Reading Summaries

Chapter 1 - Software Engineering Basics

Software engineering is the disciplined application of engineering principles to the creation of software. Unlike casual programming, software engineering involves managing the full lifecycle of a project: planning, gathering requirements, designing, implementing, testing, deploying, and maintaining the product.

The key insight of Chapter 1 is that most software projects fail not because of technical problems, but because of poor planning and management. Common failure modes include unclear requirements, unrealistic schedules, insufficient testing, and poor communication between developers and stakeholders.

Stephens stresses the importance of choosing an appropriate development model. Different project types (fixed scope vs. evolving requirements) call for different approaches such as waterfall, spiral, or agile. There is no universal methodology.

Chapter 2 - Project Management Essentials

Chapter 2 covers the tools and processes used to manage software projects effectively. Version control is central to this discussion, with systems like Git letting teams track changes, collaborate, branch, and roll back to previous states. Without version control, managing even a medium sized codebase becomes chaotic.

JBGE (Just Barely Good Enough) documentation is introduced as a pragmatic principle: write enough documentation to be useful, but no more. Over-documentation wastes time and quickly becomes outdated; under-documentation leaves teams without critical context.

The chapter also covers risk management (identifying and mitigating potential problems before they occur) and team communication practices such as daily stand-ups and written status reports.

Chapter 3 - Development Models

Stephens walks through the major software development lifecycle (SDLC) models: Waterfall, Sashimi, Spiral, Incremental, RAD (Rapid Application Development), Scrum, and Kanban. Each model has trade-offs:

- Waterfall: sequential phases, easy to manage but inflexible to changes late in development.
- Spiral: iterative with explicit risk assessment each cycle; good for large, risky projects.
- Scrum/Agile: short sprints, frequent delivery, high flexibility, best when requirements evolve.
- Kanban: continuous flow with visual boards; good for teams with unpredictable workloads.

Choosing the right model depends on how well-defined the requirements are, how likely they are to change, team size, and delivery constraints.

Chapter 4 - Project Scheduling

Chapter 4 focuses on how to estimate and schedule software projects using tools such as network diagrams and Gantt charts. The Critical Path Method (CPM) is the main scheduling technique covered.

CPM works by mapping out all tasks, their durations, and their dependencies. From this network, you can compute each task's Earliest Start (ES), Earliest Finish (EF), Latest Start (LS), and Latest Finish (LF). Tasks where LS = ES (i.e., slack = 0) form the critical path, the longest chain of dependent tasks that determines the minimum project duration. Delaying any task on the critical path delays the entire project.

Gantt charts translate the network diagram into a calendar view, making it easy to communicate schedules to stakeholders, track progress, and spot bottlenecks.

Chapter 5 - Requirements

Requirements are the foundation of a software project. They define what the system must do (functional requirements) and how it must perform (non-functional requirements). Poor requirements are the #1 cause of project failure.

Stephens identifies several audience-oriented requirement categories: business requirements (why the system is needed), user requirements (what users need to do), functional requirements (what the system must do), non-functional requirements (performance, reliability, security), and implementation requirements (constraints on how it must be built).

The MOSCOW prioritization method (Must have, Should have, Could have, Won't have) helps teams decide which features are essential for initial release and which can be deferred. This is especially important when managing scope creep.

Good requirements are clear, unambiguous, consistent, prioritized, and verifiable. Vague, ambiguous, or contradictory requirements inevitably lead to rework and schedule overruns.

HW Problems

Problem 1.1

According to Stephens (p. 13), every software engineering project must handle the following basic tasks:

- Requirements gathering
- High-level design
- Low-level design
- Development
- Testing
- Deployment
- Wrap-up

Problem 1.2

Requirements gathering: Determining what the customer needs the system to do and documenting those needs clearly.

- High-level design: Defining the overall architecture of the system, including major components and how they interact.
- Low-level design: Specifying the detailed internal design of each component, including data structures and algorithms.
- Development: Writing the actual source code that implements the design.
- Testing: Testing how multiple components work together to ensure they interact correctly.
- Deployment: Installing and configuring the finished software in its production environment for end users.
- Wrap-up: Reviewing the project afterward to identify what went well and what went poorly so future projects improve.

Problem 2.4

When using Google Docs' Version History feature, such as “Version 1" and "Version 2", appear in a sidebar on the right side of the screen. The interface highlights differences between versions using color-coding: additions appear highlighted in green, while deletions are struck through in red. Clicking a version restores a preview of the document at that point in time. You can see exactly who made each change and when, down to the timestamp. The restore function allows you to roll back the entire document to a prior state.

Compared to GitHub, both tools share the concept of snapshots of a document at a point in time, named/tagged versions, visual diffs showing what changed, attribution of who made the change, and the ability to restore previous versions. However, there are important differences:

- Granularity: GitHub tracks changes at the line level across potentially hundreds of files simultaneously; Google Docs tracks changes at the character/word level within a single document.
- Branching: GitHub supports branching and merging, allowing parallel development streams to exist and be reconciled. Google Docs has no concept of branching.
- Collaboration model: GitHub uses a commit-based model where changes are explicitly staged and committed with a message. Google Docs auto-saves continuously, and version names are optional.
- Diff display: GitHub's diff view is designed for code (line-by-line), while Google Docs' diff is designed for prose (inline character-level highlighting).

Problem 2.5

JBGE stands for "Just Barely Good Enough." It refers to a documentation philosophy that advocates writing the minimum amount of documentation necessary to be effective, rather than exhaustively documenting everything. The idea is that documentation takes time to write and

maintain, and over-documentation quickly becomes outdated, bloated, and ignored. By focusing only on what is genuinely needed, whether to onboard new team members, explain non-obvious design decisions, or define external APIs, teams can keep their documentation lean, accurate, and actually useful.

Problem 4.2

Part (a)

Task	Duration	Predecessors	Earliest Start	Earliest Finish
A. Robotic control module	5	–	0	5
B. Texture library	5	C	4	9
C. Texture editor	4	–	0	4
D. Character editor	6	A, G, I	6	12
E. Character animator	7	D	12	19
F. Artificial intelligence	7	–	0	7
G. Rendering engine	6	–	0	6
H. Humanoid base classes	3	–	0	3
I. Character classes	3	H	3	6
J. Zombie classes	3	H	3	6
K. Test environment	5	L	12	17
L. Test environment editor	6	C, G	6	12
M. Character library	9	B, E, I	19	28
N. Zombie library	15	B, J, O	11	26
O. Zombie editor	5	A, G, J	6	11
P. Zombie animator	6	O	11	17
Q. Character testing	4	K, M	28	32

Task	Duration	Predecessors	Earliest Start	Earliest Finish
R. Zombie testing	4	K, N	26	30

Part (b)

The critical path consists of all tasks with zero slack (Latest Start = Earliest Start). Performing a backward pass from the project end (day 32):

Critical path tasks (slack = 0): G → (parallel with H → I) → D → E → M → Q

More precisely, there are two parallel chains that both feed into task D:

- Chain 1: G (days 0-6) → Rendering Engine
- Chain 2: H (days 0-3) → I (days 3-6) → Humanoid Base Classes → Character Classes

Both chains complete on day 6, which is the earliest D can start. From D:

- D (days 6-12) → E (days 12-19) → M (days 19-28) → Q (days 28-32)

All other tasks have positive slack and are therefore not on the critical path.

Part (c)

The total expected duration of the project is 32 working days.

Problem 4.4

Starting Wednesday, January 1, 2024. Jan 1 is a holiday, so the first working day is Tuesday, January 2. Weekends and the following holidays are excluded: New Year's Day (Jan 1), MLK Day (Jan 20), Valentine's Day (Feb 14), President's Day (Feb 17), Alien Overlord Appreciation Day (Mar 26), Income Tax Day (Apr 15).

Task	Start Day	Start Date	End Day	End Date	Dur.
G - Rendering Engine	1	Jan 2	6	Jan 9	6
H - Humanoid Base Classes	1	Jan 2	3	Jan 4	3
I - Character Classes	4	Jan 5	6	Jan 9	3
D - Character Editor	7	Jan 10	12	Jan 17	6
E - Character Animator	13	Jan 18	19	Jan 26	7
M - Character Library	20	Jan 29	28	Feb 8	9
Q - Character Testing	29	Feb 9	32	Feb 15	4

Notes: Jan 20 (MLK Day) falls during Task D. It is skipped, stretching D from Jan 10 to Jan 17 rather than Jan 16. Feb 14 (Valentine's Day) falls during Task Q. It is skipped, so Q ends Feb 15 instead of Feb 14. The project completes on Friday, February 15, 2024.

Problem 4.6

Stephens acknowledges that software projects are routinely disrupted by completely unpredictable events, such as pandemics, hardware failures, key staff departures, platform pivots, natural disasters, and so on. The recommended strategies for handling these include:

- Build schedule buffers: Don't plan for 100% utilization. Historical data suggests people work at 50-75% productivity on any given project task due to meetings, context switching, and other interruptions. Padding estimates accordingly absorbs minor shocks.
- Risk planning: Proactively identify potential risks before they occur, estimate their likelihood and impact, and plan mitigation strategies in advance. While you can't predict every disruption, you can have contingency plans ready.
- Tracking and re-planning: When an unexpected event occurs, update the project schedule immediately. Re-run the critical path analysis with revised estimates to understand the new project end date and communicate changes to stakeholders promptly.
- Maintain flexibility in scope: Use MOSCOW prioritization so that if disaster strikes, you can cut "Could have" and "Should have" features to protect the "Must have" deliverables.
- Keep stakeholders informed: Surprises are worse when they come late. Early communication gives stakeholders time to adjust expectations and resources.

Problem 4.8

According to Stephens, the two biggest mistakes you can make while tracking tasks are:

- Not taking action when a task slips (you at least need to monitor it closely and intervene early)
- Adding more people to the task and assuming that automatically reduces the duration (ramp-up can make it take longer).

Problem 5.1

- Clear: A requirement should be easy to understand. It should use straightforward language so all stakeholders (developers, managers, customers) interpret it the same way.
- Unambiguous: The requirement must have only one possible interpretation. Vague terms like “fast,” “user-friendly,” or “efficient” should be avoided unless precisely defined.

- Consistent: Requirements must not contradict one another. If one requirement says the system allows simultaneous uploads and another says only one upload may run at a time, they are inconsistent.
- Prioritized: Each requirement should indicate its relative importance (for example, Must, Should, Could, Won't). This helps manage scope and decide what can be deferred if time or budget becomes constrained.
- Verifiable: It must be possible to test whether the requirement has been met. A requirement like "the system should be reliable" is not verifiable unless reliability is defined with measurable criteria.

Problem 5.3

Requirements can be categorized into audience-oriented categories: Business Requirements (BR), User Requirements (UR), Functional Requirements (FR), Non-Functional Requirements (NFR), and Implementation Requirements (IR).

Req.	Requirement Text	Category
a	Allow users to monitor uploads/downloads while away from the office.	UR
b	Let the user specify website log-in parameters (address, port, username, password).	FR
c	Let the user specify upload/download parameters such as number of retries.	FR
d	Let the user select an Internet location, local file, and time to perform upload/download.	FR
e	Let the user schedule uploads/downloads at any time.	FR
f	Allow uploads/downloads to run at any time.	FR
g	Make uploads/downloads transfer at least 8 Mbps.	NFR (Performance)
h	Run uploads/downloads sequentially. Two cannot run simultaneously.	IR / FR
i	If an upload/download is scheduled during another in progress, it waits.	FR
j	Perform scheduled uploads/downloads.	FR
k	Keep a log of all attempted uploads/downloads and whether they succeeded.	FR
l	Let the user empty the log.	FR
m	Display reports of upload/download attempts.	FR
n	Let the user view the log reports on a remote device such as a phone.	UR / NFR (Portability)

Req.	Requirement Text	Category
o	Send an email to an administrator if upload/download fails more than max retry count.	FR
p	Send a text message to admin if upload/download fails more than max retry count.	FR

Are there requirements in every category?

No, there are no explicit Business Requirements (i.e., no statement explaining why the business needs this system or what business objective it serves). Every requirement listed describes either a user capability, a system behavior, a performance constraint, or an implementation rule, but none explains the higher-level business goal, such as "The system must reduce the cost of manual file transfers by enabling unattended automated scheduling." A missing business requirements section is a common gap in early-stage requirement sets.

Problem 5.9 - Hangman Game Brainstorm (MOSCOW Method)

Starting from the base Hangman game described in Figure 5-1, here is a brainstormed list of possible changes and enhancements, prioritized using the MOSCOW method:

Must Have (core requirements):

- Display the mystery word as blank spaces and reveal correctly guessed letters in position.
- Track and display incorrectly guessed letters so the player doesn't repeat them.
- Progressively build the skeleton graphic with each wrong guess.
- Detect and announce win/loss conditions at the appropriate time.

Should Have (important for a good user experience, but not game breaking if absent):

- Show the number of wrong guesses remaining so the player can assess risk.
- Display a hint or category for the mystery word (e.g., "Animal," "Movie Title") to make the game more engaging.
- Allow the player to request a new game without restarting the app.
- Keep track of wins and losses across games

Could Have (nice to have but lower priority for initial release):

- Let users choose difficulty levels which affect word length or obscurity.
- Add sound effects for correct guesses, wrong guesses, wins, and losses.
- Support multiple visual themes (skins) for the skeleton
- Allow players to submit custom words for use in the game.
- Include a timer that creates urgency for more experienced players.

Won't Have (out of scope for this version):

- Multiplayer functionality where one player picks the word and another guesses.
- Online leaderboards or integration with social media platforms.
- In-app purchases or premium word packs.
- Accessibility features such as voice-over support or high-contrast mode (these should ideally be prioritized but are listed here given typical resource constraints on a first release).