of pseudo-code to allow the students to express their serial logic steps without be concerned about the exact syntax of the programming language.

In the future it would be nice to incorporate these ideas into more classes in the curriculum so that the students have a more continuous exposure to these concepts throughout their academic career. With the growing popularity of object oriented programming languages in scientific applications, introducing some of the concepts related to object oriented methodologies might become appropriate. It is worth noting that the four software development skills currently being presented do have their place in object oriented software development practices. The other skill that might need to be added is the use of a configuration management tool. This is a common tool in most software development environments, and it would be a valuable skill for the students to have, especially when the current trends are towards more engineering project artifacts (such as design documents, test strategies, and readiness review documents) being placed under configuration management. The problem with this is that there is a non-trivial learning curve that students would have to travel before they would be skilled at using a configuration management tool during their development process. However, if the need grows for this skill, then it will be integrated.

References

1 Friedman, F. L. and Koffman, E. B., *Problem Solving, Abstraction, and Design using C++,* 5th Edition, Pearson Education, 2006.
2 Kung, D. C., Hsia, P., and Gao, J., *Testing Object-Oriented Software*, John Wiley & Sons, Inc., 1998.
3 Martin, K. and Hoffman, B., *Mastering CMake: A Cross-Platform Build System*, Kitware, Inc., 2006.
4 IEEE Computer Society, *IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Std 754-1985)*, 1985.
5 Fisher, M. S., *Software Verification and Validation: An Engineering and Scientific Approach*, Springer-Verlag, 2006.
6 Zucrow, M. J. and Hoffman, J. D., *Gas Dynamics, Volume I*, Wiley, 1976.
7 Anderson, Jr., J. D., *Modern Compressible Flow: With Historical Perspective*, 3rd Edition, McGraw-Hill, 2002.
8 Anderson, Jr., J. D., *Fundamentals of Aerodynamics*, 4th Edition, McGraw-Hill, 2007.
9 Katz, J. and Plotkin, A., *Low Speed Aerodynamics*, 2nd Edition, Cambridge University Press, 2001.
10 Drela, M. and Youngren, H., *XFOIL: Subsonic Airfoil Development System*, URL: http://web.mit.edu/drela/Public/web/xfoil/ [cited December 20, 2007].
11 Abbott, I. H. and von Doenhoff, A. E., *Theory of Wing Sections*, Dover Publications, 1959.
12 van Leer, B., "Flux Vector Splitting for the Euler Equations," ICASE Report 82-30, September 1982.
13 van Leer, B., "Toward the Ultimate Conservative Difference Scheme. V. A Second Order Sequel to Godunov's Method," *Journal of Computational Physics*, Vol. 32, No. 1, 1979, pp. 101–136.