Table of Contents

Table of Contents
Project Overview
  Domain Summary
  Domain Glossary
  Project Summary
  End Users
Basic Requirements
Constraints
Development Process
  Team Roles
  Process Definition
Project Schedule: Planned and Actual
  Schedule Development
  Planned Milestones
  Actual Milestones
System Design
  Architecture and Technologies
  Domain Design
  Functional Subsystems
  Refactoring Required
Process and Product Metrics
  Metric Choice
  Metric Results
Product State at Time of Delivery
Project Reflection
  The Less-Than-Good
  The Good

Project Overview

Domain Summary
360 degree surveys are used to gather feedback about individuals in an organization from various sources, including managers, peers, subordinates, customers, and the individuals
themselves. Similar to traditional performance evaluations, questions in 360 degree surveys measure an array of personal qualities, but 360 degree surveys are not intended for formal performance evaluation. Instead, survey results are provided to individuals explicitly for personal improvement: continuing to foster strengths and improving upon weaknesses. Additionally, 360 degree survey results are often aggregated and used to draw larger conclusions about, and subsequently improve, the organization as a whole.

Domain Glossary
Important terms about the software and 360 degree surveys used throughout this document are defined here.

<table>
<thead>
<tr>
<th>Scale</th>
<th>A personal quality that is measured by questions. An example scale is “Written Communication”.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Scale</td>
<td>A high level category for many scales. An example super scale is “Communication”.</td>
</tr>
<tr>
<td>Relationship</td>
<td>A term categorizing the general interactions between two people. An example relationship is “Manager”.</td>
</tr>
<tr>
<td>Relationship Category</td>
<td>A high level category for many relationships. An example relationship category is “Work”.</td>
</tr>
<tr>
<td>Likert Question</td>
<td>A specific type of multiple choice question where options are usually sequential (“disagree”, “neutral”, “agree”) and are often presented together to save space.</td>
</tr>
<tr>
<td>Likert Grid / Grid Page</td>
<td>A space-saving way to organize several likert questions that share the same set of answers, where answer labels form the table columns and different questions form the table rows.</td>
</tr>
</tbody>
</table>

Project Summary
Team Phalanx built a web application for 360 degree surveys. The product, Easy360, hits the four main feature categories of existing 360 degree survey applications: creating surveys, taking surveys, viewing feedback, and managing system data. Easy360 improves upon existing competitors (like Qualtrics) by providing extensive domain customization, offering an easy-to-use web interface that fits the needs of experts and novices alike, visualizing survey data in a more informative, interactive, and statistically useful format, and providing these features at a low cost.

End Users
Easy360 is designed to meet the needs of 6 types of 360 degree survey users:
- **Survey Administrator** - determines what personal qualities surveys should measure and who should be involved in the process.
- **Target** - rates self and uses feedback from others for personal improvement.
- **Rater** - has an existing relationship with a target, and provides feedback about that target.
- **Coach** - helps targets understand and use feedback data for self improvement.
- **Viewer** - analyzes aggregations of survey data for organization- or department-wide improvement.
- **Site Administrator** - creates users and assigns permissions.

**Basic Requirements**

Team Phalanx used user stories to write formal requirements. As mentioned above, Easy360 addresses the needs of 6 types of users. None of the users are required to have pre-existing technical knowledge beyond basic web application usage (like using radio buttons or drop-down select boxes). Site Administrators need to understand the permissions assigned with different user roles in order to effectively use the system. Below is a short summary of the delivered functionality, organized by end user. For a comprehensive discussion of features added during development, refer to the Product State section.

- **Survey Administrator** - primarily responsible for creating and managing surveys and defining data (questions, scales, relationships) used in the surveys. A survey administrator can:
  - **Create a survey**, which involves picking an organization-wide unique survey name, selecting existing questions to include in the survey, arranging questions onto pages (or grid pages for likert questions with identical labels), selecting targets for the survey (and potentially allowing targets to select their own raters), and selecting raters for each target, potentially inviting unregistered users to rate targets. Selecting targets and raters automatically provides those users with the appropriate functionality.
  - **Manage existing surveys**, which covers functionality to edit all items mentioned above, customize the text of all email alerts associated with a particular survey, copy a survey to a new survey, export survey results to CSV (one survey at a time), delete a survey, view progress of what users have started/completed an individual survey, send email reminders to those who haven’t completed a specific survey, and uninviting specific users from a particular survey.
  - **Customize organization-wide settings**, modifying how surveys and feedback behave, such as what statistical distribution to use for calculating interrater agreement or whether to automatically append “No Answer” options to all questions.
  - **Manage super scales, scales, relationship categories, relationships, and questions**, where “manage” refers to standard create, read, update, delete capabilities. Questions are more complex, with 3 possible question types (open ended, multiple choice, and likert, where likert are able to be placed on a grid page), options to hide from specific relationships, and possible help text.
- **Rater** - responsible for completing a survey about a target to provide useful feedback. A rater can:
  - Take a survey, which covers functionality to save a survey for later and resume it, view current progress (including number of questions, number of pages, and percent complete), and submit survey.

- **Target** - can view personal feedback from surveys where designated as a target. All targets are raters, and must rate themselves. Beyond rater functionality, targets can:
  - View current feedback, which represents all data for a single survey, grouped in 2 dimensions: relationship and scale. Targets can “drill down” through super scales to scales to questions to see more granular data, and can toggle which relationships are displayed. Targets can also toggle data “confidence” (intrarater agreement statistic), scale “expectations” (which are defined by the survey administrator as a benchmark for a specific skill), and peer comparison (which aggregates data from other targets, either throughout the company or a specific department, for this survey).
  - View open-ended feedback, which includes the answers to visible open-ended questions included in the survey. Responses are completely anonymous.
  - View historical feedback, which tracks common scale and super scale scores for all surveys over time.

- **Viewer** - can view aggregations of current feedback on the organization or department level only. Like a target, the viewer can interact with the current feedback. Unlike a target, the viewer cannot see the current feedback for an individual target.

- **Coach** - can view both aggregations and individual data for current feedback, combining both target and viewer functionality. Unlike the target, the coach cannot view historical data. The coach also can:
  - Export survey data in CSV.

- **Site Administrator** - can do all actions within the system, like a coach + survey administration. Additionally, a site administrator can:
  - View all users in an organization
  - Edit users in an organization, which includes name, department, role and permissions, and management level.
  - Delete users in an organization.

Additionally, all users can modify their own account data, including password, first name, last name, department, and management level. Users cannot change their email addresses, as email addresses are used to uniquely identify users throughout the system.

All system inputs are entered by the 6 types of end users through the web interface, almost exclusively utilizing Django's form functionality. With the exception of 3 features, all system output is presented on web pages. “Export survey results” prompts coach users to download a CSV file. “Print feedback” on the current feedback or historical feedback pages creates a specially formatted PDF meant for printing. Finally, email notifications are sent when surveys are published, when targets select their own raters (after a survey is published), and when survey administrators choose to send reminder emails to raters.
Constraints
Dr. Palanski, the project sponsor, had one formal constraint for the project: it must be web-based. However, Team Phalanx transformed several of Dr. Palanski’s other key concerns and motivations into technical and process constraints.
- Dr. Palanski wanted the end product to be affordable, so the team created a constraint to only use free or open-source technologies.
- Dr. Palanski wanted to have another team continue development after Team Phalanx graduated, so the team created a constraint to only use widely-used development technologies.
- Dr. Palanski wanted to export survey data that could be used in statistical programs, so the team created a constraint to export survey data in CSV format.

Additionally, the project had a multidisciplinary aspect for the first 3 months, collaborating with 3 students from the Saunders College of Business, so the team wanted to pick a low-ceremony process. The team decided that it was more important to start planning and building the product as early as possible, instead of investing time educating team members about a formal methodology.

Development Process
Neither the sponsor nor the coach had strong preferences about the team’s process. When defining a process, the team considered the team size, multidisciplinary nature, and sponsor interaction as the most important factors to consider. Having only 3 members, the team could operate effectively with minimal ceremony. Working with non-technical students, the team shouldn’t have picked something that took a significant time to learn. Having a sponsor who was eager to see and use the product, the team wanted to put working code in his hands as early and often as possible. Finally, although the sponsor had used several competing products, the project scope was a bit fuzzy, and the team wanted to be able to respond to change. Therefore, the team selected a standard iterative and incremental process.

Team Roles
During the first week of the project, before even discussing any potential process methodologies, Team Phalanx described 11 different team roles and completed a survey about roles, selecting up to 4 desired roles and 1 undesired role. Based on the survey feedback, and after considering individual team members’ experiences, the following roles were established and included in the Project Plan:

<table>
<thead>
<tr>
<th>Role Name</th>
<th>Role Description</th>
<th>Team Member</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer Liaison</td>
<td>Primary point of contact with the sponsor, coach, and business team. Leads discussion on behalf of the SE team in sponsor</td>
<td>Phil</td>
</tr>
</tbody>
</table>
meets. Responsible for clarifying requirement and domain ambiguity.

<table>
<thead>
<tr>
<th>Role</th>
<th>Responsibilities</th>
<th>Responsible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Manager</td>
<td>Responsible for organizing and allocating tasks, recording progress, and enforcing process. Ensures that the team is meeting assigned deliverables.</td>
<td>Chris</td>
</tr>
<tr>
<td>Website Coordinator</td>
<td>Responsible for keeping the project website up to date with team artifacts and ensuring that new artifacts are added as necessary.</td>
<td>Henryk</td>
</tr>
<tr>
<td>System Administrator</td>
<td>Responsible for setting up the VM and developer tools and ensuring that all tools are working together properly.</td>
<td>Chris</td>
</tr>
<tr>
<td>Project Documentation Lead</td>
<td>Responsible for organizing the primary project documents (requirements and design), assigning sections to team members, and editing the final documents.</td>
<td>Henryk</td>
</tr>
<tr>
<td>Process Documentation Lead</td>
<td>Responsible for organizing the primary process documents (plan, effort tracking, metrics, risk), assigning sections to team members, and editing the final documents. Also responsible for keeping track of process metrics.</td>
<td>Phil</td>
</tr>
<tr>
<td>System Test Lead</td>
<td>Responsible for writing the black box acceptance test plan and tracking issues that arise from all acceptance test runs.</td>
<td>Chris</td>
</tr>
<tr>
<td>Back End Development Lead</td>
<td>Responsible for designing database models and dealing with app / database / server interaction.</td>
<td>Phil</td>
</tr>
<tr>
<td>App Development Lead</td>
<td>Responsible for coding business logic, implementing design patterns, and creating a usable platform for the front end.</td>
<td>Henryk</td>
</tr>
<tr>
<td>Front End Development Lead</td>
<td>Responsible for designing page layout and flow, as well as client-side coding.</td>
<td>Chris</td>
</tr>
<tr>
<td>Graphic Design Lead</td>
<td>Responsible for creating and designing assets (like images), picking color schemes, and tweaking interface layouts based on feedback.</td>
<td>Chris</td>
</tr>
</tbody>
</table>

The Project Plan made an explicit note that, “roles do not imply that the individual is exclusively responsible for all tasks. Instead, the assigned individual is in charge of making sure everything is done on time and to quality, overseeing all work in that sector, and acting as the final say when team debates arise.”
Ultimately, because the team contained only 3 members, several roles began to merge. Customer Liaison and Project Manager became a more generic lead role, while Website Coordinator fell under the responsibility of System Administrator. The project utilizes Django, a web application framework, so Back End Development and App Development became a single role. Documentation ended up being a team task split by deliverable, where individual members were primarily responsible for specific documents.

Process Definition

After outlining key process concerns and defining team roles, the team settled on an iterative incremental process, with the following activities:

- **Requirements Engineering** - the initial project definition, discussions with the sponsor, and research into competing products provided enough context to prepare an initial feature list, which was eventually turned into formal user stories.
- **Design** - driven by the functional requirements, the team broke design into 2 segments: domain (in an ad-hoc notation that represented both data relationships, like E-R diagrams, and basic class structure, like UML class diagrams), and user interface (described more in the project schedule section).
- **Development** - features were separated into 4 functional releases based on feature priority (for example, exporting data is more important than printing) and logic (for example, surveys must be created before taken and taken before viewed). The releases were scheduled for weeks 8 and 10 of winter and weeks 3 and 5 of spring.
- **Testing** - the team focused on acceptance testing, usability testing, and integration testing. Although not originally planned, the team also conducted security testing, using a matrix of user roles and application functionality (mostly by URL).
- **Communication** - the team put strong emphasis on face-to-face communication, with weekly SE-only team meetings on Tuesdays and sponsor/coach/business team meetings on Thursdays. Email was the secondary mode of communication. The team primarily used Google Docs to handle document deliverables, metric tracking, and other coordination activities.
- **Deployment / Maintenance** - the team had 3 levels of deployment: local development environments, a staging environment for integration testing, and several production deployments for important releases. Starting at release 3, each functional release updated a “demo” deployment that was available to the sponsor. Several other deployments (usability testing, poster day, ImagineRIT) were also used.

Project Schedule: Planned and Actual

Schedule Development

Team Phalanx created a schedule in a top-down approach, starting with two categories: product and project. Product was broken into five categories: infrastructure, coding, testing, UI, and other. Project was broken into three: documentation (further broken into requirements, project plan, technical report, and other), website, and other.
Next, the team used the deliverable schedule from the Software Engineering department and added required and recommended deliverables on the schedule, specifically required documents, presentations, and peer evaluations. Then, the team estimated dates for infrastructure, UI mockups, and coding, where coding blocks resulted in four releases, as determined by the iterative incremental process.

**Planned Milestones**
The full schedule is available here: [https://docs.google.com/a/q.rit.edu/spreadsheet/ccc?key=0AvfArRv9hM6xdHM1Wk9TYkVVY1JVVGVGTVJXdXVOVWE#gid=0](https://docs.google.com/a/q.rit.edu/spreadsheet/ccc?key=0AvfArRv9hM6xdHM1Wk9TYkVVY1JVVGVGTVJXdXVOVWE#gid=0)

A few key dates are listed below:

<table>
<thead>
<tr>
<th>Category</th>
<th>Deliverable Description</th>
<th>Due Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td>Final version of project plan</td>
<td>Week 4 winter</td>
</tr>
<tr>
<td>Documentation</td>
<td>“Final” version of requirements</td>
<td>Week 5 winter</td>
</tr>
<tr>
<td>Website</td>
<td>Set up (documents hosted as completed)</td>
<td>Week 3 winter</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>VM tools set up</td>
<td>Week 4 winter</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>Development machines configured</td>
<td>Winter break</td>
</tr>
<tr>
<td>Design</td>
<td>Full domain design</td>
<td>Week 4 winter</td>
</tr>
<tr>
<td>Coding</td>
<td>Release 1</td>
<td>Week 8 winter</td>
</tr>
<tr>
<td>Coding</td>
<td>Release 2</td>
<td>Week 10 winter</td>
</tr>
<tr>
<td>Coding</td>
<td>Release 3</td>
<td>Week 3 spring</td>
</tr>
<tr>
<td>Coding</td>
<td>Release 4</td>
<td>Week 5 spring</td>
</tr>
<tr>
<td>UI mockups</td>
<td>Paper prototypes</td>
<td>Weeks 5 and 6 winter</td>
</tr>
<tr>
<td>UI mockups</td>
<td>High fidelity prototypes</td>
<td>Weeks 7, 8, 9 winter</td>
</tr>
<tr>
<td>UI mockups</td>
<td>Functional prototype, integrated with server code</td>
<td>Week 10 (Release 2)</td>
</tr>
</tbody>
</table>

**Actual Milestones**
All documents were completed on time, although the sponsor and coach rarely provided feedback on document drafts, so few revisions were made. The website was configured on time, but the team cut corners by having many documents link to Google Docs, so the website updated automatically. The team never fully integrated the Trac project management
tool because it was too heavyweight for such a small team. The first full “vertical slice” of implementation was completed early, over winter break instead of week 4. The 2 back-end developers had personal environments set up on time, but the front-end developer never set up a development environment. Instead, he developed on a staging server, which was beneficial as it allowed constant acceptance testing.

The UI prototyping stayed on schedule, although the team underestimated how difficult high-fidelity prototypes were to create, so the first week of high-fidelity prototypes did not cover the entire application. Most UI prototypes covered “high risk” areas (like survey feedback or survey creation), as opposed to more mundane screens (like login).

The 4 planned releases stayed on schedule, although the team wasn’t able to demo for the first 2 releases. As planned, the team did not incorporate the “real” user interface until release 2, when the mockup process was complete. The team and sponsor mutually agreed that showing a demo that looked vastly different from the mockups would be both jarring and counterproductive, especially for the multidisciplinary team. Unfortunately, due to an oversight with presentation scheduling, the team wasn’t able to demo release 2 to the sponsor.

By week 2 of spring quarter, the sponsor was eager to see a demo, so the team prepared a “pre-release” deployment. However, by release 3, the sponsor began discussing plans to present at ImagineRIT, so the team had to schedule in a 5th release for the end of week 8. The 4th and 5th releases stayed on schedule, although the planned “code freezes” and “bug bashes” were skipped for additional development during release 5.

Of all the tasks planned, only one - writing a comprehensive acceptance test plan - was cut. Because the team was testing on a staging deployment every day and was behind schedule for features in release 2, implementation catch-up was deemed more valuable.

Although not formally scheduled, the team conducted security testing in week 4 of spring quarter and usability testing in week 8.

**System Design**

**Architecture and Technologies**

Architecturally, Easy360 is a standard 3-tier web application utilizing the Django framework. Django is used exclusively with the Python programming language, and all server-side code is written in Python. Django provides functionality for session management, page controllers, URL dispatching, web page templating, user management, groups and permissions, and object-relational mapping. Django is roughly based on the MVC pattern. Models (classes that extend the Django “model” class) defined in the models.py file are converted to database tables. Methods defined in views.py respond to specific URLs (from urls.py) and return web pages (often based on a Django template). The team used MySQL in production and staging, and one member used Postgres in his personal development environment.
From the client side, pages were written in the Django templating language, which is basic HTML pages with special “template tags” for accessing data passed to the template. The team used CSS for styling, Javascript and jQuery for client functionality, and Yahoo UI for some graph rendering.

The team was able to leverage so much functionality using Django, so the primary design focus dealt with domain design, described below.

Domain Design
The domain design was driven by, the relationship between key domain concepts and the requirements. Team Phalanx decided that the application should be structured in 2 top level subsystems (or Django “apps”): survey management and user/authentication management.

Easy360 needed surveys, and surveys had to contain questions. The team thought it was important for questions to be reused in multiple surveys, where different surveys could possibly change some question features, so the team added a “question instance” object. Each question is related to a scale (personal dimension measured by the survey), and scales are part of super scales. Users are invited to take surveys about targets, and the relationship between targets and raters needed to be captured when inviting each rater. All of these considerations led to the initial domain design for the survey subsystem below:
The green models, defined by the Survey Administrator, are specific to an organization (discussed below), and are available to all surveys. The blue models are specific to individual surveys. The User model (red) is a standard Django model that handles functionality like password hashing, something Team Phalanx did not implement.

The survey subsystem model design has 4 important concepts:

- **Question Instances** - designed explicitly for reuse purposes, question instances (theoretically) capture a question at a snapshot in time and are used in individual surveys. As implemented, question instances are little more than references to Questions, and changing questions can have unintended consequences. For example, changing question type of a question used in a previous survey can change the survey results.

- **Survey Results** - these objects are the core of the system. They are specific instances of “one rater, one target, one survey”, and all answer data is tied to the survey result. Survey result objects are used when taking surveys, indicating where data should be saved. They’re also used when viewing results, indicating what data should be displayed.

- **Pagination** - simple page objects associated with surveys are used to control the

- **Question Hierarchy** - in the design and for the first few releases, all question types (Likert, Multiple Choice, and Open Ended) were subclasses of Question. Later in development, this practice was not continued. Instead, developers ended up duplicating code. This part of the system is a good opportunity for refactoring in the future.

While not included above, the survey management subsystem has several models for specific question types, as well as models for custom weights and labels for these question types. These models are discussed more thoroughly in the refactoring section, as they represent one of the biggest design flaws and refactoring opportunities.
While most of Easy360’s functionality (and value) is within the survey management subsystem, the second subsystem, user / authentication management, is included below:

**Users / Authentication Subsystem Domain Design (in blue)**

The objects inside the blue box are part of the users/auth subsystem. The Management Level and Department are used for organizing users within the application, although they are simple “type” objects with a single attribute: name. The Organization model (introduced quite late in the development cycle, causing high coupling with models in the survey management subsystem) is very important. Easy360 allows multiple organizations to be included in a single deployment, and all organization-specific data (surveys, questions, relationships, users, etc.) needs to be associated with the Organization. Organization “ownership” was de-normalized (duplicated) for performance reasons, reducing the number of “joins” required in SQL queries.

The brown Survey Settings model is part of the survey management subsystem, although it deals with organization-wide behavior. Site and Survey Administrators can tweak settings here, like automatically appending “No Answer” options to all questions, or picking whether “Compare to Peers” compares to all targets in the company or just targets in the same department.
Functional Subsystems

Beyond the models described above, Easy360 has several functional subsystems. The survey management subsystem includes:

- **Email subsystem**, used for storing default email text for all emails, sending emails for all notifications, and handling the “tagging” component when customizing email text (allowing users to insert data like survey name, target, rater, and survey administrator).
- **Confidence subsystem**, used for calculating interrater agreement (IRA) statistics for current feedback. This subsystem can utilize 6 different statistical distributions when calculating IRA, which can be set through the Survey Settings model described earlier.
- **Form subsystem**, leveraging Django’s standard forms and special model forms for data input, input validation, and special saving instructions (like making a user a “rater” when selected as a rater).
- **Results processing subsystem**, used for data transformation between database representations and representations needed throughout the application. Several functions in this subsystem mold data into the CSV for export. Other functions turn the data into JSON format for the current feedback or historical feedback.
- **Organization subsystem**, an abstraction of the (over-coupled) “multiple organizations per one deployment” feature added late into development. This subsystem provides functionality to restrict data to data owned by an organization.

The users/auth subsystem includes:

- **Permissions subsystem**, leveraging Django groups to check user permissions. This subsystem is used by views, to restrict all URLs to specific roles, and by templates, to restrict certain data and visible functionality to specific roles.

Refactoring Required

The components that were designed thoroughly, including Survey Results and Pagination, were incredibly useful throughout the project. Components that were not designed up front, like Organizations, custom question weights/labels, and most importantly, question types, were not implemented in the best manner possible. The biggest opportunities for refactoring are:

- **Question Types (high priority, high effort)** - originally, all question types inherited from the Question model. During the project, this was abandoned and each question type included a foreign key to the abstract Question. Additionally, each Question “row” also included a string “type” label and the ID of the specific type. As a result, there are frequent string checks for type (instead of using polymorphism to determine the correct functionality to call), frequent areas of duplicate code, and unnecessarily duplicate data saved in the database. Also, the distinction between “Likert” and “Multiple Choice” questions is arbitrary: they do the exact same thing, except Multiple Choice questions render vertically and Likert questions render horizontally and can be included in grid pages. This minor distinction cascades throughout the entire backend.

- **Question Weights / Labels (medium priority, medium effort)** - survey administrators can customize the labels and weights for each option in multiple choice and likert questions. However, the survey management subsystem has 2 identical models to record these customizations: LikertValue and MultipleChoiceValue. There is no reason for this duplication.
● **Default email text (medium priority, low effort)** - the default email text for all organizations is saved within a Python source code file (default_email_text.py), instead of a properties file.

● **Overly complicated forms (medium priority, high effort)** - several forms, specifically QuestionForm and QuestionLabelForm, are unnecessarily complicated, often due to the issues described above. These forms should leverage Django’s functionality instead of searching request POSTs for data.

● **Organization de-coupling (medium priority, high effort)** - because organizations were added late in development, the organization-ownership subsystem has significant coupling with other models and functional subsystems. A more modular subsystem would make the code simpler and would prevent needless passing of user data between functions.

● **Better organization of functionality in results_utility.py (low priority, medium effort)** - the results processing subsystem performs many data transformations from one complicated format to another. These formats aren’t well documented, and it’s likely that methods in this file contain some duplicated, or at least hard to read, functionality.

● **Unused code (low priority, low effort)** - several modules (like custom_errors.py) contain unused code.

**Process and Product Metrics**

**Metric Choice**

Team Phalanx originally planned on recording 3 metrics:

● **Effort (hours)** - this metric, required by the Software Engineering department, tracks time spent each week for each team member across 4 categories: technology/development, documentation/artifact tracking, research, and meetings / coordinated activities.

● **Defects found / fixed** - this metric was intended to measure quality control. A high fix percentage (90%+) would indicate that the team is actively fixing open issues and preventing large backlogs of bugs. When this metric dropped below 90%, the team would know to divert more attention to bug fixes.

● **Interface change requests** - this metric was in direct response to a large concern that the sponsor and multidisciplinary team would frequently request UI tweaks.

The scheduled UI mockup process completely mitigated the risk of UI churn, and the third metric, interface change requests, became unnecessary. Instead, the team measured more elements of quality: **total defect fix percentage**, **average time to fix defects**, **percent of defects fixed within 48 hours**, and **percent of defects fixed on the same day**. Again, having a 90%+ fix rate ensured that the bug backlog stayed small. Additionally, having a 90%+ fix within 48 hour rate ensured that bugs were fixed when they were best understood. Average time to fix was originally intended to indicate timeliness of fixing bugs, but was too vulnerable to outliers, so the team introduced the “within 48 hours” metric.
Metric Results

Team Effort

As of 5/11/12, the total breakdown is:
Total hours: 857.3
Development hours: 504.6
Documentation hours: 95.5
Research hours: 26.8
Meeting hours: 220.0

As with all student projects, these metrics only reflect hours logged, not necessarily all hours worked.
As of 5/13/12, the total breakdown is:

Valid bugs found: 93
Total bugs fixed: 92 (98.9%)
Fixed in 48 hours: 87 (94.6%)
Fixed same day: 69 (73.9%)
Average time to fix: 1.3 days

Only one bug still exists within the system: poor performance when sending emails. The team investigated some solutions, including using the Celery distributed task queue tool, which was too high risk for the team to configure correctly so late into the project.
Additionally, Team Phalanx collected data about Easy360 (using Easy360 to collect the data) on both Senior Project Poster Day (5/4/12) and ImagineRIT (5/5/12). On Poster Day, 5 quantitative questions were asked:

- Team Phalanx's poster is visually appealing
- Team Phalanx's poster has quality content
- Team Phalanx provided insightful discussion about Easy360
- Team Phalanx's demo was easy to use
- Team Phalanx's product was highly polished

**Poster Day Results**

1 senior student, 2 faculty, 1 IAB member, and 1 other visitor completed the survey. As the chart shows, all users felt the demo, poster, and presentation were of high quality.
On ImagineRIT, 3 quantitative questions were asked (written by the sponsor, Dr. Palanski):

- I like Easy360
- Easy360 is easy to use
- I would recommend Easy360

ImagineRIT Results

11 students, 2 staff, and 38 visitors completed the survey. As the chart shows, all users felt Easy360 was easy to use, liked Easy360, and would recommend Easy360 to friends. The yellow (moderate confidence) for visitor indicates that not all 38 visitors agreed on an exact score.

Product State at Time of Delivery

The delivered implementation meets all of the agreed-upon requirements, described above in the Basic Requirements section. Team Phalanx originally committed to a smaller and realistic scope, and was able to respond to a large quantity of sponsor feature requests. The 23 additional features not specified in the original requirements document, yet included in the final release, include:

- Forgot password (password reset) - this feature was an oversight
- Relationship Categories - customizing relationships outside of a work context
- User first name / last name - originally, email was the only information included
- User “management level” - specify a user’s title without building an organization tree
- Invite unregistered raters - automatically create an account for raters not in the system
- Add user-identifying information in export - for raw data, include exact value, relationship, rater name, and target name
- Hide results if too few raters - to maintain anonymity for certain relationships
However, Team Phalanx was unable to implement 3 features that the sponsor requested after agreeing upon the initial requirements. The team agreed to implement both Print Macro and Current feedback “zoom”, but had to cut them from the final release. The sponsor agreed that these features were low priority and deemed them acceptable to cut. They included:

- **Print Macro** - a single feature to create one PDF for all screens for a given visualization (for example, “history” would have a screen for the super scale level and one screen for each scale level). This was the highest priority feature cut. The team justified this decision by recognizing that feedback interactivity was a primary competitive advantage of the application, making any static (printed) version significantly less valuable to targets and coaches.
- **Current feedback “zoom”** - select a single super scale, scale, or question, and “zoom in” by hiding all other questions on the axis, centering the selection, and providing more space for each relationship icon.
- **Context-sensitive help** - based on where a user is within the application, clicking help provides context sensitive explanations or tutorials

For other future work, refer to the earlier Refactoring subsection in the Design section. Other major new features are up to the discretion of the sponsor and handoff team; no significant features or stretch goals, outside of optimizing the system for scale and implementing monetization functionality, were discussed during the course of this project.

**Project Reflection**

**The Less-Than-Good**

The biggest issue throughout the project was feature creep. The first feature list was
intentionally light; the team thought it was better to plan less and be able to add new features, instead of planning ambitiously, having to cut features and being unable to respond to change. The team also planned the last release for week 5 of spring quarter, allowing 5 weeks for polish and bug fixes, knowing full well that this period could also be used for additional feature development. Despite that planning, the team originally did a poor job negotiating scope, saying “yes” to every additional feature. As the list grew (eventually to 23), the team became better at compromising on features, settling on features that satisfied both the sponsor and team. In fact, the sponsor became conditioned to the team’s questions of prioritization, framing each request with "how hard would it be to…".

Despite learning how to negotiate and compromise, the team did not do a good job uncovering assumed requirements. The first example was when the sponsor asked about a "password reset" feature that was not implemented. The team agreed this was an oversight. Ultimately, the sponsor had several assumptions that were not included in the requirements and weren't specified until late in the project.

These assumed requirements could have been uncovered if the team did a better job including the sponsor in key deliverables and being more persistent in seeking feedback. For example, the team provided the requirements document to the sponsor and coach, but neither provided critical feedback. If the team was more insistent, certain features (like password reset) might have been uncovered earlier. The team also led several demonstrations of the product and created a special deployment for the sponsor. While the sponsor was satisfied with the functionality shown in the demos, he didn't recognize missing assumed features until he used the product himself. Again, if the team was more insistent in having the sponsor use the product instead of merely viewing the demo, these assumed requirements could have been uncovered.

While the team did a good job communicating overall, there was not enough communication regarding source code changes. With only 3 members, the team didn’t have time to schedule code reviews and had to trust each other to deliver quality functionality. However, this informality led to some inconsistent design decisions that required refactoring and bugs fixes in unfamiliar code areas that could have been avoided.

Finally, while the small size provided several benefits, it also contributed to several weaknesses. With larger teams, more members can contribute opposing ideas, and decisions (ideally) form from the best of each idea; having only 3 members, the team occasionally failed to fully justify decisions, falling prey to groupthink. The team also struggled maintaining constant pace after releases; although all 3 members had to push for each release, any slacking afterward would put the team behind for the next release. Finally, the team had difficulty finding a fair balance of tasks. For each release, the team tried to assign tasks in “themes” that made sense, to prevent blocking, but these themes required inconsistent effort.

The Good
The UI mockup process was very successful in mitigating the risk of high UI churn. After 2 weeks of paper prototypes, 3 weeks of high fidelity prototypes, and a week of functional prototypes, the sponsor rarely requested any UI changes.
In order to prevent an overwhelming backlog of bugs, the team heavily prioritized bug fixes. As mentioned in the metrics section, the team finished with a 99% bug fix rate, with 95% fixed in 48 hours. Although not intentional, most bugs were found when working together, which encouraged the team to fix those bugs immediately. It’s worth noting that 2 team members shared an apartment during the project, which made communicating, diagnosing, and fixing many bugs easy. Overall, the entire team stayed in constant communication, never remaining blocked on any single task.

The team also did a good job staying focused on tasks that added value to the product. The team website was barebones HTML. The team abandoned the Trac project management tool when it became burdensome to update, far outweighing any coordination benefit it would have provided.

Finally, the team successfully leveraged 3rd party tools to handle common functionality. The Django framework allowed the team to develop core business functionality early, instead of spending time implementing session management or EE patterns or ORM. Django’s debug mode was also helpful when finding bugs, showing a error screens with quality diagnostic information. Additionally, several client side libraries simplified and minimized the amount of homegrown Javascript necessary.