CMSI 4072: Senior Project II Written Homework Assignment 01

Problem 1.1, Stephens page 13: What are the basic tasks that all software engineering projects must handle?

Problem 1.2, Stephens page 13: Give a one sentence description of each of the tasks you listed in exercise 1

Requirements Gathering and Analysis – Understanding the needs of users and stakeholders to define what the software should accomplish.

System and Software Design – Creating architecture and detailed design plans to ensure the system meets the requirements efficiently.

Implementation (Coding) – Writing and developing the software based on design specifications.

Testing and Verification – Ensuring the software functions correctly through various testing methodologies, such as unit, integration, and system testing.

Deployment – Releasing the software for users, including installation, configuration, and distribution.

Maintenance and Support – Fixing bugs, updating features, and adapting the software to new requirements over time.

Project Management – Managing time, resources, risks, and team coordination to ensure successful software development.

Problem 2.4, Stephens page 27: Like Microsoft Word, Google Docs [sic] provides some simple change tracking tools. Go to http://www.google.com/docs/about/ to learn more and sign up [if you do not have an account already]. Then create a document, open the File menu's Version History submenu, select Name Current Version, and name the file 'Version 1'. Make some changes and repeat the preceding steps to name the revised document 'Version 2'. Now open the File menu's Version History submenu again but this time select See Version History. Click the versions listed on the right to see what changed between versions.

Document what you've noticed about the information you see, and how the differences between versions are displayed.

Tracking Changes: Google Docs allows you to track changes through Version History under the File menu. Each saved version is listed on the right side, showing timestamps and the user who made changes.

Change Highlights: When viewing an older version, Google Docs highlights the changes made compared to the previous version.

Restoring Versions: You can restore an older version if needed, reverting the document back to that state.

Collaboration Insights: Multiple users can make changes, and Google Docs shows edits by different collaborators in distinct colors.

Investigation: Compare this process to what you can do with GitHub versions. How are the two tools different? How are they the same?

Google Docs is more visual and user-friendly for tracking document edits, while GitHub provides a structured, code-focused version control system. GitHub supports branching, allowing parallel development, whereas Google Docs tracks a linear history of changes.

Problem 2.5, Stephens page 27: What does JBGE stand for and what does it mean?

JBGE (Just Barely Good Enough) is a *software engineering principle* that suggests software should be developed to meet requirements *efficiently* without unnecessary over-engineering.

Instead of aiming for *perfection*, which can lead to *excessive costs, delays, or complexity*, JBGE encourages delivering a *functional, reliable, and maintainable* product that satisfies user needs.

This concept aligns with *agile methodologies,* where iterative development and continuous improvement ensure that the product remains practical and useful without wasting resources.

Problem 4.2, Stephens page 78:

1. Use critical path methods to find the total expected time from the project's start for each task's completion.
    a. Compute earliest start and earliest finish

We calculate forward pass (starting from the first task) to determine the earliest time each task can be completed.

1. Independent Start Tasks (ES = 0, EF = Duration)
    ○ A: (0, 5), C: (0, 4), F: (0, 7), G: (0, 6), H: (0, 3)
2. Following Dependencies
    ○ I: (H) → (3, 6), J: (H) → (3, 6), D: (A, G, I) → max(5, 6, 6) = (6, 12), O: (A, G, J) → max(5, 6, 6) = (6, 11), E: (D) → (12, 19), B: (C) → (4, 9), L: (C, G) → max(4, 6) = (6, 12), M: (B, E, I) → max(9, 19, 6) = (19, 28), P: (O) → (11, 17), N: (B, J, O) → max(9, 6, 11) = (11, 26), K: (L) → (12, 17), Q: (K, M) → max(17, 28) = (28, 32), R: (K, N) → max(17, 26) = (26, 30)

b. Compute latest start and latest finish

Latest Finish (LF): The latest a task can be completed without delaying the project.

Latest Start (LS): LF−LF -LF− Task Duration.

We calculate backward pass (starting from the last task).

1. End Tasks (LF = Project End Time, LS = LF - Duration)
    ○ Q: (32, 28), R: (30, 26)
2. Back Propagation
    ○ K: (min(28, 26), 17) → (17, 12), M: (28, 19), N: (26, 11), L: (12, 6), O: (11, 6), P: (17, 11), B: (9, 4), E: (19, 12), D: (12, 6), A: (6, 0), G: (6, 0), I: (6, 3), J: (6, 3), C: (4, 0), H: (3, 0)

C. Identify the critical path

The Critical Path consists of tasks where ES=LS and EF=LF, meaning they have *zero slack (float)* and determine the project's total duration.

Critical Path:

- A (5d) → G (6d) → O (5d) → P (6d) → N (15d) → R (4d)
- Total Duration = 41 days

2. Find the critical path. What are the tasks on the critical path? A → G → O → P → N → R
3. What is the total expected duration of the project in working days? 41 working days

Problem 4.4, Stephens page 78: Build a Gantt chart for the critical path you drew in Exercise 2. Start on Wednesday, January 1, 2024, and don't work on weekends or the following holidays

Task	Duration (Days)	Start Date	End Date
A. Robotic Control Module	5	Jan 2	Jan 8
G. Rendering Engine	6	Jan 9	Jan 17
O. Zombie Editor	5	Jan 21	Jan 27
P. Zombie Animator	6	Jan 28	Feb 5
N. Zombie Library	15	Feb 6	Feb 28
R. Zombie Testing	4	Mar 3	Mar 7

showing workdays (■) and non-workdays (**X** for weekends, *H* for holidays)

Task | Jan | Feb | Mar

```
----------- | ---- | ---- | ----

A (5d)     | ■■■■■ |

G (6d)     |   ■■■■■■ |

O (5d)     |       ■■■■■ |

P (6d)     |         ■■■■■■ |

N (15d)    |             ■■■■■■■■■■■■■ |

R (4d)     |                     ■■■■ |

Holidays   | H  X   X H X  X  H  X  X  H   X  X  H  X X   H
```

Problem 4.6, Stephens page page 79: In addition to losing time from vacation and sick leave, projects can suffer from problems that just strike out of nowhere, like a bad version of *deus ex machina*. For example, senior management could decide to switch your target platform from Windows desktop PCs to the latest smartwatch technology. Or a pandemic, hurricane, trade war, earthquake, alien invasion, and so on could delay the shipment of your new servers. [Not that anything as far-fetched as a pandemic might occur, right?] Or one of your developers might move to Iceland, which is a real nice place to raise your kids up. How can you handle these sorts of completely unpredictable problems?

Unpredictable problems, sometimes called *"unknown unknowns,"* can seriously impact software projects. While you can't predict every possible issue, you can prepare for uncertainty using *risk management strategies*. Here are some ways to handle such unpredictable challenges:

1. Risk Assessment & Contingency Planning

- Identify *potential risks* (even unlikely ones) and develop contingency plans.

- Maintain *backup resources* (e.g., cloud infrastructure in case of hardware delays).
- *Cross-train team members* to prevent bottlenecks if key personnel leave or fall ill.

2. Agile & Flexible Development Approach

- Use *agile methodologies* to allow quick pivots when project scope changes.
- Maintain *modular architecture* so parts of the system can be repurposed if requirements shift (e.g., from PC to smartwatch).
- Implement *continuous integration (CI/CD)* so progress isn't lost when disruptions occur.

3. Communication & Stakeholder Management

- Keep *open communication* with senior management to anticipate shifts in strategy.
- Regularly *update project plans* based on the latest information.
- Ensure *team collaboration tools* (Slack, Jira, Confluence) are in place to support remote work if needed.

4. Time Buffers & Resource Allocation

- Add *buffer time* into the schedule for unforeseen disruptions.
- Have *redundant suppliers* for critical hardware to mitigate supply chain risks.

5. Remote Work & Distributed Teams

- Support *remote work infrastructure* to allow continuity if physical offices become inaccessible.
- Maintain *offshore or distributed teams* to balance workload if one location is impacted.

6. Insurance & Legal Protections

- Secure *business continuity insurance* for financial protection against major disruptions.

- Include *contract clauses* to address unexpected delays or force majeure events.

Problem 4.8, Stephens page 79: According to your textbook, what are the two biggest mistakes you can make while tracking tasks?

Failing to track progress accurately – If you don’t monitor tasks properly, you may not realize when a project is falling behind schedule. This can lead to unexpected delays, rushed work, or failure to meet deadlines. Regular updates and clear reporting are essential to avoid this mistake.

Tracking tasks at the wrong level of detail – If tasks are tracked at too high a level, you might miss important issues until it’s too late. On the other hand, if you track tasks in excessive detail, you waste time micromanaging instead of focusing on real progress. Finding the right balance is key to effective project tracking.

Problem 5.1, Stephens page 114: List five characteristics of good requirements.

Complete – The requirement fully describes the necessary functionality, leaving no gaps or ambiguities.

Unambiguous – The requirement is clear and can only be interpreted in one way, avoiding vague language.

Verifiable – The requirement can be tested or measured to confirm that it has been met.

Necessary – The requirement is essential to the project and not an unnecessary feature or preference.

Feasible – The requirement is realistic and achievable within the project's constraints (time, budget, technology).

Problem 5.3, Stephens page 114: Suppose you want to build a program called TimeShifter to upload and download files at scheduled times while you're on vacation. The following list shows some of the applications requirements.

- a. Allow users to monitor uploads/downloads while away from the office.

- b. Let the user specify website log-in parameters such as an Internet address, a port, a username, and a password.
- c. Let the user specify upload/download parameters such a number of retries if there's a problem.
- d. Let the user select an Internet location, a local file, and a time to perform the upload/download.
- e. Let the user schedule uploads/downloads at any time.
- f. Allow uploads/downloads to run at any time.
- g. Make uploads/downloads transfer at least 8 Mbps.
- h. Run uploads/downloads sequentially. Two cannot run at the same time.
- i. If an upload/download is scheduled for a time whan another is in progress, it waits until the other one finishes.
- j. Perform schedule uploads/downloads.
- k. Keep a log of all attempted uploads/downloads and whether the succeeded.
- l. Let the user empty the log.
- m. Display reports of upoad/download attempts.
- n. Let the user view the log reports on a remote device such as a phone.
- o. Send an e-mail to an administrator if an upload/download fails more than its maximum retry number of times.
- p. Send a text message to an administrator if an upload/download fails more than it's maximum retury umber of times.

For this exercise, list the audience-oriented categories for each requirement. Are there requirements in every category? [If not, state why not…]

1. User Interface Requirements – How users interact with the system.
2. Functional Requirements – What the system must do.
3. Performance Requirements – Speed, efficiency, and limitations.
4. Operational Constraints – System constraints and restrictions.

5. Security & Logging Requirements – Security features and logs.
6. Notification Requirements – Alerts and reporting.

Requirement	Category
a. Monitor uploads/downloads remotely	User Interface
b. Specify website log-in parameters	Functional
c. Specify upload/download parameters (e.g., retries)	Functional
d. Select location, file, and time	Functional
e. Schedule uploads/downloads	Functional
f. Run uploads/downloads at any time	Functional
g. Ensure transfers of at least 8 Mbps	Performance
h. Run uploads/downloads sequentially (no parallel tasks)	Operational Constraint
i. Queue uploads/downloads if conflicts occur	Operational Constraint
j. Perform scheduled uploads/downloads	Functional
k. Keep a log of all upload/download attempts	Security & Logging
l. Allow users to empty the log	Security & Logging
m. Display reports of upload/download attempts	User Interface
n. View log reports remotely	User Interface
o. Send an email if an upload/download fails beyond retry limits	Notification
p. Send a text if an upload/download fails beyond retry limits	Notification

Are All Categories Covered?

- Yes, all categories are represented.
- User Interface, Functional, and Logging requirements are well covered.

- Performance requirements only have one entry (8 Mbps transfer speed).
- Security-specific constraints (e.g., encryption, user authentication) are not explicitly mentioned, but might be implied under functional or logging requirements.

Problem 5.9, Stephens page 115

Figure 5-1 [right] shows the design for a simple hangman game that will run on smartphones. When you click the New Game button, the program picks a random mystery word from a large list and starts a new game. Then if you click a letter, either the letter is filled in where it appears in the mystery word, or a new piece of Mr. Bones's skeleton appears. In either case, the letter you clicked is grayed out so that you don't pick it again. If you guess all the letters in the mystery word, the game displays a message that says, "Congratulations, you won!" If you build Mr. Bones's complete skeleton, a message says, "Sorry, you lost."

Brainstorm this application and see if you can think of ways you might change it. Use the MOSCOW method to prioritize your changes.

The MoSCoW method categorizes changes into:

- Must-have – Essential for the game to function.
- Should-have – Important but not critical.
- Could-have – Nice-to-have features that enhance user experience.
- Won't-have – Features that are unnecessary or too complex for now.

Change Idea	MoSCoW Priority	Reasoning
1. Prevent duplicate letter selections	Must-have	Essential for game mechanics (prevents unfair guessing).
2. Add different difficulty levels (Easy, Medium, Hard)	Should-have	Adds replay value by adjusting word length and number of mistakes allowed
3. Add a hint system (e.g., reveal one letter or category of word)	Should-have	Helps casual players, making the game more engaging.
4. Include a timer for a speed-based challenge mode	Could-have	Increases challenge but isn't essential.
5. Add a leaderboard or scoring system	Could-have	Encourages competition but isn't necessary for basic gameplay.
6. Customize Mr. Bones (different skeleton styles or themes)	Could-have	Enhances visuals but doesn't impact gameplay.
7. Multiplayer mode (e.g., two players take turns guessing letters)	Could-have	Fun feature but more complex to implement.
8. Animated Mr. Bones (funny reactions to correct/wrong guesses)	Could-have	Adds charm but isn't critical.
9. Speech-to-text letter guessing for accessibility	Won't-have	Useful but not a priority in early versions.
10. Background music and sound effects	Could-have	Improves user experience but isn't essential.
11. Themed word categories (e.g., Animals, Movies, Countries)	Should-have	Keeps the game fresh and engaging.
12. Different victory/defeat animations for Mr. Bones	Could-have	Adds flair but doesn't affect gameplay.
13. Online mode with random opponents	Won't-have	Too complex for an initial version.

Must-have: Core mechanics (prevent duplicates, functional gameplay).

Should-have: Features improving engagement (difficulty levels, hints, themed words).

Could-have: Fun but non-essential improvements (animations, leaderboards, music).

Won’t-have: Complex or unnecessary features (online mode, speech-to-text).