The report committee for Michael Brian Willis certifies that this is the approved version of the following report:

The Role of Software Engineering Process in Research & Development and Prototyping Organizations

Approved by
Supervising Committee:

K. Suzanne Barber

Thomas Graser

The Role of Software Engineering Process in Research & Development and Prototyping Organizations

by

Michael Brian Willis, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

August 2010

Dedication

To Mom and Dad:

Thank you for helping me focus and for teaching me balance (or at least trying to).

To Alana:

Thank you for putting up with graduate school, this paper and me in general…

…I hope your ear is not too red.

August 11, 2010

Abstract

The Role of Software Engineering Process in Research & Development and Prototyping Organizations

Michael Brian Willis, M.S.E.

The University of Texas at Austin, 2010

Supervisor: Suzanne Barber

Software Research and Development Organizations (or SRDs) have unique goals that differ from the goals of *Production* Software Organizations. SRDs focus on exploring the unknown, while *Production* Software Organizations focus on implementing solutions to known problems. These unique goals call for reevaluating the role of Software Engineering Process for SRDs. This paper presents six common Software Engineering Processes then analyzes their strengths and weaknesses for SRDs. The processes presented include: *Waterfall, Rational Unified Process (RUP), Evolutionary Delivery Cycle (EDLC), Team Software Process (TSP), Agile Development* and *Extreme Programming (XP)*. The results indicate that an ideal software process for SRDs is *iterative*, *emphasizes visual models, uses a simple organization structure, produces working software (with limited functionality) early in the lifecycle, exploits individual capabilities, minimizes artifacts, adapts to new discoveries and requirements,* and *utilizes collective code ownership among developers*. The results also indicate that an ideal software process for SRDs does NOT define *rigid personnel roles or rigid artifacts,* is NOT *metric-driven* and does NOT implement *pair programming*. This paper justifies why SRDs require a unique software process, outlines the *ideal* SRD software process, and shows how to tailor existing software processes to meet the unique needs of SRDs.

Table of Contents

List of Figures

1. Introduction

Albert Einstein said, "If we knew what it was we are doing, it would not be called research"[3]. This demonstrates a core difference between Research and Development Software Organizations and Production Software Organizations. The goal of Research and Development Software is to explore the unknown while the goal of Production Software is to implement the known. Most of the Software Engineering Processes available today are not tailored to Software Research and Development Organizations (or SRDs). This combined with the unique goals of SRDs calls for reevaluating the role of Software Engineering Process for SRDs.

First this paper will define and characterize SRDs. Next this paper presents an overview of six common Software Engineering Processes, followed by analysis of their strengths and weaknesses for SRDs. Finally this paper presents the results of the analysis and suggests which process traits are well suited for SRDs. After reading this paper, developers and managers at SRDs should better understand which Software Engineering Processes might be well suited for their organization.

2. CHARACTERIZING SRDS

Chapter 1: SRD Definition

Software Research and Development Organizations (SRD) primarily develop exploratory, non-production software. *Non-production Software* is software not intended for long-term use by a customer. Often *Non-Production Software* is an internal prototype software system that may or may not ever be used. *Exploratory Software* is any software system whose primary purpose is to determine the feasibility of an idea. In other words, *Exploratory Software* is software that provides the proof-of-concept for an idea. Often the Software Development in SRDs is to support R&D for new hardware platforms or new signal processing algorithms. For example, an SRD could be part of a hardware company that creates the software to provide the proof-of-concept for a new piece of hardware.

The rest of this section defines SRDs using a list of SRD *Organizational* and *Project Characteristics*. It is difficult to define and characterize R&D organizations because typically they specialize in a certain competency or type of technology that is not widespread. This makes it difficult to compare R&D organizations, which results in difficulty defining and characterizing R&D organizations such as SRDs[14]. Because SRDs are difficult to define and characterize, there is a limited amount of academic literature available regarding SRD *Organizational* and *Project* Characteristics. As a result, much of this paper utilizes the author's observations as a Software Engineer for the *Applied Research Labs(ARL),* a University of Texas research and development laboratory funded predominately by federal contracts. However, the content of this paper is applicable to many SRDs, not just *ARL*.

Section 3 will list unique traits of six selected Software Processes. Then section 4 will match the SRD *Organizational* and *Project Characteristics* to the traits of the Software Processes presented in Section 3. Section 4 will also provide an analysis of the strengths and weaknesses of the selected Software Engineering Processes for SRDs.

Chapter 2: SRD Characteristics

SRD Organizational Characteristics

SRD *Organization Characteristics* are attributes related to management and company structure. *Organization Characteristics* apply to the organization as a whole and do not define any project specific characteristics. The following *Organization Characteristics* may apply to SRDs:

1. *Flat Management*: Many SRDs have several engineers beneath one project manager, with no intermediate levels of management.
2. *Technical Managers*: Managers at SRDs likely come from a technical rather than a business background. Often, managers at SRDs are former engineers for the same organization.
3. *Small Teams*: SRDs typically are made of smaller teams when compared to a production organization. For the purposes of this paper, *small teams* are teams with less than 15 members.
4. *High Autonomy*: The developers in SRDs typically can approach solving a problem however they prefer. Although developers in a typical SRD are managed, they are not *micromanaged* and are free to implement solutions in their own way.

SRD Project Characteristics

SRD *Project Characteristics* are attributes of typical SRD projects. SRD *Project Characteristics* are independent of management or organizational structure and only describe traits of a typical SRD project. The following *Project Characteristics* may apply to SRD projects:

1. *Exploratory*: SRD Projects are *exploratory* which means their main purpose is to quickly determine the feasibility of an idea. This includes prototypes that provide the *proof-of-concept* for an idea. Research and Development organizations typically *explore* a problems space by quickly developing then fielding an applied technology [14].

2. *Unknowns*:
    - *Requirements*: SRD projects often have little or poorly defined system requirements due to their *exploratory* nature. This makes it difficult to determine if an SRD project was a success or a failure[12].
    - *Feasibility*: SRD projects by definition have never been done before, and as a result their *feasibility* is unknown[12].
    - *Future*: Because the *feasibility* of SRD projects is unknown, the future of an SRD project is also unknown. An SRD project can be cancelled at anytime if the project is determined unfeasible or not cost effective.
3. *High Risk*: SRD projects are more prone to failure, missing schedule and going over budget because of the *unknowns* involved. In addition, even if an SRD project completes on time and on budget, the end result may never be profitable or realistic to produce. As a result, this paper categorizes SRD projects as *high risk*.

Chapter 3: SRD Characteristics Summary

Different "business goals" is the biggest difference between SRDs and Production Software Organizations. SRDs are often focused on technical success while Production Software Organizations focus on commercial success[4]. In addition R&D organizations such as SRDs must promote a learning and exploratory environment to be successful[14]; while production organizations can be successful simply by implementing and customizing known solutions. As a result, SRDs and production software organizations have different *Organizational* and *Project* Characteristics. The next section will present

an overview of several popular Software Engineering Processes and their unique traits. The following section will use this information to match the unique traits of the Software Processes to the SRD Characteristics listed in this section. This will show what aspects of Software Engineering Processes are well suited for SDRs.

3. OVERVIEW OF SOFTWARE ENGINEERING PROCESSES

Chapter 1: Introduction

This section provides an overview of several popular Software Engineering Processes and describes six processes that give the reader a sampling of the Software Engineering Processes available. Figure 1 shows a matrix of software engineering process categories vs. software processes. The rows in Figure 1 are high-level traits that describe general software engineering processes. The columns in Figure 1 are the six software processes selected for this paper. The cells in Figure 1 have a *check* symbol (✔) if the process falls into the category noted on that row. Each consecutive category (or row) is the converse of the previous entry (for example: *Adaptive* vs. *Predictive*). Below is a summary of each of the software processes categories (or rows) from Figure 1.

Adaptive: A process that continually modifies design, architecture, requirements and code based on feedback from the current system.

Predictive: A process that tries to predict software requirements to create a software design before software development begins.

Small Teams: For the purposes of this paper, *small teams* is defined as teams with less than 15 members.

Larger Teams: For the purposes of this paper, *larger teams* are those with more than 15 members.

Write Tests Before Code: A characteristic of some software processes (such as *XP*) that requires the developers to create software test routines before writing any actual source code [2].

Write Tests After Code: An innate characteristic of most software processes where developers write source code first, then write software test routines.

Developers are Testers: A software process (such as *XP)* that requires the software developers that write source code to also create and perform software test routines [2].

Dedicated Test Team: A characteristic of most traditional software processes that require a software test team that is not part of the software development team. The dedicated test team creates and performs test routines for the software system.

Working Software Early: A software process requirement to create a working version of the software system (with limited capability) as early as possible in the software lifecycle (i.e. *Agile Development)* [2].

Working Software Later: A characteristic of most non-*Agile* processes where producing working software is one of the last steps of the development cycle.

Strict Roles: A characteristic of a software process that defines detailed team-member roles and responsibilities (for example *TSP* provides detailed responsibilities for all team roles[5]).

Looser Roles: A characteristic of software processes that **does not** provide detailed roles and responsibilities for each team member.

Iterative: A software process that allows the development cycle to continuously repeat until the software is complete.

Non-Iterative: A software process that has one starting point and one stopping point.

Complicated Structure: A software process that has many possible roles and many possible artifacts.

Simpler Structure: A software process that calls for a smaller number of roles and a smaller number of artifacts than processes with a *Complicated Structure*.

Artifact Intensive: A software process that encourages creating a large number of software artifacts and software documents.

Artifacts Minimized: A software process that actively minimizes software artifacts.

Difficult to Implement: A software process that would require a large amount of time, money, resources and reorganization to implement.

Easier to Implement: A software process that could be implemented at a given organization with **minimal** time, money and reorganization.

Customizable: A software process that allows the Organization to modify characteristics of the process to meet the unique traits of their Organization.

Static: A software process that is not *customizable* to meet unique needs of an Organization.

High Process Overhead: A software process that requires completing many tasks artifacts that are not part of the deliverable system. For example, collecting programmer productivity metrics contributes to *Process Overhead* because the metrics are not part of the delivered software system.

Low Process Overhead: The opposite of *High Process Overhead;* a software process that minimizes tasks and artifacts that are not part of the delivered software.

A software process listed in Figure 1 does not have to meet every category. For example, *EDLC* does not have a *check* symbol for *Artifact Intensive* or *Artifacts Minimized* because *EDLC* does not explicitly control artifact creation. It is also important to note that some of the category assignments in Figure 1 are subjective. For example, the meaning of *High Overhead* and *Low Overhead* for two Software Engineers could be quite different depending on their background and experiences. Figure 1 is not intended to definitively categorize the selected software processes, however to show the diversity of the processes selected for this paper. The *check* mark distribution from Figure 1 shows that the six software processes selected for this paper cover a broad range or software process categories.

Figure 1: Coverage of S.E. Process Categories for the Selected Processes [2,5,6,8,9,12]

Software Engineering Processes

Software Engineering Process Categories	Waterfall	RUP	EDLC	TSP	Agile	XP
Adaptive					✔	✔
Predictive	✔	✔	✔	✔		
Small Teams					✔	✔
Larger Teams		✔				
Write Tests Before Code						✔
Write Tests After Code	✔	✔	✔	✔		
Developers are Testers						✔
Dedicated Test Team		✔				
Working Software Early			✔		✔	✔
Working Software Later	✔	✔		✔		
Strict Roles		✔		✔		
Looser Roles	✔				✔	✔
Iterative		✔	✔	✔	✔	✔
Non-Iterative	✔					
Complicated Structure		✔				
Simpler Structure	✔		✔		✔	✔
Artifact Intensive		✔		✔		
Artifacts Minimized					✔	✔
Difficult to Implement		✔				
Easier to Implement	✔			✔	✔	
Customizable	✔	✔		✔	✔	✔
Static			✔			
High Process Overhead		✔		✔		
Lower Process Overhead			✔		✔	✔

The remaining chapters of this section provide a summary of each software process followed by a list of its unique traits. Next, section 4 will match the unique process traits to the *SRD Organizational* and *Project* traits listed in Section 2.

Chapter 2: Waterfall Model

The *Waterfall Model* is a famous Software Process model developed in 1970. The *Waterfall Model* is considered a 'traditional' Software Process that is non-iterative. *Non-iterative* means there is one start point and one stop point for the project and the cycle does not repeat. As a result, the *Waterfall Model* emphasizes complete requirements specification and software design before starting the implementation (or coding) of the software. There are five linear steps of the *Waterfall Model: Requirements Engineering, Design, Implementation, Testing and Maintenance*[11]. Each phase must be reached in the order listed. If you reach a step that fails to meet the specified requirements, then you can move backwards one step only. However once a step has been passed, you cannot go back to that step.

Unique traits:

1. *Customizable:* The *Waterfall Model* requires strict high-level project steps, however implementation of these steps is customizable by the software organization.
2. *Non-Iterative:* The *Waterfall* model has one starting point and one stopping point, so it is a non-iterative process.
3. *Simple Structure*: There are only five high level steps to the *Waterfall Model,* making it a simple and easy-to-learn process.

4. *Complete Requirements before Implementation:* Because the *Waterfall Model* is non-iterative, it requires complete requirements specification before any implementation (or coding).

Chapter 3: Rational Unified Process (RUP)

RUP is an iterative Software Development process that can be tailored to fit the needs of the organization. *RUP* is part of a software product sold by IBM Corporation. To use the complete *RUP,* an organization must purchase the *Rational* Software System from IBM Corporation. *RUP* divides the project lifecycle into four main stages: *Inception*, *Elaboration*, *Construction* and *Transition*. The *RUP* specifies over 30 possible roles and over 60 possible artifacts (or documents)[5,6]. However, the idea behind *RUP* is to tailor the process to fit your organization. Most organizations will not use every role or create every possible artifact [5].

Unique traits:

1. *Iterative:* The *RUP* is an iterative process, which means the development cycle continuously repeats until the software is complete.
2. *Customizable: RUP* allows you to select the aspects of the process that fit the organization.
3. *Rigid Roles and Artifacts:* Although *RUP* is customizable, the process calls for strictly defining roles and artifacts.
4. *Documentation Intensive:* Creating useful, value-added documentation is an important aspect of *RUP*.

5. *Emphasizes Visual Software Models:* UML and other visual software models are an important way to communicate information with *RUP*.
6. *Difficult to Implement: RUP* is very customizable and may require a significant initial investment to tailor the *RUP* to fit your organization.
7. *Complicated Structure:* The *RUP* has over 30 possible roles and over 60 possible artifacts[5].

Chapter 4: Evolutionary Delivery Life Cycle (EDLC)

The Evolutionary Delivery Life Cycle (or *EDLC*) is the process Microsoft uses to develop software. Some literature refers to the *EDLC* as the Microsoft Sync and Stabilize Process (or *MSS*)[5]. The main idea behind *EDLC* is to periodically synchronize all individual software components, then stabilize the software product to ensure the system works as expected. The *EDLC* process has three project phases: *Planning*, *Development* and *Stabilization;* and three roles: *Program Manager, Developer and Tester*[5]. The *EDLC* requires creating a skeletal software system early in the project lifecycle. The skeletal software system should be a working system with limited functionality that theoretically could be delivered to the customer at any time[7].

Unique traits:

1. *Iterative: EDLC* encourages frequent software rebuilds and software stabilization periods, which makes the process *iterative*.
2. *Incremental: EDLC* continuously adds functionality in small steps.
3. *Simple Structure*: With only three phases and three roles, the *EDLC* structure is simple.
4. *Produces Working Software Early: EDLC* produces working software with minimal functionality early in the life cycle.

5. *Prioritized Functionality: EDLC* adds functionality incrementally, which allows the Program Manager to assign development priority to the most important functionality.
6. *Fast Time To Market:* Prioritized Functionality allows you to assign development priority to the most time sensitive functionality. This results in a fast time to market for the software product.

Chapter 5: Team Software Process (TSP)

The Team Software Process *(or TSP)* is a series of steps (called *scripts*) that guide the team through a software project[5]. Every step is defined in detail and every role includes a detailed list of responsibilities. *TSP* is an extension of the Personal Software Processes (*PSP)*. The Software Engineering Institute at Carnegie-Mellon University developed both the *TSP and PSP*. *PSP* seeks to improve performance of individual team members in order to improve the final product as a whole. *PSP* requires software developers to keep detailed metrics during the development cycle. For example *PSP* might require a developer to track the number of *Source Lines of Code (SLOC)* they produced per hour. This information could allow managers and developers to create a productivity formula (for example number of *SLOC* created per hour). The team then could use this formula to predict how many hours to allocate for the next software development task. *TSP* defines 5 roles: *Team Leader, Development Manager, Planning Manger, Quality/Process Manager* and *Support Manager*. Notice there are no software developer roles listed for *TSP*. This is because in *TSP* everyone is a software developer and everyone shares the project management workload. (Note: Some versions of *TSP* define developer and non-developer roles[5])

Unique traits:

1. *Iterative:* Like many other processes in this paper, *TSP* allows for iterative development.
2. *Defined Roles: TSP* defines detailed team roles and responsibilities.
3. *Scripted Development: TSP* provides step-by-step instructions for each role.

4. *Document Intensive:* The default configuration of *TSP* requires the team to initially complete over 20 documents[5].
5. *High Process Overhead: TSP* requires many tasks and documents that are not part of the deliverable system. (Note: These tasks and documents might result in a better product; however they are not part of the deliverable system so they are categorized as *Process Overhead*.)
6. *Customizable: TSP* can be customized for a particular team (however, it requires completing a Process Improvement Proposal or a *PSP)*[5].
7. *Metric-Driven*: *TSP* collects performance metrics to customize the organization's process.

Chapter 6: Agile Development

Aldo Dagnino provides a concise description of *Agile Software Development* in the paper "An Evolutionary Lifecycle Model with Agile Practices for Software Development at ABB":

> A new generation of software development lifecycle models has emerged lately called 'Agile' and they embrace change, reduce development cycle time, and attempt a useful compromise between no process and too much process[9].

The term "attempt a useful compromise between no process and too much process" is an important guiding principle for *Agile Development*. There are multiple types of *Agile Software Development*. This section discusses one of the more popular Agile Development Processes called *Extreme Programming (or XP)*. However, most Agile Development Processes share the following traits:

Agile Development traits:

1. *Iterative:* Similar to *RUP, TSP* and *EDLC,* the *Agile Development* cycle continuously repeats until the software is complete.
2. *Incremental: Agile Software Development* occurs in small but steady chunks throughout the lifecycle.
3. *Artifacts Minimized: Agile Development* minimizes software documents and maximizes the amount of working software[9].
4. *Exploits Individual Capabilities:* Unlike most traditional software processes, *Agile Development* takes into account individual developer strengths and weaknesses.
5. *Adaptive: Agile Development* continually modifies design, architecture, requirements and code based on feedback from the current system. *Agile*

Development seeks to ***adapt*** to the current environment rather than ***predict*** the environment beforehand[9].

6. *Small Teams: Agile Development* is usually made of small teams.
7. *Tolerant of Loose Requirements*: *Agile Development* methods allow progress even when the system requirements are not well defined.

Chapter 7: Extreme Programming (XP)

Extreme Programming (or *XP)* is a popular *Agile Development* process. *XP* emphasizes simplicity and efficient communication. For example one of the core rules of *XP* is to "Do the simplest thing that could possibly work"[10]. *XP* seeks to reduce complexity by working on current needs rather than predicting future needs[10]. *XP* focuses on maximizing working software early in the development cycle and minimizing tasks that get in the way. *XP* still emphasizes planning before implementing software, however it seeks to keep the planning phase well scoped and grounded; this allows an *XP* software design to adapt to requirements discoveries and new technologies throughout the development cycle. *XP* shares all of the core principles of *Agile Development* from the previous section, with some additional unique traits.

Unique traits:

1. *Design Simplicity:* "Do the simplest thing that could possibly work"[10].
2. *Small Releases*: *Agile Development* uses frequent and short release cycles. This lets the organization quickly adapt to changing markets, technologies and requirements.

3. *Write Tests Before Code:* Unlike most Software Development processes, *XP* encourages writing software tests (usually unit tests) before writing the actual software.
4. *Pair Programming: XP* encourages *pair programming* for all development tasks; two developers sit together at a computer and both contribute to the coding.
5. *Collective Code Ownership*: Any developer is allowed to change any code at any time. This is inherently the case at many software organizations, however *XP* explicitly emphasizes this principle.
6. *Continuous Integration*: Similar to *EDLC, XP* adopts the "get it working early and keep it working" philosophy.
7. *Developers are the Testers: XP* does not specify a separate software tester role, so the software developers test their own code.
8. *Sustainable Pace: XP* seeks a realistic and sustainable work pace for the developers. *XP* calls for working a strict 40-hour work week if possible; and to never exceed 40 hours a week for two continuous weeks[2].
9. *Focus on Working Software*: Working software is the primary measure of success in *XP* programming.

4. ANALYSIS OF SOFTWARE ENGINEERING PROCESSES

This section will analyze the strengths and weaknesses of the Software Engineering processes discussed in section 3. This section presents the advantages and disadvantages of each Software Engineering Process as they pertain to each of the SRD Organizational and Project traits from section 2. The numbers under the Software Engineering Process advantages and disadvantages correspond to the original number assigned to that Software Engineering Process from section 3. Refer back to section 3 for a detailed explanation of each trait.

The end of this section provides a matrix diagram that maps the SRD Organization and Project Traits to the advantages and disadvantages of each of the Software Engineering Process traits. The final section of this paper provides a summary, a discussion of the results, and future research directions. As mentioned in section 1, most if this analysis is somewhat subjective and based on the author's experiences at *ARL,* however this analysis could be useful to SRDs other than *ARL*.

Chapter 1: SRD Organizational Characteristics

1. ***Flat Management:***

(-) *RUP* Disadvantages:

2. Rigid Roles and Artifacts:

With only one level of management, strictly defined roles may not be a good fit for SRDs.

7. Complicated Structure:

SRDs would have a difficult time implementing multiple roles with only one level of management.

(+) *EDLC* Advantages:

3. *Simple Structure:*

A process with only three possible role types is realistic for a *Flat Management* organization such as SRDs.

(+) *Agile* Advantages:

4. *Exploits Individual Capabilities:*

Since SRDs usually have *Flat Management* structures, this typically means that SRDs are less compartmentalized. In other words, all engineers can contribute to any part of the project. For example, a larger *Production* Software Organization may have a Quality Assurance department and a Software Development department that are completely decoupled. However SRDs are not as compartmentalized which allows managers to dynamically allocate resources based on the strengths and weakness of individual developers.

6. *Small Teams*:

Small teams are a good fit for SRDs and organizations with *Flat Management* structures; as team size grows, *Flat Management* structures become unrealistic.

(+) *XP* Advantages:

5. *Collective Code Ownership:*

Flat Management implies that all developers are on the same 'level' and developers typically do not manage other developers. This makes the *Collective Code Ownership* trait of *XP* a good fit for SRDs because any developer can easily change any other developers code without usurping any authority.

***2. Technical Managers*:**

(+) *RUP* Advantages:

5. *Emphasizes Visual Software Models*:

Technical Managers are more likely than non-technical managers to be proficient with visual modeling tools such as UML. This makes *RUP's* emphasis on Visual Software Models a good fit for SRDs.

(+) *Agile* Advantages:

4. *Exploits Individual Capabilities*:

Managers with technical backgrounds (such as former engineers) are necessary to fully exploit individual capabilities. This is because the manger must understand the technical domain as well as the developers' technical strengths and weaknesses to match development tasks with the best-suited developer.

4. *Pair Programming:*

Technical Mangers are better suited to know how to pair programmers together. For example a technical manager that understands the technical strengths and weaknesses of two developers could pair the developers so their strengths and weaknesses complement each other. This increases the impact of pair programming and makes the *Pair Programming* trait of *XP* well suited for SRDs (with respect to the *Technical Managers* SRD trait)

3. *Small Teams*:

(-) *RUP* Disadvantages:

7. *Complicated Structure:*

Small teams like those found at SRDs likely will not benefit from so many role options. As a result, the *Complicated Structure* of *RUP* is a disadvantage for SRDs.

(+) *EDLC* Advantages:

3. *Simple Structure*:

A process with a small number of roles is a natural fit for an organization with small teams such as SRDs.

(-) *TSP* Disadvantages:

2. *Document Intensive*:

A *Document Intensive* process may be difficult with *Small Teams* because there are fewer personnel resources. In addition small teams (especially those found in SRDs) are less likely to have technical writers to assist with documentation.

(+) *Agile* Advantages:

1. *Artifacts Minimized*:

Unlike *TSP, Agile* tries to minimize *artifacts* and documents. This trait makes *Agile Development* attractive for small teams that lack the resources and expertise to create large amounts of software artifacts and documentation.

5. *Adaptive*:

Often with a larger team, new discoveries about the system must travel from the developers through several levels of management before a decision is made. The decision must also travel back down to the developers before any action is taken. However, developers in a small team can quickly adapt to the latest information with minimal delay.

4. *Pair Programming*:

An *SRD* project, particularly an initial prototype, could have only one or two developers. As the number of developers on a project decrease, so does the feasibility of pair programming.

4. *High Autonomy*:

(+) *TSP* Advantages:

3. *Scripted Development:*

Scripted Development could be advantageous for organizations with high autonomy because it allows the managers to guide development (by tailoring the 'scripts'). At the same time, *Scripted Development* still gives the developers a high level of autonomy since the manager is not directly dictating the implementation details.

(-) *TSP* Disadvantages:

1. *Metric-Driven*:

High autonomy environments could make it difficult to collect and utilize performance metrics. A *High Autonomy* environment would require the developer to collect and track metrics without a manager driving the effort. *High Autonomy* environments also have less communication among managers and developers, so the rest of the team may never utilize the metrics developers collect.

(+) *Agile* Advantages:

4. Exploits Individual Capabilities:

High autonomy environments allow developers to work on the tasks they excel at, using the methods they feel more comfortable with. This *exploits individual capabilities* and makes this trait of *Agile Development* well suited for SRDs.

5. *Adaptive:*

Just as *small teams* create a more *Adaptive* environment, *High Autonomy* also creates a more *Adaptive* environment. Highly Autonomous Developers can quickly adapt based on current information because they don't' have to wait on communication or feedback form other team members.

(+) *XP* Advantages:

3. *Write Tests Before Code:*

XP emphasizes writing tests before writing software so the programmers receive immediate feedback during development[10]. However, in order to efficiently utilize immediate feedback, developers must be somewhat autonomous. The *High Autonomy* of SRDs creates a development environment that fully utilizes the *XP* trait of writing tests before coding.

(-) *XP* Disadvantages:

4. *Pair Programming:*

Developers in SRDs likely are used to such a high level of autonomy that developers might resist adopting *Pair Programming*.

4. *Exploratory*

(+) *RUP* Advantages:

1. *Iterative:*

Iterative processes inherently promote exploratory projects because developers can try new ideas at any stage in the development cycle.

(+) *EDLC* Advantages:

1. *Iterative:*

EDLC's iterative trait also promotes an exploratory environment.

4. *Produces Working Software Early*:

Producing working software early in the development cycle promotes *exploratory* projects because teams can try out new ideas at the beginning of the lifecycle. This is the best time to try new ideas because there is still time to take another approach if the new idea does not work.

(+) *TSP* Advantages:

1. *Iterative:*

TSP's iterative trait also promotes an exploratory environment.

(-) *TSP* Disadvantages:

4. *Document Intensive*:

TSP requires the development team to create a large number of documents. This could discourage *exploratory* behavior because developers may be less likely to try new ideas.

7. *Metric-Driven*:

Collecting metrics for *exploratory* projects is difficult because there are so many unknowns. For example, tracking *SLOCs* per day for a developer may not be useful because some days the developer is learning about the problem space while other days the developer is implementing (or coding) the solution to a known problem.

(+) *Agile* Advantages:

1. *Iterative*:

Agile's iterative trait also promotes an exploratory environment.

3. *Artifacts Minimized*:

Agile encourages developers to try new ideas by minimizing the artifacts the developers must create.

4. *Exploits Individual Capabilities*:

If developers are allowed to work on areas that fit their strengths, they can explore that aspect of the project in-depth.

5. *Adaptive*:

Adaptive processes promote an *exploratory* environment. Developers can try new ideas without putting the final project deliverable at risk because the project can *adapt* to new discoveries or revert to a previous approach.

(-) *Agile* Disadvantages:

3. *Artifacts Minimized*:

The previous section listed the *Artifacts Minimized* trait of *Agile Development* as an advantage for *exploratory* projects. However, this trait can be a disadvantage for *exploratory* projects if taken to the extreme. The primary

goal of *exploratory* and *R&D* projects is to learn more about a problem or a new technology. However this information may never be utilized if there are no artifacts or documents created during or after development.

4. *Exploits Individual Capabilities*:

   This could discourage *exploratory* environments because it keeps people working in the area they feel most comfortable, which may not promote creativity and *exploration*.

(+) *XP* Advantages:

5. *Collective Code Ownership*:

   Collective Code Ownership could encourage *exploratory* projects because it allows all developers to try new ideas despite their project focus.

6. *Continuous Integration*:

   The *Continuous Integration* trait of *XP* encourages *exploration* just as the *Produces Working Software Early* trait of *EDLC*.

(-) *XP* Disadvantages:

1. *Design Simplicity*:

   XP emphasizes design simplicity. However the simplest design may not be the best approach to explore a particular problem space. For example, an *exploratory* Research and Development project may require a complicated design in order to achieve an unprecedented result.

4. *Pair Programming:*

   Many believe that *Pair Programming* produces higher quality code, however it could discourage *exploration*. Developers may be less likely to try out new ideas if they

must implement and justify those ideas with another developer.

5. Unknowns (Requirements, Feasibility, Future):

(+) *RUP* Advantages:

1. *Iterative*:

RUP's iterative trait is also well suited for projects that have unknown requirements, feasibility and future because the project can mature and grow as the unknowns resolve.

5. *Emphasizes Visual Software Models*:

Visual Software Models are a good way to organize high-level designs without committing to an implementation. This is a good fit for SRD projects that have unknown requirements, feasibility and future because they can modify implementation details as the *unknowns* resolve.

(-) *RUP* Disadvantages:

2. *Rigid Roles and Artifacts*

A process that requires specific artifacts and roles could be difficult for a project with so many *unknowns*.

4. *Document Intensive:*

A process that emphasizes too much documentation may not be a good fit for *Research and Development* projects because they have so many *unknowns* and much of the documentation will drastically change.

(+) *EDLC* Advantages:

1. *Iterative*:

EDLC's iterative trait is also well suited for projects that have unknown requirement, feasibility and future.

2. *Incremental:*

R&D projects frequently have unknown feasibility. The *incremental* trait of *EDLC* allows the organization to implement the known feasibility portions, independent of unknown feasibility portions.

4. *Produces Working Software Early:*

For projects with unknown requirements and feasibility, a working software system (but with minimal functionality) is advantageous in gathering system requirements and determining the feasibility of an idea.

(+) *TSP* Advantages:

1. *Iterative*:

TSP's iterative trait is also well suited for projects that have unknown requirement, feasibility and future.

(-) *TSP* Disadvantages:

2. *Document Intensive:*

As mentioned for *RUP,* a process that emphasizes too much documentation may not be a good fit for R&D projects because they have so many *unknowns* and much of the documentation will drastically change.

5. *High Process Overhead:*

Projects that have an unknown future such as *R&D* projects may not benefit from the additional *process overhead* created by the *TSP*. For example, many *R&D* projects are never implemented in the field or in a production environment. In these cases much of the artifacts and documents created with *TSP* will never be used.

7. *Metric-Driven*:

   Projects with unknown feasibility are not well suited for *Metric-Driven* processes. Since the feasibility is unknown, many of the common metrics are also unknown. For example, programmer efficiency cannot be measured if you do not know if the programmer's task is even feasible.

(+) *Agile* Advantages:

1. *Iterative*:

   Agile's iterative trait is also well suited for projects that have unknown requirement, feasibility and future.

2. *Incremental:*

   Similarly to *EDLC,* the *incremental* trait of *Agile Development* allows the organization to implement the parts of the system that have known feasibility independent of the parts of the system with unknown feasibility.

3. *Artifacts Minimized:*

   SRD projects with unknown futures may benefit from processes that minimize artifacts. If the project is cancelled or never implemented in a production environment, then the artifacts generated during development are likely never utilized.

5. *Adaptive:*

   An adaptive process is particularly useful in a project with unknown feasibility. The project team may discover early in the project that their original idea is not feasible, however *Agile Development* lets the team adapt to the new information and still implement a scaled version of the original idea.

(+) *XP* Advantages:

3. *Write Tests Before Code:*

Writing tests before coding may be an efficient method to determine the feasibility of an *R&D* project

6. *High Risk*:

(+) *EDLC* Advantages:

1. *Produces Working Software Early:*

This trait of *EDLC* is a way to mitigate the high risks associated with *R&D* projects. Producing working software early allows the development team to deliver a working software system (yet not fully functional) if they determine that some features are not possible.

5. *Prioritized Functionality:*

Project teams can assign higher priorities to the "must have" functionality to mitigate the risk of missing deadlines.

(+) *TSP* Advantages:

2. *Document Intensive:*

One of the risks of *R&D* projects is that the idea may not be feasible. If this does occur, the documents created in the *TSP* may be useful to justify why the idea was not feasible.

(+) *Agile* Advantages:

5. *Adaptive*:

The adaptive nature of Agile mitigates the risk of an unfeasible project. If the process can adapt with new discoveries, then the project can scale down to a realistic product.

(-) *Agile* Disadvantages:

3. *Artifacts Minimized:*

SRD Projects that have minimal artifacts could make it difficult to justify to a funding sponsor why a project is unfeasible.

Figure 2: Matrix of *SRD* traits to Software Engineering Process traits

Software Engineering Processes

SRD Traits			RUP	EDLC	TSP	Agile	XP
	Org Traits	Flat Management	- Rigid roles/artifacts - Complicated structure	+ Simple structure		+ Exploits indv. capabilities + Small Teams	+ Collective code ownership
		Technical Managers	+ Emphasis on viz models			+ Exploits indv capabilities	+ Pair programming
		Small Teams	- Complicated structure	+ Simple structure	- Documentation intensive	+ Artifacts minimized + Adaptive	- Pair programming
		High Autonomy		+ Working software early	+ Scripted development - Metric driven	+ Exploits indv. capabilities + Adaptive	- Write tests before code - Pair programming
	Project Traits	Exploratory	+ Iterative	+ Iterative + Working software early	+ Iterative - Metric driven	+ Iterative + Artifacts minimized + Exploits indv. capabilities + Adaptive	+ Collective code ownership + Continuous integration - Pair programming - Design simplicity
		Unknowns: (*Feasibility Requirements and Future)*	+ Iterative + Emphasis on viz models - Rigid roles/artifacts - Documentation intensive	+ Iterative + Incremental + Working software early	+ Iterative - Documentation intensive - High process overhead - Metric driven	+ Iterative + Incremental + Artifacts minimized + Adaptive	+ Write tests before code
		High-Risk		+ Working software early + Prioritized functionality	+ Documentation intensive	+ Adaptive - Artifacts minimized	

5. CONCLUSION

Chapter 1: Results

This paper does not suggest which Software Engineering Process is definitively the best solution for all SRDs. However, this section interprets the matrix in Figure 2 to find common advantages and disadvantages for SRDs from the software processes discussed in this paper. In order to focus on the traits most applicable to SRDs, this section only considers the process traits that appear twice or more in the matrix from Figure 2. Below are the common process traits and how they relate to SRDs. As mentioned in Section 2, the analysis of the process traits presented in this section can be subjective and are largely based on the author's experience with *the Applied Research Labs*.

Positive Process Traits for SRDs

The following traits appear twice or more as positive (or advantageous) Software Engineering process traits for SRDs in Figure 2:

+ *Iterative*: All of the five Software processes discussed in this paper are *iterative*. The *Exploratory* and the *Unknown* rows (row #5 and #6) in Figure 2 both contain the *iterative* trait. *Iterative* processes are a good fit for SRDS because they promote *exploratory* development and are well suited to deal with the *unknowns* of SRD projects.

+ *Emphasis on visual models*: *RUP* is the only process in this paper that explicitly emphasizes visual software models, such as UML. However, many other processes encourage using visual software models. Visual software models communicate high-level design concepts between management and developers; even when the low level requirements are not defined. This makes processes that emphasize visual software models well suited for SRDs. In addition, creating and

interpreting visual software models such as UML, often requires a technical background. Processes that emphasize visual software models are well suited for SRDs because SRDs typically have technical managers that are proficient in visual modeling for software.

+ *Simple structure*: *EDLC* is unique from the other processes discussed because of its simple structure with a small number of defined roles. SRDs typically have a *Flat Management* structure and Small Teams, which makes processes with a *Simple Structure* such as *EDLC* a good fit for SRDs.

+ *Produces working software early:* Producing working software early in the development cycle promotes *exploratory* projects because teams can try out new ideas at the beginning of the lifecycle; this allows the team to revert to a previous approach if the new idea is unsuccessful. This trait also allows SRDs to resolve the *unknowns* early in the project. In addition, this trait helps mitigate the risks of SRD projects because at any stage, the team can deliver a working software system (yet not fully functional) if they determine that some features are not possible.

+ *Exploits individual capabilities:* Most software processes are personnel independent, meaning the process is applicable no matter who the developers are. However, *Agile Development* tries to utilize and exploit individual strengths and weaknesses to improve the process and the final product. The SRD traits of *Flat Management, Technical Managers, High Autonomy* and an *Exploratory* environment contribute to make processes that *exploit individual capabilities* a good fit for SRDs.

+ *Artifacts are minimized*: *Agile Development (*and *XP* because it is a type of *Agile Development*) attempts to reduce the number of software artifacts (or documents describing the software system). *Agile* processes still may create software

artifacts, however artifacts are kept to a minimum and are only created if absolutely necessary.

+ *Adaptive*: The terms *adaptive* and *iterative* are similar, however there is a subtle and important difference. *Iterative* means the development cycle will continuously repeat until the project is complete. However *adaptive* means the actual process (including artifacts, architecture, code and roles) might change based on feedback from the current system and the current process. As seen in Figure 2, the *adaptive* trait of *Agile Development* matches with the SRD traits of *Small Teams, High Autonomy, Exploratory, Unknowns* and *High Risk*. As seen in Figure 2, the *adaptive* trait matches to more SRD traits than any other process trait. This makes *adaptive* processes an attractive option SRDs.
+ *Collective Code Ownership*: *XP* is the only process covered that explicitly calls for *collective code ownership*; which means all developers can change any piece of code at any time. *Collective code ownership* is best for organizations with a *Flat Management* structure because all developers are on the same level; this makes *collective code ownership* more realistic and manageable. *Collective code ownership* also matches to the *Exploratory* trait in Figure 2 because any developer can spontaneously "explore" new ideas since everyone has access to all the source code.

Negative Process Traits for SRDs

The following traits appear twice or more as negative (or disadvantageous) Software Engineering process traits for SRDs in Figure 2:

- <u>*Rigid Roles and Artifacts*</u>: *RUP* has *rigid role* definitions and *artifact* requirements. In Figure 2 you can see this trait is a negative for the *Flat Management* and *Unknown* SRD traits. Processes with *Rigid Roles* may not be a good fit for SRDs because often SRDs only have two types of employees: Managers and Engineers. Processes with *rigid artifacts* may not be a good fit for SRD projects that have so many *unknowns* since many of the requirements will change. In addition since SRD projects have an *unknown future*, many of the *artifacts* created may never be used. Perhaps a better approach for SRDs is to give the team the freedom to determine which artifacts to create based on the most-likely future of the project and based on which requirements are still unknown.
- <u>*Complicated Structure*</u>: This process trait appears twice under the *RUP* column. It is listed as a negative for SRDs with *Small Teams* and *Flat Management* because small teams with *Flat Management* are less capable to deal with the numerous role responsibilities of *RUP*.
- <u>*Document Intensive*</u>: This trait is matched as a negative to the *Unknowns* SRD trait because creating too much up-front documentation for projects with so many unknowns may not be useful because of changing requirements and unknown future of the project.

  [Note that the *Document Intensive* trait is listed as a positive under the *TSP* for the *High Risk* trait of SRD projects. This is because high risk processes could benefit from a *Document Intensive* process because it provides more traceability and

justification to the funding sponsor should a project not succeed. However, overall *Document Intensive* processes are not well suited for SRD projects.]

- *Metric Driven*: This process trait is listed under the *TSP* column matched as a negative to the *High Autonomy*, *Exploratory* and *Unknown* SRD traits. Metrics are often not utilized in *High Autonomy* environments because other developers likely will not benefit from the metric collection of individuals with high autonomy. Collecting *metrics* on *exploratory* projects may be unreliable because there is a lack of historic performance data to gauge the metrics against. In addition, metric collection for projects with many *unknowns* is challenging because during most of the project you do not know which metrics to collect.
- *Pair Programming*: This process trait is listed under *XP* and matches as a negative to the *Small Teams* and *High Autonomy* SRD traits. *Pair Programming* would be difficult for SRDs with small teams due to the lack of developer resources. In addition, an environment with developers accustomed to high levels of autonomy, convincing the team to adopt *Pair Programming* would be difficult. Despite these drawbacks, many believe it is worth the effort and cost to implement a *Pair Programming* system [2].

Based on the analysis of Figure 2, an ideal software process for SRDs would have the following traits: *Iterative*, *Emphasis on visual models, Simple structure, Produces working software early, Exploits individual capabilities, Artifacts are minimized, Adaptive,* and *Collective code ownership*. In addition, an ideal software process for SRDs would NOT include the following traits: *Rigid roles and artifacts, Complicated structure, Document intensive, Metric-driven* and *Pair programming*. These process traits consist of

the positives and negatives that appeared twice or more in the process traits vs. SRD organization matrix in Figure 2.

Chapter 2: Example of a *Tailored* Software Process

This Chapter presents an example of a custom process for the author's employer, The Applied Research Labs (*ARL)*. Based on the results mentioned in chapter 1, an SRD process should include the following traits: Iterative, Emphasis on visual models, Simple structure, Produces working software early, Exploits individual capabilities, Artifacts are minimized, Adaptive, and Collective code ownership. The XP process is the closest to these traits, so XP is used as a baseline for the Tailored process. The XP process already meets the following traits: *Iterative, Simple structure, Artifacts are minimized, Adaptive and Collective code ownership* and *Produces working software early*. However, XP does not explicitly exploit individual capabilities, or emphasize visual models; these traits must be addressed in the Tailored process for ARL. .

In order to exploit individual capabilities, the managers must understand the developers' strengths and weaknesses. The software process tailored to *ARL* might require each developer to complete a skill-set inventory. This would allow project managers to better understand the developers' strengths and weaknesses and assign development tasks accordingly.

The software process tailored for *ARL* must also *emphasize visual models*. One example of this is to incorporate visual models into the requirements specification documents. In fact some SRD projects could use visual models for the entire requirements specification. For example, the process tailored to *ARL* might require a *data-flow* diagram to be created for each new binary application checked into the source code revision system. This requirement would inherently *emphasize visual models* for the software process.

This chapter presented an example of tailoring a software process to fit a particular SRD, such as *the ARL*. The *XP* process provides a good starting point because it already meets most of the needs of SRDs. The *XP* process can be tailored to *exploit individual capabilities* by creating skill-set inventories. The *XP* process can also be tailored to *emphasize visual models* by requiring a *data-flow* diagram whenever a developer adds a new binary to the source code revision system.

Chapter 3: Future Research

Chapter 3 presents some future research opportunities related to Software Engineering and SRDs.

Evaluating Visual Models for SRDs

Text-only requirements specification is often unfeasible for SRD projects. This is especially true at the beginning of the project due to unknown requirements. Visual models are more realistic than text-only for representing SRD project requirements. For example, this paper categorized *RUP's Emphasis on Visual Models* as a positive trait for SRDs because visual models quickly communicate high-level ideas without committing

to a particular implementation. The role of visual modeling in SRD projects is a future research opportunity. Future research could include a survey of SRD project managers and developers on which visual modeling techniques are successful.

Shifting Artifact Focus from the Beginning to the End of SRD Projects

This paper discussed the three *unknowns* of SRD projects: *requirements*, *feasibility* and *future*. Each of these *unknowns* makes it difficult to produce software artifacts for SRD projects until the later phases of the project when many of the *unknowns* are resolved. Traditional Software Engineering Processes focus on creating software artifacts during the first phases of the project. However, a future research opportunity is to consider shifting the focus of creating software artifacts from the beginning of the SRD project to the end of the SRD project. Software artifacts are easier to create at the end of SRD projects because many of requirements are finally defined. In addition, the artifacts are likely more useful at the end of an SRD project because the project's future is more defined. Also, creating artifacts towards the end of SRD projects communicates the working solutions of problems to the developers of future simmilar projects.

Chapter 4: Summary

Software Research and Development Organizations (or SRDs) have unique goals that differ from the goals of Production Oriented Software Organizations. SRDs focus on exploring the unknown, while Production Software Organizations focus on implementing solutions to known problems. These unique goals call for reevaluating the role of Software Engineering Process for SRDs.

Section 1 of this paper defined SRDs as organizations that primarily develop exploratory, non-production software. Next this paper characterized SRDs based on their *Organizational Traits* and their *Project Traits*. SRD *Organizational Traits* include *Flat Management*, *Technical Managers, Small Teams* and *High Autonomy*. SRD *Project Traits* include *Exploratory, Unknowns (Requirements, Feasibility, Future)* and *High Risk*.

Section 3 of this paper presented a summary of six common Software Engineering Processes and Models: *Waterfall Model, Rational Unified Process (RUP), Evolutionary Delivery Cycle (EDLC), Team Software Process (TSP)*, *Agile Development* and *Extreme Programming (XP)*. Next, section 3 listed some unique traits of each of these processes.

Section 4 mapped the *Organizational* and *Project* traits of SRDs to the unique traits of the Software Engineering processes. Figure 2 provides a graphical view of this mapping.

Section 5 presented the analysis results indicating that an ideal process for SRDs would include the traits: *Iterative*, *Emphasis on visual models, Simple structure, Produces working software early, Exploits individual capabilities, Artifacts are minimized, Adaptive,* and *Collective code ownership*; and NOT include the following traits: *Rigid roles and artifacts, Complicated structure, Document intensive, Metric-driven* and *Pair programming*. It is important to note that this paper heavily relied on the author's personnel experiences at *ARL* and some of this paper's analysis is subjective. However, this information in this paper is likely applicable to all SRDs. Next, section 5 presented two future research opportunities: 1) Evaluating the role of visual modeling in SRD projects and 2) Consider shifting the focus of creating software artifacts at the beginning of the SRD project to the end of the SRD project.

Every SRD is different and could require selecting and tailoring a process that fits their unique organization and project characteristics. This paper simply introduces what is

available for SRDs and gives an example of how to analyze the pros and cons of a software process for an SRD. A primary goal of any software organization is to find the "sweet-spot" between too much process and too little process and this is especially true for SRDs.

References

[1] T., Li, Q., Boehm, B., Yang, Y., He, M., and Moazeni, R. 2009. Productivity trends in incremental and iterative software development. In Proceedings of the 2009 3rd international Symposium on Empirical Software Engineering and Measurement (October 15 - 16, 2009). Empirical Software Engineering and Measurement. IEEE Computer Society, Washington, DC, 1-10. DOI= http://dx.doi.org/10.1109/ESEM.2009.5316044

[2] What is Extreme Programming? Retrieved June 15, 2010 from: http://xprogramming.com/xpmag/whatisxp

[3] Scientific American September 2002 Vol 287 No. 3 (a special issue on time) in the Antigravity column by Steve Mirsky.: "Einstein's Hot Time" pg. 152

[4] Shtub, A. & Bard, J. & Globerson,S (2005) Project Management: Process, Methodologies and Economics, Pearson Prentice Hall, USA.

[5] Rockwood, Justin. Choose Your Weapon Wisely: A handbook for determining the right software development method best suited for your team, client, and project (Independent Study Paper). Pittsburgh, PA: School of Computer Science, Carnegie Mellon University.

[6] Kruchten, P. 2000 The Rational Unified Process: an Introduction, Second Edition. 2nd. Addison-Wesley Longman Publishing Co., Inc.

[7] Bass, L., Clements, P., and Kazman, R. 2003 Software Architecture in Practice. 2. Addison-Wesley Longman Publishing Co., Inc.

[8] Introducing the Personal Software Process, Watts S. Humphrey. Software Engineering Institute, Carnegie-Mellon University. Annals of Software Engineering 1(1995)311-325.

[9] Dagnino, A. 2002. An Evolutionary Lifecycle Model with Agile Practices for Software Development at ABB. In Proceedings of the Eighth international Conference on Engineering of Complex Computer Systems (December 02 - 04, 2002). ICECCS. IEEE Computer Society, Washington, DC, 215.

[10] Do the Simplest Thing That Could Possibly Work Retrieved June 10, 2010 from: http://xprogramming.com/Practices/PracSimplest.html

[11] Vliet, H. 2008 Software Engineering: Principles and Practice. 3rd. Wiley Publishing.

[12] Wysocki, R. K. 2009 Effective Project Management: Traditional, Agile, Extreme. 5th. Wiley Publishing.

[13] Frank, Ronald I. September 2005 What Is Research? Technical Report. Pace University School of Computer Science and Information Systems.

[14] Standing Committee on Program and Technical Review of the U.S. Army Natick Research, Development and Engineering Center, National Research Council. World Class Research and Development: Characteristics for an Army Research, Development, and Engineering Organization, 1996.

Vita

Michael Willis is from Austin, Texas born to John and Sharon Willis of Austin, Texas. Michael attended Texas A&M University in College Station, Texas where he received a Bachelor of Science in Computer Engineering in May 2004. Currently Michael is a Software Engineer at The Applied Research Labs-University of Texas at Austin.

Permanent address: 5733 Toscana Ave, Austin, TX 78724

This report was typed by Michael Willis