CSC 478: Software Engineering Capstone

Group Project Assignment

300 points

Team Guidelines

As a software engineering team, you are responsible for building a functional software application replete with thorough documentation of all of your process activities. In order to successfully accomplish this goal, you will have to work together as a team - everyone must share the workload. No individual team member should have to shoulder the majority of the burden. There are several obstacles you will need to overcome:

1. As individuals, you likely don't know each other very well and have never worked together before, so you will have to quickly become acquainted.

2. As individuals, you are separated geographically, so you will not be able to have face-to-face meetings.

3. The time constraint is severe (you only have about 11-12 weeks) and immutable (since I have to submit final grades at the end of the semester).

4. You probably have other classes and/or a paying job and/or a family, all of which will limit the amount of time you can devote to the project.

5. You are not being paid for building the software, so money is not a motivator for doing well on the project.

I will not dictate to you how you should overcome these obstacles, but I will offer a few guidelines which you are free to adopt as you see fit.

Getting Acquainted

You should probably spend some time initially learning about each other. The group discussion board is good for this, but email or even phone calls will work as well. Here are some suggestions for things to talk about:

1. Get to know each other's strengths and weaknesses. Who is likely to be best suited for coding? Who has experience with techincal writing? Does anyone on the team develop software professionally?

2. Find out when each person is available for participating in group discussions, online chats, email conversations, etc.

3. Find out how much time each person will be able to devote to the project.

4. Will there be any times when someone will be unavailable (e.g., business trip, Thanksgiving holiday, etc.)?

Communication

Communication is always a problem where group projects are involved. Since most of you will only be communicating with each other online, via email, discussion forums, etc., communication

may or may not be more problematic than normal, depending on how much you depend on face-to-face meetings. I can assure you that in the industry, it is not uncommon for people who are distributed geographically to work on the same project, so what we are doing is not really that far-fetched an idea. For this course, the Blackboard site provides group discussion forums, and allows group emails, group file transfers, and even group online chat capabilities (if you need more synchronicity to your communication). Feel free to set up any other mechanisms for communicating, transferring files, etc. as you see fit.

Deciding on a Team Name

Having a team name often promotes team spirit and motivation. One of your first tasks as a team should be to decide upon a team name. Initially all teams will be given generic names such as Team #1, Team #2, etc. You may keep the generic name, if you wish.

Choosing a Team Structure

Once you've become acquainted, you will need to decide on a team structure. I've discussed the three major categories of team structure in the lecture. You are free to choose whichever structure you wish, but given the time constraints and the geographic separation of your team members, I recommend the democratic centralized structure. Since this is supposed to be an educational endeavor, everyone must be allowed to participate significantly. But, it is wise to give someone executive power to make final decisions to keep the project moving forward.

The decision of which team structure to use should be made democratically, to minimize the chances of creating dissension within the team.

Dividing the Workload

After coming up with your team structure, you must figure out how to divide the workload among the various team members. You may do this however you see fit, but you might consider using the following roles when dividing the workload (this is not mandatory, only a suggestion):

1. **Documentation Specialist** — responsible for writing the user's manual, documenting umbrella activities such as the project plan, and compiling all documentation in a presentable format. Coders will still document their own code, and designers will still document their own designs.

2. **Architect** — responsible for designing the software, based on the requirements.

3. **Coder** — responsible for implementing all designs.

4. **Tester** — responsible for the test plan, and conducting all tests.

A single individual can be assigned to each role, or the roles can be shared. This has important implications, because it means that you can assign someone to do nothing but documentation, someone else to do nothing but testing, etc. So yes, it is possible for you to get an A in this course without writing a lick of code. The goal of this course is not for each individual to design, implement, and test a piece of software, but to work together effectively as a team to design, implement, and test a piece of software.

Although you have a great deal of latitude in dividing the labor, I am imposing some constraints:

1. Everyone on the team must contribute equally (as much as possible) on the project. Having a team member whose sole responsibilities are to write the scope statement, or construct the Gantt chart as their sole contribution, for example, is not acceptable.

2. You are responsible for doing whatever is necessary to work together as a group. I say this because in the past some students have complained about the roles they were assigned by their teams. They wanted to be more involved in the coding or design aspect, or they were unhappy because they got stuck doing mostly documentation. For this project you may find you will need to set aside your personal goals for the good of the group.

3. If someone on the team does not pull their weight, the burden is on the other team members to do what they can to rectify the situation, and failing that, inform me of the situation. This is important! When I grade I will be grading the projects, not the individuals, unless there is compelling evidence to do otherwise. Whatever grade I assign to the project will be the grade each team member receives. So, for example, if I give your project a grade of 275 points out of 300, each team member will receive 275 points for the project.

4. In the event it becomes apparent that a particular individual just isn't going to work out for a given team, for any reason, either the individual or the team needs to inform me ASAP. Early in the project I may be able to move people around, but there are no guarantees.

Project Ideas

Experience has shown that students tend to be more motivated to work on projects they have chosen themselves than they are on projects that are assigned. Since motivation is such a key factor to success, I give you the option of choosing your own project. There are some constraints, of course:

1. You may NOT reuse a project you have built for some other class or employer. You MAY, however, take an existing project and extend it, although for this to count you must add a significant amount of functionality.

2. You must write at least 50% of the code for the project. So, stitching together pre-built components with a small amount of "glue code" is not acceptable for this course (even though it may be a viable option in the real world).

3. The functionality must not be trivial. An extreme example of trivial functionality would be the canonical, "Hello, World" program. On the other hand, you must take care not to overdo it, or you won't have time to finish everything.

4. Your software must be designed to work on a system running Windows 10, as that is the only OS I have available at the moment.

5. If you decide to build a web-based application, or an application that requires a network connection, you will be entirely responsible for ensuring the web server and other network services are operational on your end. If I am unable to test your application because of network problems, permission problems, etc., you will lose points.

6. If you know your project will have dependencies on third-party libraries, frameworks, etc., you must be able to have your installation package install those dependencies. If I am unable to test your application due to missing dependencies, or configuration issues, you will lose points. To help prevent these types of problems, make sure you test your software on a non-production computer.

7. I must give final approval of your project before you begin working on it. If you choose a project that has been built before by other people (many people have built common games like Yahtzee and poker, for example) I may require that you introduce a twist or two to make your project unique.

Examples of some past projects students have chosen include:

1. board games: Clue, Sorry, Monopoly, Battleship
2. card games: UNO, poker
3. dice games: Yahtzee

4. casino games: roulette
5. MP3 catalog and player system
6. a first person shooter game in which players drive tanks in a 3-D environment and shoot at each other
7. a fantasy role-playing game that utilizes 3-D graphics, where the player controls a character that can cast spells, fight, etc.

If you find you cannot decide on a project, let me know and I will come up with one for you.

Software Process

Several different processes, or workflows, that can be used for engineering software are covered in this course. Each process has its pros and cons. For your group project I strongly encourage you to adopt an iterative or incremental process. Given the time constraints and your limited human resources, most of the other models we've discussed won't work for you. Once you have determined and documented your requirements, I suggest dividing the proposed functionality into multiple versions. Version 1 should contain the features with the highest priority. Subsequent versions should extend previous versions by adding new features or refining existing features. The advantage of this process is that you will be more sure of having something to submit that works, even if it doesn't do everything you would like or it doesn't work as efficiently as you would like.

For example, let's say you're building the poker game used an example on the page discussing the project deliverables, and you divide your requirements into 8 major pieces of functionality. You decide to adopt an incremental approach to building the system, with four development versions:

1. dev. version 1.0.0
   - cards can be generated based on information in a data file
   - cards can be displayed in bare bones GUI window

2. dev. version 2.0.0
   - basic GUI developed
   - implement 5-card stud variant

3. dev. version 3.0.0 -
   - adds save game and load game features
   - implement 7-card stud variant

4. dev. version 4.0.0
   - implement Texas Hold'em variant
   - context-sensitive help feature added

Now, suppose that by the time you've completed development version 3.0.0 and tested it, you only have one more week before the project deadline. You begin implementing the Texas Hold'em variant, but run into problems getting it to work correctly, and you didn't have time to fully implement the context-sensitive help feature. So, you haven't fulfilled all your requirements, BUT you have a functional version 3.0.0 that you can submit.

Project Deliverables (see calendar for due date)

There will be 4 deliverables you will need to submit for your project, which are listed below, and will be explained in greater detail in the following pages:

1. **The Project Plan** — contains all the documentation for the umbrella activities

2. **The Programmer's Manual** — contains everything a software engineer needs if he or she were to maintain the software

3. **The User Documentation** — all documentation an end user would need to install and use the software

4. **The Final Product** — this is the executable software, along with all data files, library files, drivers, and any other file necessary for the software to be used

You will see that a large portion of your grade for the project is determined by your documentation. Documentation is essential to building quality software (although Agile Developers may disagree, for this course documentation is mandatory). Since all software is built to solve some problem, the process documentation serves to encapsulate a solution to that problem, as well as how that solution was arrived at and why that particular solution was used. The finished executable may seem to work flawlessly, but lack of sufficient documentation can be disastrous later on when the software needs to be maintained or changed. There is no guarantee the software will be maintained by the same development team that built it originally, and even if the original team is involved, people tend to forget things over time. Even if the software is never maintained, the documentation can be invaluable should it become necessary to build software to solve similar problems in the future.

As a rule, you should document your project under the assumption that a completely different team, which knows nothing about the problem you are solving, will need to make significant changes to the software, and this team will not have the option of contacting any of you for assistance.

One final note: Professionalism is important! Don't haphazardly slap your documentation together. Be consistent with formatting, use of fonts, etc. Make sure your documentation is well organized, and understandable, so that it's clear what I'm looking at. The guidelines on this page, along with the project rubric, should keep you on the right track. If you are unsure about anything, please e-mail ASAP!

All deliverables must be submitted no later than the due date listed on the course calendar.

1. Deliverable #1: The Project Plan

The Project Plan contains all the documentation for the process activities you perform and how you plan to accomplish your goals.

1. Scope Statement

Before going full speed ahead on gathering requirements, you should establish the boundaries of the software you will build. These boundaries should be documented in a scope statement. Examples of things that establish software boundaries are:

- What type of platform (Windows, Macintosh, UNIX, etc.) must the software work with?
- Will the software function as a standalone application on a given computer, or will it function over a network connection?
- What other software, if any, must the software interact with? For example, you might be building a subsystem component that will be integrated into a larger system. In such a case, it's important that you don't duplicate functionality provided by existing subsystems.
- If you are building a game, how many players will be supported? Will there be a computer player? If so, will it use artificial intelligence?
- What programming language will be used for the project?
- Will the software use a graphical interface or a command line interface?

Note that these are also part of the software requirements, and should therefore be included in your requirements document.

2. Org Chart

You should include a simple org chart showing how your team is organized, and what each team member's role is.

3. Gantt Chart

Once you know what project you plan to build, you need to come up with a plan for how you intend to build the software, and how long it will take. Every phase of the software life cycle except the maintenance phase should be included in your chart, although if you use an iterative or incremental process you may have multiple design phases, multiple coding phases, etc. Since you only have about three months to finish the project, most likely you will end up using it all, so the challenge will be in determining how much of the available time to allot to each task.

You should construct a Gantt chart to indicate all the tasks you will perform, and how long you estimate you will spend on each task. Use Microsoft Project for this, if at all possible. It's an excellent tool for building Gantt charts. If you indicate which tasks are dependent upon which other tasks, it will also calculate the critical path for you. You can also indicate on the chart which team members will be working on which tasks.

4. Tools and Standards

You should have a document that lists the various tools you used to build the project. This list should include things like programming language(s), editors, IDEs (Integrated Development Environments, e.g. Visual Studio, Eclipse), design tools such as UML modeling tools, etc.

For standards, you should include things like the process model you plan to follow, along with any standards (e.g., coding standards, documentation standards, etc.) you plan to use.

5. Configuration Management Plan

You should have a document that explains how you plan to maintain and keep track of the various versions you will end up developing. Be sure you clearly distinguish between your development versions and your release versions.

You need to have some kind of configuration management plan, since you will undoubtedly generate many versions of your software over the course of the semester. I suggest using something simple, like making a copy of the entire project folder for each new version, and using a numbering system like the standard major.minor.build or something similar. If you happen to run into storage problems while doing this, you can always prune the oldest versions from your hard drive and archive those to CD-ROM.

Whatever you do, you should always take care to make sure you keep a copy of versions that work. Many a project has been ruined by making one simple change that ended up cascading into a series of subsequent simple changes, to the point where the software became unstable, and the developer was unable to get back to the original, working version.

6. Weekly Status Reports

Your team is free to make its own schedule as far as the project goes. However, I do require that each team send me a brief status report once per week describing their progress. I have included this to help mitigate the damage due to inherent project risks such as:

- technical problems
- personnel changes; e.g., a student drops the course
- procrastination
- non-participation

Consider me the "department manager", in charge of all the teams. If something is not working out within a team, for whatever reason, I need to know about it as soon as possible if I am to help the team overcome the problem.

To provide incentive for submitting these status reports, each weekly status report will be worth 5 points towards the overall course grade (not the project grade). I have allotted 12 weekly status reports for a total of 60 points.

2. Deliverable #2: The Programmer's Manual

The Programmer's Manual contains everything a software engineer needs if he or she were to help develop or maintain the software.

1. Requirements Documentation

Detailed requirements are essential for building valid software (i.e., the software the customer wants). Your requirements should be as complete and as detailed as possible. You should organize your requirements in a sensible, hierarchical manner. Each requirement should be given a unique identifier that can be used to trace that requirement through the subsequent phases of the software life cycle. Your requirements document should follow the standard below, which is based on the IEEE/ANSI 830-1998 standard. For your requirements document, you do not need to include appendices (unless you have need to) or an index (an index would be nice, but unless you have a tool to generate an index automatically, this is too time-consuming).

1. Introduction

1.1 Purpose of the requirements document *(You can ignore this one.)*

1.2 Scope of the product — just use your scope statement from your project plan here

1.3 Definitions, acronyms, and abbreviations — list and define all of these that appear in your documentation

1.4 References — include references for anything that is trademarked or copyrighted here (e.g., game instructions, algorithms such as the LZW compression algorithm for images, etc.)

1.5 Overview of the remainder of this document *(You can ignore this one.)*

2. General description

2.1 Product perspective — give a little background about why the software is being built and why it will be useful; one good paragraph will do

2.2 Product functions — describe fundamentally what the software will do; again, one good paragraph will suffice, and you do not need to go in depth (the specific requirements section will cover the in-depth functions)

2.3 User characteristics — describe who the end users will be

2.4 General constraints — describe the general limitations of your software (e.g., system limitations, does not work over a network, etc.)

2.5 Assumptions and dependencies — examples of these would be: assumes end user has QuickTime installed; requires Visual Basic runtime to be installed on end user's computer

3. Specific requirements

Section 3, which deals with the specific requirements, will vary depending on the software being built. I suggest using a simple numbering scheme. Below I show a partial set of requirements for a poker game, using a similar system. These requirements are far from complete - they are simply provided as an example.

Playing Cards

1.0.0 The game should use a standard deck of 52 playing cards, with no jokers.

1.1.0 User should be allowed to choose the design used for the card backs from 2 provided designs.

1.1.1 Solid blue design should be provided.

1.1.2 Red thatched design should be provided.

1.2.0 Each non-face card should display the appropriate numerical value.

1.3.0 Each non-face card should display the appropriate number of suit symbols.

Game types

2.0.0 User should be allowed to choose the type of poker game they wish to play.

2.1.0 The game should allow the user to play 5-card draw, according to standard rules.

2.2.0 The game should allow the user to play 7-card stud, according to standard rules.

2.3.0 The game should allow the user to play Texas Hold'em, according to standard rules.

4. Appendices

You don't need to include appendices unless you feel they are warranted.

5. Index

Do not bother including an index unless you have a tool that can automatically generate an index for a document. Otherwise this will be too time-consuming.

2. Design Documentation

Numerous design techniques exist, several of which are described in Sommerville's book. Which techniques you should use for a given project depends on the type of project, and what works best for you and your team. Some techniques are not suitable for some programs. For example, a detailed data flow diagram will probably be of little use if you are building a simple calculator application. The overriding goal is that the designs should be understandable. They should clearly indicate how the requirements will be implemented. Some of the more useful, and versatile, techniques are:

1. Architecture diagrams

These show the various components that will be used and how they fit together. For software built using an object-oriented language, a UML class diagram is essential. If a procedural language is used, a diagram showing the hierarchical relationship of the various modules and methods works well. You must have at least one architecture diagram in your design documentation — a diagram that shows how the various components of your software fit together.

2. Pseudocode

Pseudocode is extremely useful when designing individual methods. The use of pseudocode has all but replaced the common flow chart. If done well, it can be incorporated into comments used for source code.

3. Decision Tables

Decision tables are valuable when a course of action depends on a particular combination of values. A decision table shows all possible combinations of a set of values, and indicates what should happen for each combination.

4. Control Diagrams

These include such diagrams as state-transition diagrams and activity diagrams.

You should indicate how your requirements trace to your designs, by identifying each design component using the identifier(s) of the requirement(s) that component is designed to meet. The following hypothetical pseudocode shows an example:

```
// Pseudocode for allowing user to choose variant of poker to play
// (Requirement 2.0.0)
//
// Assumes user has chosen the "New Game" menu option

display GUI dialog showing available game variants (5-card draw,
7-card stud, Texas Hold'em)

if (choice == 5-card draw)
      initialize 5-card draw game
else if (choice == 7-card stud)
      initialize 7-card stud game
else if (choice == Texas Hold'em)
      initialize Texas Hold'em game
```

One or two simple diagrams won't cut it for your design documentation. Your project should be complex enough to require more than that. But there is no "target" number of diagrams necessary to get all the points for this part. I can't set a target because I don't know in advance what your project will be, and even if I did, I have no way of knowing what approach you will take to build the project. Keep in mind, the goal is for you to understand the problem you are solving before you start writing copious amounts of code. You need to convince me that you didn't just sit down at your source code editor and hack out the program with all the designs in your head, or made up as you went along.

3. Source Code Documentation

You are free to use whatever code documentation standard you are most comfortable with. The code writers on your team should try to use the same documentation standard, if possible. The only stipulations I have are:

1. Your source code must be documented to receive full points for that part of the project.

2. You should indicate the traceability from your designs to your code, by including the identifier of each design component (which should correspond to the identifier of the appropriate requirement) in the header comments of the

2. source code used to implement that component. For example:

```
/**
  * Query user for background pattern to use for cards
  * (Requirement 1.1.0)
  */
public int getCardBackground();
```

4. Testing Documentation

You need to have a plan for thoroughly testing your software that should include a combination of black-box and white-box tests. Beta testing is not required, although if you have the time and resources, it would probably be a good idea. Your test plan and your test cases should be documented thoroughly. It is vital to maintain a detailed record of every test (inputs, expected outputs, rationale) you perform, since you will need to be able to rerun these tests when you make changes to the software (regression testing). Testing will probably be the least glamorous and most tedious part of the entire project, but it is absolutely vital to ensure the software you build is meets its specification (i.e., it is verified), and is valid (i.e., it meets the customer's expectations).

You should indicate traceability of requirements (for black-box testing) and of source code (for white-box testing) for each test case. An example test case is shown below:

Test Case #	**Requirement Tested**	**Rationale**	**Input(s)**	**Expected Output**	**Pass/ Fail**
105	2.1.0	User should be allowed to play 5 card draw poker	User selects "5 card draw" when prompted for game variant to play	5 card draw layout is displayed	PASS

Note that you can design many of your black-box test cases concurrently as you flesh out the requirements. For example, if one of your requirements states that the software must be able to open a Microsoft Word document, you can design black-box test cases using a series of different Word documents as input. Obviously, white-box test cases cannot be designed until there is code to test, but you can design white-box test cases as the code becomes available.

5. Known Bugs and Issues

In the event your finished software still contains one or more bugs that you are aware of, you should provide a list of these bugs, and include any other issues such as performance problems, storage requirement problems, system incompatibility problems, etc.

3. Deliverable #3: The User Documentation

The User Documentation includes all documentation an end user would need to install and use the software.

1. User's Manual

You should assume that your software will be used by someone who has only a general idea of what the software is supposed to do. For example, if you're building a word processor, a user may know that your software is supposed to allow them to create letters and other such types of documents, but they won't necessarily know things like changing fonts, numbering pages, or that they can copy and paste sections of text using CTRL-C and CTRL-V.

1. Your user's manual should be written so that a user like this should be able to use your software effectively AFTER they have read the user's manual. Don't make any assumptions about the intuitive capabilities of the end users.

2. Although it does require more effort, supplementing the text in your user's manual with screenshots is highly recommended.

Most companies these days no longer put the user manuals for their software in a separate document. Instead they embed the user manual in the software itself, and the user can usually access it from a "Help" menu option. You may integrate your user's manual into the software if you wish, or you can have a separate document.

2. Installation Instructions

Whether you include an installer for your software or not, you should include a document that instructs the user how to install your software and launch it. This document can be part of your user manual, or it can be a separate document. Either way, the instructions for installing your software should be prominently located in your user documentation. You should include the following information (at a minimum):

1. System requirements

2. Installation instructions

3. Any known issues that might affect installation or use

4. Basic troubleshooting if the software won't install

4. Deliverable #4: The Final Product

This is the executable software, along with all data files, library files, drivers, and any other file necessary for the software to be used.

1. You should assume the end user might be someone who doesn't necessarily have a lot of technical computer experience; i.e., usability is a high priority. Don't make unfounded assumptions about the intuitive capacities of your end users.

2. You need to have an installer program that installs the application on the user's computer. This can be a full-fledged installation wizard or a simple batch file, depending on the requirements of your application. **Do not use a trial version of an installer, since they often only work for a limited period of time.** There have been cases where I was unable to install students' software because the trial installer they used had expired by the time they turned in their projects.

3. Depending on the size of your final project, you may be able to simply post it to Blackboard. If you do this, please package everything into a single zip file, and post the zip file. If your project is too large to post on Blackboard, you should make arrangements for a version that can be downloaded via some other means.

Grading Rubric

At the end of this document is a table that indicates how I plan to break down the project for grading, and how much each part will be worth. Notice that the executable program itself, while worth more than any other individual part, is only worth 20% of the overall project grade. This is because the whole purpose of this course is not for you to simply build a software application, but to follow a well-defined, documented process to build the software. Thus, the majority of the points will be given for evidence that you have followed such a process.

Completeness is the degree to which every detail has been accounted for in a document. The ideal is 100%, but in practice this is very difficult to achieve. One way to think about completeness is to ask the question of whether someone who was totally unfamiliar with the project would be able to take a given document and produce the next deliverable using only that document. For example, if your team were to hand off your designs to a team who knew nothing about your project, would that other team be able to implement those designs and arrive at the correct product?

Traceability refers to whether the requirements can be traced through the different phases of development. For example, suppose you have a requirement identified with the label 3.1.5. There should be a design artifact that covers the handling of that requirement. There should also be source code that implements that requirement. Finally, there should be one or more test cases that test the implementation of that requirement. Think of a requirement trace as a path that allows you to follow the requirement through design, implementation, and testing.

Presentation refers to how well the document is formatted. If spacing, indentation, etc. are not consistent, or if the document looks like a bad cut and paste job, points will be deducted.

A Few Words on Group Grades vs. Individual Grades:

One of the biggest problems with the group project — and hence, the one that generates the most student complaints, is the possibility that someone in a group may not contribute their fair share, for any of a number of reasons. When this happens, either the entire group risks getting a low project grade, and/or the person who does not contribute receives an unfairly generous grade at the expense of the other group members' hard work. This problem is inherent in all group projects, but it is exacerbated in an academic course setting since students have many demands on their time; e.g., jobs, family, etc., in addition to other coursework. I cannot demand that you give the group project top priority, nor am I able to directly verify whether all the individuals in a group participate equally.

However, with your cooperation I can help to mediate disputes and mitigate the risks associated with non-participation. If you perceive there is a problem with non-participation in your group, and your attempts to resolve the problem have failed, I strongly encourage you to email me ASAP. Keep in mind your group will be on a very tight schedule, with only about 13-14 weeks to take your project from the conception stage to the final product. Communication is therefore key. There must be frequent and productive communication within your team, or there will likely be problems.

If non-participation does become a recurrent problem within a group, I will use any information I receive from the group members, along with the group's org chart, weekly status updates, and the content and quality of the deliverables to determine whether to assign a single group grade, or assign individual grades for the project.

Since the group project is worth nearly 50% of the total points in the course, a low project grade will significantly affect a student's overall course grade. Therefore, in addition to notifying me of any problems, I also encourage everyone to temper their expectations and have understanding for the experience levels and circumstances of their other group members when dealing with non-participation issues. It is highly recommended that each group sticks with the technologies that are familiar, and is reasonable about the scope of their project. The less confusion and troubleshooting, the better.

Point Distribution

Item	Section Points	Section Totals
Project Plan and Umbrella Activities		**35**
Scope Statement, Tools and Standards	10	
Org Chart	5	
Gantt Chart	10	
Configuration Management Plan	10	
Programmer's Manual		**40**
Requirements Documentation		
Requirements are organized and each requirement has a unique identifier	10	
Completeness	15	
Understandability	10	
Presentation	5	
Design Documentation		**40**
Architecture diagram	5	
Other design documentation (decision tables, flow diagrams, sequence diagrams, pseudocode, etc.; tailored to fit your project)	10	
Traceability to Requirements	10	
Understandability	10	
Presentation	5	
Source Code		**30**
Present	10	
Commenting	10	
Traceability to design documentation	10	
Test Plan and Test Results		**50**
Thoroughness of test cases	15	
Each test case indicates inputs, expected output, rationale, and whether it passed	15	
Understandability	10	
Traceability to source code	10	
List of Known Bugs and Issues		**5**
Final Product		
Executable Program		**60**
Ease of Installation	10	
Works	20	
Meets Specification	20	
Performance	10	
User Manual		**35**
Understandability	15	
Completeness	15	
Presentation	5	
Installation Instructions		**5**
Total:		**300**