Android dotCal

Project Plan

Version History

Version Number	Authors	Description	Date
Version 1	Team Orange	The first version	12/1

Review History

Reviewed By	Version	Comments	Date
Kelsey	0	Did not exist yet	11/23

Approvals

Approved By	Version	Signature	Date
Kelsey	0	not worthy of signature	11/23

1 Overview

2 Deliverables

Installable Android Package (APK) along with a 1.0 tagged source tree in the source control repository.

3 Assumptions, Constraints and References

The team assumes that:

- The application will run on the "T-Mobile G1" Android-platform phone
- Any updates published by Google to
  - Android SDK,
  - Android Operating System,

  - Android Calendar API,
  - Google Calendar Web API, or
  - Android Marketplace

  during the application's development will be inconsequential to the schedule and system architecture

- neutralSpace will use Eclipse to build the application, or will nonetheless find satisfactory the team providing Eclipse Project and Ant build scripts

4 Development Process

1. **Establish customer needs.** *Everyone* will meet with neutralSpace (nS) and determine overall purpose of application and expected functionality. (Complete)
2. **Write needs down in the form of a Requirements Document.** *Kelsey* and *Adam* will develop a concise list of detailed program requirements. *Sky* will create a Visio document reflecting these needs with wireframes and mockups of the application screens, along with the program flow. (Complete)
3. **Customer approval of needs.** *Everyone* will present the Requirements/Visio document to nS for review and incorporate any corrections they have into the document. (Complete)
4. **Create system architecture.** *Armando*, *Sky*, and *Adam* will meet together and split the requirements into concrete components/modules that can be developed separately by multiple team members at once. *Everyone except Nathan* has secondary roles as developers, so these members will be potential assignees for implementing these modules. (By January 1, 2009)
5. **Code the components.** Members with roles as developers will be assigned a component, with the larger and more complex components (e.g. Calendar syncing) being implemented by *Armando*. (Ongoing before February 21, 2009)
6. **Test each component.** Each component will have unit tests that will be developed by the component author. (Ongoing before February 21, 2009)
7. **Integrate components.** As individual components are completed, they can interact with other completed components eventually forming the overall application. (Ongoing before February 21, 2009)
8. **Test integrated system.** All components should be working together as expected, at this point QA can do a final sweep of functional tests and system tests allowing a

week of final bug fixing and polishing before the initial release. (February 21, 2008)

5 PERT Chart

6 Schedule

7 Calendar

8 Meetings and Reviews

9 Resource Identification

Summary of resources identified:

- T-Mobile G1 Phone Device
- Tools Provided by neutralSpace
- Development Computers
- Development Tools
- Team Members

9.1 T-Mobile G1 Phone Device

This unit is provided by neutralSpace on an as-needed basis for testing the application outside of the software phone emulator during development. While the team will use the emulator for most testing, it is helpful to occasionally run the application on the actual phone as a "sanity check" – that is, to reaffirm that the application does indeed work properly on the actual target device.

9.2 Tools Provided by neutralSpace

The project management system, called Redmine, is provided and administered by neutralSpace. The team uses this software to

- Keep track of issues: Tasks to be done, features to add, bugs to fix, weekly goals to meet

- Host a central store of frequently evolving team knowledge, in its Wiki component: Meeting agendas, tutorials
- View source code or document PDFs through the Subversion repository web interface provided by Redmine when viewing them directly from a source check out is inconvenient
- Allow neutralSpace to see our work and progress as it evolves, by working in the open

9.3 Development Computers

Portland State University's Capstone Lab has six machines which we can use for development. These machines will be set up by the team's Lead Computer Nerd. Each team member gets one machine which they can use in person or remotely as they see fit, for the application's development or related tasks. These systems will have an automated mutual-backup mechanism to keep data on them relatively safe.

Team members may also use personal machines for the project, with each member responsible for keeping their work on their personal machine backed up. Members may contact a Computer Nerd for help on configuring a backup system for their personal systems if desired.

9.4 Development Tools

The software tools to be used are almost all open-source software. These components are readily available on the Internet and will be installed on all of the team's Capstone Lab machines by the Lead Computer Nerd. Open-source components used include

- Eclipse IDE with Android Plugin
- Android SDK and emulator
- Git and Subversion source code control systems

Non-free applications used include Microsoft Visio for creation of diagrams, such as screen mockups. Members must each obtain licenses to use Microsoft applications through Portland States MSDNaa (Academic Alliance) program.

9.5 Team Members

Adam, Armando, Kelsey, Nathan, Peter, and Sky each lead one facet of the team and project (see Roles section, below).

Each member is responsible for delegating tasks for the area they lead. For example, Adam is the Lead Computer Nerd, so if he needs help with system administration or other computer nerd tasks, he can ask for help from the backup Computer Nerds, Peter and Armando.

10 Configuration Management

Source Code, Build, and Documentation are managed parts of the application's configuration.

10.1 Source Code Management

neutralSpace's Subversion repository is considered the official source of the latest code, or the "Primary Source Repository." Only the "trunk" branch is to be used in the Primary Source Repository; no new branches are to be created. Creating temporary local branches on development machines using Git is recommended when adding features.

Initially, each developer will clone the Primary Source Repository using "git svn clone," or check out said repository using "svn checkout," at the developer's preference. When creating new code or modifying existing code, developers will follow a procedure similar to this:

- Before writing code, sync to the Primary Source Repository
- Create and check out a new git branch for the feature about to be written
- Write new code with corresponding comments and JavaDoc-syntax comments
- Document expected preconditions and post conditions using debugging assertions in every function; explain non-obvious assertions with a short comment

Before checking a new feature into the Primary Source Repository, developers will follow this procedure:

- Make sure the new code builds without error or warning
- Test the new code in the emulator
- Write or modify a unit test for the new code if possible
- Verify that the new code passes the new unit test
- Verify the new unit test's falsifiability with a bad test case

- Verify that the application as a whole still passes all current unit tests (regression testing)
- Commit to the local branch created above
- Sync local master branch to Primary Source Repository again
- Merge new local branch to local master branch
- Push new commit upstream to the Primary Source Repository

Developers should not generally commit to the Primary Source Repository "work in progress" (WIP) code, such as code that doesn't build or causes the application to crash; developers can instead make several WIP commits to their working local git branches (without regard to commit message content or format) while making changes, then eventually merge the local commits into one bigger commit with a clear commit message (see Writing Commit Messages).

Developers are encouraged to

- Create small, self-contained commits
- Create cohesive commits, in general, which change code related to a single feature or bug
- Put non-cohesive changes which are not related to a specific feature or bug in their own "maintenance" commits
- Push upstream to the Primary Source Repository on a frequent basis, in order to ensure continual code integration and to keep the rest of the team up-to-date

10.1.1 Writing Commit Messages

Typically, commit messages should be in this format:

> *Summary line*
>
> *Optional longer description paragraph(s)*

Ideal commit messages:

- Are concise, clear, generally in active voice starting with a verb (e.g., "Changed X to do Y" rather than "X has been changed to be able to do Y in this revision"), though other forms can be used as deemed appropriate
- Contain relevant Redmine issue numbers
- Describe test cases added or modified, as well as code

- Are free of obvious typos and errors, such as missing or repeated words, and misspellings

10.2 Build Management

Ant targets are used to build the application, unit tests, and documentation. When adding new files, developers update the appropriate ant targets. When changing the Eclipse project, Ant targets, or other build scripts, developers include a short description of the change in their commit message.

10.3 Documentation Management

Documentation creation and modification should be done similarly to code creation and modification as outlined above (see Source Code Management). LaTeX is used for documentation in part because it allows multiple contributors to merge changes more easily than with binary document formats.

For ease of viewing the current versions of LaTeX documents, the compiled (PDF) versions of each document should be included in each Primary Source Repository commit changing a document.

10.4 Test Management

As outlined above (see Source Code Management), developers run all unit tests and any newly created unit tests before creating each commit; developers regularly (with most source code commits) create and verify the validity of new unit test cases.

11 Roles

Role	**Lead**	**Backup**
Project Manager	Nate Ertner	None
Computer Nerd	Adam DiCarlo	Armando/Peter
Lead Architect	Sky Cunningham	Adam/Kelsey/Nate
Lead Developer	Armando Diaz-Jagucki	Kelsey/Peter/Adam/Sky
User Needs Analyst	Kelsey Cairns	Adam/Sky
Quality Assurance	Peter Welte	Armando/Sky

12 Risk Management

Infrastructure- and personnel-related risks are identified below.

12.1 Infrastructure Risks

Risk:	Phone's GPS unit is inaccurate wherever it is used
Consequence:	Recorded calendar location data is meaningless; application may be considered useless by neutralSpace
Mitigation:	None. The phone's GPS unit cannot be changed. neutralSpace has agreed that we are not responsible for the phone's GPS accuracy

Risk:	Redmine (project management software) data loss
Consequence:	Delay, possible information loss
Mitigation:	Verify (now) that neutralSpace has a frequent automated backup system in place for Redmine. Ask neutralSpace (now) to provide us a method to create a backup of the Redmine Wiki. In the event of data loss, wait for neutralSpace to restore the system from backup.

Risk:	Total Redmine data loss with no neutralSpace backup source
Consequence:	Major delay and information loss
Mitigation:	The team must wait for neutralSpace to create a new Redmine instance, and then reconstruct issues in the issue tracker from Redmine notification emails. The team must reconstruct the Wiki from a team member's most recent backup if one is available.

Risk:	Phone loss or destruction
Consequence:	Inability to test on actual hardware
Mitigation:	Test using only emulator until neutralSpace has replaced the phone

Risk:	Subversion source code repository data loss
Consequence:	Delay
Mitigation:	Verify (now) that neutralSpace has a frequent automated backup system in place for the repository. Restore from backup if available, or determine who on the team has the latest repository check out and create a new repository using their check out

Risk:	Development computer data loss
Consequence:	Delay
Mitigation:	Configure development computers to be automatically backed up using a mutual backup system; in the event of data loss on one machine, its data can be restored from one of the other machines

Risk:	Android Calendar API inexistant
Consequence:	If mitigation possible, delay; if not, possible failure of project
Mitigation:	1. Contact Google Android developers and persuade them to write/finish the API (in a timely manner); or 2. Write our own Google Calendar web-API interface code and have the application put events into the web calendar (this will cause a more complicated implementation of the application in addition to the extra work of writing the web API)

12.2 Personnel Risks

Risk:	Member unable to work due to leaving team, refusing, being ill, or other circumstances
Consequence:	Other members must take on responsibility of lost member
Mitigation:	Assign lost member's primary responsibility to pre-determined understudy or understudies (see Roles section, above)

Risk:	Personal conflict among members
Consequence:	Loss of focus of members or group, delay
Mitigation:	Attempt to resolve conflict with "Radical Honesty" communication techniques, and with another group member present as mediator if necessary; if no member-mediator can be agreed upon, ask the Capstone Coordinator to be the mediator

13 Quality Assurance

For the Android dotCal project, QA activities will be integrated into all aspects of the project development process.

13.1 Preconditions and Assertions

- Refine requirements document whenever preconditions are not already determined
- Create validation functions for use by preconditions and assertions, as needed
- Write preconditions and assertions in code

13.2 Buddy Review

- Any code changes made on the main source and build components will be published to all developers
- The goal is to allow multiple developers to view changes, so developers know what is changing, can catch bugs, and are encouraged to write readable code.

13.3 Review Meetings

- Assign buddy reviewers
- Select an at-risk section of code for weekly review meetings. Issues to look for include security,
- efficiency, scalability, operability (install, upgrade, etc), and maintainability.
- Each two weeks, identify reviewers and schedule review meetings
- Reviewers study the material individually for 1 hour
- Reviewers meet to inspect the material for 1 hour
- Place review meeting notes in the wiki and track any issues identified in review meetings

13.4 Unit Tests

- Set up infrastructure for easy execution of JUnit tests (this is just an Ant target)
- Create unit tests for each class when the class is created
- Execute unit tests before each commit. All tests must pass before developer can commit, otherwise open new issue(s) for failed tests. These "smoke tests" will be executed in each developer's normal development environment.
- Execute unit tests completely on each release candidate to check for regressions. These regression tests will be executed on a dedicated QA machine.
- Update unit tests whenever requirements change

13.5 System Tests

- Design and specify a detailed automated test suite.
- Review the system test suite to make sure that every UI screen and element is covered
- Execute system tests completely on each release candidate. These system tests will be carried out on a dedicated QA machine.
- Update system tests whenever requirements change

13.6 Regression Testing

- We will adopt a policy of frequently re-running all automated tests, including those that have previously been successful. This will help catch regressions (bugs that we thought were fixed, but that appear again).
- Automated tests will be run as often as possible, preferably on check-in of new code.
- Failures will be reported via email or web.

13.7 Build Testing

- On check in of any code or build system changes, the build system will be run.
- Failures will be reported via email or web.

13.8 Beta Testing

- Involve beta testers early in the development process.
- Beta testing should focus on functionality, usability, and operability
- Issues identified during Beta testing will be reported in the issue tracker
- Do beta testing on each major release candidate or when major new milestones are reached

13.9 Stress Testing

- Tools such as Android Monkey will be used to test that the system doesn't crash given random input.
- Other tools will be developed to ensure the system can handle high volumes and high frequencies of events

13.10 QA Management

- Update this test plan whenever requirements change
- Document test results and communicate them to the entire development team
- Estimate remaining (not yet detected) defects based on current issue tracking data, defect rates, and metrics on code size and the impact of changes.

- Keep all issues up-to-date in an issue tracking database. The issue tracker is available to all project members here. The meaning of issue states, priorities, and other attributes are defined in the
- SDM (software development methodology, or glossary)

14 Deployment