Suggestions for Writing Up a SCORE Contest Project

The ICSE 2009 SCORE Organizers*

v 1.01, produced August 14, 2008

Premise

SCORE contest entries require a report of approximately 20 pages. There are no strict rules regarding the format or content of this report. However, some guidance — suggestions, rather than strict requirements — may be useful to teams. This document attempts to provide some useful suggestions.

1 Introduction

Producing a report is very much like producing a software system. There is no single "right" way to do it, and there are a wide variety of approaches, but there are also a few general problems and approaches that are largely shared across all the successful approaches. One of these (just as for programming) is that a good report requires careful design. Just sitting down to write, starting from the beginning, can sometimes (not often, actually) produce a reasonable one-page document, but is never adequate for a 5 page document, let alone a 20 page report. Also as in software production, we should anticipate from the beginning that revision will be necessary. Plan on producing several partial prototypes before you produce a full report you are happy with, and don't be reluctant to refactor the initial design as improved structure becomes clear to you.

Another way that writing the SCORE report is like constructing a software project is that we seldom start from scratch. Surely you incorporate existing software components in your projects when possible, in preference to writing new components with the same functionality. Where it is necessary to write new code, you probably start with some cut-and-paste from similar code you have written before or found, or you at least refer to example code. Much of the content you will need in a SCORE report is likely also to exist in some form already, though it will probably require some rework.

*This document was collectively edited by the SCORE organizing committee. A first draft was written by Michal Young, and revisions were contributed by Matteo Rossi, in consultation with other organizers.

2 How to get started

In the following sections, we describe several pieces of content that we would expect to find in most project reports. These will apply to various degrees in particular projects. For example, reqirements analysis might be the key concern in one project, while another project may have started from a fairly complete requirements statement and need only a reference and a brief summary for their project report. The list of topics we provide is a starting point for designing your report, rather than an outline of the report you should produce.

One way to proceed is as follows:

- Read through the suggested topics below, and make a note of the degree to which each topic applies to your project and the major points you would want to include in your report. Specific notes (e.g., "We really need to discuss how we ensure that the encrypted database would not reveal real email addresses even if an adversary managed to steal it") are more useful than general notes ("Discuss key design decisions and rationale").

  If some of the topics listed below are not relevant to your project, consider why. Perhaps your report should include a short rationale for omitting that topic.

- Consider what is missing from the list. If you covered each of the topics we have listed, what important aspects of your project would not be discussed adequately? Add those to your list.

- Make a first, very rough cut at estimating how much you should write for each topic. (Here is where being specific about content starts to pay off.) Although 20 pages may sound like a lot, there is a good chance you will find that the lengths of your topic items add up to much more. If so, now is the time to make a very rough page budget, noting how much space you plan to *actually* give to each topic. This page budget will change as you flesh out your outline and report, but even a rough page budget will help in outlining.

- Make a first content outline. A content outline doesn't just have bullets like "requirements discussion"; it includes very brief (usually one line) summaries of the actual content that will go in each part of your report. Now you should be rearranging and structuring the material, looking for a good order of presentation that will help a reader absorb it efficiently. It may be similar to the order of topics we list below, but it need not be.

- Draft an executive summary. An executive summary is at most one page long, and serves as a summary of the whole report aimed at a busy manager who will read *at most* a single page (less if he doesn't like the project or your writing). Writing such a summary is hard work and typically requires many revisions, but it has a big payoff: If you can write a coherent one-page

synopsis that effectively conveys all the key points, then writing the main body of the report becomes much easier. It is just a matter of putting back some of the details that you had to crush out of the summary.

- Revise your content outline in light of what you learned from writing the executive summary. That does not mean that they need to have precisely the same structure, but it is common to discover that some reordering will help "tell the story" more efficiently.

- Assign parts. Each member of your team should have a clear writing assignment (e.g., "John will write section 3.4 on protecting the database, Jill will write section 3.5 on the interface between database and mail forwarding subsystem, and John and Jill will be first reviewers of each others sections".) One person, possibly the team member whom you think has strongest writing skills, should be designated as the "paper boss" responsible for reviewing and integrating all contributions. The paper boss will probably need a lighter writing assignment than others, to provide enough time for review and integration.

  Treat integration of the document like integration of a software system. This means: Plan it, do it often, check it continuously. Provide high quality modules to be integrated, because the task is hopeless if checking and fitting the overall structure is mixed with excessive debugging of the individual contributions. The paper boss *must* have the ability to send a module of insufficient quality back for a rewrite by its author (or reassignment to another author) if its quality is not good enough.

- Review, revise, repeat. Treat the document as a series of prototypes, and have a "build plan" that tackles especially thorny parts first. Have many concrete milestones, and gauge your progress against them. Integrate and evaluate on a regular basis.

3 Major topics

Despite a wide variety of application domains and software development methodologies, the following topics are relevant to most projects in some form.

3.1 Development process

When developing your software project, you will (consciouosly or not) follow some kind of development process: "waterfall", "spiral", "agile", etc. If the particular development process you followed impacted in a significant way the issues you faced and the artifacts you produced, consider stating from the begninning what your process was, so that the rest of your report will be read in light of that.

3.2 Requirements: Problem statement

It's hard to build a software system if you don't have a pretty good idea of the problem you are trying to solve. It's impossible to explain your project to others — managers, users, future employers who ask to see samples of your work — unless you can clearly explain what requirements your system was intended to solve.

A problem statement differs from a requirements specification. We say it is "in the problem domain" rather than "in the solution domain". A requirements specification describes one way of solving the problem described in the problem statement, but there may be others.

For example, the following might appear in a problem statement:

> Provide to the blind or low-vision user an accessible indication of position ("cursor") in the currently displayed map.

The following might appear in the requirements specification, as the selected realization of that requirement:

> Current position (virtual "cursor") on the map, relative to a grid with major and minor divisions, is indicated in two ways. First, when the cursor crosses a grid line, a (configurable) tone is sounded. In the initial implementation this tone will be a "click" (with aurally distinct click sounds for vertical and horizontal grid lines, and for major and minor divisions). Second, each major grid division has a name, and this name will be spoken when a (configurable) key combination is pressed. Grid sounds and the key combination for speaking the current location are configurable through the map style sheet.

Common presentations of requirements statements include scenarios (in various forms including "uses cases" and "user stories"; other forms are of course possible).

3.3 Requirements specification

A requirements specification describes what a system must do. It is often interwoven with a requirements problem statement, but is logically distinct, because it describes one solution to the problem. Some development methodologies include extensive formal or semi-formal requirements specifications. In other cases, the clearest specification of system functionality as perceived by the user may be a user manual. Use cases and user stories can also express aspects of a requirements statement.

Requirements specifications can include aspects of behavior that are not directly visible to the user in normal use, but which are nonetheless important characteristics from the user's point of view. For example:

> A forwarding host for a noncemail system shall be constructed such that, even if an adversary were to obtain unlimited access to all persistent storage on the forwarding host, that information would provide the adversary with at most the individual destination addresses for which the adversary had separately obtained keys.

Formal and semi-formal models of various kinds (e.g., entity relationship diagrams, message sequence charts, use case diagrams) are often but not always part of a requirements specification. Since software specification inevitably includes design decisions (choosing to solve a problem one way rather than another way), requirements specifications are often accompanied by a rationale explaining the relation between the solution statement and the problem statement.

3.4 Architectural design

The architectural design describes how the project is broken into major parts, how those parts are related to each other, and how together they realize the overall system. Architectural design descriptions vary widely, but it is rare to find a successful system without a clear overall structure. (Unsuccessful systems with no clear overall structure are a good deal easier to find.)

Description of the architectural design is typically the starting point for describing other aspects of the project, such as the project build plan (even when the schedule is highly iterative and many major parts are developed in tandem). It is also a good place to indicate major technical and schedule risks (e.g., if some components are well understood and others less so) and planned or potential system evolution.

A familiar example of architectural design is division of a web-based application into "tiers", e.g., a database tier, a business logic tier, and a presentation tier. When a common architectural pattern (3-tier, MVC, etc.) is used, the common pattern is a starting point for describing how that template has been instantiated for a particular project.

3.5 Project plan

Most SCORE projects will have strict internal deadlines (e.g., completion by the end of an academic term) in addition to deadlines imposed by the SCORE contest. Most teams will find it useful to carefully plan their effort so that they achieve a good result in their limited time. It is appropriate to describe the project plan and its execution in a SCORE report, although in most cases a narrative account of the project does not provide a very useful overall structure for the report as a whole.

A rudimentary plan might include milestones for products including prototypes with selected product features, user manual, etc. More detailed plans are usually heavily cross-referenced to architectural design. For example, we might see the following in the development plan for a single-use email address system:

Demo 2, week 3: On Friday of week 3 the first integration of noncemail mail forwarding will take place. The following features will be demonstrated:

- Obtain a single use email address through a web page. The user will enter a destination email address and a memorandum note, and will obtain in return another, dedicated email address (the "nonce" address).
- Forward nonce messages. An email message sent to the nonce address will be forwarded by the noncemail forwarding host to the destination address, with the memorandum note inserted in the beginning of the message.

This demo will require initial, partial versions of the web interface component, the nonce database, nonce production, nonce interpretation, and email forwarding. The key goal is to provide a working end-to-end system in which more functional versions of each of these components can be incorporated incrementally. Each of the components integrated in this version is a minimal stub:

- The web interface will be a simple HTML form, lacking the planned javascript support and cookie management to minimize interactions necessary to obtain and paste a nonce address.
- Nonce production and interpretation are simple stubs with no actual encryption.
- Neither nonce management functions (e.g., deactivating a nonce address that has been identified as a source of spam) nor the web interface to those functions will be included in this prototype.
- Database functions will be limited to those required for storing and retrieving nonce forwarding entries, and will not include management (periodic backups, graceful start-up and shut-down, etc.) nor performance tuning.
- Mail forwarding will support only simple plain-text messages (RFC 822). Results of attempting to forward a MIME-structured message are undefined in this prototype.

In reporting on the project plan (and perhaps in consructing a plan), the following guidelines may prove useful:

- A major goal in project planning is to provide process *visibility*, which means the ability to monitor progress against the plan. This is as true for agile processes, in which the plan may be developed incrementally along with the project, as it is for more conventionally structured projects.

- Risk planning is closely associated with process visibility. It is very often useful to explicitly describe perceived risks and how they were addressed in the plan. (Often this means trying to make any unpleasant surprises appear earlier, rather than later in the project.)

- Like every other artifact produced in software development, a plan is the product of design. It has goals (e.g., balancing effort across time and team members, and minimizing risk of unpleasant surprises near the end of a project) and constraints (a limited supply of team members and time, a hard deadline). A *design rationale*, explaining why certain design choices were made rather than others, is useful in the description of any designed artifact, including a project plan.

An evaluation plan may be integrated into the project plan, or you may find it useful to write a separate section describing the approach you took to validation and verification.

3.6 Management plan

There may be aspects of project management that you consider important to report, but which are not captured adequately in the project plan. These can be described separately in a management plan. In some cases (particularly for agile processes), the management plan may take the place of a more detailed project plan.

Here is an example of something we might see in a management plan for a team adopting an agile process:

> In lieu of fixed development roles through the project, a portion of the regular Friday meeting was devoted to assigning roles for the following week. There were three parts to this process. First, development goals for the following week were listed on the whiteboard, with required expertise, effort estimates, and observable milestones for each. Then, as a group, we divided those development goals among team members, designating in each case a *primary* developer and a *secondary*. The *primary* developer was responsible for providing working code, an automated unit test suite, and documentation to the secondary for review by Wednesday evening of the following week. The secondary was responsible for returning a review by Thursday at five. In addition, Tuesday morning between 10:00 and 11:00 was designated as the period when a primary could request to swap the primary/secondary designation on a particular work assignment, and the secondary could either accept that swap (if his or her own work assignment was going well enough to make the extra work feasible) or refuse it.

3.7 Implementation

If implementing some part of your system required solving non-trivial technical problems (e.g., developing efficient data structures or algorithms), consider discussing those problems and your solutions and rationale in your report. For example:

> We set an objective of no more than 0.05 seconds latency between input and appropriate audio output. For our sample campus map containing a few thousand elements (buildings, streets, etc), we obtained satisfactory response using standard Java graphics collision-detection methods against each element in the whole map. However, our measurements indicated likely problems with more complex maps or less powerful computers. For a "torture test" we synthesized by combining ten slightly perturbed copies of the main campus map, exhaustive hit-testing required up to 0.10 seconds on a typical desktop computer and over 0.5 seconds on an older laptop computer.
>
> To obtain sufficiently fast hit detection with complex maps, we employ a quadtree display list data structure (see Figure 1). We maintain a separate quadtree for each map layer, and represent shapes in world coordinates rather than screen coordinates, so that the structure need not be recomputed as the map is scrolled or layers are turned on and off. As in classic quadtree display list structures, the map area is divided into four quadrants, and each quadrant is recursively subdivided, until each node in the quadtree contains no more than k elements. k is a compile-time configurable constant currently set to 4 to balance search time with space taken up by the tree.
>
> Our quadtree structures make use of one key simplifying assumption: Shapes within each layer are non-overlapping. This guarantees that recursive subdivision of a quadtree region into sub-regions will terminate. Without this assumption, it would be necessary to clip objects to quadtree nodes and discard those that are completely obscured by others. Assuming non-overlapping shapes considerably simplified our quadtree-building code and saves space since we do not need to either compute or store clipped versions of map shapes. Searching for the top-most shape covering cursor position (x, y) is particularly simple: We convert (x, y) to world coordinates (x_w, y_w) and search each map layer in order from top to bottom, stopping when we first encounter a shape that covers (x_w, y_w).
>
> Using our quadtree structure, we measured hit-detection times consistently less than 0.01 seconds, even using our torture-test example map on an older laptop computer.

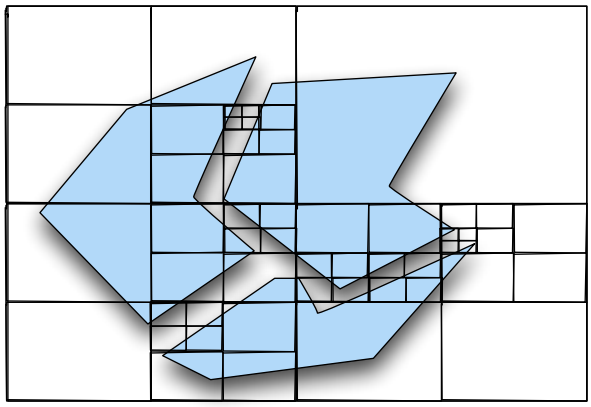

Figure 1: *The quadtree display list structure, used for fast hit detection, subdivides display regions containing any part of more than k shapes ($k = 1$ in the illustration, but $k = 4$ in the actual implementation). The root node of the quadtree represents the whole map area, and leaf nodes (containing lists of shapes) represent regions that are not further subdivided; interior quadtree nodes represent subregions that are further subdivided.*

3.8 Validation and Verification

To convince yourself of the correctness and effectiveness of your design/implementation you have most probably carried out some activity, typically (but not limited to) testing.

A description of the test plan you created, of how it was carried out, and of its outcomes (e.g. if it unearthed unexpected errors and problems) is useful to convince an evaluator of your project of the soundness of your solutions.

You may also have carried out additional verification activities during development, including automated checks or formal analysis of parts of your design or implementation. If you did, a brief description of which techniques you used and what (if any) problems were discovered can be included in your report.

For example:

> Because multiple users may interact with noncemail concurrently, we were concerned with the possibility of race conditions, which might go undetected in small-scale testing but surface in heavy usage. We avoided fine-grain data integrity problems by using transactions in the MySQL database management system to manage the table relating nonce addresses to real addresses. Grouping database operations as relatively coarse atomic transactions (e.g., "add a nonce forwarding entry to the table") freed us from reasoning about fine-grain races and reduced the burden of exercising fine-grain concurrency be-

tween operations in testing. It also clarified the invariant condition that should hold between transactions, leaving us a verification obligation of showing that each transaction takes the database from any legal state to another legal state.

We manually listed conditions relevant to each transaction (things that can vary while the invariant still holds), and the expected behavior under each condition. This prompted us, for example, to consider and document the expected behavior when a user attempts to deactivate a forwarding address that has been previously deactivated. These descriptions of expected behavior were used in four ways: As directions to the programmer implementing the corresponding module, as unit and system test obligations (we required unit and system test suites to include each of the documented scenarios), as a check-list for a developer inspecting another developer's code, and as a note to the author of user documentation to explain possible outcomes to users.

In addition to this (manual) analysis and systematic testing, we created a load test harness in which simulated users and simulated mail sources repeatedly and randomly carry out a set of noncemail scenarios at high speed. Load testing did not uncover any real errors in our implementation, but did reveal some mistakes in our case analysis. These mistakes were all due to one fundamental problem: The mail transport system itself is not under the same concurrrency control as the database, so some seemingly "impossible" sequences are in fact possible. For example, it is possible for mail forwarded through a nonce address to arrive at the real address after the nonce has been deactivated or deleted, because it spends some time "in transit" after forwarding. We revised our design documentation and user documentation to reflect this.

3.9 Outcomes and lessons learned

Some projects may be essentially completed within the contest period. Others may be (according to plan, or not) partial implementations. A SCORE report should indicate what was actually accomplished. Screen shots, input/output samples, excerpts of user manuals, and descriptions of actual use may be included as appropriate.

It is also appropriate to recount (briefly) lessons learned from the project experience, especially since SCORE projects will be carried out by students in an educational setting. Specific lessons ("In retrospect we should have recognized the technical risk in integrating an unfamiliar database component and planned an early integration test") are more valuable than generic lessons ("project planning is harder than it sounds").

4 Summary

This document is not a template for your SCORE submission. Some of the content we have suggested above may be a good fit for your particular project, and some may be irrelevant (but in that case you should have a clear reason why it is irrelevant). It may be in the wrong order. There may be other issues that were critical to your project, and should clearly be part of your report, which are not listed here. Nonetheless we hope that these suggestions will help you think about the content of your report and craft it to be useful to the judges in evaluating your project. We hope also that you find your SCORE report a useful document for other purposes, such as describing your project experience to prospective employers.