

**Computational-Based Engineering for Conceptual Design**

**CREATE-AV Program**

**Master Test Plan &   
Change Management Policy**

Version 2.0

July 22, 2011

**Distribution Statement**

Distribution Statement C: Distribution authorized to U.S. Government Agencies and their contractors due to Administrative or Operation Use as determined on 24 February 2010. Other requests for this document shall be referred to ASC/XRE, 1970 Monahan Way, WPAFB, OH 45433-7902.

**ITAR Statement**

WARNING: This document contains technical data whose export is restricted by the Arms Export Control Act (Title 22, U.S.C., Sec. 2751, et seq.) or the Export Administration Act of 1979, as amended, Title 50, U.S.C., App. 2401 et seq. Violation of these export laws are subject to severe criminal penalties. Disseminate in accordance with the provisions of DoD Directive 5230.25.

About This Document

Document Purpose

The *DaVinci* Master Test Plan (MTP) and Change Management Policy describes the strategy and procedures that will be used by the *DaVinci* project team during all aspects of testing, test objectives, and reporting. This document establishes a quality assurance policy for coordinating detailed test planning and aids in the management of testing. It specifically addresses *DaVinci* Release 1.0, with the assumption that future releases will require only minor changes to future test plans.

Document Prepared for

Gregory L. Roth, ASC/XRE, WPAFB

*DaVinci* Principal Developer

Gregory.Roth@WPAFB.af.mil

Intended Audience

All members of the *DaVinci* software product team, particularly the technical development and testing members.

Publication Date

Last saved: July 20, 2011

This printing: August 5, 2011

Page count: pages

Document Location

Request from Gregory L. Roth, ASC/XRE, WPAFB

Gregory.Roth@WPAFB.af.mil

1970 Monahan Way, WPAFB, OH 45433-7902

Prepared by

Al Cline, PMP, ASC/XRE, WPAFB

DaVinci Software Engineering Lead

alan.cline.ctr@wpafb.af.mil or alan.cline@flairsoft.net

Revision History

| Date | Version | Description | Modified By |
| --- | --- | --- | --- |
| 11 Mar 2010 | 0.1 | Initial Version | Al Cline |
| 17 May 2010 | 0.2 | Revised per review comments | Al Cline |
| 25 May 2010 | 0.3 | Formatting and Cleanup | Ray Reinhard |
| 16 Mar 2011 | 1.0 | Formatting and revisions for Release 1.0 | Sally Schehl |
| 15 Jul 2011 | 2.0 | Updated for DaVinci Release 2.0 | Al Cline |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

This page may be removed prior to publication at the discretion of the *DaVinci* product team's project manager.

Table of Contents

[About This Document i](#_Toc293055120)

[Revision History iii](#_Toc293055121)

[1. Introduction 1](#_Toc293055122)

[1.1. Purpose 1](#_Toc293055123)

[1.2. Scope 1](#_Toc293055124)

[1.3. Test Types 1](#_Toc293055125)

[1.4. Definitions 2](#_Toc293055126)

[2. Testing Strategy 7](#_Toc293055127)

[2.1. Iteration Planning 7](#_Toc293055128)

[2.2. Construction and Unit Testing 8](#_Toc293055129)

[2.3. Formal Testing 9](#_Toc293055130)

[2.4. System Test and Performance Strategy (Post-Iteration) 12](#_Toc293055131)

[3. Iteration Flow and QA 15](#_Toc293055132)

[3.2. Testing Reports 16](#_Toc293055133)

[4. Change Management 19](#_Toc293055134)

[5. Setup and Environmental Considerations 21](#_Toc293055135)

[5.1. QA Test Submission Requirements (Readiness Checklist) 21](#_Toc293055136)

[5.2. Test Area Requirements 21](#_Toc293055137)

[6. Assumptions and Risks 23](#_Toc293055138)

[6.1. Testing Assumptions/Dependencies 23](#_Toc293055139)

[6.2. Testing Risks, Prevention, and Contingencies 23](#_Toc293055140)

[7. Testing Personnel 25](#_Toc293055141)

[8. References 27](#_Toc293055142)

[9. Appendix 29](#_Toc293055143)

[9.1. Required QA Tags 29](#_Toc293055144)

[9.2. Automating Not\_Needed Test Methods 29](#_Toc293055145)

1. Introduction
   1. Purpose

The *DaVinci* Master Test Plan (MTP) and Change Management Policy illustrates how the testing process integrates with the development process, which includes change management. This document explains the test classification and purpose of various kinds of tests, and includes specific procedures for unit testing, integration testing, system and performance testing, and test forms and reporting.

* 1. Scope

Test planning and test cases cover all features, behavior, and appearance within the *DaVinci* context as defined in the project’s Software Requirements Specification (SRS), which serves as a source for black box (behavioral) test cases. Later designs to implement the individual *DaVinci* use cases will require the addition of other test cases.

* 1. Test Types

There are two operational categories for testing—tests that are done during an iteration, and tests that are done on the system following completion of an iteration. Definitions of and distinctions between *test cases*, *tests*, and *test types* are often blurred. For clarity, the [Definitions](#Definitions) section of this document provides definitions for these and related testing terms, whereas the sections below provide explanations for how the test types specific to the *DaVinci* project originate.

* + 1. Tests Within an Iteration

**Functional (platform-independent) tests:** Are transformed from the use cases of the requirements using the NEBS method (Normal, Error, Boundary, Special). These are particularly relevant for unit testing, integration testing, and documentation.

**Design (platform-dependent) tests:** Add design-oriented and platform-specific considerations to the functional tests.

**Unit tests:** Represent the testing of a class, method by method. JUnit test method code is augmented with Javadoc tags to designate the different kinds of tests derived using NEBS. Unit tests are designed by the developer. All unit tests are placed in JUnit test suites for regression. *The JUnit Standard* addresses components, classes, and system test approaches. (Refer to the *DaVinci* JUnit standard for more information.)

**Integration tests:** Are functional tests representing end-to-end testing of a use case. Integrating functional components as a separate test is not necessary in a thin-thread (agile) development approach. The integration exercises *DaVinci* one use case at a time. Although integration tests are also developed using NEBS technique, the Javadoc tags do not apply because integration tests cover multiple methods under test, instead of a single method as with JUnit tests.

* **Regression tests:** Ascertain that no unknown changes to the code (bugs) are introduced by later additions to *DaVinci* code. Previously exercised JUnit test suites, both unit tests and integration tests, must be run selectively, or in total, for regression testing. A full regression test must be run before code can be added to the shared code base.
  + 1. Tests at the End of/After an Iteration

**User demo:** Allows end users to evaluate the *DaVinci* product at pre-defined points in development to approve the current functionality for release. Often, the *DaVinci* development team drives the demo, but in some cases, the users may construct specific tests to verify and validate the *DaVinci* product. User demos are scheduled every three or four iterations.

**Performance evaluation:** Verifies the quality attributes (e.g., response time or reliability) of the *DaVinci* product after an individual iteration is approved. Performance bottlenecks are almost impossible to predict, so performance is measured and compared with the (non-functional) requirements in the SRS.

**Load testing:** Exercises *DaVinci* with a load in place (e.g., maximum number of intended users, heavy peer application traffic, highly complex and time-consuming calculations). *DaVinci* must be able to maintain the required performance during testing beneath this load.

* **Stress testing:** Exercises *DaVinci* with increasing loads (e.g., more concurrent users than designed, an overloading of *DaVinci’s* calculation abilities) until it crashes or demonstrates undesirable performance. Stress testing is performed infrequently but is used to validate design parameters and growth potential. The *DaVinci* development team observes the best practice of designing at 85% load.
  1. Definitions

This section contains a list of definitions for specialized terms used within this document. Refer to the *DaVinci* Glossary, a standalone document, for a complete listing of terms and acronyms specific to the *DaVinci* project.

1. **Backlog:** The number of use cases to be implemented in either an iteration or a release. Backlogs are usually measured in story points.
2. **Beta Test:** See *Test, Beta*
3. **Bounce:** A build that is immediately returned to the developers without further testing because the number of failure points exceeds its bounce limit, a pre-defined quality standard. For *DaVinci*, the bounce limit is 12 points. Testing a build after it reaches its bounce limit is not considered useful.
4. **Bounce Limit:** See Bounce.
5. **Build:** A code and test package with complete functionality for a desired set of use cases; a partial subset of the *DaVinci* product, before it is totally complete. Builds are checked into codebases daily, merged into the interim *DaVinci* product after each iteration, and released annually.
6. **Class:** A specification for an object, an autonomous unit of behavior, and data within an implementation. *DaVinci* is comprised of an integrated collection of class files defining its behavior and appearance. For JUnit, a test case and a test suite are a Java class.
7. **Change Management Board (CMB):** A group that resolves difficult incidents and is usually comprised of the test manager, developer, analyst, and primary stakeholder concerned with the test. Sometimes these incidents result in change requests.
8. **Component (testing context):** One of five functional areas within the *DaVinci* software architecture that maps one-to-one to Java packages: the human interface component (HIC), the component-interface validation component (CIV), the problem domain component (PDC), the data management component (DMC), and the external systems interface component (SIC). All components separate the various concerns for better maintainability and communicate through predefined protocols, and allow better test coverage.
9. **Defect:** A discrepancy between what was intended and what was observed. A code (or program) defect is a discrepancy between the behavior and appearance of the product and its requirements; but defects can also be found in test cases, design logic, requirements, and documentation.
10. **Formal Test:** (a) The phase of an iteration, after unit testing, in which the testers run use case (integration) tests, GUI tests, or system tests in the test environment; (b) A specific test that is performed during the formal test phase of an iteration.
11. **Incident:** An event identified by a failed test case. Incidents indicate a defect of some kind that originates in the code, test, requirements, or documentation.
12. **Iteration:** An interval of time in which part of a project is analyzed, designed, coded, unit tested, acceptance tested, and documented. Each DaVinci Release (annually) consists of about 13 iterations.
13. **Javadoc:** A documentation tool that automates program documentation, producing HTML output from comments embedded in the source code. See the *DaVinci* Javadoc Standard.
14. **JUnit:** A testing tool that automates unit and integration testing, producing pass/fail test results by executing test code against *DaVinci* source code at run-time. See the *DaVinci* JUnit Standard.
15. **Object Model:** A model that defines how standalone objects are logically organized and the dependencies of those objects; an abstraction of the working relations and responsibilities within *DaVinci*. The object model is the scope-of-control description of *DaVinci*, and use cases are its functional description.
16. **Planning Poker:** An agile development estimation technique whereby each member of a project team assigns story points for each use case until a consensus or a stable value is reached.
17. **Release:** A distribution of a product to its production environment. The *DaVinci* project team uses annual releases comprised of approximately 13 iterations.
18. **Regression Set:** The set of test cases used for regression testing. With JUnit, regression testing is often 100%; sometimes a small, or more targeted, set of tests are used, particularly for non-automated GUI testing.
19. **Requirements Traceability Matrix (RTM):** A table that maps the major features to the use cases (and user artifacts) to the test cases to the classes used to implement the test cases. The requirements traceability matrix is especially important during change management.
20. **Spike:** The activity of investigating a problem or use case prior to assigning it story points. Sprints are frequent in agile development, and are assigned and worked into the schedule like a use case.
21. **System Test and Performance (STP):** An evaluation or test of *DaVinci* to confirm whether its operational parameters and quality attributes are correct for the system treated as a whole.
22. **Story Points:** An arbitrary description of relative size for use case estimating, often restricted to only specific terms of the Fibonacci series (1, 2, 3, 5, 8, 13).
23. **Test:** A procedure for which a certain result (behavior, quality, genuineness) is expected, and must pass or fail that expectation. A test may exercise a use case, an entire system, or even a method within a class. For JUnit, a test is a collection of test cases that are contained in a single file.
24. **Test, Ad Hoc:** A test in which the user tries various activities without a pre-planned script. Ad hoc testing is sometimes called *random* testing. Ad hoc tests are used to allow the user to “get a feel” for the product, or explore a particular aspect of it.
25. **Test, Alpha:** An early test of a product before it is ready for a more thorough test. Alpha tests are usually internal to a project team. In the *DaVinci’s* project team’s case, selected users who want to run their own tests will do alpha tests.
26. **Test, Beta:** A pre-production test of the *DaVinci* product conducted by the Shadow-Ops group, who will run a full-quality battery of tests against the product before allowing it to be released.
27. **Test, Capacity:** See [Test, Load](#LoadTest).
28. **Test, Design:** A kind of functional test that includes platform-specific considerations.
29. **Test, Functional:** A test that exercises the behavior or appearance of the product based on the use cases. Functional tests in the purest sense are platform-independent; that is, they do not include design or system-specific considerations, but test the intrinsic logic.
30. **Test, Integration:** The end-to-end testing of a use case. Integration tests are a form of thin-thread testing. Formerly, an integration test exercised the connectivity between modules when built bottom-up.
31. **Test, Load:** An evaluation of the *DaVinci’s* performance under load. This is a kind of system test.
32. **Test, Regression:** A test whereby tests that have previously passed or failed have been run again to ensure that new code has not changed old code. A full regression test runs 100% of previously run tests.
33. **Test Script:** A description of a test containing pre-test state, inputs, expected results, and test result report. Similar tests can be contained within that same test script as *variants*. Each script is contained within its own file and given a unique number.
34. **Test, Stress:** A more severe kind of load test, where load is added until it exceeds design specifications and crashes. A stress test checks for design limits and shut-down quality.
35. **Test, System:** A test in which the system (or *DaVinci*) as a whole is tested. System tests include performance testing (evaluation), load testing, and stress testing.
36. **Test, Unit:** A test for a low-level module, such as a class method, or a single function.
37. **Test Case:** A test for a single feature or result. For JUnit, a test case is the test code that exercises a single method in a class.
38. **Test-Driven Development:** A best-practice software construction technique in which small test cases are written before the code that makes the test cases pass. This technique allows full code coverage, simplifies regression testing, and minimizes gold plating.
39. **Test Suite:** A collection of tests or test cases.
40. **Test Type**: [1] Functional tests conducted within an iteration, or system tests that are executed after an iteration. [2] Unit tests and integration tests are categorized as Normal, Error, Boundary, or Special. Test cases that cause Exceptions, or in which Null inputs are used, are classified as Error test types.
41. **Unified Modeling Language (UML):** A visual proof notation that allows use case requirements to be proven correct. UML contains a collection of diagrams for documenting and designing various features of a software product. The *DaVinci* project team uses UML for use case validation at both the analysis and design levels.
42. **Use Case:** A transaction between the user and the system; a low-level functional unit of the user’s workflow. Use cases are functional descriptions of the *DaVinci* product, and the object model is its scope-of-control description.
43. **Use Case Catalog:** The collection of use cases that comprise the *DaVinci* product’s behavior. The use case catalog contains the high-level description of how the user interacts with the system, and is the prime definition of the requirements to be built, tested, documented, and delivered.
44. **User Demo:** A briefing-style presentation or showing of a *DaVinci* feature set at predefined points in development to stakeholders and users, who approve or revise the current functionality.
45. **Use Case Validation:** The process by which the object model and the use cases are cross-matched with UML sequence diagrams. In this technique, the UML acts as a visual proofing language to rigorously prove that the (mathematically orthogonal) functional and object decompositions are consistent and correct.
46. **Wireframe:** A paper mockup or static visual representation of *DaVinci’s* graphical user interface and widgets, such as drop-down boxes, menus, text edit areas, etc., and the navigational relationships between theses areas and widgets.
47. Testing Strategy

*DaVinci* is built using the best practices of agile development, which employs a holistic and integrated set of processes for analysis, design, construction, testing, and documentation.

There are four processes focused on testing that are used to verify and validate the *DaVinci* product—Iteration Planning, Construction and Unit Testing, Formal Testing, and System Test and Performance Strategy (Post-Iteration). The iteration workflow shown in displays the steps associated with these processes in an abstract way. Specific details of this flow implemented is shown in section 3. Iteration Flow and QA.



Figure 1. The *DaVinci* iteration process.

* 1. Iteration Planning

At the beginning of each iteration, the *DaVinci* development team identifies a set of use cases from the use case catalog (which is prioritized by stakeholder preference) that can be designed, coded, tested, and documented within the twelve-day iteration window. For each use case in the set, the team analyzes the use case’s detail, works out a design to implement it, and validates that use case with UML. During this process, more use cases may be added or removed, as appropriate, from the use case catalog, now called the *backlog*. This flow is shown in Steps 0, 1, 2, and 3 of .

* 1. Construction and Unit Testing

After the use cases are designed and validated, the developers begin to code and unit test each use case using a best practice called *Test-Driven Development (TDD)*. Through this development process, developers will write a small JUnit test and then write the code to make that test pass. They will continue writing small tests and coding units until each use case is completed at all but the user interface level. The developers document the purpose of their classes, methods, and attributes in Javadoc.

The *DaVinci* development team will follow the Java coding standard, the JUnit testing standard, and the Javadoc documentation standard to perform these tasks. JUnit and Javadoc are automation tools. All JUnit tests are kept for later regression testing. All tests must pass before the build or code module can be accepted for formal testing by the quality assurance (QA) manager. This flow is shown in Step 4 of [Figure 1](#Figure1).

An important QA rule that the developers follow is the ***Four-Hour Rule***. If a developer cannot solve a problem or a sticky bit of code within four hours, he or she will discuss it with someone else. That person need not be a programmer, just someone with whom the developer can rethink the problem as they tell it, or from whom the developer can obtain a different perspective. This rule has stopped many "spinning wheels" of non-productivity and helps achieve the goal of moving the project forward again.

While the developers are coding and unit testing *DaVinci’s* non-user interface components, the requirements team and other users build the wireframes, the graphical user interface (GUI) artifacts (e.g., drop-down boxes, menus, and other user interface widgets) with the help of specific members of the development team. These mockups include the navigational relationships between screens. Often, a UI expert or requirements analyst will assist the users, or provide a straw-man design for them to review. When the user interface design is completed, it is added to the by-now working parts of the *DaVinci* product and tested. This activity is shown in Step 14 of [Figure 1](#Figure1).

Unfortunately, JUnit does not work well with GUIs, so a design approach to the problem is used. The standard GUI objects (widgets, frames, dropdown boxes and others) are paired tightly with a validator or filtering kind of object, called a CIV: Component Integration Validator (pronounced *sieve*) that allows all but 20% of DaVinci to be tested automatically. JUnit accesses into the CIV for automated testing. The CIV approach is an extension of the best practice (Model-View-Presenter) approach, augments the three-tiered software architecture well, minimizes the GUI parts that need to be manually inspected, and allows JUnit to maximize its automated testing of the GUI.

As each build (the code and test package with complete functionality for the desired use cases) is completed, it is put into the shared codebase. The *DaVinci* project team uses Subversion (SVN) as a version control tool. A build cannot be entered into the codebase until it passes a full regression test; consequently, regressions tests are run daily.

Once the use cases are coded, tested, and documented, they are released to the test manager and run through a readiness checklist. The checklist and a few automated tools are used to ensure that *DaVinci* is ready for formal testing; otherwise, the build is turned back to the developers for remedy. This activity is shown in Step 6 of [Figure 1](#Figure1).

Certain Javadoc tags are added to JUnit test method to assist the QA tool in verifying that all methods have unit tests, and that all classes are tested. See the section on QA Procedures.

* 1. Formal Testing

While the developers are coding and unit testing, and the users are designing the interface, the testers (concurrently) begin writing *formal test cases* from the detailed use case requirements. The test cases include setting up pre-conditional state situations that the tests might need as part of detailed test planning. The testers derive a set of test cases from each use case using the **NEBS transform**. Each use case is exercised with Normal, Error, Boundary, and Special conditions to verify the use case. This flow is shown in Steps 7, 8, and 9 of [Figure 1](#Figure1).

A *formal test case* exercises a use case to demonstrate that the behavior or appearance of the product confirms a single predefined result, as defined by the requirements. Test cases must be written non-ambiguously to define clearly what must be executed, and what the tester is expected to observe. Non-GUI tests are coded in JUnit to enforce such clarity, to automate the testing, which speeds validation and verification dramatically, and to provide an easy regression suite. The Javadoc tool documents those aspects of the test case.

* + 1. Test Case Naming Conventions

Each test case is described in a test script, identified using a naming convention based on the use case from which it was derived, and the kind of test it is. The ISO naming convention is used to map the test cases to the use cases. The naming convention has the form

TC<nn> - [N | E | B | S]<ddd[a]>

Where:

<nn> is the use case number;

One of the letters N, E, B, or S indicates the type of test case (Normal, Error, Boundary, or Special)

* <ddd> is the test case number, which starts at 01 for each of the NEBS categories.
* [a] is the optional variant of the test case within the test script. Variants are contained within the test script, so this suffix does not show in the script header.

Test cases usually contain small variants of the main test case, so a single test script may actually have several test cases in it. For referencing the test results, the variants are given lower case alphabetics, with the original test case having no suffix. For example, three similar test cases (variants) in test script TC03-N03 will be numbered as tests TC03-N03 (header), TC03-N03a, TC03-N03b in the test report.

New test scripts should be made instead of test variants if they have different pre-test state or different user experience (UX) artifacts (e.g. wireframes). Test case results are mapped to the use case and UX artifacts in the RTM.

Test scripts originate in a particular iteration, but tests may be added during subsequent iterations. As a result, the test case number does not restart at 01 for the same use case and test type when tests are added to later iterations; numbering continues where it left off in the previous iteration.

For example, use case 03 may have five error test scripts, from TC03-E01 to TC03-E05 in iteration 4. Later, as the result of a change request, two more error cases may be added into iteration 7. Then these cases are numbered TC03-E06 and TC03-E07. Regardless of when the CR was created, the CR is stored with the use case and its tests.

The test case identifiers are added into the Requirements Traceability Matrix (RTM) by the test manager, who also maintains the test log of failed and passed test cases. The RTM can be used as in index to find the latest test case or use case in SVN.

* + 1. Pass/Fail Criteria Scoring

All tests executed are recorded as *passed* or *failed*. Failed tests are scored for severity. If the total score for all failed tests is acceptable, the build is staged for the user demo or post-iteration testing. Test scoring also includes any regression testing conducted at this time. This flow is shown in Steps 10, 11, and 13 of [Figure 1](#Figure1).

When a test case does not provide the expected results, it is reported as an *incident*. All such incidents must be corrected before the code will be passed.

Although an incident represents a failed test, it does not necessarily imply that a code defect was the cause. A failed test case may also indicate a requirement defect or a bad test (test defect), either of which may result in a change request.

* + 1. Suspension/Rejection Criteria

The quality assurance and testing groups set suspension/rejection criteria as a minimum acceptance standard. Each failed test is scored with one of the following levels of severity. The point ranges are given to allow the tester some latitude in categorizing the incident.

**High severity incident (5–6 points):** *DaVinci* fails to execute the use case or crashes with no work-around. It is up to the test manager to decide whether to stop testing at this time. This incident will be documented as High in the QA defect tracking system.

**Medium severity incident/defect (3–4 points):** *DaVinci* fails to execute the use case or crashes, but a work-around is available. This defect or incident will need to be addressed prior to the next build. This incident will be documented as Medium in the QA defect tracking system.

* **Low severity incident/defect (1–2 points):** This failure is usually a cosmetic or screen/report layout discrepancy only. This incident will be documented as Low in the QA defect tracking system.

Testing will stop, and the build will be returned *immediately* to the *DaVinci* development team as unacceptable without further testing if a total of 12 or more defect points are discovered. In this case, the build is considered *bounced*. A Build is bounced because it is not worth the tester’s time to continue testing.

* + 1. Incident Resolution

At the end of testing activities, the analyst reviews the incidents for code defects, requirements defects, or testing defects. Code defects are returned to the lead developer; test defects are returned to the tester; and requirements defects are returned to the analyst. The analyst (a) corrects the appropriate *DaVinci* requirements documents, (b) investigates whether a change request is warranted, and (c) resolves the kinds of defects, comparing the test, the test result, and the requirements.

In some cases, the tester and/or lead developers may not agree with the analyst. These incidents, and the ones that imply a change request, are taken to the Change Management Board (CMB) for resolution. The CMB is comprised of the test manager, the lead developer, the analyst, and the primary stakeholder for the use case tested. This resolution activity is described in the [Change Management](#ChangeManagement) section of this document, and shown as part of Step 12 of [Figure 1](#Figure1).

* + 1. Regression Test Strategy

The regression test strategy defines how to retest all or a subset of the functional test cases. For automated testing, the *regression set* consists of all unit and integration tests. Automated regression testing is required to pass 100% before it can be uploaded to the shared codebase.

For GUI testing, the regression set consists of 30% to 100% of the test cases on the primary function path of use cases, tests that spawned incidents, and any other tests the test manager or GUI tester might think pertinent.

At the end of each annual DaVinci Release, all GUI tests must be executed and pass before the QA Test. The GUI test scripts are analyzed for validity and appropriateness at this time.

The selection of partial regression sets is based on the following criteria:

Normal test cases that contain primary functionality,

Boundary and error cases that are most likely to fail,

Design cases that might fail due to developer interpretation,

Previous test cases that have failed, and

* Test cases for recent code (within the last iteration or two).

Full regression testing must occur when new code is added to the code base, before the build is submitted for a user demo, and before the *DaVinci* version is released. With JUnit, a 100% regression is typical because of the automation and ease of testing.

After regression testing, incident recording, and perhaps any CMB resolutions, the iteration is approved for a user demo. This flow is shown in Steps 10, 11, and 13 of [Figure 1](#Figure1).

* + 1. User Demo and Alpha Test

If the development team leads the user through a briefing-style presentation of *DaVinci’s* latest features, that presentation is called a *user demo*. If the user downloads and tests the latest build personally (before beta release), then it is referred to as an *alpha test*.

For *DaVinci*, most users will perform ad hoc testing with a copy they can install and run on their workstations. In some cases, the users may want to run a more structured series of tests. If so, they can write their own acceptance test cases and execute them after the test manager has approved the build for user demo. Users report any incidents they find using the test incident report form (refer to the [Testing Reports](#TestingReports) section), and those incidents are given to the test manager for recording, and ultimately, to the developers, testers, analyst, or CMB for resolution.

When the user testing *DaVinci* is satisfied, he or she can approve the iteration for release. It will be staged, regression-tested periodically as new code is added to the existing product, and eventually sent for *beta testing*. This flow is shown in Steps 15, 16, and 17 of [Figure 1](#Figure1).

For DaVinci 2.0, alpha tests should include multi-user network testing and multi-platform testing for the various DaVinci and Capstone platforms supported.

* + 1. System Test and Performance Strategy (Post-Iteration)

DaVinci Release 1.0 ran on a single workstation, and consequently, the post-iteration testing was not as important as for Release 2.0, which supports collaboration through local area networks. The following post-iteration testing will be done before or in conjunction with the beta testing (see section 2.3.8).

Post-iteration testing is performed after all the iterations are complete, before the *DaVinci* version is released to the formal QA Team. The system test and performance (STP) evaluation produces a report on operational parameters and integrated testing. The testing and report contain four parts.

1. The **STP** portion evaluates response times versus selected parameters, such as depth and breadth of a system model; complexity of the lazy evaluation; or exporting to Kestrel or Helios. For *DaVinci*, it will be critical to include physics-based testing, error and uncertainty testing, and geometry evaluations. Although the CREATE Program Office generally requests all tests to be compared against actual field test data, DaVinci’s conceptual models have none. Evaluation of STP results will be compared against data collected in some other way that can corroborate DaVinci’s results.
2. **Load (capacity) testing** evaluates performance for the system under load. There are no definitive requirements yet for exactly what constitutes a load for *DaVinci* Release 2.0, although they may be available in the forthcoming Software Requirements Specification SRS 2.0.
3. **Stress testing** evaluates performance under stress. The evaluation, in addition to the load test, checks whether the system does a graceful shut down or a hard crash when it fails, and how recovery is executed. Stress testing reports the utilization trend, which should indicate that load test (design) is 85% of the stress levels that caused a crash. *DaVinci* will be designed to operate at 85% of the stress load. If *DaVinci* crashes too far below 85% stress levels, then *DaVinci* has been under-designed, and has minimal growth potential. The Quality Report documents these statistical characteristics (refer to the [Quality Report](#QualityReport) section of this document).
4. **Reliability/stability evaluations** measure performance on the number and kind of defects that occur, mean-time-to-repair these defects, percent availability of the system, and other operational parameters. This section of the STP is typically an operational procedure for the product to identify what will and what will not be tracked, according to the operational parameters required.
5. **A zero-defect product** is defined as one in which there are no defects found from the time the product went into production, and the time afterwards that it took to build the product. Example: A product that took nine months to build would be monitored for nine months after it went into production. If *no defects* were found in those nine months, the product could claim that it was a zero-defect project. A zero-defect *project* means that not only was the product zero-defect, but also that the project finished on-time, and on-budget, within low acceptable tolerances. The methods described above have resulted in over 60% zero-defect projects.
   * 1. Beta Testing and Change Control Board

Beta testing is a pre-production test of the DaVinci product conducted by the Shadow-Ops group, who will run a full-quality battery of tests against the product before allowing it to be generally released. DaVinci 2.0 must completely pass its alpha testing, and other documentation requirements, before being released for Beta testing.

The Configuration Control Board (CCB), which approves DaVinci for Beta release, may also request change requests or repairs to DaVinci. Any changes from the CCB must also be implemented and pass alpha test before Beta release.

Multi-platform testing will be done off-site by the CREATE Buildmaster after alpha test passes.

1. Iteration Flow and QA
   * 1. Readiness Checklist

The Build is considered complete for the iteration when either of two events occur:

* All designated use cases and bug repairs for the iteration are complete, and the Build passes all automated (current and regression) testing at 100%. The team decides the Build is ready to go to GUI testing.
* The iteration’s deadline for coding is reached; GUI testing must now be performed in order to complete the iteration within the time-box. All outstanding use cases and bugs are documented as carry-overs (CYOs) for later iterations.

Once the team agrees that the scope has been implemented and passed regression test 100%, the Build is “officially released” to the GUI testers for formal testing.

* + 1. Testing

*Formal Testing:* For DaVinci, formal testing refers to GUI testing after the Build is complete. Delivery to the groups outside of the DaVinci development team—QA, ShadowOps, or Beta Testers—is also formal testing, and is considered under the *Delivery* section of the *Iteration Flow* document.

*Automated Testing*: After the team approves the readiness checklist, he or she deploys the updated codebase to the test machine (completely or incrementally). If the install fails, the Build is returned to the developer. If it passes, the tester runs the GUI tests, and records the results in the test results form. If the *bounce limit* of test points is reached, the Build is returned to the developer immediately. If the tests pass acceptably but with a few defects, the GUI tester fills out the test result form and summary report, and gives it to the QA Manager.

*GUI testing*. Note that the developer does not get credit for implementing his or her story points until the tester passes that use case. The iteration manager or CMB handles disputes.

For each use case, the test cases are marked as passed or failed. Defects rates, number of use cases, and other quality metrics are recorded for the biweekly test results reports. Refer to the Testing Reports section for more details.

NOTE: Some GUI testing and automated testing may be run and identify defects before the Build is “officially released” to formal testing. These bugs may be repaired without report because any testing before “official release” to the GUI tester is considered informal, and bugs can be repaired as part of development. After the Build is delivered for GUI testing, then any defects are reported as such, and goes into the test report.

* 1. Testing Reports

The testing artifacts created to control and organize information between the concurrent tracks of development, formal testing, and change management, are described below.

* + 1. Test Case Form

Each use case generates multiple test scripts and multiple test cases, categorized into N, E, B, or S groups. The tester creates a JUnit integration test file for each use case to be tested. The Javadoc comments for each JUnit test file are used to document that test case. Each test case that passes or fails is documented and scored in the incident report. Refer to the standalone JUnit testing standard for more detail, available in the [Appendix](#Appendix).

* + 1. Build Handoff Form (BHF)

The build handoff form is used to track incomplete or defective code, and to maintain accountability for the features in the current build as it passes through development, formal testing, and product transition. The BHF records the uses cases or features that were not completed or fully debugged by the time formal testing begins on the iteration just ended. This form helps the tester know what not to test, and the user to know what not to expect at the user demo. Any use case that is not 100% implemented and tested is marked on the BHF, and becomes part of the backlog. See the [Appendix](#Appendix) for a sample of the form.

*Warning*: The BHF can be used to circumvent proper agile practices, and particularly, the time-box control. Avoid using the BHF unless absolutely necessary, else it becomes a way to avoid building the full scope of the iteration. The BHF gives the developers a “free pass” not to implement a use case, or a portion thereof, because it is recorded as not done yet. The subsequent carry-overs (CYOs) across several iterations will continue to accumulate exponentially until the team cannot catch up.

* + 1. Change Request Form (CRF)

Any request to change the *DaVinci* requirements is captured in a change request form, which will affect development, testing, and the project schedule. The change request form contains a section for the impact analysis results so the user can approve the corresponding schedule or scope change. Refer to the [Change Management](#ChangeManagement) section of this document for more information, or see the [Appendix](#Appendix) for a sample of the CRF.

* + 1. Defect Tracking Report

The defect tracking report shows the number of incidents and defects, defect origination, and change requests (approved and denied) for a given *DaVinci* iteration. The quality report summarizes these elements into release statistics. Refer to the [Appendix](#Appendix) for a sample of this report.

* + 1. Quality Report

The quality report is a summary and accumulation of the individual iteration metrics, change requests, team velocities, and other statistical results from building *DaVinci*. The quality report contains the following metrics.

1. **Team velocity:** The team velocity is the total number of story points created and successfully tested by the end of each iteration. The *developer velocity* is the number of use cases built and unit tested in the iteration by a specific developer. It starts when the developer takes the use case card to start the design, and ends when he or she has completed the readiness checklist, which includes delivering the use case code to the shared codebase and passing full regression testing. Adding a use case to the BHF does not count toward developer velocity. Developer velocity is important to provide feedback to improve his or her estimates for future use cases.
2. **Burn-down chart:** The burn-down chart shows the scope of *DaVinci’s* progress in story points throughout the current release. It shows the release backlog and the expected number of story points left to implement. It reflects the use cases added or dropped, either by discovery or as a result of a change request. The burn-down chart is the agile method’s graphical approach to earned value management, where the planned value is the beginning release backlog in story points, and the earned value is the story points ready to be released. The value can be represented as a percentage done (recommended) or converted to dollars by using the blended rate for the team.
3. **Average time for repair policy (ATR):** The average time for repair is calculated from the first time a defect is caught to the time it is repaired, even if the repair occurs across iterations. It is used to estimate time-to-complete and to guide performance requirements of future projects. This metric was not collected in Release 1, but will be collected and reported Release 2.
4. **Variance from schedule:** All story points estimated at the planning poker are expected to be completed by the end of the iteration; they become the iteration schedule. Any use cases not completed are considered CYOs and added back into the release backlog (note that the burn-down chart will show an upward peak when this happens). Increasing story points from one iteration to another is not unexpected, but if a trend begins to develop, and the story points begin to accumulate from iteration to iteration, this indicates that the release schedule and scope are in jeopardy.
5. Change Management

After the Software Requirements Specification (SRS) has been approved, any changes to *DaVinci* must go through the change management process. This process is designed to 1) ensure that all stakeholders who agreed with the original requirements will still get what they want, and 2) find how the change might impact *DaVinci* in both scope and schedule.

Anyone can submit a change request to add, modify, or delete features or use cases that are specified in the SRS. A *Change Request Form* (CRF) can also be used to change the implementation order of a use case, although that is usually handled in one of the CMB meetings.

The CRF is given to the iteration manager, and an impact analysis is performed to find the consequences of the change. Most easily, the RTM is investigated for what use cases, code classes, tests, and documentation are affected. Documentation includes the validated design model, Javadoc code documentation, and the *DaVinci* User Manual.

After the CRF is updated for impact, in both hours and story points, the CRF is shown to the requirements team. The team then decides whether they want to incorporate the change, dismiss it, or reschedule it for a later release. Often, the change impacts the project or iteration schedule more than the stakeholder or team would prefer, and the change request is dropped.

The change management procedure may be summarized as follows:

1. **Stakeholder:** Fill out a change request form indicating the desired change.
2. **Iteration manager:** Assign the CRF to a developer to find the impact.
3. **CMB:** Compare the CRF with the impact and priority to decide whether to accept or dismiss the change. If the change is accepted, the iteration manager schedules it into the next or later iterations, depending on priority and impact. **If the change is requested to be implemented in the *current* iteration, then the Primary Developer must sign off on it before it is allowed to be implemented.**
4. **Iteration manager:** If the CRF is dismissed, archive it. If the change was approved, adjust the iteration schedule and give the change to the development team to implement for the planned iteration.

NOTE: Defects *never* require a change request, and do not warrant a change in schedule or scope. The stakeholder never ordered a defect, and the development team should not be putting them into the Build. Alternatively, a change request *always* requires an impact analysis and schedule change.

1. Setup and Environmental Considerations
   1. QA Test Submission Requirements (Readiness Checklist)

Before the tester accepts a use case for formal testing, the readiness checklist must be completed. The QA manager (iteration manager) must verify that the following statements are true:

1. Each use case has a detailed use case form, and a validated analysis object model (class diagram and sequence diagram).
2. Each detailed use case references the proper UX artifact (wireframe) and the UX number is recorded in the RTM with the use case.
3. Each use case has a validated design object model (class diagram and sequence diagram) and indicates that a team review approved it. For GUI parts of the use case, the analysis object model can be used instead.
4. There is a JUnit class file for each .java file and class associated with the use case. The unit tests comply with the *DaVinci* **JUnit standards**, which includes the Javadoc QA tag for NEBS unit test code.
5. There is Javadoc documentation of the source code to explain the classes, and the various methods and attributes within each class. This documentation complies with the *DaVinci* **Javadoc standards**.
6. The Java code that implements the use case successfully passed a 100% regression test, and is checked into the shared codebase. The code complies with the *DaVinci* **Java coding standards**.
7. For each use case, the RTM has been updated with the UX artifacts, test cases, and design classes associated with it.

Any deviation from the above list requires the approval of the QA manager.

* 1. Test Area Requirements
     1. Testing Workstation

For *DaVinci* Release 2.0, a commodity-level workstation will be used on a local area network; there are no special environmental considerations for the test area.

The test machine(s) will be a duplicate of a typical single-user workstation. Three machines will be needed for networking tests. DaVinci 2.0’s minimal architecture is an ad hoc client-server topology: machine A acting as server, machine B acting as client. For true peer-to-peer (P2P), a third client machine C must be able to join, and all machines communicate to each other. (The two clients must be able to communicate and send messages between each other without going through the server machine A.)

At no time will the developers have access to the testing environment for changing tests, repositories, or other post-development work. All changes must occur on the development side of the shared codebase and be processed through the readiness checklist.

* + 1. Test Database

*DaVinci* Release 2.0 has chosen to save all tests and related documentation in its SVN repository. The same repository is able to store various configurations of unit, integration, and GUI test suites.

* + 1. Test Tools and Support Software Needed

The testing machine will be set up with same software and tools as for the developers, as identified below. All are open-source products.

Eclipse Indigo, IDE and platform framework with JDK 6 for DaVinci’s code generation

Java 1.6 SE (Standard Edition) compiler with Javadoc 1.6

JUnit 4 for unit and integration testing

Checkstyle for standards support

A customized QA software tool written by the *DaVinci* team

* *OpenOffice* versions of MS Word and MS Excel for various reports

*DaVinci*, in general, must execute many operating systems and platforms. See the SRS 2.0 for updates to this list

* Windows Vista, 32-bit and 64-bit system
* Windows 7, 32-bit and 64-bit system
* CentOS 64-bit operating system
* Linux Fedora 64-bit operating system
* Mac OS X (Snow Leopard 10.6) duel-core processors
* LAN server to support a network of these development and test machines to appropriatly emulate the user environment.

Networking tests must check against multiple permutations of these platforms; to ensure that Mac OS X client can talk to a Winddows Vista client and CentOS server, for example.

1. Assumptions and Risks
   1. Testing Assumptions/Dependencies

Testing assumptions are events or conditions on which the *testing or test management* are based, and that must be true. Project and product assumptions and dependencies are identified in other documents. If any assumption becomes false, or a dependency becomes unavailable or late, the testing, and therefore the project, may be impacted.

Below are several testing assumptions associated with the *DaVinci* product:

The testing environment is a duplicate of the eventual production environment.

Testing activities will have sufficient personnel and resources allocated to provide the level of service and effort defined in this test plan.

Developers cannot, and will not be permitted, to test their own code except unit testing; nor will they have access to the testing environment after the build is moved into the formal testing phase.

* Unit testing and regression testing will be done with the JUnit tool. Formal test cases are derived from use cases in the SRS using the NEBS method.
  1. Testing Risks, Prevention, and Contingencies

Currently, there are no identified testing risks. See the Project Risk Management Plan for a more comprehensive set of risks that may affect testing.

1. Testing Personnel

Table 1 lists the current *DaVinci* development team. Although the *DaVinci* project team members will rotate through various assignments, each person will have a specialty to which the majority of his or her time is allocated.

| **Name** | **Job Title** | **Prime/Secondary Focus** |
| --- | --- | --- |
| Gregory Roth | Primary Developer | Management/Development |
| John Livingston | Lead Designer | Modeling/ Design |
| Eric Dabbaghchi | Chief Architect | Architecture/Middleware |
| Al Cline | Software Engineering Lead, Iteration Manager, Quality Assurance | Coordination/QA/Analysis |
| Cheyney Loffing | Developer | Testing/Development |
| Alex Deschapelles | Developer | Development/Testing |

Table 1. *DaVinci* project team personnel.

1. References

Below is a list of the *DaVinci* project documents and other resources referenced within this document.

[1] Cline, A. Software Requirements Specification, *DaVinci* Project, March 10, 2010.

[2] Cline, A. JUnit Standard, *DaVinci* Project, forthcoming.

[3] Cohn, M. *Agile Estimating and Planning*. Pearson Education, Inc., Upper Saddle River, New Jersey, 2006.

Sally,

Please add the references for the other standards in this document.

1. Appendix
   1. Required QA Tags

DaVinci standards have defined a set of Javadoc tags to be applied in the Javadoc comment section of unit test code. These tags work with Javadoc and DaVinci’s QA tool to audit the test methods and classes for code coverage and types of testing.

When a new JUnit test is created, the developer should set a flag that generates stubs for each test case. The JUnit class creator will add an empty stub with a link annotation comment for each class method under test (*target method*) automatically. These annotations show up in the Javadoc documentation.

Developers should fill in the goal for each of the type of test.

/\*\* Test method for {@link pdc.myClass.method(argType1, argType2)}

\*

\* @Normal <goal of the normal flow for the test>

\* @Error <goal of the normal flow for the test; include

\* reasons for exceptions thrown>

\* @Null <expected results for null input parms>

\* @Boundary <which boundary is being tested>

\* @Special <what is special about this case>

\*/

public void test<TargetMethod>()

{

// ...

}

The first three QA tags are required for every target method; the last two are optional. Sometimes the tags are used within the same test method, sometimes they are spread across multiple test methods that exercise the same target method. (In rare cases, Error or Null test cases cause a compile error and therefore do not apply. However, the tag should still be included with a N/A comment.)

When Javadoc is run, the QA tags will explain the purpose of the each of the test case types, listed in bold face beneath the test description, which contains the target method signature. The QA tool scans the test code and source code to verify that all required tests cases are included.

* 1. Automating Not\_Needed Test Methods

In some cases, test methods are not required, such as trivial accessor methods. In this case, the @Not\_Needed tag is used and a list of target methods (and an associated list of @link comments) are required for documentation’s sake. Also, it was easy to miss having a test case method with every target method because the developer usually must switch back and forth between two views.

The JUnit automatic class created of JUnit 1.4 makes this task now much easier. After the new JUnit class is created, it is populated with empty stubs (having only fail() statements in them) and a @link comment to the target method statement. Here’s how.

1. Move all the link statements for target methods that do not need test cases into the comment section of the public void testNotNeeded()method at the bottom of the JUnit class file.
2. Remove their associated stub test methods. All remaining stubs are those tests methods that need to be implemented.
3. Remove the ‘test’ prefix from the testNotNeeded() method so that the setUp() and tearDown() methods are not unnecessarily called. The method public void NotNeeded() is purely for organization and documentation.

When Javadoc is run, the detailed method description for the NotNeeded() test case will list all the methods for which there are no tests, and the QA tools will also not flag them as target tests without test cases.