

CS32 Final Project

Computer Science 32, Summer 2009
Professor Steven P. Reiss
Professor John Jannotti

| | |
|--|---|
| <i>Team Selection:</i> | 6/24 (one per group: email cs032headtas@cs.brown.edu) |
| <i>Idea and Title:</i> | 6/24 (one per group: email cs032headtas@cs.brown.edu) |
| <i>Initial Project Requirements (I):</i> | 6/29 |
| <i>Survey or Questionnaire:</i> | 6/29 |
| <i>Project Requirements (II):</i> | 7/6 |
| <i>Initial Specification:</i> | 7/6 |
| <i>Specification Presentations:</i> | 7/6 (during class) |
| <i>Final Specifications:</i> | 7/8 |
| <i>Initial Design:</i> | 7/13 |
| <i>Design Presentations:</i> | 7/13 (during class) |
| <i>Final Design:</i> | 7/16 |
| <i>Prototype Implementation:</i> | 7/22 |
| <i>B-Level Project Implementation:</i> | 7/30 |
| <i>A-Level Project Implementation:</i> | 8/5 |
| <i>Demo Day:</i> | 8/5 (presentation) |
| <i>Report and Final Handins due:</i> | 8/6 |

Please check the dates on this handout often as they are subject to change.

1 Introduction

A lot of people come in to CS32 thinking that the final project is all about showing off your skills to your classmates. We can't stress enough how false this is. We're trying to teach software engineering, and this project is your opportunity to take what are learning and make something that you and your group members are excited about. We prefer a well-designed, well-thought out, well-implemented project to a flashy but ill-conceived one!

For the final project, we're giving you the ultimate freedom in a CS course: the choice of what problem you want to solve. In groups of three to five (four is highly preferred), you will specify, design, code, test, and demo a project completely your own.

Ever stumbled upon an idea that you thought might be **the next killer app**? Why not act upon it now? This course offers you the chance to experience firsthand "the tech startup process" all

without having to drop out of school! Plenty of revolutionary applications are written by university students such as yourself, and this would be an excellent opportunity to explore an innovative idea, or create a new approach to solving a common problem. You never know: the work you do in this class, and the relationship you make with your team may end up being the beginning of something even bigger. bottom line: We encourage everyone to *think startup* when working on your final project ideas.

This is the first time that CSCI0320 is being taught in the summer and the project schedule is a bit condensed. We are attempting to give you ample time to work on the project, but you should take the timetables into account in defining your project to ensure it is **manageable**. We will start by providing you with ample time (the first half of the course) to develop appropriate project specifications so you have a precise idea of what it is you are going to build. Then during our evaluations, we will place more emphasis on the testing, stability, and usability of your project than on how many features it has. The ideal piece of software meets all of your initial specifications, is user-friendly, and is never buggy and never crashes. If you're pressed for time, however, we prefer a project that might not meet all the initial specifications but is extremely stable and bug-free.

This document is packed with important dates, explanations, and other useful information, so read it carefully!

2 Timeline

2.1 Project Requirements

The first step in developing a project is to understand what problem you are attempting to solve, i.e. what is your project geared to accomplish. To this end we expect you to develop a requirements document that describes, from a user's point of view, what the project does. This will be done in four steps. First, we expect you to develop a general idea of the project and hand in a corresponding one-paragraph description. Second, we want you to produce an initial requirements document detailing the project from the user's perspective. (We will provide you with more details as appropriate). Third, we want you to get input from potential users (outside of your own group) on the project. To this end, we expect you to develop a survey or an interview questionnaire (whichever is most appropriate) that you will get approved and you will then use to get such input. Based on the input you receive, we want you to modify your original requirements document and hand in a revised version.

Note that requirements is an ongoing process and you should update your requirements documents as the project requirements change over the course of the semester.

2.2 Project Specifications (10% of grade)

Specifications describe your project from the programmer's point of view. They detail how your system is going to meet the various requirements and provide you with a precise checklist and description of what it is that your project has to do. Moreover, the final specifications that you develop and get approved by the staff will provide the basis on which your project will be graded. Specifications will be done in three steps. First, we want you to hand in an initial specifications

document. We will provide additional details on what we expect to see in this document. Secondly, we expect each team to do a short presentation of their specifications in class. This presentation should be a sales pitch for your project. Finally, based on the feedback you get, we want you to provide an updated and final handin that we will compare to your final project.

The final specifications document and the specification presentation will account for 10% of your final project grade. Note however, that vague or incomplete specifications will generally cause you to not really understand what you are building and thus will typically result in a lot of wasted or misdirected effort and a lower quality project overall.

2.3 Project Design

Once you have your specifications approved (or even before if you are ambitious), we want you to start designing your project. The initial design should break the project into components that can be tackled by individual team members. This should be written up in a design document (using UML as appropriate) and handed in. The important thing about this document is to define the initial interfaces between the different project components and to make those components as independent as possible.

Once this design has been checked by the staff, you should continue and do a detailed design of each of the components. Most of this design should be done by the individual in charge of the component, but the detailed designs should be checked by the whole group. One way of doing this is to arrange design presentations within the group where each person takes a turn presenting his/her detailed design to the remainder of the group in order to get appropriate feedback.

The detailed designs should be gathered together into an overall design document which you will need to create. We will arrange a design review with your team and the staff. You should bring the document to the meeting. It should consist of at least the following items:

- separate class diagrams for each package
- specifications for interfaces between packages
- descriptions of major methods and data structures
- a sample run of the project, including a discussion of flow of control
- a plan for testing all inter-package communications

This document should be complete but rough. The design review will help you refine and clarify your design, paying particular attention to the interfaces between packages and the testing plan.

Your design should address each of the following questions at least once:

- What are the separate packages (components)? How do they interact? This is one of the most important things to consider when designing. If the interface between components changes frequently, it will make integrating your project extremely challenging. Find a solid interface early on and stick with it!
- Once you've designed the interfaces between your separate packages, each group member should think about the design for each package that he/she is responsible for. This you've done many times before; think classes, associations, inheritance, capabilities, etc.

- Your testing plan should be a list of what classes will communicate across packages, what methods they will use, what actions will be taken, and what is expected to happen. Essentially, you are trying to ensure that when you integrate, all the methods of one package that are going to be used by another package will function properly. The best way of testing these methods before integration is to use a **driver**, as explained in the testing section towards the end of this document. The main point here is to test the code that you have not written yourself but that your code will rely on after integrating; it does not take the place of testing your own work as you code.

2.4 Final Design Due and Design Presentation (20% of grade)

The grade for your project design will be based on a the design presentation your team will do in class as well as the completed design documentation.

Your presentation might include slides along the lines of:

- project name, group structure, division of labor, and a paragraph or two about the project (1 slide)
- a fairly detailed timeline of individual schedule, in more detail than the deadlines, including internal integration deadlines (not to exceed a slide)
- a design snapshot: 2 slides (3 max) summarizing overall design
- testing plan (1 slide)
- a list of problems/issues (ex., a game group not having a graphics guru) (1 slide)

Although your design might change in the course of writing the project, you will save a lot of headaches if you put a great deal of thought in to this design and stick to it as closely as possible. We take this design very seriously and a poorly thought out submission will result in a grade penalty.

2.5 Project Checkpoints (30% of grade)

You are responsible for demoing your project in class at both the prototype stage and the B-level implementation stage. Come ready to present what each person has done and how it corrolates with the respected checkpoint. Your grade on this aspect of the project will be based on how successfully your implementations pass the tests that you have created in the design phase of the project.

2.6 Demo Day and Final Handin

(40% of grade)

Demo Day is your opportunity to show what you've been working on and to sell your project. It's not about one-upping your classmates or producing the biggest 3D explosions, but giving us an idea of the challenges you encountered along the way and why things turned out the way they did. We want to know why your design turned out to be good or bad, what you had to change, and what you've learned from the experience. We're also really excited to see your projects in action!

For those of you who are at a loss for words, you might want to talk about . . .

- Division of labor - each member talks about their section
- Goals of the project - were they met? If so, how? If something wasn't, what was the problem?
- Difficulties - what did they learn from this experience?
- Visuals - find something cool graphically to display, if applicable - this tends to impress viewers
- A guided tour of the program (again, focusing on the features that work if all do not)
- An amusing part of the process - a war story from the Sunlab trenches, so to speak.

Your final project handin should include up-to-date requirements, specifications, and design documentation, a listing of the code of the project, and appropriate end-user documentation. End-user documentation is a clear, easy-to-read description of how to use the program. Include screen dumps from your application or diagrams that approximate what the screens look like. Someone who isn't a programmer or UNIX sorcerer should be able to properly use your program given this documentation. The documentation may be embedded in the program, take the form of a web page or even be a handout detailing use of the program.

3 Some Final Wisdom

3.1 Requirements and Specifications

Understanding what you are building is central to building the correct product and building it efficiently. It doesn't make a lot of sense to put lots of effort into a project that nobody wants to use or that solves the wrong problem. Requirements, the task of determining what to build, is an important part of software engineering and one we want you to get accustomed to and to understand.

Your specifications detail exactly what the project is going to do. Think of it as a grading rubric. These are the things that are going to be checked when we want to determine your project grade. Vague specifications let us decide whether your project meets them or not; precise specifications let you decide this. Since we don't necessarily trust our interpretations (although we will swear by them and insist that are right), you might want to try for precision here. Specifications are again an important part of software development since they guide the rest of the development process. Its hard to build something vague; its much easier to build something where you know exactly what the interfaces should look like and what each button, command, or event should do.

3.2 Designing and Coding

If you have a good design, coding doesn't have to be the hardest part. Your design will specify what classes must be written and what they should do. This makes your job straightforward; write code that conforms to those pre-defined specifications.

You will likely think up ways to improve your design while you code. You may even discover that your carefully-written design won't work. That's OK as long as you consult with your group before making any changes. Then remember to update your documentation. Any changes you make might affect your group members, so keep everyone informed.

Remember, it's a rare design that is completely free of ambiguities or subtle shortcomings. Developing large software is an iterative process of creating, assessing, and refining; you want to make this process as smooth as possible.

The first step in coding is to have each group member working on their individual packages and test drivers. The next step is to slowly integrate your packages in to a functioning project. This is where a good design pays off and a bad one makes you miserable. Casually throwing your packages together all at once and hoping for the best is not a good idea. Gradually replace your stubs and drivers with actual packages, integrating them one by one until the entire system works together. Integrating too quickly will most likely overwhelm you with errors and bugs. **Make sure to leave plenty of time for integration!**

We recommend you give your program some sort of graphical user interface (GUI). We've prepared everyone in the class to write a GUI so we'd like to see them in your projects. In the rare case that your project is better suited for a textual interface, we'll accept this, but it should be top-notch. Also bear in mind that a text-based interface will most likely not be enough code for one group member; he/she will also have to take on other responsibilities.

3.3 Testing

You'll want to design a testing plan as you code. This is a sequence of tests that will exercise every aspect of your project's inter-package communication. If you've thoroughly tested your code before you try to integrate, it will make everyone's life much easier. Remember JUnit? Well, there are automated testing suites for almost any programming language you choose to code in (e.g. NUnit for any .Net language). We *highly* recommend automated unit testing.

Another method is using stubs and drivers. Stubs are empty methods that will eventually do real work. For testing purposes, however, they usually just print out a message and (if necessary) return a default value. Drivers are just `main()`'s that do some basic setup and feed your package any input it needs for testing. Basically, a driver is a way to test your code before integrating. For example, if you're writing a GUI that responds to another package, your driver will send a series of messages to your GUI, simulating requests from that other package.

For prototype testing, you need to produce unit tests drivers that test your **other** group members' packages with which you interface.

The intermediate project levels reflect the amount of the project that is working (relative to your specifications to get the corresponding grade. A C-level project should have at least 60% of the specifications working; a B-level project 80% and an A-level project 100%.

It is important to test things early on because you may find that your program design is either too slow, takes up too much memory, or just can't function the way you thought it could. These are critical errors that you must catch as early as possible. Having to redesign large parts of your program when you're halfway done coding is a terrible mess.

Once you've integrated your project, test it all together. Don't just use it for a few minutes and give up. Try all the functionality, all the boundary conditions. Write a test suite that stresses it to death. Remember, a **successful** test run results in a **crash!**

You should keep track of your test cases as you work. For each case, record what the test is, the input (either a file or a detailed script the tester can follow), the desired output, and the outcome

of the test. You should run through all of your tests whenever you make a major change to your system. If you document your testing plan well and automate it as much as possible, the process is much easier and more effective.

Although we only require that you show us your inter-package testing, it does not take the place of testing your own code as you write it. It makes just as much sense to come up with a concrete plan to apply to your own package before submitting it to your group members for testing. Think about every chain of events that will take place in your code and test them thoroughly. This will make Project Base I go a lot smoother.

On a related note, we cannot stress enough that you should **leave ample time for debugging**. It is nigh-impossible to write a bug-free version of your project on the first time around. As you know from past programming, sometimes even the simplest, stupidest bugs take hours to fix. There's nothing quite like getting a SEGV up on the projection screen during your final demonstration, so make sure you leave plenty of time to debug!

3.4 Technical Details

Since there will be many different projects, we will not be able to supply you with a default build file and the other niceties that we (might) have provided in the past. Also, you might want to use Subversion to keep your project files straight and make sure one person doesn't mess up everyone else's code by accident. We've already had labs covering Ant, Make and Subversion, but feel free to talk to the staff if you're having difficulties. For the more self-reliant among you, try "man svn" or visit <http://subversion.tigris.org/> for problem solving.

There are also several topics that are relevant to individual projects, but are beyond the scope of CS32. If you feel that there's a topic you really want to learn more about, inform us and we'll try to organize a help session. At the very least the staff will help you find resources on the subject.

A final thought: if you would like to use anything that we haven't taught you, feel free to ask the staff about it. There are many tools and packages out there that you may not be aware of but might be useful to you, and we may be able to recommend something if it suits your needs. Understand, however, that the staff will not be able to support everything. For this phase of the course we will offer you as much help as we can, but it's definitely possible that your group will attempt something that is as new to us as it is to you. Keep this in mind when you're planning your project.

That's all, folks!

HAVE FUN, AND GOOD LUCK!