Project Test Plan
For
eFlash

Preliminary Version
Revision 0.8

Authors: Siu-Pang(David) Chu, Sean Gabriel, Daniel Honegger, Darren Lam, Ying Tat Ng, Aretha Samuel, Kenneth Wong, Tony Wu, Anthony Yee

Date: March 22, 2006

PROJECT ABSTRACT

The eFlash team is designing a study-skills application designed to eliminate a student's need for paper flash cards. Our software intends to replace these vital learning tools while improving on the issues inherent in a pen-and-paper system.

DISTRIBUTION NOTES:

Review Deadline Date: _____
Review Meeting Date: _____

Comment(s):
_____
_____
_____
_____

Reviewer List:

Distribution List:

Document Revision History

Revision	Date	Author(s)	Comments
0.1	2006-03-20	Wu	Overview
0.2	2006-03-20	Ng	Unit Tests
0.3	2006-03-21	Ng / Lam	Integration
0.4	2006-03-21	Chu / Honegger / Yee	System Tests
0.5	2006-03-21	Lam / Samuel	Regression
0.6	2006-03-21	Gabriel	Reports Requirement
0.7	2006-03-22	Wong / Gabriel	Estimates
0.8	2006-03-22	Wong / Gabriel	Preliminary Version with Revisions

Table of Contents

Document Overview……... 4
Testing Strategy…….. 8
Unit Testing……... 8
Integration Testing…….....14
System Testing…….....16
Regression Testing…..…….....20
Test Report Requirements……21
Resource and Time Estimates……...22
Issues and Checklist……...23

Document Overview

Purpose

This document should describe all testing activities by the development group to meet the product quality criteria. Ideally these metrics are associated with the product quality criteria and the precise requirements would be described in the document known as the Product Quality Plan. Time does not always allow for this. These testing activities will be performed for the project prior to submission for release system testing. Release system testing is the system testing required before the software can be scheduled for release. The document should capture the thought processes used to design tests for this project.

Authors

Developers are the main authors of this document, though other team members have made significant contributions as well.

Target Audience

The audience for this specification is fellow developers in the work group.

Specification Flow

The Project Test Plan is written concurrently with the Implementation Specification.

Review Considerations

The primary review objective is to check that all functionality will be adequately tested.

Testing Overview

Introduction

Students from all academic disciplines need to quiz themselves constantly over the topics they learn in their classes. Flash Cards have been proven to be extremely helpful for students' self-study habits. However, the current procedure of manually creating paper flash cards can be exceedingly unmanageable and time-consuming, and can therefore be radically improved with the proper software application. Students currently spend lots of money and time either manually making paper flash cards or buying pre-made ones. They then have to categorize the cards and confirm that they are not misplaced. These actions can be tedious and inefficient. Additionally, these cards can sometimes become transparent enough so that the answers on the back of the cards become visible to the students. This unfortunate feature limits the students' ability to self-test their knowledge. Secondly, music and art students are generally at a disadvantage when it comes to quizzing themselves over the material they learnt in their classes. These students are not able to use flash cards to quiz themselves over the different art and music pieces they have seen and heard. Therefore, a suitable software application must be created that will aim to resolve all of these problems.

eFlash is the answer because it can efficiently store, and categorize all different types of student-created flash cards. This software allows for many different types of input, such as text, sound, and picture. The user can record his/her voice, and input those sound recordings as descriptions or themes. The user can also upload any images or sound-files saved on his/her computer to the eFlash database, which will then be used by the application to create customized multiple-choice and matching quizzes. The application can also simulate the flash-card learning process by utilizing the computer screen as a flash card. All these features are offered so that students may use eFlash for as many different purposes as possible. As a bonus, eFlash will also enable users to post their flash cards publicly online so that many different students can share flash cards. Students can download these publicly-shared flash cards online and can modify them on their own computer. This feature will save students even more time, as they do not have to create these flash cards themselves.

Software Product Description

eFlash provides an attractive alternative to physical flashcards by digitally simulating them to save the user from their inconveniences while retaining all of their advantages and benefits and providing additional functionality. Users can create flashcards organized into decks, just like they used to with physical flashcards, except now the organization is done automatically. Additionally, they have full control over the layout of each flashcard, and can even add images and sound clips. When reviewing, users can choose to use the conventional method of simply flipping through flashcards. Alternatively, they can use eFlash's various quizzing options, which are designed to help memorization, and keep track of their performance as they go. If the users wish, they can even export the flashcards in PowerPoint presentation

format to view on their own viewer. Finally, users can connect to the eFlash network to download thousands of flashcards made by other users, or upload their own flashcards to share with others.

Software Project Limitations

Please refer to the Design Specifications and Implementation Specifications.

Software Project Assumptions

For proper operation of our software, we assume that the user's machine has or is capable of supporting .NET Framework and MySQL database server. See Design Specification and Implementation Specification for additional details.

Testing Dependencies and Assumptions

Our final tests will be conducted on a variety of system configurations. We assume that these systems are virus-free and functional, and that users will not attempt to run eFlash on any system with lower capabilities than these. We also assume that all unit tests will fully cover all of the requirements of the unit tested, so that integration tests will be effective and valuable.

References / Contacts / Terminology

References:

Implementation Specification
Requirements Specification
Design Specification

Contacts:

General team contact	Sean Gabriel
Architecture and vision	Kenneth Wong
	Tony Wu
Customers	*Various; see requirements spec*

Terminology:

Please reference the functional design specification for an appropriate list of relevant terms and definitions.

Testing Limitations

Our major plan for testing will be to employ top-down and bottom-up testing concurrently, where they will eventually meet in the middle. The primary motivation for bottom-up testing is to ensure that the database middle layer is completely functional before plugged into the top-down scaffolding. We also intend to construct the GUI components thinly before filling in the implementation details. The limitations, therefore, may become apparent when we integrate the two testing formats in the middle (i.e. our application layer) and we will take extra precautions in this area to ensure the correctness of eFlash.

Testing Strategy

Given that the eFlash system architecture is divided into three main layers, our testing will primarily focus on

- Event-driven Graphical User Interface (GUI) layer
- Object-oriented Application layer
- Database Access layer

Unit Testing

Unit Testing Approach & Goal

Our unit tests aim to ensure functionality of individual components of each layer while we can still maintain isolation among different components and layers.
The driver functions will be the static main() method within each class(component) which help with the isolation of components and can be reused during each testing phase. The stub functions will provide simulated input or accept output from/to the driver functions whenever the method of an external component is invoked. Since the unit tests will be perform repeatedly, test cases of each phase will have to be compatible and reusable to ensure seamless integration tests.

Our goal in unit testing is to provide as bug-free code as possible before integration. With this goal in mind, we will simulate the function of each component and attempt to obtain optimum code coverage.

Scope

- **GUI Layer**

The unit testing of GUI components will involve verifying their usability and application connections. We will create a stub function for each GUI component that will ensure proper event handling of that component. Specifically, triggering each check box, tab, button and slider will lead to the correct event handling. To make sure a seamless connection between application layer and GUI layer, we will also verify the data and information presented by the GUI component is consistent and properly retrieved from the application layer.

- **Application Layer**

The unit testing of Application components can be initiated by invoking their static main() methods. The main methods are then responsible for testing the functionality of components and verifying the results. We will also supply simulated input to make sure the particular component is able to handle invalid user input gracefully. The components being tested in this layer consists of four manager modules, which deal with profiles, decks, flashcards, and collections, two modules that deal with file I/O, one to supply the logic to run quizzes, and one to handle the application's networking needs.

- Profile Manager
  - Testing of this module involves making sure the proper functionality and invalid input handling of managing user profiles, which includes profile management (creation, deletion, etc.), login authentication, and loading profile information.

createProfile()/deleteProfile()

1. Create a user profile with user name that already exists
2. Delete a user profile with user name that doesn't exist
3. Loop to create and delete 1000 profiles with randomly generated user names
4. Loop to delete all profiles.

login()

1. Login with user name that already exists
2. Login with user name that doesn't exist
3. Login with valid user name but invalid password
4. Login with valid user name and password

loadProfile()

1. Load a profile that doesn't exist
2. Loop to load all existing profiles and verify the results
3. Loop in conjunction with createProfile(), deleteProfile() (i.e create profile A, load profile A, delete profile A)

- Database Access Layer

Verify deleting a profile also removes all associated cards, decks, collections, quizzes and statistics from the corresponding tables.

- Deck Manager
  - Testing of this module involves making sure the proper functionality and invalid input handling of managing flashcard decks, which includes creating and deleting decks in the database, retrieving a list of decks given a category ID, fetching a list of flashcards belonging to a given deck, etc.

showDecks()/showCards()

1. Request to show decks given an invalid category
2. Verify Decks are displayed in an expandable hierarchal manner and sorted by Category, then Title
3. Verify descriptions of selected decks and flashcards are retrieved and displayed properly.
4. Request to show the list of flashcards belonging to a deck that doesn't exist
5. Loop to show all existing decks and the associated flashcards.

createDeck()/deleteDeck()

1. Create a deck with title that already exists
2. Delete a deck with title that doesn't exist
3. Loop to create and delete 1000 decks with randomly generated titles

- Database Access Layer

Verify deleting a deck also removes all association with the belonging flashcards and all collection to which this deck is belonged from the database. If a flashcard has no associated deck after deck deletion, it is removed from the database.

- Flashcard Manager
  - Testing of this module involves making sure the proper functionality and invalid input handling of managing various types of flashcards, which includes searching for flashcards given a search string, creating/editing/deleting flashcards, fetching the contents of a flashcard given its ID number, etc.

searchCard()

1. Verify the returned results match expected results
2. Verify Boolean operators(and, or, not) are working as desired
3. Verify invalid search strings are handled properly.

createCard()/editCard()/deleteCard()

1. Create a card with title that already exists
2. Delete a card with title that doesn't exist
3. Loop to create card A, insert a string into A, delete card A

Vocabulary flashcards

1. Create a flashcard with empty word/definition
2. Create a flashcard with word already exists

Visual/Music flashcards

1. Enter or select an invalid/empty file name
2. Create a flashcard with image/mp3 already exists
3. Verify previewing of mp3/image works

- Database Access Layer

Verify deleting a card also removes the association with the deck that the card belongs to from the database. If there is no card associated with a deck after deletion, the deck (also the possible association with some collection) has to be removed from the database.

- Collection Manager
  - Testing of this module involves making sure the proper functionality and invalid input handling of managing collections, which includes fetching a list of flashcards belonging to a specific collection, adding/removing decks to/from a collection, creating/editing/deleting collections, etc.

addDeck()/removeDeck()

1. Add same deck to a collection
2. Remove all decks from a collection

createCollection()/editCollection ()/deleteCollection()

1. Create a collection with title that already exists
2. Delete a collection with title that doesn't exist
3. Loop to create collection C, add Deck D, delete collection C

- Database Access Layer

Verify removing a deck from a collection delete the association from the database. If there is no deck associated with a collection, the collection is also removed.

- Quizzer
  - Testing of this module involves making sure the proper functionality of conducting quizzes. This includes picking the next flashcard to display, loading/updating current user statistics, etc.

showNextCard()

1. Check against array index out of bound
2. Verify flashcards are displayed properly(i.e. objects are properly aligned)

verifyAnswer()/updateStat()/loadStat()

1. Verify user answers and retrieve expected answer from database correctly
2. For each correct or wrong answer, verify we update the statistics accordingly.
3. Verify statistics retrieved from database show accurate data

  - Database Access Layer

- File Importer
  - Testing of this module involves making sure the proper functionality of importing files. This includes reading in data stored in text files and converting into eFlash format, which then stores as application layer data structure.

readFile()

1. Open a file that doesn't exist
2. Read in a file that contains invalid characters/format

convertData()

1. Check again array index out of bound
2. Import file to a non-existing deck

- File Exporter

- Testing of this module involves making sure the proper functionality of exporting files. This involves outputting flashcard and deck information in the database to various file formats.

writeFile()

1. Open/Write to a file that is currently opened by other application
2. Check consistency of exported files
3. Verify exported files can be imported correctly

exportPowerPoint()/exportEflash()

1. Verify the PowerPoint document has correct object placements and presentations
2. Verify we make correct use of the Microsoft PowerPoint API
3. Verify the exported eFlash documents complies with eFlash format and standard

- Networking
  - Testing of this module involves making sure the proper functionality of networking services. This includes server login creation and authentication, retrieving lists of decks and flashcards in the server repository, uploading and downloading decks to and from the online repository, etc.

login()

1. Create a network user name that already exists
2. Ensure user password is properly encoded before sending to server
3. Verify connection to the remote server is properly established

getDeck()

1. Loop to get the same deck from remote server 100 times. (Should not be overwritten)
2. Try to get other's non-shared decks

uploadDeck()

1. Upload an empty deck
2. Verify when overwriting a deck, network deck database gets updated
3. Verify when upload fails (due to connection loss), entire transaction gets dropped

- **Database Access Layer**

In our system architecture, components in the application layer depend heavily on the database access layer to gain access to the database. In order to reduce bugs in this critical layer, our database access layer is assumed to be generic and simple. In this way, codes regarding to complex logic decisions or alike can be put in each calling component and thereby reducing the complexity of the database access layer. There need not be a formal unit test for this layer. Unit testing of this layer depends on the calling component and it is the calling component's responsibility to provide correct input and retrieve the corresponding output. (Please refer to the Database Access Layer section for each component below.)

Creation Schedule & Execution Schedule

Our Unit Tests will follow Extreme Programming testing strategy which means that the coding of each component should proceed only after unit test for that component is created and executed. Therefore, we will execute unit tests with integration and regression repeatedly when a new component is added or modified throughout the project development.

Integration Testing

Integration Testing Strategy & Goal

In order to focus on components and interfaces interaction rather than the correct functionality of individual components, the integration test can proceed only after the components being tested passes all unit tests. Our integration testing strategy is bottom-up so that smaller and light-weighted components are thoroughly tested first and if any individual component fail during integration, it will be documented and a bug test will be added to the regression tests. Since our application is heavily dependent on the database access layer, starting with this layer ensure that any database accessibility or interfacing problems can be detected and fixed at an early stage. The smaller components on the application layer are integrated into bigger modules in a bottom-up manner and eventually a full application layer is properly tested and formed. Finally, the fully integrated application layer module will be linked to the GUI layer, which will then be passed on to system level testing.

Our goal of integration tests is to build larger and complex modules from the smaller unit components that define the application layer and database access layer. By integrating from a bottom-up manner, complex modules can be easily extensible and having wider functionality that complex operation or behavior can be tested on.

Scope

- Deck Manager / Flashcard Manager ➔ File Importer/ File Exporter

The interface between File Importer/ File Exporter and Deck Manager / Flashcard Manager is a call and return relationship. When an eFlash document is to be exported/ imported, Deck/ Flashcard Manager will call the Importer/Exporter to perform the requested operation. The Exporter/Importer will then read in or write out the eFlash document. For reading in a file, the importer will store the file into application layer data structure which will then be processed by the Deck/Flashcard Manager. Integration testing will involve making sure the Deck/Flashcard Manager is able to access the correct data structure storing the file being read in and the requested eFlash document can be successfully exported to a file.

- Deck Manager➔➔ Flashcard Manager ➔➔ Collection Manager

The interactions between Deck/Flashcard/Collection Managers are performed via the underlying database access layer. When an eFlash card is created in Flashcard Manager, the card is stored into the database via the database access layer while the Deck Manager retrieves the card also via the database access layer. Similarly, the Collection Manager creates a collection of decks by retrieving decks through the database access layer. Integration testing will focus on making sure the consistency and robustness of the database access layer between transactions.

- Viewer And Quizzer ➔ Collection Manager

The interface between Collection Manager and Viewer And Quizzer is a call and return relationship. Viewer And Quizzer retrieves the collection being quizzed from Collection Manager. Integration testing will focus on verifying the correct collection is chosen and accessed by Viewer And Quizzer.

- Viewer And Quizzer ➔ Profile Manager

The interface between Profile Manager and Viewer And Quizzer is also a call and return relationship. Quizzes scores and statistics are updated into the database via Profile Manager, who will invoke the corresponding method on the database access layer. Integration testing will mainly focus on verifying the scores and statistics for a particular user are correctly updated into the database.

- GUI ➔➔ Application Layer Modules

The scope of integration between GUI and the application layer modules is part of the unit testing of the entire GUI layer. The integration will involve making sure the corresponding function in the application layer module is correctly invoked. For example,

when a profile on the Profile Pop-up Screen is selected, we will have to make sure that profile is successfully reflected in Profile Manager; when a deck is edited on the Deck Edit Screen, the changes can be reflected to the Deck Manager.

Creation Schedule & Execution Schedule

Integration test will be executed continuously throughout our project development phases. After the unit test of all individual components passed, we will start to run integration test. Adhering to the Extreme Programming module, the integration tests will take place every week. At the later stage of project development, as larger modules are integrated, integration test become more important to the system level testing and will be executed nightly.

System Testing

System Testing Strategy

Due to our system architecture, namely model view controller, there is not a large amount of inter-module information passing. Most information is transported through successive read and writes to and from the database, so database consistency and unit and integration testing of the queries does a lot towards making the system work well when assembled. However, system verification is still incredibly important. We will use execution threads specified in this section to ensure the use-cases specified in previous documents are met. The threads will cover these eleven general sections:

1. User profile creation and management.
2. User options properly saved and loaded.
3. Deck creation saves all information and transitions fluidly from card to card.
4. Deck importation generates and saves fully populated card and deck entries.
5. Collection management activity is stored properly and operates quickly.
6. Deck exporting creates properly formatted output readable by external applications (e.g. PowerPoint for *.ppt exporting)
7. Card viewing does not corrupt storage and runs smoothly.
8. Quizzing and self-correcting is unobtrusive and correctly stores statistics.
9. Network connecting and deck viewing is properly displayed
10. Upload and download to the network results in consistent database views locally and on the server.
11. All screen and manager switching occurs with prompts if information is not saved, and properly saves state if save is selected. Also, if save is not selected, no information is saved.

Scope

The scope of our testing will follow the general categories stated above in the goal and fleshed out above that in the testing strategy. Specifically, the program components to be merged are as follows:

- Profile Manager
- Creation Manager
- Exporter
- Collection Manager
- Viewer
- Quizzer
- Networking Manager

System tests also cover external dependencies, and checking that these interactions are completed successfully. This means verifying the *.ppt exporter outputs correctly formatted slides that PowerPoint can open, the printer output is properly formatted, and the network transfers complete.

User A: Create a new profile name: "User A," password "cs169." (1,2)

Test 1: Create and View

1. Create a new deck of vocabulary cards in the creation menu. (3)
2. View the newly created deck in the viewer. (7)

1. Create a new deck of music cards in the creation menu, editing the length of each sound file. (3)
2. View the newly created deck in the viewer. (7)

Test 2: Create and Quiz

1. Create a new deck of vocabulary cards in the creation menu. (3)
2. Use the newly created deck for a matching/multiple choice quiz. (8)

1. Create a new deck of media cards in the creation menu. (3)
2. Use the newly created deck for a matching/multiple choice quiz. (8)

Test 3: Upload

1. From the network mode, connect to the server. (9)
2. Select cards of an existing flash card deck, "test1," and upload it to the server. (10)

Test 4: Export deck

1. Create a new deck of vocabulary cards. (3)
2. Save the deck to the local database. (6)

1. Create a new deck of music cards. (3)

2. Save the deck to the local database. (6)

1. Create a new deck of vocabulary cards. (3)
2. Replace duplicate cards in the deck on the local database with the newly created one. (6)

1. Create a new deck of music cards. (3)
2. Replace duplicate cards in the deck on the local database with the newly created one. (6)

Test 5: Open viewer, test navigation

1. Open cards of an existing vocabulary deck launching the viewer. (5,7)
2. Navigate through the entire deck. (7)
3. Verify acceptable performance in switching/flipping cards in a deck.

1. Open cards an existing music deck launching the viewer. (5,7)
2. Navigate through the entire deck. (7)
3. Verify acceptable performance in switching/flipping/playing music cards in a deck.

Test 6: Quiz existing deck

1. In Quiz mode, select cards of an existing vocabulary deck for multiple choice/matching quiz.(5,8)

2. In Quiz mode, select cards of an existing music deck for multiple choice/matching quiz (5,8)

Test 7: Deck deletion

1. In the deck manager, delete an existing deck from the local database.
2. Verify that the deck is deleted in the viewer. (5,7)

1. In the deck manager, attempt to delete cards of an existing deck from the server.
2. Verify that the deck was not deleted if the user did not upload it.
3. Verify that the deck was deleted if the user uploaded it. (9,10)

Test 8: Exiting program

1. Close the program in each mode. (11)
2. Verify successfully shut down. (11)
3. Verify any network connections are closed. (9,11)

Test 9: Saved sessions

1. Login after exiting program. (2)
2. Verify previous session's data was saved.

Test 10: Quiz profile consistency

1. Take the matching/multiple choice quiz multiple times. (8)
2. Verify the quiz results are accurate.

User B: Create a new profile name: "User B," password "cs169"

Test 1: download cards of "test1" deck file from network

1. From the network mode, connect to the server. (9,10)
2. Select cards of deck "test1" from the server and download it to the local database.
3. Open in the viewer. (7)
4. Verify the deck corresponds with the one uploaded by User A.

Test 2: Deck editing

1. In Creation mode, open an existing vocabulary/music deck and modify the contents (3)
2. Save the newly modified deck. (6)

Test 3: PPT export

1. In Viewer mode, open an existing vocabulary/music deck and export the file to a power point slide. (6)
2. Open the newly created power point file.

Test 4: Printing

1. In Viewer mode, open an existing vocabulary deck and pretty print the deck. (6)

Goal

The goal of system testing is to guarantee that the features listed in our Design Specification, which in turn originate with our customers' user-stories, are present and operational in the assembled program. Through unit and integration testing functional correctness and invariants should be verified, so system testing focuses on larger scale correctness and also program speed. As stated in our initial documents, user-instantaneous reaction times are necessary for this product. Specifically, this entails testing the eleven components listed in the testing strategy section. The system test alone tests dependencies external to our program such as file system interaction, printing, and

networking. Particular attention to these aspects is necessary, in system testing, as they are not verified in the other sections.

Creation Schedule

Because most information will be sent to and retrieved from the database on the fly, system testing is less critical than unit and integration testing. System interconnections when dealing heavily with a central repository should be correct if database consistency is maintained. Therefore, system testing routines will be fully developed mid-way through the development cycle after unit and integration tests are developed, following a bottom-up testing approach. This approach also allows time to solidify all options, functional buttons, and menu items, in turn allowing development of specific module switching and option combination testing. Also, specific system tests for any additional use-cases can be added later.

Execution Schedule

System testing will begin once all modules involved in the specific system test have passed unit and integration tests. System tests should be repeated for all subsequent debugging fixes and other code changes to the program.

Regression Testing

Regression Testing Strategy & Goals

Regression tests fall into two categories. The first category is the basic functionality tests, which is in response to user scenarios. The second category is Bug tests, which is in response to bugs found during system, integration, and unit testing. Both categories of tests will be permanently added to the unit test suite because they are written and run with all subsequent unit tests to ensure that changes to the product do not cause the behavior of the product to regress and a particular defect does not reappear.

Functionality tests are written at the user level, which typically test command level functionality for both the actions of malicious and behaved users. Tests should check normal operating inputs for the program, as well as unusual, invalid, and boundary conditions. Functionality tests can be developed by defining correct input through user scenarios and testing on unusual and boundary conditions.

Writing bug tests depends on detailed documentation of all bugs that are discovered throughout all states in the testing process. Detailed documentation of bugs should include where and what causes the defect in our system. After the bug is fixed and passes the test, the test cases are stored in the test suite to ensure that the same bug does not occur again in response to some other change in our product.

The overall goal of regression testing is to make sure the system continues to conform to the Design Document. Regression testing ensures that current system does not regress (i.e. previously fixed bugs do not reappear or cause new bugs).

Scope

Regression testing is designed to prevent old bugs from recurring in the main functionalities of the program. Because integration and system testing will continue independently during this time, regression testing will focus on finding basic problems that may occur as a result of adding new features into the application. The basic functionalities covered will include:

1. Creating text/sound/picture Flash cards either through manual user input or through importation of text files.
2. Editing/deleting Flash cards
3. Viewing these Cards as a PowerPoint presentation or in the eFlash viewer.
4. Take multiple-choice, matching and other quizzes based on the Flash cards written.
5. Upload/Download decks of Flash cards from the Network.

Creation Schedule

Because of the nature of regression testing, we will not enforce a strict schedule for the creation of regression tests. In general, the basic functionality tests that are built in response to user scenarios should be written and run as soon as the proper components are ready. Any changes to the product will require the rerunning of these tests to ensure a bug-free user experience.

The second category of regression tests, however, will be created when bugs are found and fixed. Please reference the implementation timeline for an estimate of the time period in which regression tests will be created most frequently. Any situation involving new fixes will require rerunning these regression tests to ensure that old bugs do not creep back into the program.

Execution Schedule

Regression tests should be run at all times on the components which have been written. Any time a new component is written, the functionalities of previously written components should be tested again as well. Execution will begin once the first bugs are found and fixed, and periodically, we will also run our existing battery tests on a weekly basis to ensure the steady decrease of the bug curve.

Test Report Requirements

To ensure the quality of eFlash meets and/or exceeds the expectations of our customers, we intend to compile a test report to be reviewed for further action before eFlash enters product release system-integration testing. This report will contain the following sections:

1. Analysis and recommendations to improve
2. Graphs or other visualizations for:
    a. Bug curve of items found/triaged/fixed
    b. Estimated code coverage of current test cases
    c. Progress through testing timeline
3. A comprehensive list of the test cases run, each with:
    a. Name and brief description
    b. Reference to relevant use case or activity diagram
4. A comprehensive list of bugs found, each with:
    a. History (first occurrence, fix date, etc.)
    b. Severity and triage decision
    c. Reference to relevant test case

Resource and Time estimates

Unit Testing Estimates

Unit tests will be created by April 1st, before any code is written. We will set aside three days for creation of tests. After the completion of a unit of code, the corresponding unit test will be executed. Every unit test should take a maximum of one days for execution. To plan for the correction of difficult bugs, we will set a side a total of three days. Thus total estimated unit testing time should be around 1 week.

Integration Testing Estimates

Integration tests will most likely take an extended amount of time due to the complexity of interacting systems. Integration tests will be created by April 10th. Writing integration tests should take a maximum of three days to code. Integration testing will begin as soon as two connected units have finished unit testing. Because all units may finish at different times, and time to test will grow exponentially as more units are added, we will set aside two and a half weeks for integration test executing and debugging. Thus total estimated integration testing time should be about three weeks.

Regression Testing Estimates

Regression tests are mainly defensive in nature, and as such will be run continually throughout every phase of testing. The initial battery of tests should be completed by April 10th as well, and new tests will be added to this set as bugs are fixed. Therefore, we do not expect to begin regression testing until some units have been completed (see above), but once started, we will run regression tests for approximately three weeks concurrent to the other phases as well.

Issues

Open Issues

-Fix up old documents to reflect new design decisions

Resolved Since Last Revision

None

Checklist

The following checklist is provided to help the reviewers (and author) prepare for the review by providing a set of questions for the reviewer to answer about the specification document. If the answer to any question is "no", that item should be identified as an issue at the review. The checklist is only a guideline; it should not be solely relied upon for a complete review. Reviewers may want to add their own questions to the checklist.

Y	N	CONTENT
___	___	Has the Implementation Specification been reviewed and accepted by the appropriate reviewers? (This is a prerequisite).
___	___	Will the goal for each type of testing be met with the testing that is described?
___	___	Are the testing activities scheduled at the appropriate times?
___	___	Are you satisfied with all parts of the document?
___	___	Do you believe all parts are possible to implement?
___	___	Are all parts of the document in agreement with the produced requirements?
___	___	Is each part of the document in agreement with all other parts?
		COMPLETENESS
___	___	If the regression test strategy is not covered, is it covered in the Product Quality Plan?
___	___	If the resource and time estimates for testing activities are not included, are they part of the Product Plan?
___	___	Do the testing activities which are described meet the product quality criteria described in the Product Quality Plan?
___	___	Where information isn't available before review, are the areas of incompleteness specified?
___	___	Are all sections from the document template included?
___	___	Are you willing to accept the specification with the items in the open issues section unresolved?
		CLARITY
___	___	Is the test strategy for each type of testing clearly described?
___	___	Are the test report requirements clearly described?
___	___	Are all items clear and not ambiguous? (Minor document readability issues should be handled off-line, not in the review, e.g. spelling, grammar, and organization).