



IST-2001-32133

GridLab – A Grid Application Toolkit and Testbed

D14.2 “Methodology for Quality Assurance”

Author(s): Jarek Nabrzyski
Title: Methodology for Quality Assurance
Subtitle: GridLab Project Management
Work Package: 14
Lead Partner: PSNC
Partners: ALL
Filename: GridLab-14-D.2-0001
Version: 1.0
Config ID: GridLab-14-D.2-0001-1.0
Classification: IST

Abstract: This document presents a general project management methodology used in the GridLab project to provide the highest possible quality assurance.

Project Manager: Jaroslaw Nabrzyski
Institute of Bioorganic Chemistry PAS
Poznan Supercomputing and Networking Center
ul. Noskowskiego 12/14
61-704 Poznan, Poland
Phone: +48 61 858 2072, Fax: +48 61 852 5954
Email: naber@man.poznan.pl



- 1. Introduction..... 4
 - 1.1. SOFTWARE DEVELOPMENT 7
 - 1.2. BACKGROUND 8
 - 1.3. COMPARISON TO OTHER SOFTWARE ENGINEERING MODELS 9
- 2. Gecko Lite: Underlying Principles..... 10
 - 2.1. FLEXIBILITY AND ADAPTABILITY.....10
 - 2.2. APPLICABILITY AND SCALABILITY.....10
 - 2.3. REQUIREMENTS MANAGEMENT12
 - 2.4. CONFIGURATION MANAGEMENT19
 - 2.5. PROJECT MANAGEMENT AND TRACKING21
 - 2.6. QUALITY ASSURANCE.....22
 - 2.7. SOFTWARE ERROR/DEFECT CLASSIFICATION24
 - 2.8. ACCEPTANCE CRITERIA24
 - 2.9. USER GUIDE/P ROGRAMMER REFERENCE G UIDE DEVELOPMENT25
- 3. Life Cycle Overview 27
 - 3.1. PROJECT INITIATION PHASE.....27
 - 3.2. PROJECT PLANNING PHASE.....28
 - 3.3. CONCEPT DEFINITION PHASE (OPTIONAL)28
 - 3.4. SYSTEM REQUIREMENTS PHASE (OPTIONAL)28
 - 3.5. SOFTWARE R EQUIREMENTS ANALYSIS/DEFINITION PHASE28
 - 3.6. BUILD PLANNING P HASE29
 - 3.7. BUILD DEVELOPMENT PHASE29
 - 3.8. BUILD INTEGRATION PHASE.....30
 - 3.9. TEST PHASE.....30
 - 3.10. DELIVERY AND INSTALLATION PHASE.....32
 - 3.11. PROJECT CLOSURE P HASE.....32
- 4. Common Problems and Solutions 33
- 5. Implementation Phases 36
 - 5.1. IMPLEMENT SOFTWARE CONFIGURATION MANAGEMENT (CM).....36
 - 5.2. IMPLEMENT SOFTWARE REQUIREMENTS MANAGEMENT (RM).....38
 - 5.3. IMPLEMENT SOFTWARE ENGINEERING TRAINING (TR)40
 - 5.4. IMPLEMENT SOFTWARE DESIGN P RACTICES (DE)41
 - 5.5. IMPLEMENT SOFTWARE DEVELOPMENT PRACTICES (DV)42
 - 5.6. IMPLEMENT SOFTWARE TESTING PRACTICES (ST)43
 - 5.7. IMPLEMENT SOFTWARE DELIVERY & INSTALLATION PHASE PROCESSES (DL)45
 - 5.8. IMPLEMENT SOFTWARE PROJECT MANAGEMENT, TRACKING AND OVERSIGHT AND QUALITY ASSURANCE PRACTICES (PM).....46

- Appendix A: Software Engineering Implementation Checklists 47
 - SOFTWARE CONFIGURATION MANAGEMENT (CM)48
 - SOFTWARE REQUIREMENTS MANAGEMENT (RM)50
 - SOFTWARE TESTING PRACTICES (ST)51
 - SOFTWARE ENGINEERING TRAINING (TR)53
 - SOFTWARE DELIVERY & INSTALLATION PHASE PROCESSES (DL)54
 - SOFTWARE DESIGN PRACTICES (DE)55



SOFTWARE DEVELOPMENT PRACTICES (DV)	56
SOFTWARE PROJECT MANAGEMENT, TRACKING AND OVERSIGHT AND QUALITY ASSURANCE PRACTICES (PM)	57
Appendix B: Life Cycle Phase Entry/Exit Criteria and Products	58
Appendix C: Summary of Gecko Lite Principles	64

1. Introduction

GridLab project is a very hard project to manage. Among many difficulties the following ones seem to be the most important:

- Multi-nationality and multi-culture
- Team distribution over Europe and US
- Many different approaches to project management built over the years in all the GridLab teams
- Limited access to all team members
- Many workpackages (12 technical workpackages)
- Many different partners (15+)

Most of the GridLab's work is software development. Software development is pretty complex human artifact. Taking into account all the difficulties, there is a high risk in GridLab to progressively project many mistakes over the whole project management. Thus, it is extremely important to adapt to GridLab the most accurate and proven methodology. Adherence to a well-defined methodology improves the likelihood that a project will succeed.

The good thing about the GridLab is that the involved teams have proven their ability to work together. We have been doing together many demos and common unfunded and funded projects, where a close collaboration between developers and deployers was a fact.

In GridLab this success is being proved again, however the scope and the size of the project has never been such large as it is now. That's why we have decided to spend some time on choosing the proper methodology. We conducted discussions on applying the Rational Unified Process and others. But Rational Unified Process has turned out to be too "heavy", the project management itself would lead here to overloading the staff with working on too many documents and etc. Also, it is too much focused on the Rational Rose tools and, as such, can not be used in GridLab environment, where participants are used to many different tools. So, we had to find a methodology, which is not tool dependent.

After a long discussion we came up to the decision of using the GeckoLite methodology and adopted it to the GridLab project. GeckoLite eliminates much of the finer details and additional bureaucracy associated with heavy weight methodologies. The end result is a methodology that provides a high degree of structure, rigor, documentation and engineering process support without adding significantly to the cost of the overall project. GeckoLite is used for example by the Information Power Grid project in the States.

In this document the GeckoLite methodology has been described. The following terms are used to express the relations with the GridLab project management and structure:



- The *customer* is a GridLab application team and the potential GridLab software deployers
- *Development team* - all GridLab developers
- *Project Manager* – Grid Lab Project Manager
- *Team lead* – GridLab Technical Board Chair
- *Configuration Manager and Quality Assurance Officer* – Each WP nominates the Configuration Manager and the

Whole GridLab management structure is defined in the Annex1 of the project contract.



The complete Gecko Lite Methodology and all of its components can be located and is available at:

<http://intranet.AMTI.com/GeckoLite>

1.1. *Software Development*

Computer software is potentially the most complex human artifact. If we think of a computer program as a virtual machine and we then consider each line of code as though it were a moving part, a large computer program could contain more than a million lines of code! Imagine a real machine such as the space shuttle with that many working parts. We would expect that a machine like the space shuttle would be carefully designed, assembled, planned and tested. And in fact it is, but not all software development receives the same level of methodological rigor.

Software development is typically viewed as an art form. Software developers are usually referred to as "authors" of their code. Software design and development does require a high degree of creativity and ingenuity. However, the process behind software development is often overlooked or skipped entirely. Software projects usually focus on "creating" the code. After all, the code is what is delivered. What is lacking in so many projects is a structure and process to guide and channel the creative solutions.

Software engineering process do not dampen creativity, they foster it. Focusing on the right aspects of the problem and the solution at the right point in time and with the right people allows maximum use of creative solutions so that development teams do not spend most of their time solving process, organizational and programmatic problems that have already been solved many times before.

The application of a good methodology becomes essential to maximizing the problem solving potential of every member of the development team. The methodology functions like the musical score for a symphony. Individually, the members of the orchestra may be highly talented musicians, but collectively their true potential to make music is captured in the score.

1.2. Background

Gecko Lite Principle	
1	Adherence to a well-defined methodology improves the likelihood that a project will succeed.

Gecko Lite is AMTI's standard software engineering methodology. This methodology is named after the Gecko lizard because the lizard is small, lightweight and can move relatively quickly. The philosophy behind Gecko Lite as a software engineering methodology is to provide software engineers and developers with a methodology with similar characteristics. Gecko Lite provides the discipline, structure and management necessary to develop and deploy successful high quality systems while imposing only minimal levels of management and engineering discipline overhead to achieve these results. The goal of Gecko Lite is to facilitate the creative and iterative process that is essential to developing high quality client solutions

Software projects require structure, timeliness and client involvement to succeed. Discipline is needed to insure manageability, predictability and quality. Software design and development, by definition, is a creative and intellectual process. These creative facets must, be tempered with a disciplined approach to gain the greatest advantage. Developers typically fear that software and system development will be reduced to its lowest common denominator. Gecko Lite is designed to empower the project team and its members with the flexibility to adapt to changing requirements.

The often-misunderstood process of developing software can now become collaboration between, the client and the software development team. AMTI's philosophy of software development is that when the customer is involved, the end-result is the customer's product. If the client is not involved, it is no-one's product. Gecko Lite requires client involvement in each major step of the process.

Gecko Lite has been developed over a period of several years and applied successfully on numerous projects. It is based on best practices and is currently in its third major revision. Gecko Lite works well on small to medium and even large projects to provide developers with a software engineering process that is well grounded in commonly accepted industry practices of successful development projects.

Successful development projects are those that come within budget and schedule with a margin of error of not more than twenty percent. Although twenty percent over budget or schedule may seem quite high, by comparison to the majority of software projects in industry and government, this is significantly better than the majority of projects. Additional success qualifiers include the ability to obtain signed customer acceptance of the system after successfully demonstrating that the system satisfies all documented requirements and acceptance criteria and that the system has been judged stable and reliable enough to commence operational use of the software.

Gecko Lite employs its own written procedures for all aspects of the software development life cycle from planning through delivery, installation and maintenance. The Gecko Lite documentation suite is based on a set of document templates designed to speed up the documentation process and improve completeness and consistency of documentation. Reasonable and maintainable documentation is of paramount importance in developing sustainable systems that can be readily maintained throughout the life span of the system. Clear, concise and consistent documentation also facilitates better communication between the development team and the customer.

1.3. Comparison to other Software Engineering Models

Gecko Lite follows the basic concepts of requirements definition, design, managed and iterative development, various phases of testing and formal acceptance and delivery. Gecko Lite incorporates aspects of project planning, tracking, oversight, software quality assurance through peer reviews and periodic software engineering process assessments. Gecko Lite is intended to follow the spirit of larger methodologies, which may have been assessed at Level Two, or Three of the Carnegie Mellon, Software Engineering Institute (CMM/SEI) Capability Maturity Model (CMM). By following the spirit of its larger counterparts, Gecko Lite incorporates the core processes and activities generally accepted as standard industry practice. However, Gecko Lite eliminates much of the finer details and additional bureaucracy associated with heavy weight methodologies. The end result is a methodology that provides a high degree of structure, rigor, documentation and engineering process support without adding significantly to the cost of the overall project.

Like any methodology, Gecko Lite provides methods and guidance for each phase of the software development life cycle. Gecko Lite does not depend upon any specific technique or product for any phase of development. This makes Gecko Lite an open methodology so that suitable techniques and products can be applied within Gecko Lite's framework.

2. Gecko Lite: Underlying Principles

2.1. Flexibility and Adaptability

Gecko Lite Principle	
2	A methodology must be flexible and adaptable.

All projects are not the same. Any good methodology should be flexible and adaptable so that changes can be made to the basic process or accompanying standards to adapt to the specific needs of a project. However, changes made to Gecko Lite or its supporting materials and processes should be made judiciously and only after concluding that the standard Gecko Lite process or materials will not satisfy specific process needs.

Developers and software engineers should resist the temptation to simply change something because they do not completely understand the mechanisms or the philosophy behind a particular aspect of the methodology. Changing the methodology too much can result in much wasted effort trying to redefine a process has already been refined significantly to take into account the issues and needs of a wide variety of software engineering projects.

2.2. Applicability and Scalability

Gecko Lite is appropriate for a wide range of projects of varying size. Small, medium and even large development teams can effectively utilize Gecko Lite. The following table defines very small, small, medium and large-scale projects in terms of the number of developers, or total life cycle of the project.

Type	Developers	Total Life -Cycle	Methodology
Very Small (short duration/standalone)	1-3	< 3 months (stand-alone application)	eXtreme Gecko
Very Small (long duration/standalone)	1-2	> 3 months (stand-alone application)	eXtreme Gecko
Very Small	1-3	> 3 months (part of a larger)	Gecko Lite

(long duration/tightly coupled)		set of tightly coupled applications)	
Small	3-14	> 3 months	Gecko Lite
Medium	15-30	> 3 months	Gecko Lite
Large	31+	> 3 months	Gecko Lite

In the case of Small, Medium and Large-scale projects, Gecko Lite is appropriate. Very Small projects should only use Gecko Lite if the project is part of a set of separate applications that are tightly coupled. If a Very Small project is effectively a stand-alone piece of software such as a utility program or other small application, then Gecko Lite would require too much overhead effort. In the case of a Very Small stand-alone project, eXtreme Gecko would be appropriate. Extreme Gecko is a very lightweight methodology intended for very small projects.

2.2.1. Small to Large Projects

Gecko Lite Principle	
3	Large projects with 15 or more developers should be broken into smaller segments (applications) for ease of management.

Any large project consisting of 15 or more developers should be broken into a number of segments or separate components or applications for ease of management. With each application, the requirements, design, development and testing can all be managed with the Gecko Lite life cycle. Coordination among multiple related applications or software components is simply a matter of coordinating release dates, interfaces and functionality to be addressed in any given release or build.

Very large projects that consist of multiple separate, but tightly coupled software components developed by different teams can be integrated and the definition of their interfaces managed easily with the use of Gecko Lite's "*Interface Control Document*" (ICD). A separate ICD should be created to define the interface between each unique pair of software components, systems or applications. Large projects comprising many components or segments also benefit from the optional development of a "*System Operations and Concept Specification*." This document is an exercise in determining the scope and goals of the project; determining what problem is to be solved and what are the "right" requirements.

Once the concept or vision of the system has been clearly, discretely documented for all stakeholders to review, the System Requirements can be developed. The System Requirements Specification defines the total hardware, software and other types of requirements that must be addressed by the total compliment of applications, segments or software components. This complete body of requirements is then subdivided and "allotted" to the individual applications. The System Requirements define the roles of the individual applications and are further refined in the "*Master Software Requirements Specification*" for each individual application.

2.3. Requirements Management

2.3.1. What are Requirements?

Gecko Lite Principle	
4	Requirements are the features or behavior of the system that are externally observable.

Requirements are generally defined as those features, aspects or behavior of the system that are externally observable by the users. Of course if the system to be built is an application program interface (API), then there is no externally observable behavior in the sense that something appears on a computer screen. In the case of API's or embedded systems, the externally observable behavior is simply the interface to a client application. Design information is generally any aspects or behavior of the system that is internal, invisible or otherwise not observable from a perspective outside of the external interface(s) of the system.

2.3.2. Changing Requirements

Gecko Lite Principle	
5	Requirements change and are often derived through a process of discovery and definition.

Requirements change. This is an immutable law of software development. Either the customer is not certain what they need or they must "discover" the requirements after

something has been built or the customer does specify what they need, but these needs change throughout the life cycle. In either case, the developers and the software engineering process must be capable of incorporating change and managing it effectively.

Requirements management is potentially the most difficult activity in any project because requirements drive every activity of the project. If requirements are not well understood or are changing too rapidly, design, code, test plans, schedules and budgets can become impossible to control. Documenting requirements and accounting for what is needed and what is to be tested and verified prior to delivery is critical to the success of a project at a technical level. If requirements are not met, then organizational goals are not met and the project has failed. Time and budget overruns can often be absorbed, but at the end of the project it must work and it must provide what the user and stakeholders need.

Gecko Lite Principle	
----------------------	--

6	High-level Master Requirements should be defined in the beginning and detailed requirements defined "just-in-time" for development.
---	---

Defining the basic requirements at the beginning makes sense, but detailed requirements should be defined iteratively as the system is evolving. The end-user and development team's understanding of requirements will evolve as the effort progresses.

Requirements are tracked from phase to phase through the use of a requirements traceability matrix which maps the requirements from one phase to the next to insure accountability of requirements across all phases.

2.3.3. How Gecko Lite Manages Requirements

Identifying the "Right" Requirements:

The *Gecko Lite Requirements Development Procedure* walks developers through the process of identifying the correct stakeholders who will be involved in specifying, reviewing and signing off on the documented requirements. It is imperative that the customer task lead identifies and approves the individuals who will be responsible for dictating requirements. These may be domain experts or key users or other customer personnel for whom the system is to satisfy key goals and objectives. Without the input



and agreement of the correct group of stakeholders it is very easy to develop a wonderful system that meets the wrong set of requirements.

Gecko Lite Principle	
7	Identifying the "correct" stakeholders who will specify and sign-off on requirements is critical to building the "right" system.

Master Software Requirements Specification:

Gecko Lite Principle	
8	Master Requirements are the "Outline" of the system. They can be rapidly documented and easily changed in successive builds.

Gecko Lite requires that the development team work with the stakeholders to develop a Master Software Requirements Specification (MSRS). This document is equivalent to an Executive Summary of the behavioral and non-behavioral requirements that will be satisfied by the system. The philosophy behind the MSRS is that after spending a sufficient amount of time iterating through the requirements with the stakeholders, the MSRS can be rapidly developed from the notes of the analysts. The MSRS is very lightweight and does not delve into a significant level of detail. The MSRS does provide an overview and narrative description of the functionality, services, features and environment and responsibilities of the system. The information provided in the MSRS represents the collective agreement and understanding of the requirements at this first level of specification. Specific implementation-level requirements will be developed later in the Build Definition Specification (BDS)

Build Definition Specification:

The Build Definition Specification takes requirements to the next level of iteration on details. The reasons for breaking the requirements and development effort into "builds" are:

- 1. Requirements are dynamic:** Don't define all of the requirements in great detail. They will change and have to be re-defined. This wastes effort. Just define small sets (called builds) of requirements at a time in enough detail so that

developers can develop a simple architecture and test plans before proceeding to coding.

2. Development can begin sooner: If you define all of the requirements in detail, it will take longer to produce this documentation, thus delaying the start of implementation. Not all requirements may be equally clearly understood at the beginning, so define in detail only those requirements that are clearest and defer detailed definition of other requirements until later Build Definition Specifications.

3. After each build, re-assess requirements: Once a build has been developed and tested, it can be demonstrated to the customer as evidence of progress. At this point the customer can re-examine the priorities of the remaining builds or even decide to return to the Master Requirements Specification to consider modifications to the overall requirements for the system. If changes are to occur, then the MSRS is updated as a new revision and then the contents of the next build are identified.

Gecko Lite Principle	
9	A "build" represents a complete life cycle in miniature. Requirements, Design, and Testing.

2.3.4. Defining Builds

Breaking up the system design and development into "builds" allows management and technical personnel to focus on relatively small, manageable portions of the system. In effect, the development of a large system actually becomes the development of many smaller systems, which are assembled into the completed product. The duration of the development and testing within a build can range anywhere from one week to 90 days. However, generally speaking, the shorter the build, the more manageable and predictable the effort. Short-term horizons are easier to manage and estimate than something longer term such as 60 to 90 days worth of effort. A lot can change in 60 to 90 days that can cause a project schedule to go astray.

The Build Concept of Gecko Lite takes the approach of dividing the total set of software requirements across a number of builds. The assigned requirements are then designed, implemented and tested as specific sub-sets of the total set of system requirements. Builds are strictly internal. That is, a build is not released for use by the customer, it is an internal building block. A release is a deliverable version of a software system that is

installed and prepared for customer use. A release is developed over the course of numerous builds. When all defined builds are completed and tested, the system is ready to enter the formal testing phase, which includes validation test, performance testing, alpha testing and beta testing. Maintenance is handled the same way, changes and new requirements are determined, bugs are prioritized and a number of builds is defined which will lead toward the development and testing of the next release which can be considered a maintenance release such release 2.1 of a system. These minor interim or maintenance releases may be made to fix problems or modify or add new minor features between major system releases.

2.3.5. Build Planning

Requirements are not always fully defined at the very beginning of a project. There will undoubtedly be changes to existing requirements or new requirements that will arise well after the initial requirements definition phase. The Build concept takes these factors into account.

Build planning is the process of assigning requirements to builds. There will always be some requirements that are better understood or specified than others. Ideally, those requirements that are the most completely defined should be assigned to earlier builds. The client should be involved in the build planning process to help establish build priorities. Remember that the level of effort that constitutes a build includes the coding and testing of the build. Upon completion of any given build, the next build is specified in a Build Definition Specification (BDS). The BDS should be developed and signed off in a fairly rapid manner. A couple of weeks is generally enough time to get the necessary details from the stakeholders on the functionality to be developed, create a simple architecture for the code to be developed and develop the validation test plans that will be used at the end of the coding to test and verify that the code created during the build is working. Once the BDS is signed off, the build officially begins. That is the point where the development and testing can range anywhere from one week to 90 days.

It is usually a good idea to assign requirements that involve system functionality that is complex or high risk to earlier builds, however the first build should be the simplest. Keeping the first build simple allows the team to have an early success, build momentum and to identify issues of procedures, policies, standards and so on that may need to be modified or clarified. Assigning complex or key issues to earlier builds make it possible to evaluate the development approach and make modifications in the early stages in the event that the design or development approach needs to change or will impact the approach to future builds.

The first build and the last two builds are the most crucial. As described, the first build should be the simplest – just to get the development process underway. The last build should be the next most simple – containing any final clean-up fixes and small, essentially miscellaneous issues that were deferred from previous builds. The next to last build should essentially represent a fully functional system with all major functional requirements satisfied.

Builds are internal deliveries to the test team or the development team – not to the client. A build is not intended for production or live operations. The disadvantage of the build approach is that customers quickly become accustomed to the relatively short completion cycles of builds and come to view builds as required products. They are not. Builds are used only as an internal management tool to help insure that the actual contract deliverables and requirements will be satisfied.

2.3.6. Segmenting Large Projects into Smaller ones

If the project is large enough and implementation of multiple builds must occur in parallel, then the system should be partitioned into multiple sub-systems or segments each with their own Master Software Requirements Specification and Build Definition Specifications. Essentially this is a case of creating one or more smaller applications out of a single large application. Partitioning the total system into several segments requires careful specification of the interfaces between segments. The individual segments should have build plans that are synchronized so that the release of build software to a test team will be approximately at the same time. By synchronizing the build releases for the same time, the test team can test the segments in parallel. This is especially useful in a large effort where many separate applications are tightly coupled.

2.3.7. Configuration Control Board (CCB): Managing Changes After Requirements have been Defined

The Configuration Control Board (CCB) reviews and approves of changes in the overall project scope and direction. The CCB is comprised of key individuals from all affected groups of customer and contractor personnel. The CCB meets periodically to review changes and overall project status. The CCB is the key mechanism by which change is coordinated, communicated and controlled.

Changes to requirements are discussed with the customer and agreed to by all affected individuals or groups in the Configuration Control Board (CCB). The impact of changes to requirements is assessed and the Project Management Plan and other affected documents and plans are updated to reflect any changes in requirements that impact these documents or work products.

The CCB meets periodically to review batches of changes to requirements – change requests. Software Change Requests (CRs) are changes to the approved requirements in either the Master Software Requirements Specification (MSRS) or one of the Build Definition Specifications (BDS). Changes can be approved, rejected or deferred. These changes should be documented in a Change Request database or change log.

4.8.3.2.3.8. Tracking of Problem Reports and Change Requests

All software enhancements, new-requirements, database changes and developer problem reports are tracked in the Change Request Tracking Database. This database maintains records on each individual Change Request (CR) and the CR's status is tracked to closure. Changes are managed according to the *Gecko Lite Change Control Procedure* and the *Gecko Lite Change Control Checklist* and the *Gecko Lite Baseline Maintenance Procedure* and the *Gecko Lite Baseline Maintenance Checklist*.

2.4. *Configuration Management*

Baselines of project products must be established for any level of quality assurance to take place. Quality cannot be assured if there is not a controlled set of software and documentation reflecting the current state of the system. The baseline software and documentation is what QA and the customer will use to determine the progress and acceptability of the products being delivered. Refer to the Configuration Management section of the Project Management Plan for additional guidance.

The baseline is the master set of project products to which changes are made and the vehicle through which changes can be tracked. Ignoring the importance of configuration management is an invitation to disaster.

~~4.8.3.~~2.3.9. **Types of Baselines**

Prototype Baseline

A separate baseline is kept in the configuration management library for any prototypes created.

Development Baseline (during any given build development phase)

All software created during the development phase of the project must be baselined. The baseline can be updated daily by development team members to insert their evolving code and on-going changes during development. Changes in the Development baseline for any build can be made at any time even during the validation testing of the build.

Validation Test Baseline

Once development of all builds is completed, the validation of the entire set of requirements in the Master Software Requirements Specification (MSRS) will be performed. The baseline will be updated as often as needed by the configuration manager with any changes that have been reviewed and approved by the project manager or team lead. At this point the baseline is more tightly controlled with approved changes ONLY being added to the baseline.

System (or Performance) Test Baseline

Once validation testing is completed, system testing will be performed. The system testing will include stress testing, communications testing, security testing, fail-over and recovery testing as appropriate to the project. The baseline will be updated as often as needed by the configuration manager with any changes that have been reviewed and approved by the project manager or team lead.

Alpha Test Baseline

Once system testing is completed, the System Test version of the software will be promoted to the Alpha Test Baseline for commencement of Alpha Testing. Once the Alpha Test has concluded and all software problems have been fixed or prioritized for resolution in parallel during the Beta Test period, the Beta Testing will begin. Changes are not made to the Alpha baseline during Alpha Testing, but rather to an Alpha-2 baseline. Once the Alpha testing is completed and changes to the Alpha-2 baseline are completed, the Alpha-2 baseline can be promoted to the Beta Test baseline. Creating an Alpha-2 baseline prevents changes from being made to the Alpha baseline while it is being tested.

Beta Test Baseline

Once Alpha Testing is completed, the Alpha-2 version of the software will be promoted to the Beta Test Baseline for the commencement of Beta Testing. Once the Beta Test has concluded and all software problems have been fixed or granted waivers in a Beta-2 baseline, and acceptance testing of the system has been successfully completed and customer signoff of the Beta-2-version of the product has been acquired, the Beta-2 Test Baseline will be promoted to become the Operational Baseline.

Operational Baseline

This baseline is the master copy of the operational software. It is from this baseline that any computer software delivery media will be generated or copied from to produce the operational configuration of the software for installation and checkout in the operational environment.



2.5. *Project Management and Tracking*

The project manager or team lead notifies the customer of any realized or anticipated deviations from the project milestones so that corrective action may be taken. Weekly progress meetings should be held with the development team to assess the progress and status of the project.

Periodic (monthly) progress meetings should be held with the customer to assess the project and status of the project. Weekly and monthly progress/status reports should be produced by the project manager or team lead. These reports should be provided to the customer and be used as the basis for discussion at project status review meetings.

2.6. *Quality Assurance*

Gecko Lite products are designed to provide a seamless transition from one phase to the next. The system development life cycle moves easily from one phase to the next with products that facilitate rather than hinder this progression.

Gecko Lite requires that each product undergo various formal and informal inspections and certifications. Potential problems can be detected at many points to minimize expensive backtracking or rework. The final system meets client expectations by insuring that each step is producing results consistent with client expectations throughout the project. Project teams can modify their efforts before significant amounts of resources are expended unnecessarily.

Quality assurance is concerned with the consistency, readability, usability, maintainability, reliability and other attributes of the completed system and the work products produced throughout the project life cycle. Quality is assured through multiple review points with the customer to identify errors, inconsistencies, misunderstandings and omissions in each interim work product.

Peer Reviews

Peer reviews are internal reviews conducted on products prior to external formal customer reviews. Peer reviews are conducted on all deliverable documents and plans as well as selected units of source code and system user interface screens. Peer reviews are conducted using a review checklist specific to the type of product being reviewed.

Once a product has been reviewed, corrections are made and a remedial or follow-up review is conducted. After the reviewer evaluates the product as "Accepted" it can then be reviewed by the customer and prepared for submission to configuration management as the latest baseline of the product.

4.8.4. Analysis of Software Error Metrics

Software errors should be logged and tracked to closure. Data on software errors should be collected and categorized by the nature of the error: computational, logical/control, clerical, requirements, and data. Collection and analysis of these simple metrics allows software teams to identify areas of the software development effort that require further

attention and improvement. Over time, teams can track the effect of improvements to the process on the number and types of errors that are introduced into the effort.

Categories of Software Errors	
Clerical	Typographical errors or incorrect variable names.
Computational/Algorithmic	An error in the implementation of a formula or algorithm. The error may be that the wrong algorithm is being used.
Data	An error with the acquisition/production/transmission of data. This includes I/O errors.
Requirements	An feature or function is correctly implemented and traceable back to the requirements, but there is an error in the specification of the requirement.
Logical/Control	Incorrect logical conditions have been specified or branching and control structures are improperly implemented. The error may be that the right algorithm is being used, but was improperly implemented.

2.7. Software Error/Defect Classification

Software errors are generally classified by severity according to the impact they have on the usability of the application, which is generally measured in terms of the scope, or scale of impact that the error has on the application.

Severity of Software Errors	
5 - System Level CATASTROPHIC	The entire application or system is unusable, catastrophic failure, usually requires restart or re-initialization or reboot.
4 - Sub-system/module Level CRITICAL	A sub-system or module becomes unusable
3 - Feature/Function Level SERIOUS	Feature or function does not work at all or aborts
2 - Sub-Feature Level MINOR	A particular aspect of a function/feature does not function properly, but the overall feature/function is still usable.
1 - Formatting/cosmetic SIMPLE	The layout or format of data, reports, messages, screens and other cosmetic issues require changing; no impact on usability. If there is an impact on usability, then the error may need to be reclassified at a higher level.

2.8. Acceptance Criteria

Acceptance criteria are at the discretion of the customer. However, it is important to remember that no software system is without bugs. If we were to wait until an application was completely bug-free, no software would ever be put into use for its intended purpose.

One common approach to defining acceptance criteria is to say that all requirements must be satisfied in order for the system to go into operational use, but at what level of detail? Another is to decide some subset of requirements that must be tested and demonstrated to be working completely before the system can go into operational use. Any requirements not completely satisfied in the initial release can be completed or satisfied in a subsequent

minor or maintenance release within 1-3 months (or some defined period) of the major release.

Another approach to defining acceptance criteria is to define criteria by phase. Since alpha test phase is generally the first time that users are allowed to "touch" the software and delivery is the time when the completed system is installed in the production environment, we can establish successively more stringent criteria for each phase of testing leading up to release. Depending upon the overall timeframe of the project and the number of errors encountered during each phase, the customer may want to determine which specific errors of which type should be resolved before moving to the next phase or toward release. Below is an example of how criteria might be defined:

Acceptance Criteria (example)	
Validation Testing	No level 5 errors by the start of System Testing.
System Testing	No level 4 errors by the start of Alpha Testing.
Alpha Testing	No level 3 errors by the start of Beta Testing
Release	Only level 2 and level 1 errors may remain.

2.9. *User Guide/Programmer Reference Guide Development*

The Master Requirements Specification (MSRS) defines the Goal Model for the requirements. The Goal model represents the goals that the system must accomplish and the roles it must play to satisfy the user requirements. The goals represent the things that the user must be able to do when using the system and the things that the system must provide in terms of features, functions and services. The user guide should reflect these goals and have sections to explain from a user perspective how each goal and sub-goal is accomplished via the GUI. If the application is an API and not a GUI, then it should more appropriately be labeled as a Programmer API Reference Guide. In either case the same principle applies. Demonstrate how the programmer, via the API calls that are provided by the application, can accomplish the various goals and provide explanations and examples.

If the application is a GUI-based system you should start by identifying the various types or categories of screens and provide a legend which explains the general components of

the interface and what they are used for and give these components names such as the left side navigation bar, or the top command bar or the option menu, or the main work area of the screen etc.

The user guide mirrors the master requirements. If the master requirements discuss how the system must be able to run in several different operating systems or environments, then the user guide should cover this and define and discuss any differences between what the GUI or the API will look like. If there are different API calls depending upon the particular operating system then these differences must be documented with appropriate examples. By combining the user guide which is based on the master requirements and the validation test plans which are also based on the requirements you will end up with everything needed for testing and using the system. Both the user guide and test plans should point back to the requirements which specify what the system is to do.

The user guide can actually be "tested" by executing all of the commands/actions/options as described in the guide to insure that the guide is correctly written and that the instructions are in the proper sequence and at a sufficient level of detail. It should be possible to run the user guide commands/calls or actions based only on the information in the guide. If someone cannot accomplish the goals in the guide, then it must be revised until those goals can be achieved.

The user guide should begin with some explanation in the first chapter about background, purpose, system objectives/goals (from a high level perspective), then zoom in to the second chapter to discuss environments and system configurations as applicable and then in chapter three begin covering the goals/commands/actions/options etc.

All of the system goals specified in the master requirements do not have to go in chapter 3, of the user guide. It would be more logical to break down the goal model in the MSRS and take each group of high-level goals and start a new chapter with each group of high-level goals and within the chapter cover the sub-goals.

3. Life Cycle Overview

The typical "Waterfall" or "Big Bang" approach to software development prescribes that no development work begins until the entire set of requirements has been fully specified and the design of the entire system is complete. The Gecko Lite approach of "build a little", "test a little" allows for frequent customer reviews and involvement and earlier views of the system as it begins to evolve.

Breaking any system development effort into multiple "builds" makes the total development effort more manageable by allowing the customer and the developers to focus on one major aspect of the system a time. As each build is developed, it is coupled to the previous builds and tested cumulatively. The process is additive and the testing of the current build takes place while other members of the team begin work on the next build.

Gecko Lite Principle	
10	Testing each build as it is added to the evolving system provides more opportunities for developers to find and fix problems with the design and fit of the components, the functionality and to gain further clarification of any requirements that may not be fully understood or finalized.

3.1. Project Initiation Phase

The project initiation phase helps get the project started effectively. Many projects hold a project kickoff meeting with no clear objectives and without accomplishing many crucial activities at the beginning of the project, which can help insure the ultimate success of the project. For example, key stakeholders must be identified, project resources for hardware, software and personnel must be identified and the schedule of availability of these resources must be determined. If such crucial issues as stakeholders are not addressed and resolved at the beginning of the project, then requirements definition can become a nearly impossible task with no clearly defined customer representatives involved in the requirements gathering process or in the signoff and acknowledgement of the documented requirements.

3.2. Project Planning Phase

Project planning occurs throughout the life of a project. Initial planning occurs at the beginning of the project. The Project Management Plan outlines the general nature of the system, the project, personnel, schedules, development approach, applicable standards, procedures and quality assurance and configuration management plans.

3.3. Concept Definition Phase (optional)

Prototyping is highly recommended. Prototyping can help verify whether a given approach is feasible or will work as desired. Evaluating prototype results gives the design team and the client a clear idea of what changes may need to be made to requirements, resources and schedules as well as what design approach to use before beginning detailed design.

3.4. System Requirements Phase (optional)

Very large projects benefit from defining system-level requirements. These requirements represent the overall hardware and software requirements of a system that may actually be composed of a number of independent, but tightly coupled software applications or components developed by different teams. System requirements give the total picture of the functionality of the larger 'system' and allow the project team and customers to allot specific groups of requirements to specific segments or applications of the total project. Agreement must be reached at this level about the scope and role of each application or software component in the larger system before delving into the details of any one specific application.

3.5. Software Requirements Analysis/Definition Phase

Gecko Lite Principle
11 Successful projects focus on basic core requirements.

Successful projects focus on basic core requirements. Trying to develop everything that everyone wants will likely result in delivering something that no one wants. Individual requirements are uniquely identified for tracking and testing purposes. Requirements are

defined at a very high-level in outline form with short, concise descriptions of the functionality to be developed.

3.6. *Build Planning Phase*

The requirements identified in the MSRS are partitioned into multiple builds. The Build Plan identifies which requirements will be developed in which build and provides a simple build schedule to indicate the start and end of each build. The Build Plan can and should be updated upon the completion of each build to reflect any changes in build priorities or schedule.

3.7. *Build Development Phase*

Requirements for a specific build are defined and reviewed with the customer. The architecture for the build is laid-out and the validation test plans are defined in the Build Definition Specification.

The Build Definition Specification (BDS) contains: requirements, design, test plans, and system environment changes pertaining to a given build. The BDS is a lightweight, but crucial artifact of the plans for the next round of development. In effect, a build represents a total life cycle in miniature.

During the course of the current build, the functionality of the system will be incrementally enhanced in a controlled and planned manner. If requirements change, they will be addressed in subsequent builds. The build and test process is repeated incrementally throughout the life of the project.

Individual software units are developed and tested by the developers and reviewed by peers to insure that each unit satisfies the requirements allotted to it. All units must comply with established project coding style and conventions.

Incremental Build Validation Testing is important to verify that the system fulfills its allotted requirements and to identify design and performance issues, which could affect future builds.

Gecko Lite Principle	
12	Incremental testing as each build is completed results in a more stable, better-tested system by the time the system is tested as a whole.

3.8. Build Integration Phase

Build integration is a phase taking place after the previous round of build development. The Build Integration Phase provides developers a chance to couple the previous builds to the current build and test the combined functionality.

Tests performed during build integration are focused on exercising the various connections and communications between components developed in separate builds. This incremental build-a-little, test-a-little helps insure system stability and insure that testing time does not get eliminated or negatively impacted at the end of the project when the schedule may be starting to slip.

3.9. Test Phase

If every line of software code is considered a moving part in the software machine, then we must test thoroughly to ensure that all parts operate, as they should. Software testing begins with the unit testing or White Box testing of key units of source code within a given build.

As each build is produced, an integration and test phase may follow a build that is to be coupled to a previous build. This integration and test phase allows the team to mate the previous build with the current build and make appropriate adjustments so that they communicate and function cohesively.

Incremental Testing Phases	
Unit Testing (White-Box) - <i>optional</i>	Provides an internal view of the system. Detailed testing of highly critical units of code only. This type of testing may not be required or necessary for many typical MIS systems. White-box testing involves testing all paths of logic that flow through a unit, all combinations of conditions that may be encountered by logical comparisons within the code and all looping conditions to verify correct loop performance and exception handling.
Build Validation Testing (Black-Box)	Provides an external view of the system. Tests the requirements to insure that all requirements and constraints are satisfied.
Build Integration Testing	Where two successive builds must be coupled, the linkages between these components are identified

	<p>and testing in integration tests. Two or more successive builds may simply be additions to the same portion of the system where no coupling of separate components is taking place. In this case, Build Integration Testing may not be necessary when one build simply adds functionality to a previous build.</p>
--	---

The various types of testing of the entire system are identified in the table below. These phases are conducted after the entire system has been built.

Testing Phases for the Completed System	
Validation Testing	Software requirements are re-tested in Validation tests, which insure that the cumulative functionality of previous builds has been verified once development of the release is complete.
System Testing	Exercises performance, recovery, security and stress tests.
Alpha Testing	Very small number of users who are concerned with verifying the complete functionality of the system from a user perspective rather than from the system and performance point of view.
Beta Testing	As close to operational as possible, simulating usage or mission conditions realistically.
Acceptance Testing	Verifies the critical criteria needed for the customer to deem the system acceptable.

3.10. Delivery and Installation Phase

The delivery and installation phase involves execution of the *System Delivery and Installation Checklist* and procedure. This checklist identifies the required personnel, equipment, preparations and software necessary for a smooth transition from the software development phase to the system operations and maintenance phases of the project.

Many software projects may be successfully planned, managed and executed through to acceptance testing. Significant delays, incompatibilities and obstacles to initiating live operations with a completed software application can be avoided through careful planning and execution of a delivery, installation and checkout phase.

This phase is concerned with the orderly planning and execution of the preparation, delivery, installation, configuration and checkout of software systems in their target production environment. The final stage of this phase is the Operational Readiness Review, which is the point in the development lifecycle when the new system or the current release of the software is assessed for its suitability for live operations.

3.11. Project Closure Phase

Project Closure or task completion is managed via the Gecko Lite Project Closure Checklist which helps insure the all project deliverables have been met and delivered, all project documentation is up to date, project property is accounted for and that all project records are complete and correct.

4. Common Problems and Solutions

There are many problems that typically occur on a software development project. The majority of the problems can be avoided by simply following a good software development process. This frees developers to focus on the unforeseen project-specific problems and issues.

Problem #1:
Too many requirements. (or too few requirements). Either of these extremes is a serious problem.
Solution:
Projects can fail due to over-specification of requirements or at the other extreme – complete lack of specification of requirements. Focus on core requirements that can be specified rapidly and built into a production-ready release. Getting a first release in the users hands that does much of what they need can contribute significantly toward the end-users meeting their organization objectives. It is also desirable not to try and specify every possible requirements or bell and whistle because this will overcomplicate the development effort and increase the likelihood that the project will ultimately not satisfy any requirements, thus failing. Once the users have been able to use and evaluate the functionality of the first release, the next set of Master Requirements can be defined for a second release that will build on the core functionality.

Problem #2:
Lack of stakeholder involvement.
Solution:
The end-users were not considered important in the development effort. Their needs should have been the primary focus of the effort. The system will affect the way they do their work. And who better understands the process than those who perform it every day? The customer task lead must be called upon to take action to obtain commitments from any organization that contains identified stakeholders in the system. These stakeholders are mandatory to specify, review and approve the requirements.

Problem #3:
Constantly changing requirements.



Solution:

This is not really a problem if your team and your software development process are focused on incremental development and specification of requirements.

Requirements change frequently during the project and are often too lofty to begin with. Define the system as a series of small incremental builds leading to a usable initial release. With each subsequent build, add functionality and then review the results with the customer to define the work to be performed in the next build. Continue the evolution until the system accumulates enough functionality to be put into use as a production-ready release.

Problem #4:

What requirements are REALLY needed? What is the core functionality that must be developed to get a practical, useful system into the hands of the users?

Solution:

Build from the basics first. Bells and whistles can be added to later releases as refinements and enhancements.

Just define the core functionality needed to field a useable first release. Additional functionality can be added in later releases as the users get a better feel for how the system works and how it can be further evolved to meet their needs.

Gecko Lite seeks the middle ground on requirements. Define high-level software capabilities up front in the Master Software Requirements Specification – and revisit this document every time there is a significant course change in the customer's needs.

Problem #5:

No disciplined process was followed.

Solution:

Once the initial coding begins, projects often end up in a development free-for-all.

Each month a small increment or 'build' of functionality should be built, tested

and demonstrated to the customer before proceeding further. Incremental testing helps maintain stability of the application. Incremental integration of previous components helps prevent the 'house of cards syndrome'. A build represents the detailed requirements of the Build Definition Specification, the Design and Test Plans for that build. A build should not exceed three months from start to finish. Builds can even be as short as one week to develop a small subset of functionality, review, test and demonstrate before continuing.

Problem #6:
Too much documentation too soon.
Solution:
It does not require a mountain of documentation to develop a good, stable system that meets customer needs. It DOES require constant review of small sets of requirements, incremental development followed by incremental testing.
Defining the basic requirements at the beginning makes sense, but detailed requirements should be defined iteratively as the system is evolving. The end-user and development team's understanding of requirements will evolve as the effort progresses. These evolving requirements, design and tests go into the Build Definition Specification. One of these small documents contains requirements, design, test plans and system updates that apply to the current build.

Problem #7:
Lack of adaptability to changing requirements.
Solution:
Constant review and definition of requirements allows the customer to make reasonable course corrections and adjustments to the desired functionality as the project progresses. These changes are reviewed monthly to assess the impact on the work-to-date so that subsequent builds can accommodate the new functionality while previous builds are adapted accordingly.

5. Implementation Phases

This section describes the various phases of implementing Gecko Lite on a software development project.

5.1. *Implement Software Configuration Management (CM)*

CM1.0 Establish Project Notebook

This should be established according to the Gecko Lite Project Initiation Procedure.

CM1.1 Document the Development and Production Environment HW/SW Configurations

Document the HW and SW configuration of the production environment and development or test environment for any application software being developed. Place the configuration information in the project notebook.

CM1.2 Create Source Code and Document Repository

CM1.2.1 Identify source code repository server

CM1.2.2 Identify version control software package

CM1.2.3 Acquire and Install version control software

CM1.2.4 Identify Project-Level CM Manager

The project lead or a designee for small project teams generally plays the role of CM. Larger project teams have an individual, sometimes the DBA who performs CM and baseline maintenance activities. Some projects may have a dedicated individual for the CM role.

CM1.2.5 Create project directory structures for source code

CM1.2.6 Identify project document repository server (if different than source code server)

CM1.2.7 Identify version control software package (if different than source code version control package)

CM1.2.8 Create project document directory structures

CM1.2.9 Identify and baseline all project documents

CM1.2.10 Identify and baseline all project source code

CM1.3 Implement Backup and Recovery Procedure



CM1.4 Implement Change Control Tracking Database

CM1.4.1 Customize any project tables that require adaptation for projects

CM1.4.2 Configure user accounts Project lead and CM have highest level of access.

CM1.5 Implement SW/HW Inventory Management Database

CM1.5.1 Identify secured location (preferably lockable) for storage of COTS software media.

CM1.5.2 Assign access (keys, or other means) to project lead and CM
There should be two individuals on a project with access to the software storage location.

CM1.6 Inventory Licensed Software

Inventory all licensed software in the Inventory Management Database. This database will track the individual CD-ROMS and their serial numbers for every valid licensed copy of software that is owned by the project. It should be indicated in the database the owner of the software e.g. AMTI or the customer name, NASA/Ames, NAS down to the division level or lower if necessary.

CM1.6.1 Update project notebook
Update the project notebook with a copy of the initial and subsequent monthly software inventory reports.

CM1.6.2 Distribute Software Inventory Report
Distribute copies of the initial software inventory report to the customer and AMTI management. This report should be generated monthly and the project notebook updated.

CM1.7 Inventory Hardware

Identify all hardware, printers, peripherals and other physical items such as tables, chairs, desks, etc. that are under control of the contract which must be returned to the government or are the property of AMTI or a sub-contractor or partner.

CM1.7.1 Update project notebook

Update the project notebook with a copy of the initial and subsequent monthly software inventory reports.

CM1.7.2 Distribute hardware Inventory Report

Distribute copies of the initial software inventory report to the customer and AMTI management. This report should be generated monthly and the project notebook updated.

CM 1.8 Evaluate the Status of the CM Implementation and Processes

Use the CM Process Evaluation Checklist to review and assess the strength or weakness of the implemented processes.

5.2. *Implement Software Requirements Management (RM)*

RM1.0 Identify and Document Baseline Requirements

Document project requirements according to Gecko Lite Software Requirements Specification template and using the Gecko Lite Requirements Definition Procedure.

RM2.0 Conduct Internal Peer Review of Requirements

RM3.0 Conduct Software Requirements Review (SRR)

This is a formal review with the customer. The SRR should be conducted according to the SRR Checklist.

RM4.0 Develop Software Build Plan for Development

The build plan addresses which requirements will be developed for each software build of each release of the system.

RM5.0 Develop SW Acceptance Criteria

The customer generally identifies and prioritizes the functionality of the system that must be functioning properly in order to consider the system acceptable or suitable for delivery and installation and the start of live operations. It is acceptable and commonplace for the key requirements to be demonstrated as satisfied and for other bugs or lower level problems to potentially still remain unresolved at the time of the operational readiness review and the start of live operations.



So long as the critical acceptance criteria are satisfied prior to the start of operations, any other unresolved issues or problems can be managed in subsequent maintenance releases or the next release. The customer can modify acceptance criteria with the issuance of a software waiver identifying the particular requirement that is being waived temporarily. Waivers should be used judiciously.

RM6.0 Conduct Customer SW Acceptance Criteria Review

This formal review is intended to finalize the actual criteria or requirements that constitute the acceptance criteria by which the system will be judged prior to the start of operations.

~~5.4~~-5.3. *Implement Software Engineering Training (TR)*

TR1.0 Conduct SW Engineering Overview and Fundamentals Course

TR2.0 Conduct Project Planning, Management and Oversight Course

TR3.0 Conduct Configuration Management, Baseline Maintenance and Release Management Training Course

TR4.0 Conduct Requirements: Analysis, Definition Management Training Course

TR5.0 Conduct Software Testing Training Course

TR6.0 Conduct Software Delivery and Installation Training Course

TR7.0 Conduct Software and Development Design Course

5.4. Implement Software Design Practices (DE)

DE1.0 Produce Software Design

Produce the software design according the Gecko Lite Software Design Specification template and using the projects Software Requirements Specification as the basis for the design.

DE2.0 Trace Software Requirements

Trace the software requirements from the SRD to the Software Design.

5.5. *Implement Software Development Practices (DV)*

DV1.0 Review the Gecko Lite Coding Style

DV2.0 Identify Key Source Code Units

Identify the key source code units to be updated with source code documentation and converted to follow the adopted coding style for the project.

DV3.0 Modify Key Source Code Units

Update the key source code units to comply with the adopted coding style for the project.

DV4.0 Perform Peer Code Reviews

Review and modify key source code units.

DV5.0 Perform Unit Tests

Re-test the units after modification to insure functionality.

5.6. *Implement Software Testing Practices (ST)*

Depending upon whether the software is to be tested on the same machine on which it was developed or if the software is to be installed in a separate test environment it may be necessary to properly configure the separate test environment prior to the start of any testing beyond the Unit Test Phase.

ST1.0 Trace Requirements

Trace the software requirements from the Software Requirements Document (SRD) to the code. Create a requirements traceability matrix (RTM) to map between the requirements and the application source code being developed. This matrix helps identify what source code modules are responsible for satisfying which requirements.

ST1.0 Develop Unit Tests

Identify requirements allotted to each unit through the RTM and develop tests to verify the functionality of the requirements allotted to each unit.

** Normally Software Unit Tests are developed and performed as part of the development phase, but this implementation of the activity is for existing projects that are already in development.*

ST2.0 Perform Unit Testing

Execute the unit tests and document the results. Record any problems, issues or bugs in the Task Management Database.

ST3.0 Develop Build Test Plan

ST3.1 Identify couplings between units (internal interfaces) that will be tested.

ST3.2 Identify external interfaces to units that will be tested.

ST3.3 Identify couplings between the previous builds (internal interfaces) and the current build that will be tested.

ST4.0 Conduct Peer Review of Build Test Plan

Use the Integration/Build Test Plan Checklist to review the plan and identify and make any necessary changes.

ST5.0 Perform Build Testing

ST5.1 Execute the Build/Integration test plan.

ST5.2 Document any problems, bugs or issues in the Task Management Database.

ST6.0 Develop Software System Test Plan

ST6.1 Identify system-level test plans of major processes/functions and external
ST6.2 interfaces to be tested.

ST6.3 Identify major use-case scenarios to be tested.

ST7.0 Perform Software System/Release Testing (all builds integrated)

This phase concerns the testing of all subsequent builds which constitute the finished product or the current release. Once testing is completed, the Task Management Database should be updated with problems, bugs or issues requiring resolution that were discovered during the testing.

ST8.0 Perform Alpha Testing

ST8.1 Execute the System Tests with a minimal number of users.

ST8.2 Document the problems/issues found during testing in the Task Management Database

ST9.0 Perform Beta Testing

ST9.1 Execute the System Tests with a larger number of users.

ST9.2 Document the problems/issues found during testing in the Task Management Database

~~5.5-~~5.7. *Implement Software Delivery & Installation Phase Processes (DL)*

Gecko Lite specifies a *System Delivery and Installation Procedure*. This procedure describes the steps involved in delivering and installing a new hardware and software system to a customer site. The installation procedure provides guidance about the activities and products concerned with delivering and installing a software system product.

The *System Delivery & Installation Checklist* provides a simple and convenient means of accounting for these activities. Continuing to follow a simple, but disciplined process for installation of a finished product helps ensure that the product potential of the work performed in the previous life-cycle phases is realized in the finished and delivered system.

When software is ready for release it will include a release or delivery package which contains at a minimum:

- Installation Instructions
- Release Notes
- List of all required files
- List of all required database files/tables/components
- List of all required or applicable documents

DL1.0 Develop a System Delivery and Installation Plan

DL.2.0 Implement the Delivery and Installation Plan

Use the System Delivery and Installation Checklist to aid execution of the plan.

5.8. Implement Software Project Management, Tracking and Oversight and Quality Assurance Practices (PM)

PM1.0 Conduct Weekly Development Team Meetings

Track action items in the Task Management Database that were taken in subsequent meetings.

PM2.0 Produce Weekly Status Report

PM3.0 Produce Monthly Status Report

PM4.0 Conduct Monthly Customer Status Review

PM5.0 Conduct Weekly Telcons for Process Implementation

PM6.0 Conduct Operational Readiness Review (ORR)

PM7.0 Implement SW Help Desk/Call Tracking Database

PM8.0 Conduct SW Help Desk and Problem Tracking Status Reviews



Appendix A: Software Engineering Implementation Checklists

This section contains checklists for the various phases and activities within these phases that should be conducted in order to fully satisfy and implement Gecko Lite on a software development project.

Software Configuration Management (CM)

- ❑ **CM1.0 Establish Project Notebook**
- ❑ **CM1.1 Document the Development and Production Environment HW/SW Configurations**
- ❑ **CM1.2 Create Source Code and Document Repository**
 - ❑ CM1.2.1 Identify source code repository server
 - ❑ CM1.2.2 Identify version control software package
 - ❑ CM1.2.3 Acquire and Install version control software
 - ❑ CM1.2.4 Identify Project-Level CM Manager
 - ❑ CM1.2.5 Create project directory structures for source code
 - ❑ CM1.2.6 Identify project document repository server (if different than source code server)
 - ❑ CM1.2.7 Identify version control software package (if different than source code version control package)
 - ❑ CM1.2.8 Create project document directory structures
 - ❑ CM1.2.9 Identify and baseline all project documents
 - ❑ CM1.2.10 Identify and baseline all project source code
- ❑ **CM1.3 Implement Backup and Recovery Procedure**
- ❑ **CM1.4 Implement Change Control Tracking Database**
 - ❑ CM1.4.1 Customize any project tables that require adaptation for projects
 - ❑ CM1.4.2 Configure user accounts
- ❑ **CM1.5 Implement SW/HW Inventory Management Database**
 - ❑ CM1.5.1 Identify secured location (preferably lockable) for storage of COTS software media.
 - ❑ CM1.5.2 Assign access (keys, or other means) to project lead and CM
- ❑ **CM1.6 Inventory Licensed Software**
 - ❑ CM1.6.1 Update project notebook
 - ❑ CM1.6.2 Distribute Software Inventory Report
- ❑ **CM1.7 Inventory Hardware**
 - ❑ CM1.7.1 Update project notebook



- CM1.7.2 Distribute hardware Inventory Report
- **CM 1.8 Evaluate the Status of the CM Implementation and Processes**

Software Requirements Management (RM)

- ❑ **RM1.0 Identify and Document Baseline Requirements**
- ❑ **RM2.0 Conduct Internal Peer Review of Requirements**
- ❑ **RM3.0 Conduct Software Requirements Review (SRR)**
- ❑ **RM4.0 Develop Software Build Plan for Development**
- ❑ **RM5.0 Develop SW Acceptance Criteria**
- ❑ **RM6.0 Conduct Customer SW Acceptance Criteria Review**

Software Testing Practices (ST)

- ❑ **ST1.0 Trace Requirements**
- ❑ **ST2.0 Develop Unit Tests**
- ❑ **ST3.0 Perform Unit Testing**
- ❑ **ST4.0 Develop Build Test Plan**
 - ❑ ST3.1 Identify couplings between units (internal interfaces) that will be tested.
 - ❑ ST3.2 Identify external interfaces to units that will be tested.
 - ❑ ST3.3 Identify couplings between the previous builds (internal interfaces) and the current build that will be tested.
- ❑ **ST5.0 Conduct Peer Review of Build Test Plan**
- ❑ **ST6.0 Perform Build Testing**
 - ❑ ST6.1 Execute the Build/Integration test plan.
 - ❑ ST6.2 Document any problems, bugs or issues in the Task Management Database.
- ❑ **ST7.0 Develop System Validation Test Plan**
 - ❑ ST7.1 Combine the validation test plans from each build into a single comprehensive Validation Test Plan.
 - ❑ ST7.2 Peer Review the Validation Test Plan
- ❑ **ST8.0 Perform Software Validation Testing (all builds integrated)**
- ❑ **ST9.0 Develop System Performance Test Plan**
 - ❑ ST9.1 Identify system-level test plans of major processes/functions and external interfaces to be tested.
 - ❑ ST9.2 Identify major performance characteristics/scenarios to be tested.
- ❑ **ST10.0 Perform System Performance Testing (all builds integrated)**
- ❑ **ST11.0 Perform Alpha Testing**
 - ❑ ST11.1 Execute the System Tests with a minimal number of users.



- ST11.2 Document the problems/issues found during testing in the Task Management Database
- **ST12.0 Perform Beta Testing**
- ST12.1 Execute the System Tests with a larger number of users.
- ST12.2 Document the problems/issues found during testing in the Task Management Database

Software Engineering Training (TR)

- ❑ **TR1.0 Conduct SW Engineering Overview and Fundamentals Course**
- ❑ **TR2.0 Conduct Project Planning, Management and Oversight Course**
- ❑ **TR3.0 Conduct Configuration Management, Baseline Maintenance and Release Management Training Course**
- ❑ **TR4.0 Conduct Requirements: Analysis, Definition Management Training Course**
- ❑ **TR5.0 Conduct Software Testing Training Course**
- ❑ **TR6.0 Conduct Software Delivery and Installation Training Course**
- ❑ **TR7.0 Conduct Software and Development Design Course**

Software Delivery & Installation Phase Processes (DL)

- **DL1.0 Develop a System Delivery and Installation Plan**
- **DL.2.0 Implement the Delivery and Installation Plan**
- **DL3.0 Review the Software Delivery and Installation Plan – Peer Review**
- **DL4.0 Review the Software Delivery and Installation Plan – Formal Customer Review**

Software Design Practices (DE)

- **DE1.0 Produce Software Design**
- **DE2.0 Trace Software Requirements**
- **DE3.0 Review the Software Design – Peer Review**
- **DE4.0 Review the Software Design – Formal Customer Review**

Software Development Practices (DV)

- ❑ **DV1.0 Review the Gecko Lite Coding Style**
- ❑ **DV2.0 Identify Key Source Code Units**
- ❑ **DV3.0 Modify Key Source Code Units**
- ❑ **DV4.0 Perform Peer Code Reviews**
- ❑ **DV5.0 Perform Unit Tests**

Software Project Management, Tracking and Oversight and Quality Assurance Practices (PM)

- ❑ **PM1.0 Conduct Weekly Development Team Meetings**
- ❑ **PM2.0 Produce Weekly Status Report**
- ❑ **PM3.0 Produce Monthly Status Report**
- ❑ **PM4.0 Conduct Monthly Customer Status Review**
- ❑ **PM5.0 Conduct Weekly Telcons for Process Implementation**
- ❑ **PM6.0 Conduct Operational Readiness Review (ORR)**
- ❑ **PM7.0 Implement SW Help Desk/Call Tracking Database**
- ❑ **PM8.0 Conduct SW Help Desk and Problem Tracking Status Review**

Appendix B: Life Cycle Phase Entry/Exit Criteria and Products

Life Cycle Phase	Project Initiation Phase
Entry Criteria	Customer-signed Statement of Work (SOW) or Customer-signed Contract or Customer-signed Task Order or Task Assignment
Exit Criteria	Kickoff Meeting has occurred Project Acceptance Criteria have been identified and documented
Products	Project Acceptance Criteria <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	Project Planning Phase
Entry Criteria	Project Acceptance Criteria SOW/Contract/Task Order – (Customer-signed) Kickoff Meeting Minutes
Exit Criteria	Completed Project Schedule (e.g. Microsoft Project) – for PMP List of top 10 Project Risks – for PMP List of HW/SW and other Resources – for PMP Baselined (customer-signed) Project Management Plan (PMP) (<i>which includes the schedule</i>)
Products	Project Management Plan (PMP) <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	Concept Definition Phase (optional)
Entry Criteria	This phase can occur in parallel with/Project Proj. Planning Phase Project Acceptance Criteria Kickoff Meeting Minutes
Exit Criteria	Any applicable prototypes or GUI mockups for requirements analysis. Baselined (customer-signed) System Operations and Concept Specification (SCS)
Products	System Operations and Concept Specification <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	System Requirements Definition Phase (optional)
Entry Criteria	Project Acceptance Criteria SOW/Contract/Task Order – (customer-signed) Kickoff Meeting Minutes
Exit Criteria	Baselined (customer-signed) System Requirements Specification (SRS) (<i>for large systems composed of multiple applications or separate software components</i>)
Products	System Requirements Specification <i>* A customer or peer-signed copy of all of the above</i>

	<i>products should be stored in the project file for auditing purposes.</i>
--	---

Life Cycle Phase	Software Requirements Analysis/Definition Phase
Entry Criteria	Baselined (customer-signed) System Requirements Spec – (if produced) Baselined (customer signed) System Operations and Concept Specification (if produced) Project Acceptance Criteria Contract/SOW/Task Order – (customer signed)
Exit Criteria	Baselined (customer-signed) Master Software Requirements Specification (MSRS) (optional) Baselined (customer-signed) Use-Case Specification (UCS)
Products	Baselined (customer-signed) Acceptance Test Plan (ATP) Peer Review Checklists for MSRS Follow-up Peer Review Checklists for MSRS Formal Requirements Review Checklists for MSRS Follow-up Requirements Review Checklists for MSRS Master Software Requirements Specification (MSRS) Acceptance Test Plan - ATP <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	Software Design Phase – design is largely covered in the Software Build Development Phase
Entry Criteria	Baselined (customer-signed) Master Software Requirements Specification Baselined (customer-signed) System Requirements Specification – (if produced)
Exit Criteria	Baselined (optional) System Architecture or Design Specification - SDS Baselined (customer-signed) Software Build Plan - SBP Baselined (customer-signed) Build Definition Specification BDS – one for each build. Baselined Build Integration Test Plan - BITP (one for each pair of builds to be integrated – as needed)
Products	System Architecture or Design Specification – SDS Software Build Plan – SBP

	<p>Peer Review Checklists for BDS Follow-up Peer Review Checklists for BDS Formal Review Checklists for BDS Follow-up Review Checklists for BDS Build Definition Specification – BDS Build Integration Test Plan - BITP</p> <p><i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i></p>
Life Cycle Phase	<p>Software (Build) Development Phase – <i>(additional Build Definition Specifications (BDS) would be created for each subsequent build and, as needed, the Master Software Requirements Spec (MSRS) would be updated prior to creating the next BDS.</i></p>
Entry Criteria	<p>Baselined (customer-signed) Build Definition Specification Properly configured development environment</p>
Exit Criteria	<p>Baselined source code integrated for all builds. Baselined (customer-signed) Validation Test Plan Baselined (customer-signed) System Test Plan</p>
Products	<p>Source Code – Validation Test Baseline (ready for testing) Validation Test Plan System Test Plan</p> <p><i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i></p>
Life Cycle Phase	<p>Validation Testing Phase</p>
Entry Criteria	<p>Baselined (customer-signed) Validation Test Plan Source Code – Validation Test Baseline Properly configured test environment</p>
Exit Criteria	<p>Updated Validation Test Software Baseline (testing completed and bugs fixed) Log or database of software bugs discovered during validation testing.</p>
Products	<p>Source Code – System Test Baseline (ready for System Test)</p> <p><i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i></p>

Life Cycle Phase	System/Performance Testing Phase
Entry Criteria	Source Code – System Test Baseline Baselined (customer-signed) System Test Plan Properly configured test environment
Exit Criteria	Updated System Test Software Baseline (testing completed and bugs fixed) Log or database of software bugs discovered during system testing.
Products	Source Code – Alpha Test Baseline (ready for Alpha Test) <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	Alpha Testing Phase
Entry Criteria	Source Code – Alpha Test Baseline Properly configured test environment
Exit Criteria	Updated Alpha Test Software Baseline (testing completed and bugs fixed) Log or database of software bugs discovered during alpha testing.
Products	Source Code – Beta Test Baseline (ready for Beta Test) <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	Beta Testing Phase
Entry Criteria	Source Code – Beta Test Baseline Properly configured test environment
Exit Criteria	Updated Beta Test Software Baseline (testing completed and bugs fixed) Log or database of software bugs discovered during beta testing.
Products	Source Code – Acceptance Test Baseline (ready for Acceptance Test) <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	System Acceptance Testing Phase
-------------------------	--



Entry Criteria	Source Code – System Acceptance Test Baseline Properly configured production environment
Exit Criteria	Updated Acceptance Test Software Baseline (testing completed and bugs fixed) Log or database of software bugs discovered during acceptance testing.
Products	Source Code – Delivery Baseline (ready for system delivery and installation) <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	System Delivery and Installation Phase
Entry Criteria	System Acceptance Letter from customer Source Code Delivery Baseline
Exit Criteria	Baselined (optional) (customer-signed) Training and Operations Procedures – <i>no specific Gecko Lite document template.</i> Baselined (customer-signed) System Delivery and Installation Plan Operational Test Results (after installation) – <i>no specific Gecko Lite document template.</i> Customer signoff/approval letter to commence operations
Products	Training and Operations Procedures System Delivery and Installation Plan Operational Test Results Customer signoff/approval to commence operations <i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i>

Life Cycle Phase	Project Closure Phase
Entry Criteria	Customer approval letter to commence operations Delivered and installed source code
Exit Criteria	Customer Property Inventory Report – <i>use Gecko Inventory Template or comparable inventory system report.</i>



	<p>Project File – deliver completed customer copy of project file and records. Keep a copy for corporate records.</p> <p>Customer Project Acceptance Letter – to signify completion and compliance with all project/contract deliverables and issues.</p>
Products	<p>Customer Property Inventory Report Project File Customer Project Acceptance Letter</p> <p><i>* A customer or peer-signed copy of all of the above products should be stored in the project file for auditing purposes.</i></p>

<p>Life Cycle Phase: All Periodic Deliverables/Products</p>
<p>Weekly Status Report Monthly Status Report Monthly Project Inventory Report/Listing (of customer property) Software Engineering Assessment questionnaire (generally every 2 – 4 months)</p> <p><i>* A copy of all of the above products should be stored in the project file for auditing purposes.</i></p>

Appendix C: Summary of Gecko Lite Principles

Gecko Lite Principles	
1	Adherence to a well-defined methodology improves the likelihood that a project will succeed.
2	A methodology must be flexible and adaptable.
3	Large projects with 15 or more developers should be broken into smaller segments (applications) for ease of management.
4	Requirements are the features or behavior of the system that are externally observable.
5	Requirements change and are often derived through a process of discovery and definition.
6	High-level Master Requirements should be defined in the beginning and detailed requirements defined "just-in-time" for development.



7	Identifying the "correct" stakeholders who will specify and sign-off on requirements is critical to building the "right" system.
8	Master Requirements are the "Outline" of the system. They can be rapidly documented and easily changed in successive builds.
9	A "build" represents a complete life cycle in miniature. Requirements, Design, and Testing.
10	Testing each build as it is added to the evolving system provides more opportunities for developers to find and fix problems with the design and fit of the components, the functionality and to gain further clarification of any requirements that may not be fully understood or finalized.
11	Successful projects focus on basic core requirements.
12	Incremental testing as each build is completed results in a more stable, better-tested system by the time the system is tested as a whole.