Summer 2008

Using Agile Software Development Practices in a Research Oriented Distributed Simulation

Douglas James Mielke
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/msve_etds

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
https://digitalcommons.odu.edu/msve_etds/21

This Thesis is brought to you for free and open access by the Modeling, Simulation & Visualization Engineering at ODU Digital Commons. It has been accepted for inclusion in Modeling, Simulation & Visualization Engineering Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.
USING AGILE SOFTWARE DEVELOPMENT PRACTICES IN A
RESEARCH ORIENTED DISTRIBUTED SIMULATION

by

Douglas James Mielke
B.S. December 1987, University of Minnesota

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

MASTER OF SCIENCE
MODELING AND SIMULATION
OLD DOMINION UNIVERSITY
August 2008

Approved by:

John A. Sokolowski (Director)

Yiannis E. Papelis (Member)

Bryan E. Barmore (Member)
ABSTRACT

USING AGILE SOFTWARE DEVELOPMENT PRACTICES IN A RESEARCH ORIENTED DISTRIBUTED SIMULATION

Douglas Mielke
Old Dominion University, 2008
Director: Dr. John A. Sokolowski

Although sometimes controversial, agile methodologies have proven to be a viable choice for some software development projects. Projects suited to agile methodologies are those that involve new technology, have requirements that change rapidly, and are controlled by small, talented teams. Much literature about agile software development leans towards business products and non-government entities. Only a handful of literature resources mention agile software development being used in government contracts and even fewer resources mention research projects. NASA's Airspace and Traffic Operations Simulation (ATOS) is a research oriented simulation that doesn't follow the traditional business project mold. In an effort to gain a better understanding if agile could be used effectively in a NASA contract for a research oriented simulation project, this research looked at what agile practices could be effectively used to help gain simulation reliability while simultaneously allowing routine maintenance, current experiment support, new modeling additions, and comprehensive architectural changes.
To: Jan for your love, patience, and help with my thesis project.
<table>
<thead>
<tr>
<th>TABLE OF CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>LIST OF TABLES ..................................................................... vi</td>
</tr>
<tr>
<td>LIST OF FIGURES ................................................................... vii</td>
</tr>
<tr>
<td>INTRODUCTION .......................................................................... 1</td>
</tr>
<tr>
<td>Thesis Statement ..................................................................... 1</td>
</tr>
<tr>
<td>Problem Statement ................................................................... 1</td>
</tr>
<tr>
<td>Motivation ............................................................................... 2</td>
</tr>
<tr>
<td>Approach ................................................................................ 3</td>
</tr>
<tr>
<td>Contributions .......................................................................... 4</td>
</tr>
<tr>
<td>Thesis Organization .................................................................. 4</td>
</tr>
<tr>
<td>BACKGROUND ............................................................................ 6</td>
</tr>
<tr>
<td>Agile software development methodologies ................................ 6</td>
</tr>
<tr>
<td>What is ATOS the system? ......................................................... 17</td>
</tr>
<tr>
<td>Current and past research using ATOS ....................................... 19</td>
</tr>
<tr>
<td>Strengths of ATOS .................................................................... 20</td>
</tr>
<tr>
<td>Weaknesses of ATOS .................................................................. 22</td>
</tr>
<tr>
<td>THE AGILE APPROACH ............................................................. 28</td>
</tr>
<tr>
<td>Introduction ............................................................................ 28</td>
</tr>
<tr>
<td>Changes to the development process ......................................... 28</td>
</tr>
<tr>
<td>Need for a new methodology ................................................... 29</td>
</tr>
<tr>
<td>Methodology requirements for ATOS .......................................... 31</td>
</tr>
<tr>
<td>Agile practices best suited for ATOS ......................................... 40</td>
</tr>
<tr>
<td>Other similar research simulation projects ................................. 50</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Critical error count—baseline</td>
<td>72</td>
</tr>
<tr>
<td>2. Compiler warning reduction</td>
<td>73</td>
</tr>
<tr>
<td>3. Runtime error reduction</td>
<td>75</td>
</tr>
<tr>
<td>4. Install time reduction</td>
<td>87</td>
</tr>
<tr>
<td>5. File size reduction</td>
<td>88</td>
</tr>
<tr>
<td>6. Build time reduction</td>
<td>89</td>
</tr>
<tr>
<td>7. Increased log messages</td>
<td>90</td>
</tr>
<tr>
<td>8. Log messages old versus new coding</td>
<td>91</td>
</tr>
<tr>
<td>9. Critical error end result</td>
<td>92</td>
</tr>
<tr>
<td>10. Comparison to Lindvall et al.</td>
<td>95</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. General SCRUM Process</td>
<td>9</td>
</tr>
<tr>
<td>2. Dimensions of Crystal methodologies</td>
<td>11</td>
</tr>
<tr>
<td>3. Processes of FDD</td>
<td>13</td>
</tr>
<tr>
<td>4. The ASD cycle</td>
<td>16</td>
</tr>
<tr>
<td>5. Typical cluttered code example</td>
<td>47</td>
</tr>
<tr>
<td>6. Refactored code example</td>
<td>48</td>
</tr>
<tr>
<td>7. Scenario design</td>
<td>57</td>
</tr>
<tr>
<td>8. ATOS system architecture</td>
<td>61</td>
</tr>
<tr>
<td>9. Typical top-level simulation code structure</td>
<td>82</td>
</tr>
<tr>
<td>10. Modified top-level simulation code structure</td>
<td>82</td>
</tr>
<tr>
<td>11. Refactoring error handling for a potential divide by zero problem</td>
<td>84</td>
</tr>
<tr>
<td>12. Successful build days over a 30 day period</td>
<td>85</td>
</tr>
</tbody>
</table>
1 INTRODUCTION

1.1 Thesis Statement

Agile practices can be used in the software development of research oriented distributed simulations. Furthermore, a set of agile practices can be utilized on NASA Langley's Airspace and Traffic Operations Simulation to simultaneously increase reliability and functionality.

1.2 Problem Statement

NASA Langley's Airspace and Traffic Operations Simulation (ATOS) was originally developed for the purpose of investigating advanced, distributed air-ground traffic management concepts [1] under the direction of NASA's Advanced Air Transportation Technologies (AATT) project [2]. Throughout the past three years, additional types of Air Traffic Management (ATM) research were conducted utilizing ATOS. This new research required a significant size development team to design and implement new modeling capability to the simulation. The future brings more versatility to ATOS with researchers expressing desires to expand upon the simulation's traditional research functionality.

Unfortunately, due to budget cuts, the size of the development team has drastically decreased and the demand to successfully accomplish future experiments places unique challenges on this small team. They are faced with maintaining the existing system,

---

1 Citation and reference list format for this manuscript are taken from the journal SIMULATION: Transactions of the Society for Modeling and Simulation International.
supporting existing demonstrations and experiments, while at the same time trying to
grow the system to meet future modeling requirements. The existing system is
fragmented with several different architectures that lengthen the learning curve for new
developers. Talented individuals are wasting effort trying to understand how to modify
the simulation system, what to modify, and how much to develop.

The development team previously worked under a formal software design
methodology. Separate modeling and architecture groups provided formal designs and
often enlisted another group of programmers to implement. In addition, an Integration
and Test team checked and merged the code base to a build suitable for releasing to
NASA’s Air Traffic Operations Laboratory (ATOL). The need to keep the simulation
running and improving has sidelined the above methodology. However, it is not prudent
to keep on developing in the same manner. A new software development methodology is
needed to complement the changed environment. This new methodology must allow
routine maintenance, current experiment support, new modeling additions, and
comprehensive architectural changes.

1.3 Motivation

The foundation and underlining objectives have changed in the ATOS system from what
they were just a few years ago. ATOS development has a smaller budget and fewer
developers. However, NASA is demanding a wider variety of research, more
sophisticated modeling, and shorter lead times from concept to experiment. ATOS has
the additional duties of providing demonstrations and supporting connectivity to outside
simulation laboratories that were never included in its original design. The size of the
laboratory hosting ATOS has doubled in square feet, and the number of computing units has increased by an order of magnitude.

The development methodology has also significantly changed. In the past, there was a formal methodology in place that served its purpose. With a shrinking budget and team size, the formal methodology has given way to undefined informal practices. Elaborate UML diagrams created to aid the analysis phase and lengthy design phases with complete documentation are practices of the past. Instead, quick informal meetings utilizing simple whiteboard drawings are now sufficient before new coding begins.

Agile methodologies are the wave of the future for many software development teams. This thesis research intends to investigate the usefulness of implementing many of the key agile practices mentioned in literature into the development environment of a complex distributed real-time research oriented simulation such as ATOS.

1.4 Approach

The goal of this thesis is to explore the most popular agile methodologies and choose a set of agile practices that are best suited for the ATOS development environment. The approach to achieving this goal will require the following:

Survey the literature on existing agile software development methodologies.

Present a set of agile practices that are best suited for ATOS development.

Validate the set of agile practices by demonstrating significant improvements to the reliability of the ATOS system while performing routine maintenance, new modeling additions, and comprehensive architectural changes for an upcoming experiment.
1.5 Contributions

This thesis discusses the usage of a set of agile software development practices as an effective way to reduce reliability problems and increase system functionality in a research oriented distributed simulation.

The author of this thesis has contributed the idea of introducing agile software development to the ATOS development group as well as championing the implementation of the agile practices and providing a significant portion of the reliability solutions and functionality enhancements mentioned in this thesis.

In summary, the main contributions of this thesis are:

A set of agile software development practices that are practical for a small development team in a simulation research facility.

Validation that using a set of agile software development practices is effective in reducing reliability problems.

Demonstration that a successful experiment requiring increased functionality can be conducted using the ATOS simulation.

1.6 Thesis Organization

The remainder of this thesis is organized as follows:

Section 2. Background: This section starts by summarizing the literature research on the most popular agile software development methodologies. Section 2 continues with a description of ATOS and the types of research utilizing this complex system. Finally, Section 2 concludes with a description of the strengths and weaknesses of this simulation.
Section 3. The Agile Approach: This section analyzes the existing demands of everyday development tasks (system maintenance, demonstration capabilities, current experiment support, new modeling additions, and system wide architectural changes) of ATOS, and presents a set of agile software development practices best suited for this simulation.

Section 4. Research Project: This section describes the validation of the agile methodology plan examined in Section 3 by demonstrating significant improvements to the reliability of the ATOS system, while adding functionality for an upcoming experiment.

Section 5. Conclusion: This thesis concludes with a summary of the research results.
2 BACKGROUND

This section starts by summarizing the research literature on the most popular agile software development methodologies. Section 2 continues with a description of ATOS and the types of research utilizing this complex system. Finally, it concludes with a description of the strengths and weaknesses of this simulation.

2.1 Agile software development methodologies

Various methodologies are used to direct the life cycle of a software development project. While each methodology is designed for a specific purpose or reason, most have similar goals and share many common practices. In the last decade, methodologies have emerged and are referred to as agile software development methodologies. All of these agile methodologies share common principles, which have been expanded on as described in the Agile Manifesto [3]:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Agile software development processes are built on the foundation of iterative development with changing requirements. To that foundation they add a lighter, more people-centric viewpoint and rely heavily on the tacit knowledge of users. Agile processes are less document-oriented, usually emphasizing a smaller amount of
Agile documentation for a given task. Agile techniques utilize feedback, rather than planning, as their primary control mechanism. The feedback is driven by regular tests and releases of the evolving software. Lightweight methods provide frequent increments to already functioning software. The team who published the *Agile Manifesto* also expressed a set of principles behind the manifesto [4]:

*We follow these principles:*

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity—the art of maximizing the amount of work not done—is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
The most popular agile methodologies used today are SCRUM, Dynamic Systems Development Method (DSDM), Crystal Methods, Feature-Driven Development (FDD), Extreme Programming (XP), and Adaptive Software Development (ASD). These methodologies share common principles, but differ in practices, and provide different levels of flexibility. A development team may choose which agile approach to take among the variety of agile methodology surveys. Abrahamsson et al. [5] has one of the most extensive reviews and analyses of the subject. Cohen et al. [6] and Williams [7] produced additional surveys looking at only the most popular agile methodologies. Finally, what many consider one of the most authoritative experts in agile methods, Jim Highsmith, has written a book titled *Agile Software Development Ecosystems* [8] that examines and compares major agile methodologies.

### 2.1.1 SCRUM

SCRUM, named for the scrum in rugby, was initially developed by Ken Schwaber [9] and Jeff Sutherland [10], with later contributions from Mike Beedle [11]. SCRUM provides a project management framework that focuses development into 30-day sprint cycles in which a specified set of backlog features are delivered. The core practice in SCRUM is the use of daily 15-minute team meetings for coordination and integration. SCRUM has been in use for over ten years and has successfully delivered a wide range of products.

The SCRUM approach is an empirical approach applying the ideas of industrial process control theory to systems development, resulting in an approach that reintroduces the ideas of flexibility, adaptability, and productivity [12]. SCRUM in itself does not define any specific implemental software development techniques. Instead, it
concentrates on how team members should function in order to produce a flexible system in a constantly changing environment.

The SCRUM process includes three main phases (Figure 1): Pre-Game, Mid-Game, and Post-Game.

<table>
<thead>
<tr>
<th>Pre-Game</th>
<th>Mid-Game</th>
<th>Post-Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning and High-Level Design</td>
<td>Develop</td>
<td>Closure</td>
</tr>
<tr>
<td></td>
<td>Wrap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Adjust</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Review</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. General SCRUM Process - Clifton and Dunlap [13]

The Pre-Game phase consists of two primary activities: planning and architecture. During planning, a new release is defined based on a current known backlog, then a schedule is created, and cost estimates are conducted. Architecture activities include designing how the backlog items will be implemented in a high-level design. The Mid-Game phase consists of several development sprints, focusing on new release functionality. During these sprints, constant respect to the variables of time, requirements, quality, cost, and competition are considered. Finally, in the Post-Game phase, preparation for release of the code begins, including tasks such as integration, system testing, and documentation [9].
2.1.2 Dynamic Systems Development Method (DSDM)

Dynamic Systems Development Method (DSDM) was developed in the U.K. in the early 1990s. DSDM is a non-profit and non-proprietary framework [14] for Rapid Application Development (RAD), and maintained by the DSDM Consortium [15]. According to Stapleton [16], DSDM has gradually become the number one framework for RAD in the U.K. DSDM recommends the application of a series of principles, which provide a fast and effective way to develop software systems within limited deadlines and manpower resources.

The fundamental idea behind DSDM is, instead of fixing the amount of functionality in a product (and then adjusting time and resources to reach that functionality), it is preferred to fix time and resources, and then adjust the amount of functionality accordingly.

2.1.3 Crystal Methods

Alistair Cockburn is the founder of the Crystal Methods approach [17]. His focus is on people, interaction, community, skills, talents, and communications with the belief that these have the first-order effect on performance. Process, he says, is important, but secondary. Cockburn has interviewed dozens of project teams worldwide trying to differentiate between what actually works from what should work. The Crystal philosophy recognizes that each team of people has a different set of talents and skills, and that each team should use a uniquely tailored process. Additionally, the process should be minimized to the point that it is barely significant.
Cockburn's methods are named Crystal to represent a gemstone, i.e. each facet is another version of the process, all arranged around an identical core. As such, the different methods are assigned colors arranged in ascending opacity. The most agile version is Crystal Clear, followed by Crystal Yellow, Crystal Orange, Crystal Red, etc. The version of Crystal used depends on the number of people involved, which translates into a different degree of emphasis on communication.

Figure 2. Dimensions of Crystal methodologies - Cockburn [17]

Figure 2 illustrates Cockburn's Crystal Methods. Upon adding people to the project, you translate right on the graph to more opaque versions of Crystal. As project criticality increases, the methods harden and move upwards on the graph. The character symbols indicate a potential loss caused by a system failure (i.e. criticality level): Comfort (C), Discretionary Money (D), Essential Money (E), and Life (L). For example, criticality
level "C" indicates that a system crash due to defects causes a loss of comfort for the user, whereas defect in a life critical system may literally cause loss of life, “L” [17].

2.1.4 Feature-Driven Development (FDD)

Jeff DeLuca and Peter Coad collaborated on Feature-Driven Development (FDD) in the late 1990s. DeLuca was contracted to save a failing, highly complicated lending system. The previous contractor had spent two years producing over 3,500 pages of documentation, but no code was actually developed [8]. DeLuca started from the beginning and hired Coad to assist with the object modeling. From this experience they developed the feature-oriented development approach that came to be known as FDD that was written by Coad et al. [18].

The FDD approach does not cover the entire software development process, but rather focuses on the design and building phases [19]. However, the FDD approach has been designed to work with the other activities in software development and does not require any specific process model to be used. The FDD approach embodies iterative development with the best practices found to be effective in the industry. Emphasis is on quality aspects throughout the process and includes frequent and tangible deliveries, along with accurate monitoring of the project’s progress.
FDD consists of five sequential processes during which the designing and building of the system is carried out (Figure 3). FDD’s processes are brief (each is described on a single page), and utilize two key roles: Chief Architect and Chief Programmer. According to Palmer and Felsing these processes include [19]:

**Develop an Overall Model**: Domain experts are already aware of the scope, context, and requirements of the system being built. The development team then discusses and decides upon the appropriate object models for each of the domain areas. The end result is the overall model shape created for the system.

**Build Features List**: By walking through the object models and the known requirements, a comprehensive feature list is created and reviewed with the client for its validity and completeness.

**Plan by Feature**: A high-level plan is created for each feature by a Chief Programmer and then passed down to the individual developers.
Design by Feature and Build by Feature: Each feature can be split into smaller features as it goes through an iterative development cycle of a few days to a maximum of two weeks. After a successful iteration, the completed feature is promoted to the main build.

2.1.5 Extreme Programming (XP)

Extreme Programming (XP) has undoubtedly garnered the most interest of any of the agile approaches. XP evolved from the problems caused by the long development cycles of traditional development models [20]. Haungs [21] claims XP first started as "simply an opportunity to get the job done" using effective practices of the preceding decades that were found in typical software development processes [22]. The XP methodology was conceptualized on key principles and practices after a number of successful trials [23].

Highsmith [8] lists five key principles of XP, all of which are enhanced by its practices: community, simplicity, feedback, courage, and quality work. XP also provides a system of 12 concise dynamic practices [22]:

The planning game: At the start of each iteration, customers, managers, and developers meet to analyze, estimate, and prioritize requirements for the next release. The requirements, also known as user stories, are captured on story cards in a language understood by all parties.

Small releases: An initial version of the system is put into production after the first few iterations. Subsequently, working versions are put into production anywhere from every few days to every few weeks.
**Metaphor:** Customers, managers, and developers construct a metaphor, or a set of metaphors, to model the system.

**Simple design:** Developers are urged to keep design as simple as possible, to say everything once and only once.

**Tests:** Developers work with a test-first approach, writing acceptance tests for their code before writing the code itself. Customers write functional tests for each iteration, and at the end of each iteration all tests should run.

**Refactoring:** As developers work, the design should evolve to keep it as simple as possible.

**Pair programming:** Two developers sit together at the same machine.

**Continuous integration:** Developers integrate new code into the system as often as possible. All functional tests must still pass after integration or the new code is discarded.

**Collective ownership:** The code is owned by all developers, and they may make changes anywhere in the code at any time as deemed necessary.

**On-site customer:** A customer works with the development team at all times to answer questions, perform acceptance tests, and ensure that development is progressing as expected.

**40-hour work weeks:** Requirements should be selected for each iteration such that developers do not need to put in overtime.

**Open workspace:** Developers work in a common workspace set up with individual workstations around the periphery with common development machines in the center.
2.1.6 Adaptive Software Development (ASD)

Adaptive Software Development (ASD) was developed by Highsmith in 2000 [24]. Many of ASD's principles stem from Highsmith's earlier research on iterative development methods. The most notable ancestor of ASD, *RADical Software Development*, was co-authored by Bayer and Highsmith in 1994 [25].

ASD focuses mainly on the problems in developing large, complex systems. The method strongly encourages incremental, iterative development, with constant prototyping. Fundamentally, ASD's aim is to provide a framework with enough guidance to prevent projects from falling into chaos, but not too much, which could suppress creativity.

An Adaptive Software Development project is carried out in three-phase cycles: Speculate, Collaborate, and Learn (see Figure 4).

![ASD Cycle Diagram](image)

**Figure 4.** The ASD cycle - Highsmith [8]

The phases are named to emphasize the changing roles in a process. *Speculation* is used instead of *Planning*, as a *plan* is generally seen as something where uncertainty is a weakness, and from which deviations indicate failure. Similarly, *Collaborate* highlights
the importance of teamwork as the means of developing high-changing systems. Learn stresses the need to acknowledge and react to mistakes, and that requirements may change during development.

ASD is explicitly component-oriented rather than task-oriented. In practice, the focus is more on results and their quality, rather than the tasks or processes used for producing the result. The way ASD addresses this viewpoint is through adaptive development cycles that contain the Collaborate phase, where several components may be under concurrent development. Planning the cycles is part of the iterative process, as the definitions of the components are continuously refined to reflect any new information.

2.2 What is ATOS the system?

In collaboration with other government agencies, industries, and the international R&D community, NASA is developing, researching and nurturing components of the Next Generation Air Transportation System (NGATS). The NGATS vision calls for a system-wide transformation of the National Airspace System (NAS), leading to a new set of capabilities that will allow the system to respond to future needs of the nation's air transportation [26]. One of the more visible tasks under NGATS vision is the Air Traffic Management (ATM) Airspace project [27]. The primary focus is to explore and develop integrated solutions that provide research data to define and assess the allocation of ground and air automation concepts, including the human roles necessary for the NGATS. A portion of this work is being conducted in NASA Langley's state-of-the-art simulation facility known as the Air Traffic Operations Laboratory (ATOL).
The ATOL facility allows NASA to evaluate new ATM concepts. The main focus is to ensure appropriate levels of compatibility with real-world avionics system architectures and emerging NAS infrastructure. This is achieved through a simulation environment, known as Airspace and Traffic Operations Simulation (ATOS). Finkelsztein et al. [28] and Peters et al. [29] describe the ATOS system in further detail.

ATOS is comprised of dozens of computer workstations that can be used as either pilot stations flown by real pilots, or batch processing stations run automatically. These simulated aircraft can interact with each other in a simulated airspace and traffic environment using various configurations and air traffic scenarios. ATOS includes the abilities to insert hundreds of additional automated aircraft via the Traffic Manager (TMX) [30], connect high-fidelity flight-deck simulators [28], and can be connected to other ATM simulation labs over the Internet using the open standards of AviationSimNet [31]. In addition, the lab can support “pseudo-pilot” (i.e. multi-aircraft) control, remotely piloted and non-piloted aircraft operations [28].

This concept-level distributed traffic simulation environment is used for:

Operational feasibility assessments,

System-level requirements definition,

Airborne and ground-based communication, navigation, and surveillance (CNS) technology requirements determination, and

Human-centered design and assessment of ATM concepts and flight-deck systems.

A workstation-based aircraft simulation referred to as the Aircraft Simulation for Traffic Operations Research (ASTOR) [32], is designed to replicate the displays and controls of the modern transport aircraft [33]. An enhanced avionics data bus that
achieves conceptual compatibility, rather than hardware compatibility, with existing ARINC 429 avionics data bus standards is used for inter-component communication [34]. Each ASTOR hosts advanced decision aids and airborne CNS systems, including high-fidelity Automatic Dependent Surveillance Broadcast (ADS-B) models [35].

2.3 Current and past research using ATOS

The Airspace and Traffic Operations Simulation (ATOS) system was used in all the experiments conducted in the Air Traffic Operations Laboratory (ATOL). In addition, many demonstrations of the system’s capabilities are conducted on a weekly basis.

To support NGATS research, ATOS is supporting the continuous development of several interactive flight-deck decision support tools, such as the Autonomous Operations Planner (AOP) [36, 37], the Paired Dependent Speed (PDS) tools [38], and an Altitude Change Request Advisory Tool [39].

AOP is a tool set that functions as an Airborne Separation Assistance System (ASAS) for advanced performance-based 4D trajectory flight operations. 4D-ASAS development efforts supports NGATS research to significantly increase capacity of the NAS while maintaining or improving safety. 4D-ASAS enable aircraft pilots to maintain traffic separation while conforming to traffic flow management constraints assigned by ground-based air traffic service providers.

NASA is also developing Paired Dependent Speed (PDS) flight guidance to help increase arrival efficiency and throughput at capacity-limited airports. PDS allows the pilots to manage their speeds during descent and approach while precisely spacing their aircraft relative to another aircraft. By increasing precision with which aircraft are
spaced, they can be safely spaced more tightly, allowing more aircraft to land during a period of time and decrease en-route delays.

Oceanic operations, due to the extended period in which aircraft are out of radar coverage, have large longitudinal and lateral separation minima. The separation minima provide safe operations, but are often not fuel-efficient. Current research uses the oceanic domain as a place to investigate a phased approach to integrating the various levels of separation authority delegation in the constrained 4D environment, which creates the opportunity to consider 4D-ASAS. The ATOS provides an exceptional environment for this research.

2.4 Strengths of ATOS

ATOS has performed exceptionally well in its past experiments because of its many strengths.

2.4.1 Optimized for HITL studies

ATOS is optimized for real-time Human-In-The-Loop (HITL) experiments. The system consists of up to 12 pilot stations with each setup located in a private cubical. The collection of pilot stations is separated from the simulation operations center in their own closed-off room. Each pilot station has a realistic cockpit display that represents a modern day large-bodied air transport jet’s glass cockpit. Data collection can be performed on all the pilot interactions with the cockpit displays.
2.4.2 **Excellent design of HLA data**

Most studies performed in ATOL consist of potential future applications of ADS-B communications between aircraft. High Level Architecture (HLA) allows ATOS's simulated aircrafts to share data between each other, in effect modeling the ADS-B communications. The data is broken down into various interactions between aircraft and uses the HLA interaction concept as the mechanism of transfer. This robust design [29] has remained unchanged for several years and continues to meet most of ATOS's needs.

2.4.3 **Multiple simultaneous experiment support**

One of the most powerful features of the ATOS system is its reconfiguration capabilities. The simulation supports a baseline configuration with various options that can be turned on to give the simulation additional modeling capabilities, such as AOP [36] and PDS [38]. Each of these modeling capabilities has multiple configuration possibilities that sets up easily with scenario files. For each experiment, a set of scenario files are created expressing the various configurations appropriate for study. This set-up approach is extremely powerful because ATOS can support multiple experiments with an assortment of simulation models at one time with the same code base.

2.4.4 **Simple build and installation methodologies**

The ATOS software has a simple build and installation methodology. Every day, two build machines automatically create a development build and an integration build. Development builds are considered the tip of the development iceberg and tend to be unstable. Integration builds include merges of stable versions of the software and are used for testing, demonstrations, and experiments. Custom builds can be performed at
any time and installed similar to the daily builds (useful for testing potentially harmful changes). The builds are installed on all computers with one command from a central computer (this is crucial with a lab of over 100 computers). For each computer, all the software is installed so that it can be configured to a multitude of options. The old builds exist simultaneously with the new builds and can run by changing one statement in a command file located on a central computer. All builds have a unique name labeling system with a prefix and a date of the build.

2.4.5 Robust Avionics Bus architecture

The individual ASTOR aircraft simulation components are held together by a robust Avionics Bus (AvBus) architecture, allowing individual components to be glued together with a consistent interface and easily swapping out similar components. The AvBus simulated architecture is modeled after the ARINC 429 bus structure as explained by Palmer and Ballin [34].

2.5 Weaknesses of ATOS

For all the strengths that ATOS software architecture exhibits, unfortunately there are an equal number of weaknesses that prevent the lab from working at its full potential. This section will identify the major weakness in the simulation and how they affect the system as a whole.

2.5.1 Inconsistent coding practices

The largest and most challenging obstacle keeping ATOL from reaching its full potential is the inconsistent coding methodologies spread throughout the entire code base. This
inconsistency can be found in the manner in which applications start up, pass data around, handle unexpected conditions, deal with algorithmic complexities, and in the code readability. These inconsistencies make large system-wide changes extremely difficult for such a small team of developers.

2.5.2 Overly complex

On many occasions, developers program their applications with the ability to handle future enhancements by placing hooks into the code base, or creating algorithms that can handle every possible situation imaginable. Usually these "super algorithms" are not a requirement of the ATOS system, but are included by the developer "just-in-case" it will be needed in the future. Other times, code was written as an ATOS requirement and the requirement changed, but developers choose to keep the code in place "just-in-case" the requirement came back. When these developers leave the program, they take with them the knowledge of these hooks and the location of this dead-end logic. What they leave behind is overly complex code that appears to do more than it actually does, thus confusing the current development team about what exactly a particular piece of code will do.

At its peak, ATOS had over thirty developers working directly on the code base, plus an unknown number of developers working on the code that was lifted from other projects and used as the initial baseline. The current team understands ATOS requirements and the strengths and weaknesses of the simulation, but is struggling with the code where this unnecessary complexity is built in.
2.5.3 Poor error handling

The ATOS software does not have a uniform method of handling errors during run-time. The worst offenders are the applications that assume that they will always work, and do nothing about protecting themselves from unexpected events. The second worst offenders are applications designed to detect faults; as a result, they simply crash. An example of this is source code that calls for an abort function when a pointer is undefined. The assumption behind these simple types of error handling is that when applications crash, a developer will be available to analyze the crash error immediately.

For the applications that attempt to address errors, there are inconsistencies in how the applications handle the errors. Some applications will check for potential error conditions; if a problem is identified, the application ignores the next section of code without logging the error event. Other applications will report the problem to the console window and/or to a log file. Many applications do not distinguish between severity levels of errors, or have consistency in the severity level meanings (e.g. a warning message might be the last item logged after a critical shutdown of the simulation).

Often times, when the system reports an error, the log file is not analyzed by the appropriate people. Even though a log file is created, an error will go undetected unless directly observed. Some applications report hundreds of errors; thus the log files are useless in determining which error is a "noisy message," or which error is causing a system problem. Frequently, the existing error messages in the system are not reported as defects; therefore, uncorrected errors are never planned into the scheduled development activities.
Many times a system failure is due to a series of cascading errors, in which the failure to address the first error creates a second error, which in turn creates another error until the entire system is in an unnatural state. This disarray is observed by the researchers and other users of the system, which could lead to misguided decisions about future activities of the lab. Because of poor error handling, ATOS has a failure rate that is unacceptable for the types of future experiments NASA wants to conduct.

2.5.4 Non-optimal framework library

The ATOS software heavily utilizes the LaSRS++ object-oriented framework for real-time simulation [40]. However, the LaSRS++ framework was designed for a monolithic-type application and not a distributed-type application. LaSRS++ efficiently solves typical problems found in the types of simulations it supports and allows for a development team to quickly add/remove simulated objects, hardware devices, and support equipment. LaSRS++ was designed to be the central authority for the entire simulation, but the role LaSRS++ plays in ATOS is that of partial authority on individual ASTOR simulations. To make LaSRS++ work in an ASTOR, many of its strong features had to be compromised in order to allow command of key components to be taken over by another higher level entity. This in turn left a handicapped framework that complicates overall development work.

2.5.5 Multiple HLA libraries

One of the key strengths of ATOS is the HLA interface design. However, the HLA design is hampered by its use of multiple HLA interfacing libraries. Most of these libraries are not robust enough to handle the full demands required by a distributed
simulation environment. Each library was developed by a different group and has vastly different characteristics. ATOS's current implementation of the HLA interface data has remained unchanged for several years because changing each library is too time consuming and could produce undesired side effects. Future requirements for upcoming experiments will require that the HLA interface is modified.

2.5.6 Poor system monitoring capabilities

ATOS is run by one operator who has full control of starting and shutting down the entire simulation system. There are anywhere from one to dozens of additional simulation stations on any typical run. These stations can exist in different rooms and are not necessarily observable from the operator's position. Future planned experiments will require the use of up to 96 simulation stations, and many of these stations will not include a physical monitor, which leads to a significant problem for handling all the remote stations. Currently, the operator receives only limited information about the health of each system, and except for a partially working tri-color coded health status indicator for each federate, the operator has no way of knowing if the system is having problems. Unless there are human observers at the remote stations indicating the system is functioning correctly, the operator does not have any way to verify that the simulation run was a success.

2.5.7 Does not thoroughly support batch studies

Although ATOS was primarily designed to support HITL experiments, many of the future experiments are focused on batch studies. Complex batch scenarios will automatically run the simulation throughout an extended period of time, often without
human observers present. Batch studies place a new demand on the system as a whole, such as running the simulation for extended periods, when the time synchronization between individual simulations drifts for unknown reasons. In addition, the simulation does not handle reliability issues effectively or records an error log when something goes wrong with the system. These artificial behaviors could potentially affect data results for the batch studies.
3 THE AGILE APPROACH

This section analyzes the existing demands of everyday development tasks (system maintenance, demonstration capabilities, current experiment support, new modeling additions, and system wide architectural changes) of ATOS and presents a set of agile software development practices best suited for this simulation.

3.1 Introduction

Why does the ATOS development team need a new development methodology to support the simulation? Would a change in methodology support any future plans of the simulation? What has changed in the makeup of the team that would justify a change? This section answers these types of questions and presents a set of agile practices that will address the needs of the team from this point forward.

3.2 Changes to the development process

Many aspects changed in perspective to the make-up of the ATOS development team since the inception of ATOS several years ago. Development has undergone two distinct phases in terms of the size of the team and its operating budget. These changes justify a fresh look at the software development methodology being used. During the beginning phases of ATOS development, the team consisted of more than thirty individuals (spread out over five different locations, the majority centered in one office building), who were directly responsible for the design, development, and integration of the ATOS system. A substantial budget was in place to support a large Capability Maturity Model Integration (CMMI) based software development methodology, complete with the proper tooling and
enforcement personnel to verify that the methodology procedures were indeed being followed. The main focus was on the initial design and integration of a complex framework to support a set of software components [28] that supported NASA's advanced, distributed air-ground traffic management concepts [1], which was under the direction of NASA's Advanced Air Transportation Technologies (AATT) project [2]. The development work also included the assembly of several heterogeneous pieces of existing software into the newly developed system.

Compare the above to the current development team of the ATOS system. After completing the AATT project, a significant reduction in budget and team size occurred when the initial start-up of NASA's new NGATS vision [27] took place. The team was still spread out, but in fewer locations. The new budget did not allow for purchasing new tools, equipment, or hiring personnel unless it was deemed critical to an actual experiment underway. Everyone assumed that the old CMMI software development methodology would still be followed, but CMMI noncompliance was not addressed. The development phase also changed as ATOS was considered to be in its mid-life. Other issues of importance included stability of existing code, expansion of the system to support a wider variety of experiments, coordination of the consistency between heterogeneous components, enhancement of the simulation's modeling capability, and comprehensive architectural changes.

3.3 Need for a new methodology

Without enforcement of the old CMMI based methodology, the development was transitioning into an ad-hoc approach. Attrition of key members left several development roles open (an architect, a systems engineer, a configuration management specialist, and a
test and integration specialist), leaving the remaining team members in a state of confusion in terms of CMMI compliance. The old processes such as code reviews, defect tracking, regression testing, and documentation were being dropped, and accountability of many of these processes was being ignored.

NASA still had numerous experiments planned out into the future, with several projects requiring significant changes to ATOS. However, without a development methodology in place it was extremely difficult to design and implement any future plans. Part of this difficulty is attributed to ineffective communications on how exactly to achieve these goals. In addition, the sense of architectural direction was rapidly fading, and developers didn't know how their work would fit into the simulation's future.

Resurrecting the old CMMI approach was not feasible since the people who understood CMMI had left the project, and the lack of training with the remaining individuals prevented them from championing an approach that was not a developer-driven methodology. Typically, management drives CMMI methodologies as a way to control and monitor the progress of a project [41]. However, since the budget had been drastically cut, the additional costs of building up the CMMI approach would have made it too difficult. The processes associated with the old methodology would all have to be reanalyzed, further increasing the costs.

Yet, there was a reality that something needed to be done. The ad-hoc approach would eventually slow down the major architectural changes so desperately needed. The benefits to using a software development methodology are described best in Berard's [42] paper, "What Is a Methodology", by using the term "engineering" instead of "software development." Nevertheless, his theory is still applicable: "Probably the most important
idea behind engineering is that one can systematically and predictably arrive at pragmatic, cost-effective, and timely solutions to real world problems."

Berard further describes that the most worthwhile engineering techniques (i.e. software development methodologies) are those that are described quantitatively and qualitatively, used repeatedly so that similar results are achieved, taught to and applied by others within a reasonable timeframe, consistently achieve better results than an ad-hoc approach, and are applicable to a relatively large percentage of development types.

3.4 Methodology requirements for ATOS

Not one single software development methodology fits all types of situations. Flexibility is an important choice of software development, and Ambler [43] lists a few reasons:

Different technologies used will require different techniques to handle.

Every individual is unique, with regard to background, preferences, and cognitive style and cannot be considered a replaceable part.

Every team is unique, simply because it is made up of individuals.

Every project's external needs vary, such as conforming to government regulations.

3.4.1 Principles

When selecting a project's development methodology, there are numerous factors to consider, such as the group's size, the system's criticality, the project's budget constraints, and the team's communication methods. Cockburn [44] has grouped these factors into four principles:

**Principle 1:** A larger methodology is required for a larger group.
**Principle 2:** The more critical a system is, the more need for public visibility into its correctness.

**Principle 3:** Relatively small increases in methodology size will add relatively larger project costs.

**Principle 4:** Face-to-face and interactive communication is the most effective form of communication.

The following paragraphs will take each one of these principles and apply them to the needs of the ATOS development team.

### 3.4.1.1 Principle 1: Group size

One of the primary objectives of a software development methodology in supporting a simulation is to coordinate people. More people require more coordination. Coordination is achieved by various methodology elements, such as roles, work products, reviews, standards, etc. Any time a team member joins or leaves, group communication between team members needs to adjust accordingly. Because the ATOS team is small, it can utilize a very light methodology.

### 3.4.1.2 Principle 2: System criticality

Cockburn describes four levels of system criticality [44]: loss of comfort, loss of discretionary moneys, loss of irreplaceable moneys, and loss of life. Each level justifiably requires a larger development expense. Consider a system where a failure simply means that a few people are in an uncomfortable state. In this case, it makes sense that the methodology can be more relaxed in an effort to cut costs. On the extreme end of Cockburn's criticality levels, a failure means there is a potential for somebody to
lose his or her life. By all means, the software development methodology should go to
great measures to keep defects from creeping into the system. Costs for such a
methodology are justifiable as it involves a loss of life.

ATOS is a research simulation operated by a small group of talented individuals who
test the simulation repeatedly in preparation for an experiment until the system presents
reliable results. If at any time an error occurs, a team member will look into why it
occurred and produce a fix or a work-around to the problem. In relation to Cockburn's
system criticality levels, a failure on the ATOS system indicates a loss of comfort;
therefore, the software development methodology can stay light in this regard.

3.4.1.3 Principle 3: Size of methodology related to cost

In this principle Cockburn is stating larger methodologies require greater costs. He does
not question whether the coordination activities and deliverables associated with a larger
methodology are beneficial. Cockburn is simply stating that these activities have a cost
and lesser methodologies results with lower costs.

The original ATOS development team had a much larger budget than the current
team. However, that does not mean that NASA wants less from this team. On the
contrary, more experiments are planned, with a shorter duration between experiments. In
addition, more sophisticated features and models to the ATOS system are required than
ever before. However, with a smaller budget to work with, it doesn't make sense to
produce the same level of documentation and keep up the same level of coordination
designed for a large team. Keeping the software development methodology as lean as
possible will cut down on the costs of maintaining the methodology.
3.4.1.4 Principle 4: Effective communication

If productivity and costs are key issues to a development project, then effective communication needs to be applied to the group. Physically locating the team close to each other helps so that they can easily work in small groups. Avoiding the use of extensive documentation will allow the team to concentrate more effectively on their tasks.

The total development team for ATOS spreads out over several contractors at three site locations. Each site has team members in close proximity. However, the communication level between sites needs to compensate for the distance factor, requiring the size of the methodology to slightly increase.

3.4.2 Why agile for ATOS

How suitable an agile practice is to a development team can be looked at from multiple perspectives. From a product perspective, agile practices are more suitable when requirements are emergent and rapidly changing. Agile practices are less suitable for systems with high criticality, and reliability and safety requirements (although there is no consensus on this point [6]). From an organizational perspective, the suitability can be assessed by examining three key dimensions of an organization: culture, people, and communication. In relation to these areas, a number of key success factors have been identified [6]:

The culture of the organization must be supportive of negotiation.

People must be trusted.

Utilize fewer but more competent people.
Organizations must cope with the decisions developers make.

Organizations need to have an environment that facilitates rapid communication between team members.

Project size is the most important factor [6]. As size grows, face-to-face communication becomes more difficult. Therefore, agile methods are more suitable for projects with small teams with fewer than 40 people.

The development team for ATOS has changed from a large group of computer programmers and engineers, all with a variety of backgrounds and experience levels, into a small group of experienced engineers each with extensive simulation and programming backgrounds. The current team understands the research concepts and has a close relationship with the research team. Furthermore, there is extensive interaction with off-site development groups whose components are tightly integrated into the system.

Agile methods work best on projects where high levels of change are expected. Changing requirements, technological uncertainty, the need to experiment, and evaluating different approaches are factors that indicate agile methods are a good choice [45].

Research for large problems such as the future of air traffic management is, by its nature, an exploratory endeavor. As researchers gain new insight into their investigations, new avenues for inquiry come into view. This dynamism means that new requirements for ATOS support are constantly emerging. Cooperation and competition can also change priorities. ATOL has conducted and will conduct several cooperative experiments with other research labs. These experiments will always cause the priorities and the requirements of development to change. Similarly, keeping pace with other researchers with aspirations to publish novel work can also drive changes.
3.4.3 Plan-driven methodologies verses agile methodologies

The two most common software development methodology categories are predictive and adaptive and each has unique characteristics. While some authors try to demonstrate that the two can co-exist [46, 47], others believe there is a fundamental philosophical difference between the two. The differences infiltrate all levels, such as McMahon's approach to scheduling and designing projects [48]. McMahon suggests that when planning out a schedule with a traditional project, similar past projects staffed with people of similar skills are used to determine the new schedule. The agile approach is much different. Instead of predicting schedule performance based on results from a different project with different people, the plan is based off the current team's velocity and performance. Agile practices do not limit design to a fixed time slot within a fixed phase. Agile encourages deferring design details until the optimum time (e.g. when data is available) and the important requirements have been clarified to minimize rework and maximize team velocity.

Because software development is different from the traditional engineering disciplines, the development methodology should also be different. However, many proponents of the traditional approaches try to correlate these two as suggested by Fowler [49]. Fowler discusses the differences between plan-driven (engineering) methodologies and agile methodologies. The engineering methodologies are designed to impose a disciplined process upon software development with the aim of making development more predictable and efficient. Fowler uses the civil engineering discipline as an example of how bridge design is completed before the construction begins. The design team requires expensive and creative people who can produce a construction plan that
can be implemented by less-skilled people. For a bridge building project, the construction phase is always longer and requires more people than the design phase. Because of this, it is beneficial to use the lower cost people in the construction phase.

In software development, Fowler uses Reeves' claims [50] that source code is actually a design document, and the construction phase is simply using the compiler and linker. Fowler came to the following conclusions:

- In software development, construction is so cheap it can be considered to be free.
- In software, all the effort is in the design, thus requiring creative and talented people.
- Creative processes are not easily planned, and therefore not as predictable.
- The traditional engineering metaphor for building software really does not apply to modern software development and requires a different process.

In other differences, Cockburn and Highsmith claim that rigorous processes are designed to orient people to the organization, while agile processes are designed to capitalize on each individual and each team's unique strengths [51]. While Constantine writes about how agile methods enable clients to begin using a simplified working core with limited but useful capability early in the project, traditional approaches require the design to be complete before coding begins [52]. Finally, a comprehensive comparison was conducted between agile and Tayloristic (traditional) methods by Chau et al. [53] and highlights the many differences Chau's team found between the two approaches.

Overall, agile methods separate themselves from traditional software development practices with agile stressing frequent and incremental delivery of functional software through intense collaboration. The traditional approach attempts to be predictive and flows the design to the development team. Due to the nature of ATOS development, a
predictive approach is not feasible. However, the benefits of agile are greatly pronounced.

3.4.4 Commonalities between agile methodologies

Many sources depict the commonalities between all the various agile methodologies listed in Section 2.1. A complete comparison between all commonalities would go beyond the scope of this thesis. In a rather extensive write-up on the subject by Cohen et al. [6], the authors claim that the common characteristics of agile methods include: iterative development, focus on interaction, communication, and the reduction of resource-intensive intermediate artifacts. Fowler has also provided a concise but detailed overview of agile commonalities in his article, “The New Methodology” [49].

Constantine writes [52] that agile is a philosophy, and has a set of practices that takes genuine discipline to follow. He continues with this statement that best summarizes the commonalities between agile methodologies:

The rules of the agility game are relatively simple. Work in short release cycles. Do only what is needed without embellishment. Don't waste time in analysis or design, just start cutting code. Describe the problem simply in terms of small, distinct pieces, then implement these pieces in successive iterations. Develop a reliable system by building and testing in increments with immediate feedback. Start with something small and simple that works, then elaborate on successive iterations. Maintain tight communication with clients and among programmers. Test every piece in itself and regression test continuously.

In summary, the ATOS team does not need to follow any particular agile methodology to make this simulation successful. The ATOS team only needs to keep the agile philosophy in mind to reap the benefits of agile.
3.4.5 Agile is for people

Agile is an approach to software development that is oriented around people as stated by Ambler [43], who believes that agile enables people to respond effectively to changes of a working system that meets the needs of its stakeholders. Because the ATOS team is staffed with talented individuals, it makes sense to enable them with decision making authority. In order to keep costs down, the team needs to self-organize and exploit the strengths of each individual. Any software development methodology that encourages this approach would benefit the overall project. In an article titled “Agile Software Development: The People Factor” [51], Cockburn states: "... people working together with good communication and interaction can operate at noticeably higher levels than when they use their individual talents. We see this time and again in brainstorming and joint problem-solving sessions."

In their book, Buckingham and Coffman discuss the outcome of a long-running research program by the Gallup organization, in which 80,000 managers in 400 companies were interviewed over a 25-year period [54]. They evaluate the interplay between talent, skill, and knowledge among managers in high-performance working environments. In their research, they found major differences between organizations that legislate the process of performance versus organizations that are people-based whose objective is not the steps of the journey, but rather the end results. The distinction between the two types of organizations is real. Step-by-step organizations are designed to battle the inherent individuality of each employee, whereas strength-based organizations are designed to capitalize on individuality.
People are not commodities. An individual from a team simply cannot be pulled out
and replaced by another individual to produce the same results from that team [46].
Agile requires development to capitalize on the people's strength. Fortunately, the ATOS
team has the right people to use an agile approach and to capitalize from it, benefiting the
overall simulation.

3.5 Agile practices best suited for ATOS

This section looks at several typical practices of any agile methodology. Most of the
individual practices of agile are not new to software development because they have been
successfully incorporated in other methodologies. In an interview about his book
Managing Agile Projects, Aguanno elegantly states [45]:

Agile methods are not all that radical—these techniques have existed for
many years, and indeed are just evolutions of what previously existed. When you look at the history of agile, you will see that the techniques we
have packaged into our named methods (XP, Scrum, etc.) were used long
ago; we have just grouped them together and formalized their interactions. We have cobbled together methods from existing techniques—there is
little revolutionary in the techniques, but much in the thinking behind the
methods. Agile methods are not a mental revolution, but a technical
evolution.

The number of agile practices are numerous, and most likely very difficult to
implement all known practices into a methodology. Extreme Programming [55] has the
most written about the various "rules and practices" (as an XP user would call them).
This section only concentrates on the practices that were found useful in ATOS
development to support this robust simulation.

The usage of the following practices is powerful in terms of software development.
An abundant amount of research examined the "hows" and "whys" of these practices at
work. Following is a list of practices that can be used successfully by the ATOS research software development team:

- Continuous integration,
- Delayed decisions,
- Good enough software,
- Iterative development,
- Refactoring,
- Simplicity, and
- Sustainable pace.

Section 4.0 will illustrate the effectiveness of these practices as they are used to drive ATOS towards their development goals.

### 3.5.1 Continuous integration

Fowler defines continuous integration as an agile practice where members of a development team integrate their work frequently into a single source repository, leading to multiple integrations per day [56]. Each integration is verified by an independent, automated self-testing build machine to detect integration errors as quickly as possible. All results of the build and self-test can be viewed by all. Finally, each integration can be easily installed and tested as a whole in a cloned production environment by any team member.

Continuous integration is not a tool; rather, it is an attitude shared by the team that the latest code from the repository will always build successfully and pass all tests. Each individual on the development team will always check in their code every few hours and
commit to the health of the code base. The biggest benefit of continuous integration is the reduced risk [56] when compared to deferred integration. If a bug is introduced into the system and immediately detected, it is easy to determine which code produced the bug because only a small change was made to the simulation. One of the main benefits of projects that practice continuous integration results in significantly fewer bugs [56].

Like most agile practices, continuous integration is not a totally new concept. In fact McConnell [57] wrote about the benefits of a daily build and smoke test back in 1996. Understanding the importance of keeping the build working, McConnell wrote the following: "Make it clear from the beginning that keeping the build healthy is the project's top priority. A broken build should be the exception, not the rule. Insist that developers who have broken the build stop all other work until they've fixed it. If the build is broken too often, it's hard to take seriously the job of not breaking the build."

Many teams find that the continuous integration approach leads to significantly reduced overall integration problems and allows a team to develop cohesive software at a rapid pace [56].

3.5.2 Delayed decisions

Agile practices encourage delaying irreversible decisions until the last responsible moment, because this is the point when the most information is available on which to base the decision. Although controversial, the key focus is on the "responsible moment," not the last "possible moment." Designs emerge as they develop a growing understanding of the problem; they do not emerge from collecting mass amounts of requirements.

In his essay "Delaying Commitment," Harold Thimbleby observes that the main difference between amateurs and experts is that experts know how to delay commitments
and conceal their errors for as long as possible, and then repair flaws before they cause problems [58]. Amateurs try to get everything right the first time, thus overloading their problem-solving capacity so that they end up committing early to wrong decisions.

Szalvay explains other benefits gained by delaying decisions [59]: "Many software development teams that implement Agile software development are finding they get something they never expected: options. Rather than locking into decisions at the beginning of a project, organizations can reduce risks by leaving options open to decide at a better time when more accurate information is available."

The development team isn't the only group who can delay its decisions. Agile also allows the customer to delay decision making. Often times, the customer don't know exactly what they want, and by observing the system under development, they can make better informed choices about the future direction of the project. With the iterative development practice of agile, a delayed decision by the customer can be easily implemented on the next iteration.

### 3.5.3 Good enough

One definition of agile developed software is that it's just barely good enough [60] upon its release to the customer. If the software is not yet good enough, then there is more work to do before it is released. On the other hand, if the software is more than good enough, then too much time was invested in software development and the customer's money has been wasted.
Ambler [60] has several points about when one can justify if the software is just barely *good enough*. *Software should be:*

Fulfilling a purpose,

Understandable to the audience,

Sufficiently accurate to meet the intended purpose,

Sufficiently consistent for users to work with,

Sufficiently detailed to give the information required,

Provide a positive value to solve the problem, and

As simple as possible.

The real key to *good enough* software is that understanding what the customer needs and to give them no more or less than that. Bach [61] states: "To be good enough is to be good at knowing the difference between important and unimportant; necessary and unnecessary."

When solving problems for the customer, a case-by-case decision needs to be made for each problem on whether it is important to solve or fix. Bach believes that in the eyes of the stakeholders, something is *good enough* when the potential positive consequences of software acceptably outweigh the potential negatives.

*Good enough* software fits well with agile development, which has fast iterations of releases to the customer. Each release supplies the functionality needed for that release and no more. More functionality can be implemented for later releases if it is required for that release. Yourdon [62] argues that late software is never better than on-time software, and that *good enough* software helps releases to be on time. A project that delivers late software with unnecessary features is not doing anybody a service.
3.5.4 Iterative development

Iterative development is not new to agile; Larman and Basili wrote a paper about the long history of this topic [63]. The approach of iterative development is one of continuous discovery, invention, and implementation, with each iteration forcing a development team to drive to closure a project's many features in a predictable and repeatable way.

Martin [64] writes that the waterfall method [65] taught development teams to think before they started coding. Martin believes it is more efficient to think in small increments and then use code to verify that the thinking was correct. The feedback from each coding cycle produces the next level of discovery about the problem. Martin continues with the argument that people are not capable of holding entire complex systems in their minds. Rather people need to be able to examine, explore, and evolve complex systems. Doing analysis without this feedback is nothing more then an educated guess into the problem solution. The more complicated an application is, the more feedback is required.

Kruchten [66] looks at the iterative process from a risk point of view. The team reduces risks earlier in the development process because typically it's during the integration phase where risks are discovered. Kruchten further explains that it's the early iterations that allow the team to exercise many aspects of the project, including tools, off-the-shelf software, and people skills. Perceived risks will prove not to be risks, and new, unsuspected risks will be discovered. Kruchten explains that iterative development allows the discovery of misguided course directions to be discovered early on, before a lot of time and effort are expended in up-front analysis and design.
3.5.5 Refactoring

Refactoring is a disciplined practice for altering the internal structure of an existing section of code without changing its external behavior. At the core is a series of small behavior preserving transformations. Each transformation, also called a refactoring, does very little, but a sequence of refactorings can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can break during the alternations [67].

The practice of refactoring is powerful, and a lot of research is devoted to this subject. Both Mens et al. [68] and Du Bois et al. [69] discuss the current set of refactoring applications and tool support, and also the direction refactoring research is heading. The decluttering of code is the most common type of refactoring used (see the next section for an example). However, refactoring can also be used for feature reduction and simplification.

3.5.5.1 Cluttered code

Refactoring does not require an understanding of what exactly is being done with the code. The goal is to keep the logic flow the same, while at the same time making the code more readable. Simple decluttering techniques can be applied to any code such as that shown in Figure 5. Upon inspection of the code, it's not immediately understood what the routine's responsibility is. There is very little structure to the code, as it appears to be all clumped together, and has single letter variable names, which does not offer useful information.
int ClassA::MethodX(int x)
{
    int r, i, s;
    if(x<5){r=x/5;
     for (i=0;i<5;i++){s+=i*r;}
    }
    return s;
}

Figure 5. Typical cluttered code example

Refactoring cluttered code without changing its logic flow can allow programmers a new insight into what the code actually does, and even allow discovery of potential harmful flaws. Figure 6 shows the same code as in Figure 5, but it has been refactored without changing the logic flow. White space is inserted in the code, along with variable renaming to make better sense of the code. In addition, initial values were assigned to the local variables to give some sort of meaning before they are used, rather than leaving it up to the compiler to decide. This first refactoring might not inform the programmer exactly what this routine is trying to accomplish. However, the increased readability allows the programmer and future programmers to visually inspect the code and gain a rudimentary understanding of its purpose. As the programmer learns more about the application, further refactorings could change variable names again to actually match newly understood functionality along with comments that state the real purpose of the method.
int ClassA::MethodX(int nInput)
{
    const int MAX_VALUE = 5;

    int nRatio = 0;
    int nSum = 0;

    if (nInput < MAX_VALUE)
    {
        nRatio = nInput / MAX_VALUE;

        for (int i=0; i<MAX_VALUE; i++)
            nSum += i * nRatio;
    }

    return nSum;
}

Figure 6. Refactored code example

Finally, by refactoring the above code, a potential source of unexplainable error was eliminated. The original code defined the return variable "s," but did not give it an initial value. Although not a factor when the code was first written, the programmer might have known that the routine always calculated the value "s." However, changes made by other programmers (i.e. addition of the "if" statement) has created a situation where "s" could be undefined if the input value were greater or equal to five. The refactored code eliminated this hole by simply defining an initial value for all variables. Poor coding can cause unexplainable errors in the simulation models and applying declutter refactoring techniques often finds these errors without relying on a debugger.
3.5.6 Simplicity

Agile processes introduce the idea of simplicity, meaning never produce more than what is necessary for new development. Only concentrate on the functions initially needed and wait until a current requirement dictates an extra layer of complexity before introducing it. Simplicity is often lost when continuously adding new functionality to existing software. Wirfs-Brock [70] believes, whether accidentally or out of necessity, that developers tend to tolerate this loss of simplicity. Wirfs-Brock states: "You can make complex solutions more manageable by spot-reducing the complexity of any single design element: Simplify overly long methods. Break them into smaller, more understandable ones. Refactor complex behaviors into new classes. Redesign interfaces to make them simpler."

Other techniques that follow the simplicity principle are removing old functionality if it isn't needed, refactoring internal naming conventions to allow for better readability, and finally refactoring logic for ease of maintainability.

3.5.7 Sustainable pace

One of the principles behind the Agile Manifesto [4] is that agile processes promote sustainable development, which means that the sponsors, developers, and users should be able to maintain a constant pace indefinitely. A sustainable pace is one of the 12 dynamic practices of Extreme Programming [22] and requires hard work at a pace that can be sustained indefinitely. Langr [71] explains that an eight hour work day is the maximum amount of energy a person has to burn, and that this energy can easily be replenished after a full night's sleep. Langr further explains that the notion of an eight
hour work day has proven sustainability over long periods of time. By following the sustainable pace practice, a development team can plan out what they can accomplish in a work day and strive for that goal.

A team can only produce so much regardless of its size. XP uses a concept called the team velocity, which is the team's speed to develop new features. Velocity changes when new members join or old members leave the team. As the team gains more experience, the velocity increases. The team’s velocity is considered when planning new simulation features. Langr [71] points out that when the simple practices of agile are followed, an emergent behavior arises that allows quality software to be predictably and successfully delivered at a sustainable pace over long periods of time.

3.6 Other similar research simulation projects

The agile practices defined in the above section can easily be used for any research-oriented simulation project, especially with a large complex distributed simulation implemented on a variety of computing platforms. Where responsibility is held by a small senior team of engineers, who are driven by constantly changing requirements, and development work is split between maintaining and enhancing legacy software, along with designing and developing new software.

NASA Ames Research Center is such a facility that utilizes multiple distributed-simulation systems that provide real-time, HITL environments to evaluate Air Traffic Management (ATM) concepts that could benefit from a like set of agile practices as mentioned in this research. The Virtual Airspace Simulation Technology Real-Time (VAST-RT) project [72] combines a multitude of other smaller projects, such as the FutureFlight Central (FFC) simulator [73], the Crew Vehicle Systems Research Facility
(CVSRF), the Flight Deck Display Research Lab, and the Airspace Operations Lab (AOL) [74].

In addition, there are a variety of other institutes and industries that cooperate to develop similar systems that could benefit from agile, such as the following:

The Dutch national collaborative SIMULTAAN project has a flexible, re-usable component based Flight Simulator [75].

The MITRE Corporation's Center for Advanced Aviation System Development (CAASD) ATM Laboratory [76].

3.7 Agile projects in similar environments

Most agile development methodologies are designed for typical business and Web-based applications, with many published case studies or experience reports based upon these applications, although there are a few that are similar to ATOS development. Listed below are three projects that have implemented agile techniques.

Wood and Kleb [77], two NASA Langley Researchers, conducted a study evaluating if Extreme Programming would work in a research environment. Their study was designed to explore the possibility of extraordinary gains in productivity, or enable entirely new applications using nontraditional methodologies, for the field of aerospace engineering research. The results of their study indicated that the Extreme Programming approach to software development was approximately twice as productive as similar projects undertaken previously by the researchers.

A U.S. Army simulation program, known as the One Semi-Automated Forces (OneSAF) Objective System (OOS) software development methodology blends agile and
traditional methods [78]. The techniques used for their processes come from various agile approaches, including Extreme Programming. In addition, they are certified at Capability Maturity Model Integration (CMMI) Level 5 and have all the development processes well documented. The commonalities between the OOS simulation and the ATOS simulation are numerous, such as the OOS simulation is used extensively for analysis and experimentation. The program utilizes technically qualified engineers, from multiple contractor companies, with advanced degrees in software-related disciplines and years of real engineering experience. The majority of the team are also located in one facility along with the customer representatives.

A development team, similar in nature to the ATOS team, is working on software for the Genomics and Bioinformatics Group in the Laboratory of Molecular Pharmacology of the National Cancer Institute. The team successfully introduced an agile approach to their software development methodology [79]. While the development of ATOS supports the research of the domain of Air Traffic Management, Kane's group supports the development of applications for the research of a large and complex domain of biology cancer. In both facilities, a group of researchers depend upon the tools created by the development team to be consistent and reproducible so that the results can be included in scientific publications, and can pass the scrutiny of peer review. The agile methodologies used by Genomics and Bioinformatics group are necessary because of the nature of scientific research, particularly in that the effort is characterized by dynamic requirements. The ATOS simulation is also characterized by this same reasoning for dynamic requirements.
3.8 Does agile work?

Often times, practitioner’s of agile methodologies are the ones who make the most claims on the effectiveness of agile. Mark Johnson, Principal of Shine Technologies, states [80]: "Our experience is that any methodology is only as good as the people using it, and this is a central belief of all Agile processes. In addition, our experience has proven that smaller teams of great people can achieve results far beyond those achieved by larger teams. It is our belief that Agile processes reap the greatest results with these small, highly skilled teams. That is why Agile works for us."

Other agile proponents are known to give a lot of rhetoric on the subject and make unfounded claims regarding the benefits of agile processes [81]. Questions are always being asked about who is doing agile, how are they doing it, and are they actually benefiting from it? Lindvall, et al. [82] answers these questions by collecting and analyzing empirical evidence about the effectiveness and classification for agile projects. They examined the results of an expert discussion panel and made an attempt to collect the experiences of a successful agile team. A summary of these findings follows:

Any sized team can be agile if attention to communication is addressed.

Part of the team make-up must be experienced with past development of a similar system.

Agile methods require less formal training than traditional methods.

Attention must be paid to culture, people, and communication in order to be successful.

Refactoring should be done frequently. Large scale refactoring is more feasible using agile methods then traditional designs’ up front approaches.
Significant architectural changes do not need to be risky if a set of automated tests is maintained.

Documentation should be assigned a cost, and the extent should be determined by the customer.

Several surveys on agile have been published, including one by Shine Technologies in 2003 [80], which asked 10 questions to 131 respondents. Shine Technologies found 95% of respondents believed that costs due to agile were the same or less compared to using a more traditional method. They also found that 93% of projects cited improved productivity, and 88% cited improved quality after applying agile techniques.

VersionOne [83] conducted a survey in 2006 that had 722 respondents from several industries: financial services, health care, education, video game entertainment, government, and defense. The main goal was to highlight the value of agile development teams in terms of productivity, defect reduction, time-to-market, and reduction of costs. The most notable findings were that close to 66% of respondents belong to software organizations of 250 people or fewer. Agile teams experienced 10% or more improvement in the following areas: accelerated time-to-market (86% of respondents), increased productivity (87% of respondents), and reduced software defects (86% of respondents). Finally, 81% of respondents considered the agile projects within their organization as “somewhat successful” or “very successful,” compared with only 29% for non-agile projects.

One of the largest and most recent surveys about agile’s success was conducted by Ambler [84] with a response from over 4,000 IT professionals. Ambler’s survey was
modeled after the earlier Shine Technologies 2003 survey [80]. Ambler's March 2006 survey found:

65% of the respondents work in organizations that have adopted at least one agile development technique. 

60% have experienced increased productivity using agile techniques. 

66% have reported an increase in the quality of their product.

58% reported improved customer satisfaction from using agile techniques.

3.9 The growth of agile

Agile has been gaining momentum, as witnessed by the numerous conferences in recent years. These conferences allow participants to submit research papers on agile topics and a chance to exchange ideas and thoughts with each other. Two recent conferences that take place annually are the Agile 2007 Conference in Washington, D.C., [85] and the Agile Business Conference 2007 in Europe [86].
4 RESEARCH PROJECT

This section describes the validation of the agile methodology plan examined in Section 3 by demonstrating significant improvements to the reliability of the ATOS system, while adding functionality for an upcoming experiment.

4.1 Introduction

The goal of this research project is to use a set of agile practices to increase the reliability of the Airspace and Traffic Operations Simulation (ATOS) while at the same time increasing the system’s functionality required for an upcoming NASA experiment. The end result is an acceptable simulation that is fully functional and reliable enough to start experimental trails. Due to the small development team size, the existing software methodology would have been too heavy to successfully implement the same level of reliability and functionality in the allotted time span.

This research project starts with an introduction to the NASA planned experiment, along with all the required modeling and functionality changes. Following is a section highlighting the reliability problems with ATOS at the beginning of this project. The next section explains what approach was taken to address these reliability problems and examines the results of this approach.

This section concludes with a brief synopsis about the extension of the NASA experiment and the continued progress of using agile practices since the end of the research period.
4.2 Safety Performance of Airborne Separation experiment

Researchers at NASA Langley’s Air Traffic Operations Laboratory (ATOL) conducted a Safety Performance of Airborne Separation (SPAS) simulation experiment in the summer of 2007 [87]. The SPAS experiment's main goal was to investigate the effect of traffic demand on safety performance of distributed airborne separation, while adjusting various variables that have the potential to impact system safety. A series of Monte Carlo simulation runs performed on the ATOS platform were designed to analyze and quantify safety behavior of airborne separation.

![Diagram of scenario design](image)

**Figure 7.** Scenario design - Consiglio et al. [87]

A series of simulation runs consisted of scenarios that randomly placed aircraft, each starting with a straight random route traversing a circular test area that represents a generic, high density, en route air traffic control sector (see Figure 7). The routes are
defined by three fixes, where the first and second fixes are randomly placed upon the outer circle and the corresponding opposite side of the inner circle, respectively. The third fix was created 500 nautical miles away from the first fix, outside the test region along a straight line route created by the first and second fixes. A time constraint, or required time of arrival (RTA), was placed on the third fix; thus, the aircraft was required to fly at a speed to have its predicted time of arrival match the RTA. This entire study was conducted at a constant altitude.

The reasoning behind this setup was to allow aircraft to self-detect conflicts with other aircraft and to resolve those conflicts via the usage of the Autonomous Operations Planner (AOP) [36] without any two aircraft penetrating the minimal Federal Aviation Administration's (FAA) required loss of separation (LOS) zone of five nautical miles and to continue to meet their assigned RTA at the third fix. Each aircraft would initialize somewhere on the outer circle boundary and fly through the Initialization Region into the Test Region, where the experiment metrics were collected (see Figure 7). All aircraft were terminated as soon as they exited the test region and reinitialized for reinsertion on a new random route.

4.2.1 New system requirements

In preparation for the SPAS experiment, the existing ATOS system needed to undergo critical changes of such a magnitude that were never accomplished before by the small development team. In addition, reliability problems that were tolerated in the past needed to be aggressively addressed. The old ATOS system, consisting of multiple medium-fidelity aircraft simulators referred to as Aircraft Simulation for Traffic Operations Research (ASTOR), never had more than 12 simulators in operation at one time, never
ran a scenario longer than 90 minutes, and never had to recycle an ASTOR back into the same scenario. The new system requirements involved synchronizing up to 96 ASTOR simulators, with scenarios lasting up to 16 hours, and each ASTOR recycling dozens of times per scenario.

The entire lab's scenario generation capability needed a new architecture, because the old system was only capable of slowly producing limited scenarios of 12 aircraft or fewer, and required manual operation for every level of usage. The new scenario generation system was required to create scenarios involving thousands of aircraft quickly and automatically. In addition, this new system needed to be backwards compatible with the old system's input and output files.

A real-time scenario generation application was required to complement the static scenario generation capability. The application consisted of a model of the SPAS experiment airspace and was immersed into the HLA messages between each ASTOR. The responsibilities of the real-time scenario generation application were to:

Launch each ASTOR simulation at the appropriate time,
Create a customized mini-scenario for each ASTOR that is randomly positioned on the outer circle (see Figure 7) and initially conflict-free with the other ASTOR aircraft,
Handle the completion of each ASTOR flight through the inner circle in order to reuse this ASTOR for the next flight, and
Monitor the health of each ASTOR in order to preserve overall system health.

Several major component models of the ASTOR simulation needed enhancements and refinements to meet the needs of the experiment. Some of these components were
controlled by outside contractors. However, it was up to the development team to identify those changes, present the requirements to the contractors, manage the work, and test changes when they were made available.

4.2.2 Where to begin

A significant amount of work was required to support the SPAS experiment. If the development team continued to follow the past development methodology (Section 3.2), the time it would take to complete this work would exceed the time given to complete the tasks. The team needed to work smart and utilize the agile practices discussed in Section 3.5. The team needed to come up with new architectures, design and develop new simulation models, modify existing models, test changes for normal and stressed operations, and verify backwards compatibility with original simulation modeling and functionality.

The sum of all requirements pulled from the planned usage of ATOS placed new demands on the system that have never been addressed before. Although the ATOS system has performed quite well for past experiments, it was due to the work of a talented team of developers who put this system together and made it work. The development team has been drastically reduced since the last major simulation experiment preparation and most individuals were not familiar with the inner workings of the software base. Nobody could pinpoint the problem areas, what was weak and required replacing, and what was strong and could perform successfully with just minor changes. The demands of fulfilling these requirements would require a clear approach and one that could be handled with the skill set of the team. Many of these requirements touched common
areas of the system with known problems that needed to be addressed before they could be justifiably enhanced.

4.3 ATOS reliability problems

There are many components that make up the ATOS system, with each component having a job that it must successfully accomplish. The individual component in general tends to be reliable when performing unit tests and simple system tests. Unfortunately, ATOS as a whole quite often fails in many different aspects when running complex scenarios. There are four high-level components of the ATOS system, the ASTOR aircraft, Traffic Generator, Gateway and Simulation Manager. By design, these components also make-up the HLA federates. Each one of these components can be broken down into subcomponents (or models) with the most complex hierarchy of components being the ASTOR as shown in Figure 8. This thesis research project concentrates on the ASTOR component.

![Figure 8. ATOS system architecture](image_url)
The ATOS system is driven by scenarios that dictate which components are to be included, and when components start up and shut down. Before the start of this thesis a typical scenario could contain anywhere from 0 to 12 ASTOR federates, a Traffic Generator, and a Simulation Manager. Each ASTOR is highly configurable by including particular configuration files, which specify what subcomponents (or models) are executed, and what options are activated.

The old ATOS system occasionally failed. Low-level failure rates were accepted on well tested-out scenarios and deemed good enough, but if a failure happened during an experiment, then the Simulation Operator would simply bring down the system and start over again. With the requirements of the SPAS experiment, ATOS had to dramatically improve its reliability problems. The experiment called for up to 96 ASTOR federates running overnight without any human monitoring. Each ASTOR would cycle through dozens of flights throughout the night, and if a particular ASTOR failed, the scenario could not be started over.

The rest of this section highlights some of the most critical reliability problems of ATOS, including start up/shut down, HLA, and timing.

4.3.1 Start up / Shut down problems

The majority of failures ATOS experienced were the startup and shutdown processes, even though the real-time processes of the system were highly reliable. The majority of these problems were believed to be caused by poor or nonexistent handshaking between the various subcomponents of the simulation. However, nobody could prove or disprove that this was the actual cause.
The start up and shut down problems were not consistently repeatable, making it difficult to track down errors. Generally, the system performed successful start ups and shut downs the majority of the time, as shown later in this research project, which meant that the user group of ATOS would just tolerate these failures. If a failure occurred at start up the Simulation Operator would simply restart and usually the second attempt at running the simulation would work. If the failure was upon shut down the Simulation Operator would manually kill any left over components before the next scenario was executed.

Failure to start a particular ASTOR was dealt with on a case-by-case basis. If the ASTOR was critical to the run at hand, and it failed to start, the entire simulation was brought down and restarted. If the ASTOR was not critical to the simulation run, but rather it played a background role, then it would be just left in its broken state until the scenario was finished. The SPAS experiment required that individual ASTOR federates start at various times throughout the run, and that each ASTOR is used multiple times in one scenario. Because of these requirements, the start up and shut down of the ASTOR had to be flawless.

4.3.2 HLA problems

The largest reliability concern with the ATOS system was within the HLA implementation. All the design and development efforts for the HLA code were accomplished by a group of developers who no longer worked on the project. The code base was complex, undocumented, and tied itself into the system architecture. There were several known issues with the HLA interface architecture that caused critical failures to the system.
In the most severe problem, if one federate ungracefully leaves the federation without properly resigning, the whole federation hangs. In the case for the SPAS experiment, this would mean up to 96 ASTOR federates, a Traffic Generator federate, and a Simulation Manager federate would all stop executing and fail to respond to system commands. In the past, a hung ASTOR would be left in that state by the Simulation Operator until the scenario was completed to avoid crashing the entire ATOS. This approach was not sufficient for the SPAS experiment because it required recycling ASTOR federates after each individual run.

4.3.3 Time drift problems

On many occasions, the ATOS system would suffer from time synchronization problems that would cause various federates to drift apart in simulation time. The problem was so severe that by the end of an hour long run, a federate could be anywhere from several seconds to several minutes off from the other federates. Considering that one of the most powerful features of the ATOS system is in its ADS-B modeling [35], this was a critical problem. Each aircraft in the simulation broadcasts its current state and future intentions to all other aircraft in the vicinity. When time synchronization is off, some aircraft receive information from the future, and other aircraft receive information in the past, causing corruption in the collected data.

Since the large time drifts were an occasional problem, the work around was to bring down the simulation, and hope that upon starting up again the time drift would go away. Various theories existed as to why the time drift problem occurred, but due to the lack of proper system logging, a solution was not identified. The SPAS experiment could have
scenarios running up to 16 hours and even the smallest time drift problem would cause serious problems within the experiment, and these needed to be fixed.

4.4 Approach taken

This section highlights the approach taken in this research project for reducing the reliability problems found in the ATOS system while preparing for the SPAS experiment.

4.4.1 Implement continuous integration

At the beginning of this project, the ATOS development team had a practice similar to what McConnell explained in his paper about a daily build and smoke test [57]. On average, the build was installed bi-weekly in a cloned ATOL environment by a specialist from the test group, who would look for new problems introduced by the build and report to the development team for further investigation. Unfortunately, this approach did not allow for multiple iterations per day or allow developers to test a build with only one new feature added per iteration.

To gain immediate benefits from a continuous integration capability as described by Fowler [56], several goals needed to be implemented into the ATOS build system:

- Any developer can kick off a new build at anytime.
- The build must be quick and easily installed in a cloned ATOL environment.
- Every developer must have a mini-cloned ATOS installed on their desktop.
- Developers must have confidence that any build on any day will work.

The development team did not completely follow Fowler's approach to continuous integration [56]; the goal of a self-testing build was left out because of the time and effort
it would take to fully implement. In addition, Fowler also advocates a single-source repository. However, due to the complexities of the current ATOS simulation, it was determined that short-lived test branches to the source repository should also be allowed to exist. Since the ATOS simulation consists of many executables that share the same libraries, a change to one library could affect multiple components. The test branch would allow a complete compile of the entire system and a complete install package to allow system level tests without ever affecting the mainline.

4.4.2 Reduction of useless code

ATOS has a significant code base (assembled by teams of developers for nearly a decade) that is stored in a version control system. Unfortunately, the code base has significantly more code than what is actually used by the ATOS system, as the extra code caused some issues with the current development team. First, it was unknown exactly what code was unused in the working simulation; the unused code included deactivated features, new features that were never quite finished, or simply interesting code a previous developer stored in the version control system. Second, while investigating the code base, the differences between working code versus useless code could not be determined without digging further into the system.

The goal of this phase was to remove any code easily defined as useless code without negatively affecting system functionality. This thesis defines four categories of useless code:

**Dead files:** Complete files containing code not used anywhere in the installed system or test harnesses.
**Dead code:** Code never executed, but contained in files as part of the installed system or test harnesses.

**Dead logic:** Code executed, although no useful results are ever produced from this logic path.

**Bloated code:** Unnecessary excessive code typically created by generic programming solutions.

The benefits from reducing useless code are numerous. Muller [88] states this best by claiming that lean code enables faster development time, enabling faster feedback, which in turn improves the specification, and leads to an even greater reduction of code. Muller continues with the claim that by having a smaller code base, the development team size can also be reduced. Fewer people will mean easier communication, closer proximity of the team, less need for bureaucratic control, and less organizational overhead. All of the above in turn reduces the number of people needed.

The ATOS development team was significantly reduced before preparation of the SPAS experiment when compared to the number of people involved in past experiments. If Muller's claims are indeed correct, then out of necessity the amount of useless code needs to be reduced just to maintain par with past development activities. In addition, the smaller code base should give other obvious benefits such as improved reliability, maintainability, and testability for all current and future work.

### 4.4.3 Identify reliability problems

Many of the applications previously running in ATOL were thought of as a program monitored by a person. With this mentality, when something went wrong with the
application, a person would be present to record the symptoms of the problem and pass this on to the appropriate team member to solve. Quite often, the only method to record bugs came from eyewitness accounts of the chain of events that occurred while the system exhibited problems.

With the shift in focus to the SPAS experiment, where up to 96 ASTOR federates, a Traffic Generator federate, a Simulation Manager federate, and a real-time scenario generator federate, all running simultaneously under no supervision, the human monitoring dependency became impossible. Furthermore, since many of the ASTOR federates are without monitors, it is impossible to visually inspect the health of each federate. Most imperative is that each component running in ATOL is capable of monitoring its own health and to record any problems via the existing log file system.

The log files are vital to gaining insight into any problems that occur within the simulation. In the past, the log files were only viewed after a crash occurred in the system, with the hope that some clues were supplied in the log files to indicate the reason. The goal of this phase was to utilize the log files and actively search for recorded problems. A manual process would be too inefficient at searching; thus, a decision was made to develop a scanner to quickly aid in identifying system reliability problems. The scanner’s job was two-fold, first: it would summarize to one file a list of found errors, and second it would use the list as a benchmark for subsequent runs. The benchmark allowed verification that errors were addressed and new errors did not creep into the system on subsequent runs.
In past experiments, a test team verified changes made to the system, identified new problems, and added problems to a defect tracking application. Due to funding cuts and attrition, this test team was completely dissolved. A new approach needed to be developed, and the hiring of a new test team was not an option. Since agile processes use feedback as their primary control mechanism, the feedback is driven by regular tests and releases of the evolving software. Frequent testing by the development team basically duplicates the service provided by the original test team, thus eliminating the need for a dedicated test team. This thesis will not argue whether a dedicated test team is required for a large complex system such as ATOS. However, when lacking the proper resources to provide for such a team, the agile practices satisfactorily fill in the need.

Most existing agile methodologies promote the usage of automatic unit tests executed after every build cycle, and Fowler writes about this as a key ingredient of the continuous integration practice [56]. A noble goal for the ATOS development team is to implement such a system. However, this requires a significant effort that the team could not afford to take on in this early implementation of agile. The agile practice of good enough needs to be heavily applied to this phase and the task for building up a large automatic test harness can be pushed to a later date.

The research team was renowned for presenting capabilities and advanced concepts of the simulation from previous experiments in the form of demo scenarios. Because these demo scenarios cover most of the functionality that NASA desired, utilizing these demo scenarios as a test approach allowed the quickest implementation of a test plan. The usage of demonstration scenarios for an approach to testing takes advantage of the
fact that the simulation looks noticeably different in either screen display or its results when functionality breaks. With a trained eye and an understanding of the concepts being simulated, errors in the system are easily noticeable. Each member of the development team understands the concepts being studied and has a trained eye required for performing regression tests in this manner. In this regard, the test approach of running demo scenarios for determining if the system still works after every build is good enough in this phase of development.

Another advantage for testing can be exploited with the fact that the SPAS experiment is a batch run, without any requirements of human intervention to run. Test scenarios are created to push the simulation to its limits and log files can record the results. Scripts can be written to harness this information and to look for degradation of performance. A third set of test scenarios are developed for the sole purpose of testing functionality not covered by the demonstration and pre-experimental scenarios. These scenarios can be called on a case-by-case basis depending on the types of changes being made to the system in each development iteration. Finally, some components of ATOS are written by other contractors, in which this team has only a little influence with software changes, making the system-wide test plan an effective means of identifying problems related to those other groups.

4.4.5 Find causes of reliability problems

The remainder of this section ties everything together including: implementing the agile continuous integration practice, reducing large amounts of useless code, implementing a method to automatically scan log files, assembling a comprehensive test plan, and using the prescribed agile practices best suited for ATOL from Section 3.5. All these combined
practices should allow the development team to efficiently find many of the real causes of the simulation's reliability problems, and to quickly design and implement solutions to these problems.

By no means is the intention to solve every reliability issue. The ATOS computing platform is far too complex to address every possible situation. The goal is that ATOS should be expected to work reliably in most situations. However, ATOS does not need to be fail-safe; an occasional crash of the system would be acceptable, such as a power outage or faulty hardware. A reasonable goal would be that during an experiment or demonstration, all known reliability issues need to be solved, unless they are cost prohibited. One of the benefits of the agile approach for dealing with costly solutions is that simple attempts at making the system good enough might be all that is needed.

The remainder of this section demonstrates a baseline of ATOS reliability problems at the beginning of the SPAS experiment preparation, along with a description of some obvious problems that were initially addressed. Following are some ground rules on what is expected in the reduction process and the approach taken, which heavily relies on the agile practices mentioned in Section 3.5.

4.4.5.1 Reliability baseline

The overall reliability of ATOS is essential to the successful completion of the SPAS experiment and in the demonstrations of the system to various VIPs. Due to the many integrated computing platforms of ATOL, each with a multitude of synchronized components, failures occur more often than what would be deemed acceptable. The typical failures of ATOS can be placed into three phases of the simulation: Startup, Operate, and Terminate. Table 1 shows all critical failures in ATOS tabulated by
Startup/Operate/Terminate over a two week period in August 2006, using a string of five ASTOR simulation stations. A critical failure is defined as a condition that abruptly halts one of the system's components, causing the simulation as a whole to either hang up, or cause a corruption in the modeling.

**Table 1.** Critical error count—baseline

<table>
<thead>
<tr>
<th>Date</th>
<th># Runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>08/07/06</td>
<td>60</td>
</tr>
<tr>
<td>08/08/06</td>
<td>150</td>
</tr>
<tr>
<td>08/09/06</td>
<td>210</td>
</tr>
<tr>
<td>08/10/06</td>
<td>40</td>
</tr>
<tr>
<td>08/11/06</td>
<td>170</td>
</tr>
<tr>
<td>08/14/06</td>
<td>130</td>
</tr>
<tr>
<td>08/15/06</td>
<td>260</td>
</tr>
<tr>
<td>08/16/06</td>
<td>50</td>
</tr>
<tr>
<td>08/17/06</td>
<td>200</td>
</tr>
<tr>
<td>08/18/06</td>
<td>150</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1420</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Failures</th>
<th>Startup</th>
<th>Operate</th>
<th>Terminate</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>11</strong></td>
<td><strong>2</strong></td>
<td><strong>17</strong></td>
<td><strong>26</strong></td>
</tr>
</tbody>
</table>

| Failure Rate | 0.77% | 0.14% | 1.2% | 1.8% |

4.4.6 Obvious problems

Obvious problems can be quickly solved when looking at reliability problems in a complex system such as ATOS. This research project addresses several problems, such as the reduction of compiler warnings, noisy log message errors, and the addition of system self protection logic.

4.4.6.1 Reduction of compiler warnings

Subramaniam and Hunt [89] explain in their book, *Practices of an Agile Developer*, that when a program has a compilation error, the compiler refuses to produce an executable. The developer does not have a choice other than fixing the error before moving on.
Warnings, on the other hand, are different. The program generating compiler warnings can still be executed. When developers ignore warnings and continue to develop code, they are essentially sitting on a ticking time bomb, which most likely will go off at the worst possible moment. Subramaniam and Hunt describe several typical compiler warnings and possible side effects if warnings are ignored.

Each warning is the compiler’s method of informing the developer that something is not right with this logic, and ignoring these warnings has the potential of causing unexplainable system crashes when executing the system. In addition, as code is developed or refactored, a new compiler warning could appear in the list of warnings, but could go unnoticed because the developer can not see them through all the clutter of the old warnings.

Table 2. Compiler warning reduction

<table>
<thead>
<tr>
<th>Date</th>
<th>Number of Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun-06</td>
<td>1,081</td>
</tr>
<tr>
<td>Jul-06</td>
<td>0</td>
</tr>
</tbody>
</table>

The complete build of the ATOS system was full of warnings at the beginning of this research project. The agile practices introduced in Section 3.5 were utilized through a series of quick iterations, allowing those warnings to be removed. Table 2 shows the sum of all compiler warnings across all applications at the beginning of this exercise, and then again after this exercise was over. Most notably, the majority of compiler warnings were found to be in much of the dead code, and these warnings simply disappeared when the dead code was removed. Nevertheless, these dead code warnings masked real warnings.
4.4.6.2 Reduction of noisy logged errors

The reliability of the simulation system depended on aggressively tackling any error messages generated by a scenario run. If error messages were ignored, then other developers or users of the system would learn to ignore them as well, thus reducing their overall effectiveness. The same arguments for reducing compiler warnings, as explained above, can be used in the reasoning for eliminating as many system-logged errors as possible before continuing. Past developers placed ERROR and WARNING messages in their code to indicate something happened that was not expected to occur. The definition of what would be considered a warning versus an error was not defined, with each individual responsible to classify the type of log message displayed upon identifying a problem. Also of note, ERROR log messages by far outnumbered WARNING messages. The basis of the rest of this section is how to deal with the large amount of ERROR messages.

In preparation for the SPAS experiment, test scenarios were created that would exercise as much of the functionality required as possible. At first these scenarios did not run very long, because system-wide problems would tend to bring the ATOS system to a complete halt. When inspecting the log files, thousands of errors were being logged before the eventual CRITICAL error occurred and crashed the entire system. Many of these errors were the same error constantly repeated, leading to questions regarding if these errors were real or just noisy. If the log file scanner tool was to be effective, then identification of noisy messages needed to be addressed. This research project will not go into the techniques used to reduce the number of noisy errors (other than that the agile practices mentioned in this research were applied).
Table 3 demonstrates that in the beginning, an incredible number of errors is found in the system after just a five ASTOR simulation batch run. The table also shows how quickly the errors were reduced in one month’s time. Also of note, the test run in August lasted less than two hours before failures brought down the system, whereas the subsequent months all had the same execution time of 15 hours.

<table>
<thead>
<tr>
<th>Date</th>
<th>Number of Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aug-06</td>
<td>101,342</td>
</tr>
<tr>
<td>Sep-06</td>
<td>325</td>
</tr>
<tr>
<td>Oct-06</td>
<td>87</td>
</tr>
<tr>
<td>Nov-06</td>
<td>16</td>
</tr>
<tr>
<td>Dec-06</td>
<td>0</td>
</tr>
</tbody>
</table>

4.4.6.3 System self-protection

During preliminary development for the SPAS experiment, there were theories that many of the system-wide crashes could be prevented by some simple commonsense self-protection logic. Each of the logic implementations was worked in as an agile iteration between other system development tasks. The team started with a quick whiteboard design, and immediately coded using the agile good enough practice, which gave a form of protection implemented within the first day it was realized that these items could potentially crash the system. All of these self-protection steps not only prevented the propagation of errors throughout the system, they also allowed longer scenario runs because they gracefully shut down the offending ASTOR federate, rather than crashing the whole ATOS system. The remainder of this section highlights two of the most important self-protection implementations.
In the early stages of this research project, there was a lack of logging information that indicated time drift problems. Time drift had to be observed by a person. However, it was easy to write logic to detect that time drift was occurring within one particular ASTOR federate. Thus, a solution was to simply monitor time, and when the system detected a significant time drift, the ASTOR federate would log this to a file and perform a self-terminate. Not only did the ASTOR with a time drift problem go away, but the time critical information being shared via ADS-B messaging was also eliminated from the system, preventing other ASTOR federates from receiving corrupt data.

The amount of data produced by the new SPAS scenarios was extremely large in comparison to previous experiments; a typical overnight run could generate over 10GB of data per computer. The computers that make up ATOL do not have large hard disk drives, because disk speed was favored over disk size. On several occasions, one of the computer's hard disks was nearly full, with less than 1MB of free space after a system crash. A simple agile feature, iteration, was performed to include monitoring of hard disk free space. When the hard disk free disk space was within a pre-defined size, the ASTOR would log the hard disk drive as near full and then self-terminate. After the free disk space protection logic was implemented, there were numerous occurrences of the error message.

4.4.7 Basic principles

The project continues with a set of basic principles, consistency and simplicity, that are followed throughout the length of the project.
4.4.7.1 Consistency

There are many ways to design, develop, test, and display a set of applications. Every method has advantages and disadvantages, but the disadvantages are typically overcome in a design if there is consistency with the other applications under the team’s control. Efficiency is gained when a series of applications all have a consistent look and feel to them. The most obvious efficiency is a shortened learning curve; a user only has to learn something once and then can reuse that knowledge across all the applications. For example, if a user finds a quirk/flaw/bug in one application, they will find it in all applications. For each quirk/flaw/bug there usually is a fix, or at least a workaround that allows the user to continue. Developing this fix or workaround could potentially be applied to all applications. Now take a set of inconsistent applications, each one will have its own set of flaws that need to be discovered independently by the user group.

Consistency requires teamwork and a strong discipline to maintain the code base. Team members need to agree on the approach taken and have an opportunity to address concerns. A good team listens to all ideas from the group and then decides on the most appropriate way to move forward. Better decisions will result by using the agile practice of refactoring to fix inconsistencies in the older code base, and following a strong coding standard [90] for new development.

Developing in a multi-contractor environment, the majority of ATOS software is lifted from other projects, and the obvious differences between code baselines will probably never be consistent. However, new software can follow an accepted standard, and refactored software should try to follow the standard of the existing baseline.
4.4.7.2 Simplicity

A good architecture tries to optimize various aspects of the system being developed, including data flow, speed, setup time, or memory usage. There is not any harm in developing systems with the above optimized features. However, if the system turns out too complex, few users will understand it and never be able to contribute their ideas into the system. Cunningham and Elliott [91] write about the cost of complexity and how it can seriously inhibit productivity. They believe if companies can introduce simplicity into their organizations, they will find an improvement in their productivity. The goal of any ATOS software or process is to keep it simple. Simplicity is quick and easy to code, easy to test, and easy for future developers to understand and modify. Simplicity avoids the use of generic coding if it is not needed. A complex messaging mechanism setup is not needed if only passing around a few parameters. Simplicity allows for a component or application to do one thing and do it well, and avoids creating the do-it-all application. Since ATOL has a skilled user-group, they do not need a super application that figures everything out for them; rather they work best with a series of simple applications that can be assembled to achieve what is needed at the moment.

ATOL is unique in that the goals of this simulation are not to create a finished product to pass on to the customer, but rather create an environment that meets the changing needs of the researcher. In other words, the software is continuously changing and requires a development team that can support the necessary coding. Key areas of the software base should address the research interests of ATOL, with the key areas written in a clear and maintainable style. Generic coding that would be difficult for any typical programmer to understand should be minimized. The developers’ team has experienced
a 100% turn over rate in personnel in the past, and any design should take this into consideration because the team could turn over again in the future; therefore, a simplistic approach to the coding is the most desired approach.

4.4.8 Ground rules

There are two basic ground rules that are important in addressing the system's reliability problems. The first one is to avoid speculation in determining what a particular symptom of the simulation means. Second, use caution when deciding to rewrite a faulty component of the system.

4.4.8.1 Avoid problem speculation

Many of the causes of reliability problems in the ATOS system are at best an educated guess on what is thought to be the true problem. Speculation of the actual causes can often waste time as the developer attempts to implement solutions to the wrong sections of code. In these instances, solutions could take weeks, if not months, to fully implement correctly.

This research project is taking the approach of avoiding problem speculation if possible. By implementing well-placed log messages and utilizing the strengths of the agile practices listed in Section 3.5, the real causes of problems within the system can be isolated. Once problems are accurately identified, then decisions can be made on how best to correct them.
4.4.8.2 Refactor or rewrite

Determining whether an unreliable component of ATOS should be refactored (an agile approach) or rewritten from scratch can often be a difficult decision. Refactoring a piece of software is an easy decision when there are only a few known bugs associated with the component. But what determines if the faulty component should be totally rewritten? The wrong decision can have drastic consequences, as in the case of Netscape versus Microsoft in their next generation browser upgrades [92]. Netscape lost significant market share to Microsoft because Netscape chose to totally rewrite their browser software despite not fully understanding what exactly their original browser did well; instead they only understood its weaknesses.

Although the stakes are not as high with ATOS software, a smart decision needs to be made that benefits the lab in the long-run. As a rule of thumb with ATOS software components, the key deciding factor involves how integrated the component is with other components of the simulation. If the component acts as a central hub of other components, then in general a rewrite is considered dangerous, and refactoring is the only way to proceed.

4.4.9 Approach

The overall goal of this research project is to increase the reliability of the ATOS system by using the agile practices listed in Section 3.5. This involved enhancing the logging of messages of suspected problem areas in the ATOS code base. Since the code base is large, a systematic approach of finding problems would simply take too long. In addition to fixing reliability issues, the development team was tasked to enhance the simulation to
new capabilities that are significantly different than in previous experiments. Accomplishing both improvement of reliability and enhancement of functionality, in the short time frame required for preparation of the SPAS experiment, using the traditional development methodology was deemed impossible.

To systematically add log statements throughout the entire code base from the beginning was unrealistic. Furthermore, no useful information was gained while log statements were being implemented. Rather, the _good enough_ agile practice was applied. As described in Section 3.5.3, the practice of _good enough_ software requires only the necessary efforts to achieve the desired results of the project. The _good enough_ approach dictated only to add log statements where they would give useful information leading to identifying system problems.

Since the majority of reliability problems occurred in the start up and shut down of the system, a two-step approach was implemented. Step one involved adding log statements to all top-level components that will indicate how far the execution proceeds before a crash. Figure 9 shows a typical top-level simulation code structure without any log information contained within the code base. The second step added log tracing statements to all the major sections of code as they were constructed, initialized, and destructed (see Figure 10), allowing for a quick traceability into the logic flow when a crash occurred.
```cpp
main()
{
    classA.initialize();
    classB.initialize();
    while (System_is_running)
    {
        classA.execute();
        classB.execute();
    }
    classA.terminate();
    classB.terminate();
}

Figure 9. Typical top-level simulation code structure

```cpp
main()
{
    LOG("Begin initialization of classA");
    classA.initialize();
    LOG("Begin initialization of classB");
    classB.initialize();
    LOG("Begin main loop");
    while (System_is_running)
    {
        classA.execute();
        classB.execute();
    }
    LOG("Begin terminate of classA");
    classA.terminate();
    LOG("Begin terminate of classB");
    classB.terminate();
}

Figure 10. Modified top-level simulation code structure
For the real-time portion of the software, logging was implemented at places where occasional events would occur. The theory was that the system cycles continuously, exercising most of the real-time code. It's only on occasion that a system crash occurs, and it is triggered by some specific event. Logging these events is not time consuming as long as the event is occasional and does not affect the real-time performance.

Normal feature enhancement development continued on using the agile practices of quick iterations and continuous integration. When a failure occurred after system tests, the log files was examined for possible clues into which section was the cause of the crash. If the suspected area of the crash was too large to deal with, then another layer of log statements were supplied to the offending code base; this continued until the section of logic was small enough to consider refactoring or rewriting.

If a section of code was refactored, it was accomplished in two phases. The first phase cleaned up code readability, but the logic flow was not change. The second step swept through the logic looking for potential coding pitfalls and introduced proper warnings and error log statements (see Figure 11 as a simple example).

Finally, common areas of the code base without a recorded failure at this time were also refactored with the lessons learned from the original failure coding, thus preventing future failures and making the code base more consistent. If rules of simplicity could be applied they were done so at this level.
// Original code with potential divide by zero problem
int nRatio = 1/nInput;

// Refactored code with LOG message identifying problem
int nRatio = 0;

if (!nInput == 0)
    nRatio = 1/nInput;
else
    LOG("ERROR: nInput = 0 - Divide by zero problem")

Figure 11. Refactoring error handling for a potential divide by zero problem

4.5 Results

The development team was drastically smaller compared to previous efforts to prepare for a major simulation experiment. New software needed to be developed, while at the same time the existing software had to become more reliable. Following the same development methodologies that helped make the simulation experiments in the past successful was simply not an option, as these methodologies required a larger team in place. Due to time constraints, the team did not have the luxury to define in detail a development process to follow. Because of these constraints, it would have been very easy to fall into an ad-hoc development approach. However, fortunately the practices
adhered to by the agile community seemed to be very promising, even to the extent of bringing in the particular agile practices iteratively and allowing the team to grow into a new agile approach throughout the length of this research project.

4.5.1 Results from continuous integration

The most notable result of having implemented continuous integration is reflected in the dependability of every build. Any developer could confidently install a new build in minimal time, encouraging developers to keep current with the latest build. Comparing this result with a past 2004 similarly complex experiment as the SPAS, the number of working build days was recorded over a period of 30 days (see Figure 12).

![Figure 12. Successful build days over a 30 day period](image)

In 2004, the build was stable for several consecutive days because check-ins to the version control repository were frozen. Notably, one of the successful build days was related to a managerial-directed freeze for checking in code (unless the code change was directly related to stability of the build). Due to poor stability of the daily build,
developers were forced to test their new software changes on old builds, which did not have all the latest features. Often it is these new features that will cause the incompatibility problems with the other new features, thus perpetuating the problem.

Fowler's approach to continuous integration states that developers should commit to the version control repository verified by an automatic build, thus leading to multiple builds per day (if multiple developers commit during the day) [56]. In the past, the ATOS development team only had one build per day, which was an automatic build kicked off at midnight. Although possible to manually start another build, the process was cumbersome, and the builds were lengthy. The process has been improved significantly; starting a new build went from manually typing in a series of confusing command lines to simply double-clicking a desktop icon. Developers can also choose to make a mainline build or a test branch build at any time of the day.

The install process was also significantly improved. Previous installs required a complex command line text entry that was difficult to remember due to its peculiarities. The install process was slow because it would unzip one installation file at a time (total of six files) on one computer, and then repeated this process for the next computer. On a seven computer integration string, the install was 20 minutes. On the ATOL string of 14 computers, the install was 45 minutes. An official install was not available for a developer's desktop simulation station; instead, an undocumented process was used by the few individuals who knew how to implement the process.

The new install process was easier to invoke and quicker to complete. To invoke the install, a user double-clicks an icon on the desktop and types the date of the build into a pop-up window. All installation files are quickly distributed to each computer on the
simulation string, and then in parallel every installation file is uncompressed, reducing the time to install by 90% (see Table 4). Finally, each developer has the same install capabilities as the test lab and ATOL. The private simulation capability for the developers allows them to develop and test with the latest build, which in turn leads to significantly reduced integration problems.

Table 4. Install time reduction

<table>
<thead>
<tr>
<th>Date</th>
<th>ATOL install (min)</th>
<th>Integration string install (min)</th>
<th>Developer desktop install (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun-06</td>
<td>45</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>May-07</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Difference</td>
<td>40</td>
<td>28</td>
<td>18</td>
</tr>
<tr>
<td>Reduction Pct.</td>
<td>89%</td>
<td>93%</td>
<td>90%</td>
</tr>
</tbody>
</table>

Implementing the agile continuous integration practice is an ongoing task, in which the implementation itself utilizes the other agile practices of Section 3.5. At the time of this research project, the continuous integration approach was not fully implemented. However, the results are encouraging, and the benefits gained from continuous integration are aiding in other tasks.

4.5.2 Reducing useless code

Code reduction was by far the longest phase of this research project, and it still continues today. The code base is large and it takes time for individuals to decipher what exactly is useless code. In addition, like continuous integration, the reduction of useless code was considered a background task. Each of the four categories of useless code (defined in Section 4.4.2) had their own peculiarities, and each demanded their own techniques for identification and reduction. Dead files were found to be the most numerous of all
useless code. The very removal of useless code (a code base that seemed dauntingly large) suddenly decreased by close to half of its former size (see Table 5).

<table>
<thead>
<tr>
<th>Date</th>
<th>Number of Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun-06</td>
<td>41,996</td>
</tr>
<tr>
<td>May-07</td>
<td>21,747</td>
</tr>
<tr>
<td>Difference</td>
<td>20,249</td>
</tr>
<tr>
<td>Reduction Pct.</td>
<td>48%</td>
</tr>
</tbody>
</table>

Although dead code is typically removed by the linker, the reduction quite often helps in the identification of what a particular piece of code actually does. Identifying any type of dead logic can be a time consuming task, and the risk is high if a developer is wrong and removes a piece of code that is key to a seldom used, albeit important feature of the system.

Typically dead logic and bloated code removal was reserved for when large gains in useless code reduction could be made. Because the development team was experienced in real-time simulation, often times they could recognize the core functionality of some bloated software and replaced it with simplified logic. In one case, a bloated generic library consisting of nine files and over 2,700 lines of code was reduced to its core functionality of just 29 lines of code. A variety of tools and techniques are available for identifying each of the listed useless code categories as defined in Section 4.4.2. This paper will not go into how removal was accomplished, but rather how removal was staged in loosely defined iterative refactoring sessions with a series of system level tests conducted between each iteration.

Besides the benefits listed in Section 4.4.2 for reducing useless code, another benefit of a reduced build time was also realized. At the start of this research project, the time to
build a complete ATOS installation package was measured at 105 minutes. An analysis of total build time was conducted and it was found that 48 minutes, or 46% of the total time, was due to managing the large quantity of files. This involved copying of files from a networked version control repository into a local disk drive, and creating an install package in the form of TAR files after the compiling portion of the build was complete. Table 5 shows that almost half (48%) of the total amount of files that represent the ATOS system had been removed, causing a direct benefit of a 65% reduction to the build times simply because there were fewer files to manage. An equal reduction also was realized in the overall compile times due to the fact that there was less code to compile. The net effect in removing useless code shaved off a full 67 minutes (or 64%) from the total build time (See Table 6).

Table 6. Build time reduction

<table>
<thead>
<tr>
<th>Date</th>
<th>Build File Management (min)</th>
<th>Compile time (min)</th>
<th>Total build time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun-06</td>
<td>48</td>
<td>57</td>
<td>105</td>
</tr>
<tr>
<td>May-07</td>
<td>17</td>
<td>21</td>
<td>38</td>
</tr>
<tr>
<td>Difference</td>
<td>31</td>
<td>36</td>
<td>67</td>
</tr>
<tr>
<td>Reduction Pct.</td>
<td>65%</td>
<td>63%</td>
<td>64%</td>
</tr>
</tbody>
</table>

The dramatic decrease in build time has also made the agile continuous integration practice more practical. Faster builds gave quicker turnaround times in testing new functionality with the new build. Thus, encouraging less functionality added per iteration, led to reduced integration problems, and allowed a more rapid development pace of cohesive and reliable software [56].
4.5.3 Increasing log statements

Before this research project started, log messages were used sporadically throughout the ATOS code base. The majority of log messages fell into one of the four categories below:

**INFO:** A brief informative message to indicate location and time a coding event took place.

**WARN:** A message intended to notify various users of possible issues with respect to running applications.

**ERR:** A message that details a programmatic error that will most likely affect the results of the particular application.

**CRIT:** Programmatic violations that will usually result in application failure if continuation of the application is allowed.

Table 7 shows the increased log messages added throughout the code base over a period of a year. Most of these log messages were strategically placed in an effort to uncover reliability problems in ATOS.

<table>
<thead>
<tr>
<th>Date</th>
<th>INFO count</th>
<th>WARN count</th>
<th>ERR count</th>
<th>CRIT count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jun-06</td>
<td>13</td>
<td>37</td>
<td>253</td>
<td>35</td>
</tr>
<tr>
<td>May-07</td>
<td>627</td>
<td>139</td>
<td>456</td>
<td>358</td>
</tr>
<tr>
<td>Increase Pct.</td>
<td>4723%</td>
<td>276%</td>
<td>80%</td>
<td>923%</td>
</tr>
</tbody>
</table>

Table 8 shows a breakdown of log messages between old and new coding. Even though the size of the old coding is a few orders of magnitude larger than new code, the number of INFO log messages is only double in size. This is indicating that the old software only had messages added to the top level of code and in several key areas of
suspected unreliable code, whereas the new software systematically applied INFO messages throughout the code base.

Table 8. Log messages old versus new coding

<table>
<thead>
<tr>
<th>Code</th>
<th>INFO count</th>
<th>WARN count</th>
<th>ERR count</th>
<th>CRIT count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old</td>
<td>410</td>
<td>73</td>
<td>385</td>
<td>44</td>
</tr>
<tr>
<td>New</td>
<td>217</td>
<td>66</td>
<td>71</td>
<td>314</td>
</tr>
<tr>
<td>Total</td>
<td>627</td>
<td>139</td>
<td>456</td>
<td>358</td>
</tr>
</tbody>
</table>

Another interesting finding from the data presented in Table 8 is that the number of CRIT messages in the old code is significantly lower than that in the new code. Since the new code was written with reliability issues in mind, every obvious identifiable problem area was coded up as a critical error. With the old code there were simply too many potential problem areas to justifiably add critical error handling in the short time available. In addition, the majority of old code was already proven reliable and was successfully executed in ATOS for years.

4.5.4 Reduction of reliability problems

Over the course of four months the reliability problems of ATOS were reduced significantly. Table 9 shows a summary of all critical failures that occurred over a two week period during the month of December compiled over three independent federations of 5, 21, and 34 ASTOR simulation stations. The net number of ASTOR runs was 77,360 with only seven failures, or a 0.0090% failure rate. When comparing this table with the baseline Table 1, only 1,420 runs were executed with 26 failures, representing a 1.8% failure rate.
Table 9. Critical error end result

<table>
<thead>
<tr>
<th>Date</th>
<th># Runs</th>
<th>Startup</th>
<th>Operate</th>
<th>Terminate</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/11/06</td>
<td>760</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/12/06</td>
<td>780</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/13/06</td>
<td>756</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12/14/06</td>
<td>1,173</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/15/06</td>
<td>900</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>12/18/06</td>
<td>150</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/19/06</td>
<td>613</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/20/06</td>
<td>476</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12/21/06</td>
<td>1,034</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>12/22/06</td>
<td>1,094</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>77,360</strong></td>
<td><strong>3</strong></td>
<td><strong>0</strong></td>
<td><strong>4</strong></td>
<td><strong>7</strong></td>
</tr>
</tbody>
</table>

Failure Rate | 0.0039% | 0.0000% | 0.0052% | 0.0090%

During the months following this data collection period, the number of ASTOR federates grew from 34 to 96. Improvements in the simulation’s reliability have continued, although the concentration of work is less on reliability issues and more focused on scenario modeling and architectural improvements.

4.5.5 Added functionality of ATOS

During the same four month period that the reliability problems were reduced, ATOS had many new features added to the system. The majority of this functionality was concentrated on the scenario capability of the system.

The concept of a master scenario file was fully exploited to be the cornerstone of the ATOS scenario system. Previously, a master scenario file was only a starting point for the manual creation of the entire scenario suite of files. This suite of scenario files was stored with every build, since the sum of all scenario files numbered in the thousands; maintenance of these files was a difficult problem. The new system reduced the burden of file maintenance by extracting the commonalities between scenario files and placing
these into named configuration files. A configuration file could define initial airframe, propulsion, avionics, environmental, or simulation system properties. The master scenario files were modified to include these configuration files such that each aircraft of a master scenario file could be exactly identified in terms of all its initial properties.

A scenario assembly tool was created that could automatically generate the complex suite of scenario files from the master scenario file and its referenced configuration files in less than a second. This automation allowed further development of a real-time scenario generation application that modeled the SPAS experiment’s airspace while simultaneously controlling and monitoring the mode status and health of all ASTOR federates. The application was designed to initialize an ASTOR aircraft object along the outside edge of the outer circle (figure 4.x) with a random position, heading, speed, and start-time while maintaining a fairly consistent density of aircraft within the inner circle. In addition, the ASTOR federates were modified with new messaging that allowed the real-time scenario generator to optimize the flight time of the aircraft object.

All of the new scenario capability developed for the SPAS experiment was designed to be backwards compatible with functionality developed in past experiments, as this functionality was required to continue to work in the almost weekly demonstrations that NASA conducted with ATOS. To simplify the handling of this backwards compatibility and the amount of detail needed to define a particular aircraft, the concept of experiment based scenario template was created. Each template file would specify all the common configuration files and fine details that a typical ASTOR would need in a particular experiment. The master scenario file would simply include an experiment specific template and then only maintain the unique changes, thus keeping the size and
complexity of the master scenario file to a minimum. Overall, the newly enhanced scenario capability allowed significant changes to be made to the system without breaking any of the capability developed in the past.

Although the ASTOR was designed as a Human-In-The-Loop simulator, it could also be easily controlled by an optional rule based pilot model in the loop during batch experiments. This pilot model was enhanced with new functionality such as a new sensor input algorithm for detecting conflicts and resolutions via the on-board autonomous operations planner and new rules for the pilot model to handle dynamic in flight situations such as cancelation of conflict resolution operations.

Finally, a self termination concept of the ASTOR was implemented. Basically, this was a series of checks that the ASTOR would conduct looking for system problems such as near full hard-disk, time synchronization problems, and any unexpected errors thrown by lower level code. This self termination capability was also exploited to simplify the modeling of when to terminate an aircraft from leaving the SPAS experiment testing area. A simple distance calculation from the test area center point was the determination of an ASTOR self terminating when it left the test region.

4.5.6 Comparing with other results

Lindvall, et al. [82] collected and analyzed empirical evidence about the effectiveness for agile projects by examining the results of an expert discussion panel and made an attempt to collect the experiences of the group. A summary of these findings was presented in Section 3.8 and is depicted in Table 10 along with corresponding comments and findings related to this research project.
Table 10. Comparison to Lindvall et al. [82]

<table>
<thead>
<tr>
<th>Lindvall, et al. Survey Findings</th>
<th>Thesis Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any sized team can be agile if attention to communication is addressed.</td>
<td>The team for this project consisted primarily of three closely working developers, followed by seven other developers working from different sites. Communication consisted of frequent unscheduled standup meetings, emails, and phone calls.</td>
</tr>
<tr>
<td>Part of the team make-up must be experienced with past development of a similar system.</td>
<td>All three primary developers are skilled and experienced with simulation software development.</td>
</tr>
<tr>
<td>Agile methods require less formal training than traditional methods.</td>
<td>Training consisted of a few informal meetings and several written articles about agile development (such as: [3, 44, 49, 51, 56, 58, 60, 70, 71, and 92]).</td>
</tr>
<tr>
<td>Attention must be paid to culture, people, and communication in order to be successful.</td>
<td>All agile practices followed were in consideration of the backgrounds of the team members, and the existing culture of the NASA customer.</td>
</tr>
<tr>
<td>Refactoring should be done frequently. Large scale refactoring is more feasible using agile methods than by using traditional design approaches.</td>
<td>Refactoring was the primary practice used to reduce the simulation’s reliability problems with the legacy code. Even new coding was quickly implemented, and through a series of refactoring sessions, the design was shaped into what was needed rather than designing every detail up front.</td>
</tr>
<tr>
<td>Big architectural changes do not need to be risky if a set of automated tests is maintained.</td>
<td>Some big architectural changes were made, such as the way scenarios are created and distributed through the simulation, HLA interfacing libraries, and simulation timing interactions. Each night the simulation was run through an automated test scenario that was compared with the previous night’s run, and new problems were immediately noticed.</td>
</tr>
<tr>
<td>Documentation should be assigned a cost, and the extent should be determined by the customer.</td>
<td>The only documentation created for this project was that requested by the customer, such as user guides and system overviews. Since the customer did not ask for any low level documentation that was provided in the past, none was written.</td>
</tr>
</tbody>
</table>
The surveys conducted by Shine Technologies [80], VersionOne [83], and Ambler [84] all found that by using some form of agile development techniques that projects had increased productivity and quality over those projects that used a more traditional form of development. The findings for this research project also found that the development team appears to have achieved higher productivity than in the past, and that the quality of the ATOS system was remarkably improved, as shown in Table 9.

Ambler's survey also found that 58% reported improved customer satisfaction from using agile techniques. This research project also found that the NASA customer was very satisfied with the improved results, as shown in NASA's semi-annual reviews of the contractor, and the large increase in scope of the original SPAS experiment. Finally, satisfaction was personally observed by this author from hearing more positive comments about the improved stability of the simulation.

4.6 Continuing progress

The SPAS experiment was a huge success, and this avenue of research will continue into the future. Research scheduled within the next two years will examine higher levels of aircraft density covering multiple altitudes, and include other sources of error and uncertainty incorporating the coordination of a centralized approach [87]. Other types of experiments not related to SPAS are also being actively prepared for using the ATOS system; agile development practices will be a part of the preparation.

Since the period representing this research project, the development team size has grown significantly. With a larger team, communication has become a challenging issue, but the agile practices listed in this research will continue to play an important role. However, the way in which the practices are used has already changed, and this is to be
expected. Because several new individuals have been added to the team, a change in the agile practices is in order [43]; as a team increases in size, a larger methodology is required to support the larger team [44].
5 CONCLUSION

The motivation for this research was based on recent changes to the development environment. A reduced budget and a significantly smaller team size left the old way of doing development ineffective. A new approach was sorely needed and this research analyzed a set of agile practices as a replacement to the old CMMI based development approach. Although there are numerous software development practices that fall under the agile umbrella, this research concentrated only on a few practices that could be implemented within the ATOS development team in a short time. These practices were: continuous integration, delayed decisions, *good enough* software, iterative development, refactoring, simplicity, and sustainable pace.

This research has shown that implementation of the above agile practices have indeed shown favorable results during the preparation phase for an upcoming NASA experiment. ATOS has shown a remarkable improvement in reliability, and this was done while the development team simultaneously enhanced the required modeling and system functionality as required by the experiment plans. Agile methodologies have proved to be successful in NASA’s simulation environment and will be continuously implemented and modified into the future.
6 REFERENCES


VITA

for

DOUGLAS JAMES MIELKE

DEGREES:
Master of Science (Modeling and Simulation), Old Dominion University, Norfolk, VA, August 2008
Bachelor of Science (Aerospace Engineering and Mechanics), University of Minnesota, Minneapolis, MN, December 1987

PROFESSIONAL CHRONOLOGY:
Science Applications International Corporation (SAIC), Hampton, VA
Section Manager (Technical Lead / Architect), May 2003—Present

Shared Healthcare Systems, Anacortes, WA
Software Engineer, October 1998—March 2003

Raytheon Systems Co. (Multiple locations)
Sr. Systems Engineer, October 1987—September 1998