Project (Theory)

Learning Software Engineering by Doing

ASE

ASE @ Northern Kentucky University

Chapter 1:
Meet the Team

First day of Software Engineering Team Project. Students gather to form teams for the semester-long project. Some have experience, others are just beginning their journey.

Quick Reference Guide

Team Formation Essentials

- Form teams by the end of Week 2
- Select a team leader
- Identify each member's strengths
- Establish communication channels

Core Principles Introduced

- Rule 1: No surprises (communicate early)
- Rule 2: Show no emotions (stay professional)
- Software Engineering ≠ Just Coding
- Managing complexity in a team using the rules and tools is key

Team Formation

Prof. Cho

"Welcome! This semester, you'll work in teams to build a real application.[1] Let's form teams and get to know each other today."

Prof. Cho

"If a student cannot or does not find a team for whatever reasons, I'll assign the student randomly to a team. So, everyone joins the team by the end of week 2."

Prof. Cho

"One more thing - each team needs a leader. If you haven't led a team before, this is your chance. I strongly encourage new leaders this semester.

After the class, Tim looks around and finds Jane.

---

[1]There are two projects: team and individual. This story is mainly for the team project, with an introduction of the individual project, but the tools and rules for the team project can be used for the individual project.

Tim

“Hi, Jane. Looking for a team? I like making things and understanding how they work, but I’ve never done a team project before.”

Jane

“Hello, Tim. Yeah, I was looking for teammates! I love solving puzzles - I’ve built some small applications on my own. Sometimes I feel like building software is just another kind of puzzle.”

A junior walks over.

Ken

“Hey, I’m Ken. Need another teammate? I have been developing applications since high school, and I completed an ASE course as a team member.

Tim

“Perfect!

Understanding Team Roles

Team Member Profiles

Member	Strengths	Growth Areas
Tim	Curious, eager to learn, asks good questions	First ASE course, needs process guidance
Jane	Problem-solver, independent developer	Only individual project experience
Ken	Prior team experience, understands process	First time as team leader

Why diverse skills matter: Teams succeed when members bring different perspectives. Tim's curiosity drives learning, Jane's problem-solving tackles technical challenges, and Ken's experience provides structure.

Selecting a Team leader

Jane

"What is the team leader's role?"

Ken

"It's mainly managing projects. The team leader should manage the schedule, report weekly progress, manage tests, manage documents, and evaluate team members."

Tim

"I'm not familiar with managing schedules and weekly progress reports."

Ken

"You'll learn about them as soon as we start the agile process. Anyway, we should select a team leader."

Jane

"Ken, as you already finished a team project before, how about leading the team this time?"

Ken pauses, but after a short thought, he says:

Ken

“Well… last time I was just a team member. But sure, I can try! I learned what NOT to do, at least.”

Tim

“I’m okay with that. We will support you!”

Jane

“Then, what is the team member’s role?”

Ken

“Basically, what the real-world software engineers do: define features, design and implement the features, and make tests/documents.”

Tim

“So, are you going to tell me what to do? I also expect you to teach me how to do it.”

Ken

"Well. That's not exactly how software engineering works. We will discuss what to solve together, but it's your job to find the correct solution."

Jane

"Tim, don't worry, you will learn how to solve problems effectively as you learn various problem-solving skills and technologies in this and other courses."

Ken

"Yes, exactly. You will learn by doing projects."

Tim

"And, maybe, by making mistakes and learn from them."

Tim smiles.

Team Leader and Member Responsibilities

Aspect	Team Leader	Team Member
Schedule	Creates & maintains timeline	Commits to personal deadlines
Communication	Reports to class weekly	Updates individual contribution for the team leader
Quality	Ensures testing & documentation	Writes unit tests alone, but writes other tests with the leader
Coordination	Resolves conflicts, assigns tasks	Collaborates, helps teammates
Accountability	Overall project success	Individual task completion

Leadership Tip:

First-time leaders often worry about "knowing everything."

Reality: Good leaders facilitate communication and keep the team organized. You don't need all the technical answers—you need to know when to ask for help and how to coordinate the team effectively.

Common Mistake #1: The 'Hero Leader' Anti-Pattern:

❌ Leader does all the work to “ensure quality.”

✅ Leader distributes work and supports team success

Why? If the leader does everything, team members don’t learn, and the project fails when the leader is unavailable. Leadership is about enabling others, not replacing them.

Key Takeaways

- **Team leader** coordinates and manages; doesn’t do all the work
- **Prior experience** helps but isn’t required for leadership
- **Learning** from past mistakes is valuable leadership preparation
- **Support** from team members makes leadership sustainable

The Mentor Appears

A senior student approaches them.

Julie

"Hey, new team? I'm Julie, finishing my capstone project. I'll be around if you need advice. I learned a lot from these courses—some from mistakes, some from breakthroughs."

Tim

"Mistakes? What kind?"

Tim remembers he just talked about 'learning from mistakes.'

Julie

"All kinds! Not writing requirements, no architecture, no testing, trying to solve unsolved problems alone and wasting time… basically everything you can do wrong."

Julie

"But we learned and recovered. That's what these courses teach - not just coding, but the whole **software engineering rules and tools in a team**."

Tim

"Oh! I remember we already discussed that software engineering is not just about writing code."

Julie

"Coding or programming is maybe, at most, 30% of software engineering, in most companies. The rest is understanding problems, designing solutions, working as a team through communication, and most importantly, **managing complexity** to produce **high-quality software products**."

Software Engineering vs Just Coding

Activity	Just Coding	Software Engineering

Time Spent	90% coding	Less than 30% coding, more than 70% other activities
Focus	“Make it work”	“Make it maintainable (bug fix) & extensible (add features)”
Planning	Minimal, jump to code	Requirements, architecture, design first
Testing	Manual testing, if any	Automated tests, TDD, CI/CD
Documentation	Code comments only	Architecture docs, API specs, READMEs
Teamwork	Individual work	Collaboration, code reviews, communication
Problem Solving	Trial and error	Systematic analysis & design

Jane

“Managing complexity? I remember Prof. Cho mentioned it when he discussed ASE courses.”

Julie

"Yes, software engineering is about managing complexity to deliver high-quality software products, and software engineers are **problem solvers** by **managing complexity** in a **team**."

Tim

"What is the high-quality software? What is the software product?"

Julie

"Good question! High-quality software is easy to fix bugs and add features, and software products are products people pay to use."

Jane thinks

"I see. Software engineers don't just make any software. It should be easy to fix and extend, and also useful so people are willing to pay to use."

Setting Expectations

Prof. Cho

"Ken's team with Tim and Jane - good! Ken, first time as team leader?"

Ken

"Yes. But I was a team member last semester and want to learn how to lead a team successfully."

Prof. Cho

"Excellent! This semester, you'll build a complete web application.[2] - something you'd be proud to show others and yourself. You'll learn requirements, architecture, testing, and the full development process."

[2]This is an example; each ASE course has different objectives for team and individual projects.

Jane

“What if we can’t figure something out, I mean, in trouble or something?”

Prof. Cho

“You have your team, me, and mentors[3]

Julie

“Every senior struggled at some point. The hardest part is learning that software engineering is more than just coding or programming - it’s communication, teamwork, and systematic problem-solving.”

Ken

“Any advice for us starting?”

[3]In this story, we have Julie as a mentor. But students can ask anyone who has finished projects before. The real failure would be not trying.“

Julie

"Maybe the #1 team rule will help: **No surprises**. Communicate early and often. When you're stuck, say something immediately. Don't wait until the deadline. And trust the process - it seems like overhead at first, but it saves time later."

Prof. Cho

"Also don't forget the #2 team rule: **Show no emotions**.[4]"

Prof. Cho

"As a team, you will experience all kinds of issues among team members. Under any circumstances, we professionals should not show emotions, but focus on problem solving."

[4]Do not show negative emotions to other team members. Focus on problem-solving. We all know that showing emotions can never solve any real problems.

> **Prof. Cho**
>
> "Stay calm, and act like a professional. And the issues will be resolved sooner or later, without invoking any unnecessary issues."

The Two Core Team Rules

Rule #1: No Surprises:

What it means:

- Communicate blockers immediately, not at the deadline
- Share progress (or lack thereof) as quickly as possible
- Alert the team when you need help
- Update task status in project boards regularly

Why it matters: Early communication allows for replanning. Surprises at the deadline create crisis mode and destroy team trust.

Example scenarios:

- ✅ "I'm stuck on the authentication API—can someone pair with me this afternoon?"
- ❌ **[Silent for 3 days, then at deadline]** "I couldn't get it working, sorry"

Rule #2: Show No Emotions:

What it means:

- Don't react defensively to code reviews or feedback
- Don't express frustration at teammates in meetings
- Separate technical disagreements from personal conflicts
- Focus on problem-solving, not blame

Why it matters:

Emotional reactions create toxic team dynamics. Professional composure enables productive conflict resolution.

Example scenarios:

- ✅ "I see your concern. Let me explain my reasoning, and let's find the best solution together."
- ❌ "You always criticize my code! Why don't you write it yourself?"

Common Mistake #2: Waiting Too Long to Ask for Help:

❌ "I spent 10 hours stuck on this bug, but didn't want to bother anyone."

✅ "After 30 minutes stuck, I posted in our team chat asking if anyone's seen this before."

Why? The "30-minute rule"[5]: If you're stuck for 30 minutes, ask for help. Someone might solve it in 5 minutes. Waiting wastes time and delays the whole team.

Professional Communication Examples

Unprofessional	**Professional**
"This is stupid, it doesn't work!"	"The current approach isn't working. Let's debug together."
"Why did you break my code?"	"I see my tests are failing. Can we review the recent changes?"
No response to messages[6]	"Got your message. Investigating. Will update in 1 hour."
"That's a terrible idea"	"I have concerns. Here's an alternative approach to consider..."

[5]Or 1 hour rule, 2 hours rule, set the time for yourself. This is just an example.

[6]This is one of the most unprofessional behaviors as a professional software engineer. We must respond to any communication under any circumstances.

Ready to Begin

Prof. Cho

"Next class, we'll start with what software engineering really means. Ken's team - get to know each other this week. Brainstorm what kind of application you want to build."

As they leave, the team feels excited and nervous.

Ken

"Coffee tomorrow to brainstorm?"

Tim

"Definitely! I have so many questions."

Jane

"I think we picked a good team."

Ken

"My team project experience, Jane's problem-solving, Tim's curiosity, and Julie as our mentor. We've got this!"

The team doesn't know it yet, but they are about to learn that software engineering is about:

- Understanding problems in the real world
- Start with why, not what or how
- Designing solutions
- Building quality into every step
- Working effectively as a team
- Managing complexity systematically

Their journey *from* students who can **code** *to* engineers who can **solve problems** has just begun.

Chapter Summary: Key Concepts

Core Concepts Introduced

Concept	Key Point
Team Dynamics	Diverse skills create stronger teams; leadership is about coordination, not doing everything
Software Engineering	At most 30% coding, more than 70% other activities (requirements, architecture, testing, communication)
Managing Complexity	Core skill that enables scaling from 500 to 500,000 lines of code
Rule 1	No Surprises—communicate early, often, and honestly
Rule 2	Show No Emotions—stay professional, focus on solutions
Problem First	Understand the problem and users before jumping to code
Process Over Heroics	Systematic approaches scale; individual heroics don't

Notice

This storytelling explains the software engineering process and related ideas.

In the real classroom team project:

1. Teams can be formed in a variety of ways: for example, the professor can assign students to teams.
2. Each ASE course has a goal, and a corresponding team project can be given: for example, in ASE 420, we use Tetris (or a similar game) as the team project.
3. In most cases, students are given prototypes before requirements, and each team should finish MVP at the end of the 1st sprint (iteration) of the project.
4. Before the start of the project, students are ready to dive into the project through the discussion of the technologies and ideas for the project in the classroom meetings and assignments.

Chapter 2:
Road Map

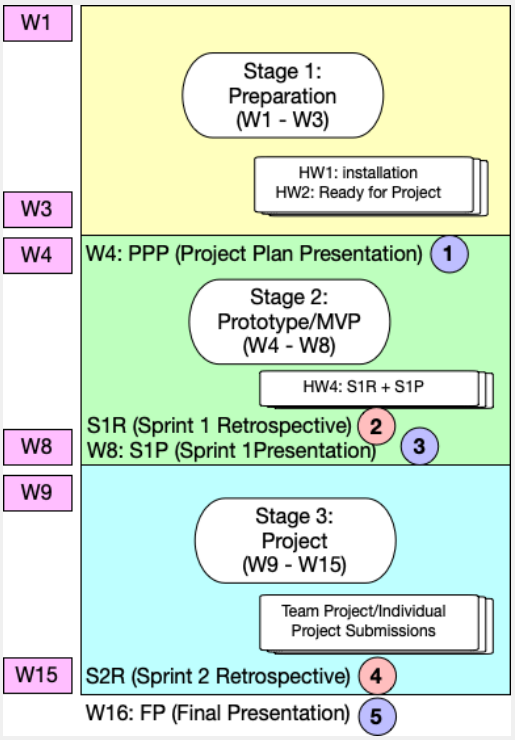
Now that Tim, Jane, and Ken have formed a team and selected Ken as the team leader, they need to plan their two sprints with five ceremonies and three presentations.

Quick Reference Guide

Five Ceremonies with Three Presentations

1. PPP (Project Plan **Presentation**)
2. S1R (Sprint 1 Retrospective)
3. S1P (Sprint 1 **Presentation**)
4. S2R (Sprint 2 Retrospective)
5. FP (Final **Presentation**)

Three Stages



- Stage 1: Preparation
  - Weeks: W1 - W3
  - Goal: **PPP (W4)**
  - Related Assignments
    - HW1: Team setup & installation
    - HW2: Ready for project

- Stage 2: Prototype/MVP
  - Weeks: W4 - W8
  - Goal: **S1P (W8)**
  - Related Assignment
    - HW4: S1R + S1P

- Stage 3: Project
  - Weeks: W9 - W15
  - Goal: **FP (W16)**
  - Related Assignments
    - Team project submission
    - Individual project submission

16 Weeks Schedule

Prof. Cho shows the 16-week schedule on the screen[7].

Stages	Weeks (M/W)	Sprints	Topics (Theory)	Modules	HW
Stage 1 (Preparation)	Week1 (1/12, 1/14)		Introduction		
	Week 2 (~~1/19~~, 1/21)	Project Discussion** Team Setup*	Project Preparations		HW1 (Installation) (1/25)*
	Week 3 (1/26, 1/28)		Project Preparations		HW2 (Project Plan) (2/1)*
Stage 2 (Prototype /MVP)	Week 4 (2/2, 2/4)	**Project Plan Presentation (2/2)** Sprint1 - 1	High-Level Languages	Module 1	
	Week 5 (2/9, 2/11)	Sprint1 - 2	High-Level Languages		
	Week 6 (~~2/16~~, 2/18)	Sprint1 - 3	High-Level Languages		
	Week 7 (2/23, 2/25)	Sprint1 - 4	Software Process		HW3 (Module 1) (2/27)**
	Week 8 (**3/2, 3/4**)	**Sprint 1 Presentation (3/2)**	**Midterm (3/4), on Canvas no class no office hours**		HW4 (Sprint 1 Retrospect + Sprint 2 Plan)(3/1)*
Stage 3 (Project)	Week 9		Spring Break (no class) **No classroom/office hours**		
	Week 10 (3/16, 3/18)	Sprint 2 – 1	Git	Module 2	
	Week 11 (3/23, 3/25)	Sprint 2 – 2	Git		
	Week 12 (3/30, 4/1)	Sprint 2 – 3	Security		
	Week 13 (4/6, 4/8)	Sprint 2 – 4	Security		
	Week 14 (4/13, ~~**4/15**~~)	Sprint 2 – 5	**Midterm (4/15, on Canvas no class/office hours)**	Module 3	HW5 (Module 2) (11/22)***
	Week 15 (4/20, 4/22)	Sprint 2 – 6 **Project submissions deadline (4/26)***	Deployment		
	Week 16 (4/27, 4/29)	**Presentation (4/27, 4/29)**			

> **Prof. Cho**
>
> "We have 16 weeks in this semester, in one semester we have three project stages, and three modules for classroom discussions."

[7] This is an example; each course differs in detail, but the structure is almost the same

Prof. Cho

"For classroom meetings, we have about one or two week break, two midterms, and one week final presentation. So, we have about 12 - 13 weeks for classroom meetings divided into three modules."

Prof. Cho

"And for projects, there are three stages: preparation, prototype/MVP, and project."

Tim thinks

"So, three modules for theories, and three stages for projects. Interesting!"

Prof. Cho

"In the first project stage, we are ready to dive into projects, then we build an MVP, Most Viable Product, from the prototype in the second stage, and we call it the 1st sprint, and finally, we build high-quality software in the final stage that we call the 2nd sprint."

Jane thinks

"I see, the yellow one is the 1st stage, and the green and blue ones are for the 2nd and final stage."

Tim

"What is the sprint?"

Prof. Cho

"You can think of it as an iteration. In other words, we have two iterations to build and deploy high-quality software products."

Prof. Cho writes five ceremonies on the whiteboard.

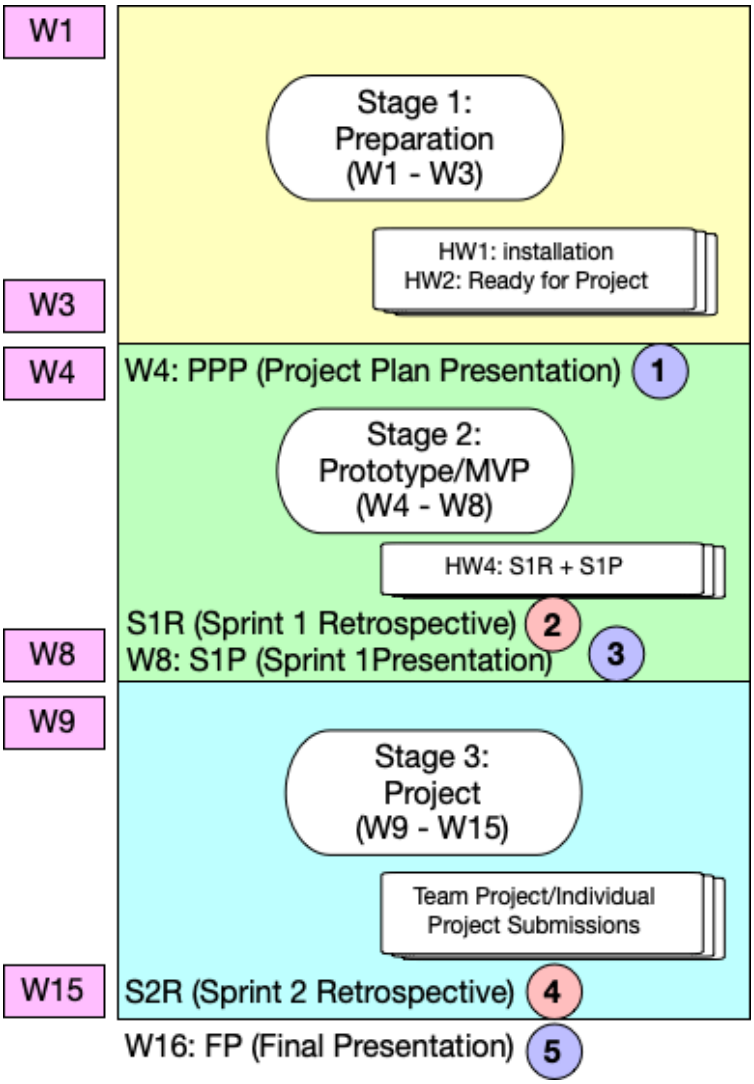
1. PPP (Project Plan **Presentation**) <- Stage 1 Goal
2. S1R (Sprint 1 Retrospective)
3. S1P (Sprint 1 **Presentation**) <- Stage 2 Goal
4. S2R (Sprint 2 Retrospective)
5. FP (Final **Presentation**) <- Stage 3 Goal

Prof. Cho

“Each stage has a goal, and we have five ceremonies with three presentations to prepare and close each stage.”

Three Project Stages

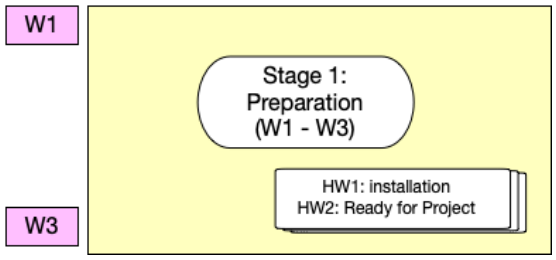
Prof. Cho shows a project diagram.



Prof. Cho

"Let's talk about each stage one by one."

Stage 1: Preparation



- Stage 1: Preparation
  - Weeks: W1 - W3
  - Goal: **PPP (W4)**
  - Related Assignments
    - HW1: Team setup & installation
    - HW2: Ready for project

1. ***PPP (Project Plan Presentation)*** *<- Goal*
2. *S1R (Sprint 1 Retrospective)*
3. *S1P (Sprint 1* ***Presentation****)*
4. *S2R (Sprint 2 Retrospective)*
5. *FP (Final* ***Presentation****)*

Prof. Cho

"The goal of the 1st stage is to be ready for the project. Specifically, teams (1) upload all of their project goals and plans on GitHub & Canvas pages, and (2) present them on the Project Plan Presentation (PPP) in week 4.

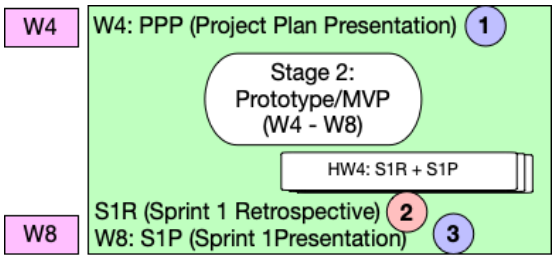
Jane

"So, we prepare for the two iterations, I mean sprints, during weeks 1, 2, and 3."

Prof. Cho

"Yes, HW1 and HW2 will help you prepare for the projects."

Stage 2: Prototype/MVP



- Stage 2: Prototype/MVP
  - Weeks: W4 - W8
  - Goal: **S1P (W8)**
  - Related Assignment
    - HW4: S1R + S1P

Prof. Cho

"The 2nd stage starts with the PPP (Project Plan Presentation) in week 4 and ends with S1P (Sprint 1 Presentation) in week 8."

Jane

"What is the S1R marked with circle 2?"

Prof. Cho

"S1R stands for Sprint 1 Retrospective. After completing Sprint 1, teams should hold a retrospective meeting to discuss what went wrong and what went well during the sprint. This helps teams learn and improve for the next sprint."

Tim

"Basically, we need to discuss our project progress to be better next sprint."

Prof. Cho

"Yes, and the gaol is to present team progress and contribution at the Sprint 1 Presentation."

Ken

"And HW4 is about the retrospect of the 1st sprint and make ready for the 2nd sprint."

1. PPP (Project Plan Presentation)
2. S1R (Sprint 1 Retrospective)
3. **S1P (Sprint 1 Presentation)** <- Goal
4. S2R (Sprint 2 Retrospective)
5. FP (Final **Presentation**)

Prof. Cho

"Yes. At the end of the 1st sprint, teams know what they didn't know when they first started the project."

Jane

"Please explain the difference between prototype and MVP again."

Prof. Cho

"Sure. When we first start the project, we are not sure if our ideas will work technically or if users will actually want what we're building. A **prototype** is like a rough sketch - it helps us test if something is possible and learn the technology."

Prototypes:

Prototypes are for quick feasibility checks, not production code.

Build a prototype to see if something can be done. Once confirmed, **throw it away** and write proper code with good design and testing. Prototype code is meant to be discarded, not shipped.

Jane

"So a prototype doesn't need to be perfect?"

Prof. Cho

"Not at all! A prototype is for learning. But an **MVP** - Minimum Viable Product - is different. It's the simplest version that actually works and delivers value to users. It has fewer features than the final product, but those core features must work reliably."

Jane

"Do we have to make both the prototype and MVP?"

Prof. Cho

"In most ASE courses, students are given prototypes so that they can focus on understanding the required technology to solve the given problem."

Jane

"Oh, so we start with a working prototype?"

Prof. Cho

"Yes. For example, in ASE230, you might receive a basic Laravel application as a prototype. Your job is to understand how it works, then extend it into an MVP by adding the core features your project needs. This way, you spend less time on basic setup and more time on software engineering practices."

Jane

"That makes sense! So during Sprint 1, we're turning that prototype into our MVP?"

Prof. Cho

"Exactly! You'll study the prototype, understand the technology stack, plan your features, and implement the most important ones. By week 8, your S1P should demonstrate a working MVP - simple, but functional and useful."

Tim

"Wait, I'm confused. You told us we should prepare for the PPP in Stage 1, but now you're saying we plan our features in Stage 2?"

Prof. Cho

"Good question! Let me clarify. In Stage 1, during your PPP, you create an **initial plan** based on what you think you know. You propose what features you want to build and estimate the work. But here's the thing - you haven't actually worked with the technology yet."

Jane

"Oh, so the plan can change once we start building?"

Prof. Cho

"Exactly! Once you dive into Sprint 1 and start coding, you discover things you didn't expect. Maybe a feature is harder than you thought, or you find a better way to solve the problem. So you **adjust** your plan based on what you learned. This is how we solve problems as software engineers."[8]

Tim

"So planning isn't a one-time thing?"

Prof. Cho

"Not at all! Even though we don't know exactly what we're going to face, we plan and execute first. Then we discover what was missing, correct our plans, features, and goals, and do it again. It's a cycle of continuous learning and improvement."

[8] As you gain more experience and know more software engineering rules and tools, your estimation becomes more correct. This makes you a professional and competent software engineer.

Jane

“I remember from class - it’s the transition from ‘unknown unknowns’ to ‘known unknowns’!”

Prof. Cho

“Perfect! Before you start, you don’t even know what you don’t know - those are unknown unknowns. After Sprint 1, you’ve identified the challenges and gaps - those become known unknowns. Much easier to deal with!”

Ken

“This sounds like the agile methodology we discussed in the classroom meetings.”

Prof. Cho

“Exactly right, Ken! The agile process is one of the proven methodologies that embraces this cycle. Instead of trying to plan everything perfectly upfront, agile accepts that change is inevitable and builds in regular opportunities to learn and adjust - like our sprint retrospectives and planning sessions.”

Tim

"So the PPP is our best guess, and we refine it as we go?"

Prof. Cho

"Yes! Your PPP shows you've thought through the problem. But the real learning happens when you build. If we waited until we understood everything perfectly before starting, we'd never start at all. Plan, build, learn, adjust, repeat - that's the engineering process."

Jane

"What if we want to add more features than the MVP needs?"

Prof. Cho

"That's what Sprint 2 is for! The MVP proves your concept works. Then in later sprints, you refine it, add more features, and improve the user experience. Remember: working software first, fancy features later."

Tim

"How many features do we need to implement?"

Prof. Cho

"I expect students to build meaningful features, not trivial ones. That said, most successful teams implement about two substantial features per sprint."[9]

Jane

"So, two features for Sprint 1, and another two features for Sprint 2. Right?"

Prof. Cho

"Yes, that's the typical pattern. Most students report that two features per sprint is a reasonable workload - enough to demonstrate progress without overwhelming you while you're learning the technology."

[9]Making trivial projects can turn out to be a waste of time, so students should be careful to choose features that are challenging but not overwhelming.

Tim

“What makes a feature ‘substantial’ versus ‘trivial’?”

Prof. Cho

“Good question! A substantial feature requires you to apply software engineering principles - things like proper data modeling, user authentication, error handling, or integration with external services.[10] A trivial feature might just be changing UI colors or adding static text that doesn’t involve real functionality.”[11]

Jane

“So we should aim for features that challenge us to learn?”

[10]These are just examples. The rule is that students should show that they can solve problems by building applications with useful features.

[11]Keep in mind that your grade reflects the complexity and engineering effort of your features. Substantial features naturally earn higher scores than trivial ones.

Prof. Cho

“Exactly! The goal is to grow as engineers. Pick features that push you slightly beyond your comfort zone, but not so far that you get stuck for weeks.”

Tim

“What about HW4?”

Prof. Cho

“HW4 is about finalizing Stage 2, I mean, the Sprint 1, and preparing for the final stage. It includes both the S1R documentation and your S1P presentation.”

Jane

“I see! Just like PPP prepared us for Sprint 1, the S1R prepares us for Sprint 2, right?”

Prof. Cho

“Exactly! The S1R is where you reflect on Sprint 1 - what worked well, what didn’t, and what you learned. This helps you plan Sprint 2 more effectively.”

Ken

“So we should also update the original plan we made in the PPP, based on what we discovered during Sprint 1?”

Prof. Cho

“Yes! Your PPP was your initial plan when you knew the least about the project. After Sprint 1, you have real experience with the technology and a working MVP. You should definitely revise your feature list, timelines, and goals based on what you now know is realistic.”

Tim

“So S1R isn’t just looking back - it’s also planning forward?”

Prof. Cho

"Precisely! That's why retrospectives are so valuable. You look back to learn, then look forward to improve. The insights from S1R directly feed into your Sprint 2 planning."

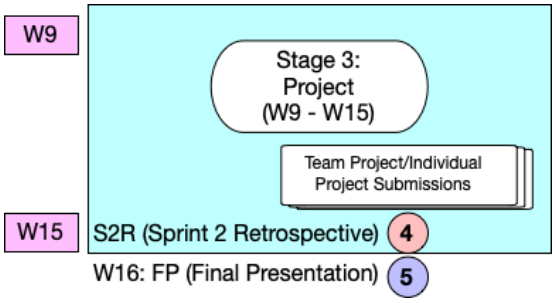
Prof. Cho

"Exactly, HW4 will help teams to be ready for the final stage of the project."

Stage 3: Project

Prof. Cho

"After the S1R and S1P, finally teams understand what they didn't know, so they can solely focus on the project."



- Stage 3: Project
  - Weeks: W9 - W15
  - Goal: **FP (W16)**
  - Related Assignments
    - Team project submission
    - Individual project submission

Jane

“So Stage 3 is when we complete the final project?”

Prof. Cho

“Yes! By this point, you’ve learned the technology in Stage 1 and built your MVP in Stage 2. Now you have about 7 weeks to refine your product, add more features, and prepare for the Final Presentation in week 16.”

Tim

“I see there are two submissions - team project and individual project.”

Prof. Cho

"Yes. So far we've mainly discussed the team project, but students also need to complete an individual project. In total, you'll finish two projects per semester. We'll discuss the individual project in detail later."[12]

Prof. Cho

"For now, let's focus on understanding the overall roadmap."[13]

He writes down on the whiteboard.

1. *PPP (Project Plan Presentation)*
2. *S1R (Sprint 1 Retrospective)*
3. *S1P (Sprint 1 Presentation)*
4. *S2R (Sprint 2 Retrospective)*
5. ***FP (Final Presentation)*** *<- Goal*

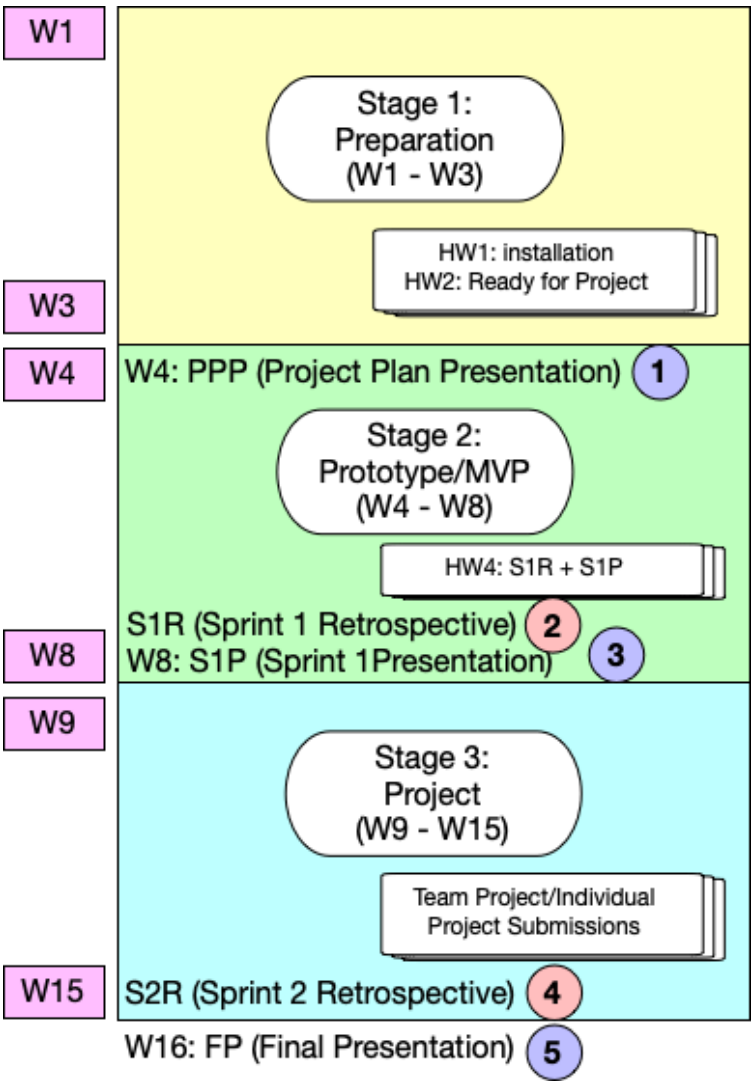
[12]The individual project follows the same process and uses the same tools as the team project, but you work independently rather than with a team.

[13]Note: Some ASE courses only require an individual project, not both.

Prof. Cho

"Now you can see the pattern emerging. Teams work toward the Final Presentation in week 16, but before that, you'll complete Sprint 2 with another retrospective - the S2R (Sprint 2 Retrospective) - just like you did after Sprint 1."

Prof. Cho shows a project diagram once more.



Prof. Cho

"Now you can clearly see the roadmap for this course - the stages, sprints, and ceremonies that will guide your learning journey."

Prof. Cho

"I strongly recommend that you start early, complete tasks ahead of deadlines, and identify your unknown unknowns as quickly as possible. Plan your time wisely and communicate regularly with your team."[14]

Prof. Cho

"By following this process, you'll learn not just how to code, but how to solve problems effectively - both as part of a team and independently. These are the skills that make you a professional software engineer."

[14]The most successful students are those who don't procrastinate and who actively seek help when stuck. Don't wait until the last minute to discover problems.

Three Modules

Prof. Cho shows the schedule again.

Stages	Weeks (M/W)	Sprints	Topics (Theory)	Modules	HW
Stage 1 (Preparation)	Week1 (1/12, 1/14)		Introduction		
	Week 2 (~~1/19~~, 1/21)	Project Discussion** Team Setup*	Project Preparations		HW1 (Installation) (1/25)*
	Week 3 (1/26, 1/28)		Project Preparations		HW2 (Project Plan) (2/1)*
Stage 2 (Prototype /MVP)	Week 4 (2/2, 2/4)	**Project Plan Presentation (2/2)** Sprint1 - 1	High-Level Languages	Module 1	
	Week 5 (2/9, 2/11)	Sprint1 - 2	High-Level Languages		
	Week 6 (~~2/16~~, 2/18)	Sprint1 - 3	High-Level Languages		
	Week 7 (2/23, 2/25)	Sprint1 - 4	Software Process		HW3 (Module 1) (2/27)**
	Week 8 (**3/2, 3/4**)	**Sprint 1 Presentation (3/2)**	**Midterm (3/4), on Canvas no class no office hours**		HW4 (Sprint 1 Retrospect + Sprint 2 Plan)(3/1)*
Stage 3 (Project)	Week 9		Spring Break (no class) **No classroom/office hours**		
	Week 10 (3/16, 3/18)	Sprint 2 – 1	Git	Module 2	
	Week 11 (3/23, 3/25)	Sprint 2 – 2	Git		
	Week 12 (3/30, 4/1)	Sprint 2 – 3	Security		
	Week 13 (4/6, 4/8)	Sprint 2 – 4	Security		
	Week 14 (4/13, ~~4/15~~)	Sprint 2 – 5	**Midterm (4/15, on Canvas no class/office hours)**	Module 3	HW5 (Module 2) (11/22)***
	Week 15 (4/20, 4/22)	Sprint 2 – 6 **Project submissions deadline (4/26)***	Deployment		
	Week 16 (4/27, 4/29)	**Presentation (4/27, 4/29)**			

Prof. Cho

"Now let's talk about the three modules for classroom discussions."

Module 1 & Module 2

Prof. Cho

"In the stage 1, we focus mainly on the technologies and tools for projects."

Prof. Cho

"In the stage 2 and 3, we discuss about the technology and tools in more detail."

Prof. Cho

"Remember our P4M4 terminology? In Stage 1, the technology feels like Magic (M1) - you don't fully understand how it works yet. But by Stages 2 and 3, it becomes a Machine (M2) - you understand the inner workings and can use it confidently to build your features."[15]

Jane

"What about the relationship between the exams and our projects/modules?"

[15]Refer to the other story telling about ASE courses if you are not familiar with the P4M4 terminology.

Prof. Cho

"The first midterm covers all the topics we discuss in Module 1. The second midterm is comprehensive, so it covers both Module 1 & Module 2."

Tim

"So there are no project-related questions on the midterms?"

Prof. Cho

"Not directly. However, the module topics and your project work are closely connected. The concepts you learn in Module 1 are the same ones you'll apply in your projects. So studying for the exam actually helps you understand what you need to do in your project, and vice versa."

Jane

"Oh, so the exams test our understanding of the concepts, and the projects test our ability to apply them?"

Prof. Cho

"Exactly! The exams mainly focus on theory and principles, while the projects focus on practical application. They reinforce each other - you need both to become a complete software engineer."[16]

Module 3

Tim

"Then what is Module 3 about?"

Prof. Cho

"Module 3 covers more advanced topics or concepts that aren't directly applied to your current project. These are important for your overall software engineering knowledge, but they may go beyond the immediate scope of what you're building."

Jane

"So it's kind of optional material?"

[16]Students will have a guideline for the exams in each course.

Prof. Cho

"From an exam perspective, yes. As you might notice, there are no exams after the second midterm, so Module 3 topics won't be tested. However, I wouldn't call them truly 'optional' for your education."

Ken

"Yeah, I found that Module 3 topics were really important for my other courses and even helped me solve problems in my own projects."

Prof. Cho

"Exactly, Ken! Module 3 is where we explore broader software engineering principles, and they'll make you a better engineer in the long run."

Jane

"So we should still pay attention even though it's not on the exam?"

Prof. Cho

"Absolutely. The best students see beyond just the exams. Module 3 gives you knowledge that separates junior developers from senior ones - the 'why' behind the 'how' you learned in Modules 1 and 2."

Tim thinks

"OK. Now I understand what I need to do this semester. I usually procrastinate until the last minute, but this time I'll change my attitude and stay prepared. To do that, I think I need to find a good project management tool. Or wait... couldn't I just build one myself?"

Building software to solve your own problems is actually an excellent starting point for any project, whether team or individual. When you're both the developer and the user, you understand the requirements intimately and can iterate quickly based on real needs.

Chapter 3:
Managing Complexity

Tim thought software engineering was just making code run—his 100-line calculator worked fine. But Julie's 20,000-line app showed him the truth: real software must scale, change, survive failures, and support teams.

That's the difference between a coder and an engineer—managing complexity with structure and disciplined problem-solving.

Quick Reference Guide

Core Concept

- Complexity grows exponentially with size
- 100 lines ≠ 100,000 lines (different challenges)
- Software Engineering = Managing Complexity
- Rules and structure prevent chaos

Why It Matters

- Small projects: ad-hoc coding works
- Large projects: need systematic approaches
- Team projects: everyone needs the same rules
- Real-world: handle failures, scale, changes

Tools for Managing Complexity

- Data models (define structure once)
- Modules (organize functionality)
- Interfaces (define contracts)
- Constraints (enforce rules)

Key Principles

- High-quality = easy to fix bugs + add features
- Everything keeps changing
- Problem-solving through structure
- Team collaboration requires rules

The Student's Misconception

Tim scratches his head. He still does not understand the meaning of **managing complexity** in software engineering.

> ***Tim thinks***
>
> *"I can already make a calculator app in Python, no complexity at all!"*

🐍 Python

```
# Tim's calculator - "This works perfectly!"
def calc():
    a = input("First number: ")
    b = input("Second number: ")
    op = input("Operation (+,-,*,/): ")
    if op == "+": print(float(a) + float(b))
    elif op == "-": print(float(a) - float(b))
    # ... and so on
```

> ***Tim thinks***
>
> *"If it runs, it's good software engineering. Right?"*

He keeps asking questions.

Tim thinks

“So why all this talk about ‘managing complexity’? Isn’t coding just… coding?

And ‘problem solving’? That sounds like solving math puzzles, not programming.

Plus, I have VSCode and Node.js. Why do I need ‘rules’? My code runs fine without any rules!“

“It Works!” Trap

Common Mistake #1: Confusing 'It Works' with 'It's Well-Engineered':

❌ “My calculator works, so it’s good software engineering.”

✅ “My calculator works for **this specific use case**—but can it scale, be maintained, and handle edge cases?”

Why this matters:

- “It works” means it passes one test case
- “Well-engineered” means it handles edge cases, is testable, maintainable, and extensible

Example edge cases Tim’s calculator doesn’t handle:

- Division by zero
- Non-numeric input

- Huge numbers (overflow)
- Multiple operations (2 + 3 * 4)
- Operator precedence
- Persistent calculation history

Then, Julie passes by. Seeing Tim's confused face, she stops to help.

Tim

"Julie, honestly, I don't understand this definition of software engineering as **managing complexity** and **problem solving**."

Tim

"I made a calculator for homework using Python in VSCode - it was not that complex at all, and I don't think I solved any problems. It was so straightforward and almost mechanical."

Julie

"Oh, nice! How many lines of code were in your calculator?"

Tim

“About 100 lines.”

Julie

“Perfect! Now imagine this…” **pulls out her laptop** “Here’s my recent team project for one of my ASE courses - a food delivery app. Guess how many lines?”

Tim

“Maybe… 1,000?”

Julie

laughs “Try 20,000 lines. And that’s just the backend! In the real world, even a simple project can have hundreds of thousands of lines of code. A modern OS has about 20–30 million.”

Tim

“WHAT?!”

Understanding Scale: Lines of Code (LoC)

Project Type	Lines of Code	Complexity	Team Size
Calculator (Tim's)	100	Single file, one function, basic logic	1 person
Class Assignment	500-2,000	Multiple files, some structure needed	1-2 people
Food Delivery App	20,000-50,000	Multiple modules, databases, APIs, testing	3-5 people
E-commerce Platform	100,000-500,000	Microservices, complex business logic, security	20-50 people
Operating System	20-30 million	Kernel, drivers, system calls, extreme optimization	1,000+ people
Google Search	2 billion+	Distributed systems, AI, global scale	10,000+ people

Key Insight: Complexity Isn't Linear:

Going from 100 to 1,000 lines isn't 10x harder—it's more like 50x harder because:

- More files to navigate
- More interactions between components
- More potential for bugs

- More people need to understand the code
- More edge cases to handle
- More requirements to satisfy

This is why techniques that work for 100-line projects fail catastrophically at 10,000 lines.

Julie

"And it's not just about size. Let me show you what happened when we DIDN'T manage complexity..."

Chaos When There are No Rules

We start simple, but we add complexity!

```
# Week 1: Tim's style - "It works!"
user1_order = {"food": "pizza", "price": 10}
user2_order = {"food": "burger", "price": 8}
# ... copy-paste for 100 users
```

Problems with Week 1 approach:

- Copy-paste programming (violates DRY principle)
- No validation (what if price is negative?)
- No type safety (price could be a string)
- Doesn't scale (manual coding for each user)

```
# Week 4: Chaos begins
user1_order = {"food": "pizza", "price": 10, "address": "123
Main"}
user2_order = {"food": "burger", "cost": 8}  # Oops! 'cost'
not 'price'
# ... now need to update 100 places
```

Additional problems at Week 4:

- Inconsistent field names ("price" vs "cost")
- Schema drift (some orders have "address", others don't)
- No single source of truth

- Impossible to query or aggregate data reliably

```
# Week 8: Complete disaster
# Different team members used different formats:
orders = [
    {"item": "pizza", "price": 10, "user_id": 1},
    {"food_item": "burger", "amount": 8, "customer": 2},
    {"meal": "pasta", "total": 12.5, "userId": "three"}  #
string ID?!
]
```

Critical failures at Week 8:

- Three different naming conventions for the same concept
- Mixed data types (integers vs strings for IDs)
- Impossible to write reliable processing code
- Bug fixing requires changes in dozens of places
- Adding features becomes a nightmare

> **Tim**
>
> "Oh… that's messy when people keep adding their own code."

Team Collaboration Chaos

Julie

"Exactly! Especially when many people are working on this. Sarah adds delivery tracking, Mike adds payment processing, Lisa adds restaurant management… all without rules."

Tim

"It would be chaos!"

Real-World Team Chaos Scenarios

Scenario	Without Rules	With Rules
Adding feature	• Search through 20 files to find related code • Hope you found all the places to change • Break existing features accidentally	• Check architecture docs • Modify a single module • Tests catch any breaks
Bug appears	• No one knows which code caused it • Blame game starts • Takes days to isolate	• Error logging shows the exact module • The responsible team owns it • Fixed in hours
New developer joins	• Spends weeks learning "tribal knowledge." • Makes mistakes due to undocumented rules	• Reads architecture docs • Follows code standards • Productive in days

	• Slows down the team	

Julie

"That's where 'managing complexity' comes in. Watch this:"

Julie

"This is when we architect the whole system by defining **data models**, **modules**, **interfaces**, and **constraints**, and design features based on the architecture.

🐍 **Python**

```python
# With Software Engineering principles:

from dataclasses import dataclass
from typing import List, Optional
from datetime import datetime

@dataclass
class Order:
    """Single source of truth for order structure"""
    order_id: int
    user_id: int
    items: List['OrderItem']
    total_price: float
    status: str
    created_at: datetime
    delivery_address: Optional[str] = None
```

```
class OrderService:
    """Centralized order management - everyone uses this"""
    def create_order(self, user_id: int, items: List[dict]) -
> Order:
        # Validation, business logic, consistency guaranteed
        pass
```

Tim

"Wow. It looks clean, and even I can understand what is going on!"

Benefits of the structured approach:

1. **Single Source of Truth**
   - `Order` class defines the structure once
   - Everyone uses the same definition
   - No inconsistencies possible
2. **Type Safety**
   - `order_id: int` prevents string IDs
   - `Optional[str]` documents that the address is optional
   - Editor autocomplete helps developers
3. **Centralized Logic**
   - `OrderService` handles all order operations
   - Validation in one place
   - Easy to add logging, caching, etc.
4. **Documentation**

- Docstrings explain purpose
- Type hints document expected data
- Code is self-documenting

5. **Maintainability**
   - Adding fields? Change one class
   - Adding validation? Change one method
   - Bug in order logic? Check one service

Tim begins to understand the problems complexity causes and the definition of software engineering as managing complexity with tools and rules within a team.

Managing Complexity

Julie

"Yes. We could manage complexity using software engineering tools such as architecture and design."

Tim

"Oh. You solved the problem of 'messy code' by managing complexity when you introduced the software design!"

Julie

"Exactly! And it gets more complex. What about:"

- "Payment failures?"
- "Restaurant is closed?"
- "User wants a refund?"
- "Database crashes?"
- "10,000 users ordering at once?"

Real-World Complexity: Beyond Happy Path

Challenge	Beginner Approach	Engineering Approach
Payment fails	`try: charge_card()` `except: print("error")`	• Retry logic with exponential backoff • Transaction rollback • User notification • Admin alert for repeated failures
Database crash	Code crashes, user sees error page.	• Connection pooling with failover • Graceful degradation • Queue pending orders • Automatic recovery
High traffic	The server slows down or crashes.	• Load balancing across multiple servers • Caching frequently accessed data • Rate limiting per user • Database query optimization
Data inconsistency	Hope it doesn't happen	• Database transactions (ACID) • Data validation at multiple layers • Automated tests catching edge cases

Tim

"I… never thought about all that."

Julie

"Your 100-line calculator handles maybe five requirements. Our app handles 50+ requirements. Without rules and tools, it's impossible to manage that level of complexity."

Tim

"So that's why we need software engineering! It's not about making code work once, it's about making it work always, for everyone, even when things go wrong!"

Julie

"Now you're getting it! It is about building high-quality software that is easy to fix bugs and add features."

Julie

"And don't forget - everything keeps changing; clients want new features, team members come and go, libraries get updated, security threats emerge... the list never ends."

The Nature of Change in Software

Type of Change	Example	Frequency
Requirements	Client wants new payment method	Weekly
Dependencies	Library updates with breaking changes	Monthly
Team	Developer leaves, new one joins	Quarterly
Technology	New framework becomes standard	Yearly
Security	Vulnerability discovered, must patch	Unpredictable
Scale	User base grows 10x	Varies
Business	Company pivots, priorities shift	Varies

Why 'High-Quality Software' Means 'Handles Change Well':

Code isn't written once and forgotten. Over its lifetime, software will:

- Be modified hundreds or thousands of times
- Be read by dozens of different developers
- Interact with evolving external systems
- Serve changing business needs

Therefore, high-quality software is designed to **accommodate change**:

- Modular architecture allows replacing components
- Comprehensive tests catch regressions when changing code

- Clear documentation helps new developers understand and modify
- Consistent patterns make changes predictable

Tim

"So, to build high-quality software, we should be able to update software from possible changes."

Julie

"Yes. We keep solving problems by managing complexity. This is the essence of software engineering."

Tim's Realization

Tim

"Now I think I understand why software engineers are problem solvers who manage complexity."

Julie

“Exactly! Software engineers must choose the right rules and tools for each problem; there’s no one-size-fits-all solution.”

Julie

“ Also, they should follow the common rules for all projects. Furthermore, they solve the problems in a team, so they have to manage all the possible human-related problems through effective communication.”

Tim

“Wow, it’s like juggling while riding a bicycle!”

Julie

“Perfect analogy! The good news is that software engineering gives you the skills, I mean rules and tools, to keep all those balls in the air.”

Tim

“I see the point now.”

Julie

"Don't forget that software engineering is relatively young. You can think of software engineering as the accumulation of knowledge to build high-quality software since the invention of computers."

Tim

"From the experiences of managing complexity."

Julie smiles, knowing Tim finally gets it.

Julie's Four Pillars of Managing Complexity

Julie

"Let me share my four pillars of managing complexity"

Julie shows her notebook.

Julie

"I learned these rules from my ASE courses, and I practiced it through my individual and team projects."

1. Structure (Architecture)
   - Define clear boundaries between components
   - Each module has a single, well-defined responsibility
   - Minimize dependencies between modules

   Example:

   ```
   # Bad: Everything in one place
   def process_order():
   ```

```
        # 500 lines of mixed logic
        pass

    # Good: Separated concerns
    class OrderValidator:
        def validate(self, order): pass

    class PaymentProcessor:
        def charge(self, order): pass

    class InventoryManager:
        def reserve_items(self, order): pass
```

2. Consistency (Standards)

- Everyone follows the same naming conventions
- Same patterns for similar problems
- Shared code style (enforced by linters)

Example:

```
# Team standard:
#  always use dataclasses for data models
@dataclass
class User:
    user_id: int
    email: str

@dataclass
class Order:
```

```
    order_id: int
    user_id: int
```

3. Validation (Testing)

- Unit tests for individual components
- Integration tests for component interactions
- End-to-end tests for complete workflows

Example:

```
def test_order_total_calculation():
    order = Order(items=[
        OrderItem(price=10.0, quantity=2),
        OrderItem(price=5.0, quantity=1)
    ])
    assert order.calculate_total() == 25.0
```

4. Documentation (Communication)

- Architecture decisions recorded (ADRs)
- API contracts documented
- Code comments explain **why**, not **what**

Example:

```
class PaymentProcessor:
    def process_refund(self, order_id: int) -> bool:
        """
```

```
        Process a refund for an order.

        Note: We use eventual consistency here
        because immediate refunds caused the race
        conditions with the inventory system.
        Refund is queued and processed
        within 1 hour.

        Returns:
            True if the refund was queued
successfully
        """
        pass
```

Common Mistake 2: Ignoring Complexity Until It's Too Late

❌ "Let's just code it quickly now, we'll organize later."

✅ "Let's spend 30 minutes designing the structure before we write any code."

Why? Refactoring messy code is 10x harder than designing structure upfront. "Later" never comes—technical debt compounds and eventually the codebase becomes unmaintainable.

Chapter Summary

Tim's Journey in This Chapter

Aspect	Beginning Belief	Understanding After Discussion
Complexity	"My 100-line code isn't complex"	"Complexity explodes with size & requirements"
Quality	"If it runs, it's good"	"Quality = handles edge cases, changes, scale"
Engineering	"Just coding in an IDE"	"Systematic problem-solving with structure"
Problems	"Math puzzles"	"Real-world challenges with many constraints"
Rules	"Unnecessary overhead"	"Essential for team collaboration & maintainability"

Notice

As we understand the importance of managing complexity, we can now discuss the detailed process for managing it in software engineering.

The Software Engineering Process has three key steps:

1. **Requirements** - Define the problem clearly
2. **Architecture & Design** - Plan the solution structure
3. **Implementation & Testing** - Build and verify

These steps aren't strictly sequential - we often cycle back, refining requirements as we learn from design, or redesigning when tests reveal issues. This iterative approach is essential for managing complexity.

Let's examine each step in detail.

Chapter 4:
Requirements

After understanding the concepts of software engineering as managing complexity, Tim, Jane, and Ken begin their team project journey. Their first lesson: understanding WHAT to build.

- Requirements tell us WHAT clients want, not HOW to build it
- Actor-Goal format helps write clear requirements
- Use Case Diagrams visualize the entire system at a glance

Quick Reference Guide

Core Concepts

- Requirements = bridge between World (problem) and Machine (solution)
- Start with WHY (problem domain)
- Then WHAT (features & requirements)
- Finally HOW (architecture & code)

Three-Step Approach

1. Understand Why & Define Problems
2. Translate into Features/ Requirements
3. Provide High-Quality Solutions

User Story Format

- As a [WHO - user type]
- I want to [WHAT - action]
- So that [WHY - benefit]

INVEST Checklist

- Independent
- Negotiable
- Valuable
- Estimable
- Small
- Testable

Notice

In this story, the team is expected to build a time management web application.

The Eager Start

The team gathers for their first project meeting, excited and ready to code.

Tim

"Let's start coding! I'll make the login page first!"

Ken

"Wait... what exactly should the login page do?"

Tim

"Uh... let users log in?"

Ken

"With email? Username? Social login? What happens if they forget their password?"

Tim pauses. He hasn't thought about any of this.

The Hidden Complexity of "Simple" Features

What 'Simple' Login Actually Involves:

Aspect	Decisions Needed
Authentication	Email/username? Social login (Google, Facebook)? Multi-factor?
Password	Minimum length? Special characters? Password strength meter?
Recovery	Forgot password flow? Security questions? Email verification?
Session	How long does login last? Remember me option? Auto-logout?
Security	Rate limiting? CAPTCHA? Brute force protection?
Errors	"Wrong password" vs "User not found"? Security implications?
Accessibility	Screen reader support? Keyboard navigation?

Key Insight: What seems like "just a login page" actually involves dozens of decisions. Without requirements, you'll make these decisions arbitrarily—and likely wrong.

Julie's Timely Warning

Julie, who is working on her capstone project nearby, overhears the confusion.

Julie

"You're about to make the same mistake my team made last semester."

Team

"What mistake?"

Julie

"We spent 3 weeks building features nobody asked for, then had to throw it all away."

The Pizza Ordering Disaster

Julie pulls up a chair and shares her story.

Julie

"We thought we knew what to build: a pizza ordering app. Simple, right?"

- Week 1-2: Built beautiful UI with animations
- Week 3: Added shopping cart with fancy features
- Week 4: Client says: 'Why can't I customize toppings?'
- Week 5: Client asks: 'Where's the delivery tracking?'
- Week 6: Client confused: 'Why is checkout so complicated?'

Julie

"We never asked WHAT they actually needed. We just started building what WE thought was cool."

The Cost of Skipping Requirements

Metric	Julie's Team (No Requirements)	With Requirements
Wasted Code	3,000 lines deleted	Minimal rework
Rework Time	3 weeks (50% of timeline)	2 days for refinements
Client Satisfaction	Frustrated, lost trust	Happy, informed throughout

Team Morale	Demoralized, blamed each other	Confident, collaborative
Final Grade	C (late, missing features, chaos in presentation, missing tests)	A (on time, complete)

Ken nods thoughtfully.

Ken

"Think of requirements like a recipe for cooking."

Without Recipe (No Requirements):

- *"Make breakfast" → What kind? For whom? How many? → You might make pancakes when they wanted eggs!*

With Recipe (Requirements):

- *"Make scrambled eggs for two people with cheese, ready in 10 minutes." → Clear, specific, achievable!*

Common Mistake #1: Jumping to Code Without Requirements:

❌ "I'll just start coding, and we'll figure it out as we go."

✅ “Let’s spend 2 hours writing requirements, then 1 hour reviewing them with the client before writing any code.”

Why? Every hour spent on requirements saves 10 hours of rework. Starting to code without requirements means you’re building the wrong thing, which is worse than building nothing at all.

What are Requirements?

Professor Cho walks by and joins the discussion.

Prof. Cho

"It looks like you are discussing **what** part of requirements, but what's more important is **why** part of requirements."

Tim

"Well, I think we already talked about it; without requirements, we might end up implementing something wrong. Julie shared her experience with us already."

Prof. Cho

"It's great, but we need to understand the basic ideas behind the requirements."

Prof. Cho

"I believe you already understand the meaning of professionals."

Jane

"Yes, unlike amateurs who do the work because they love it, professionals make money by doing the work."

Prof. Cho

"Correct. It means that we have clients who pay us, and we solve the **clients' problem**, not the problem that we like to solve."

Jane

"Ouch! I was there before."

Tim

"Also, I remember you also mentioned **high-quality product** and **responsibilities**."

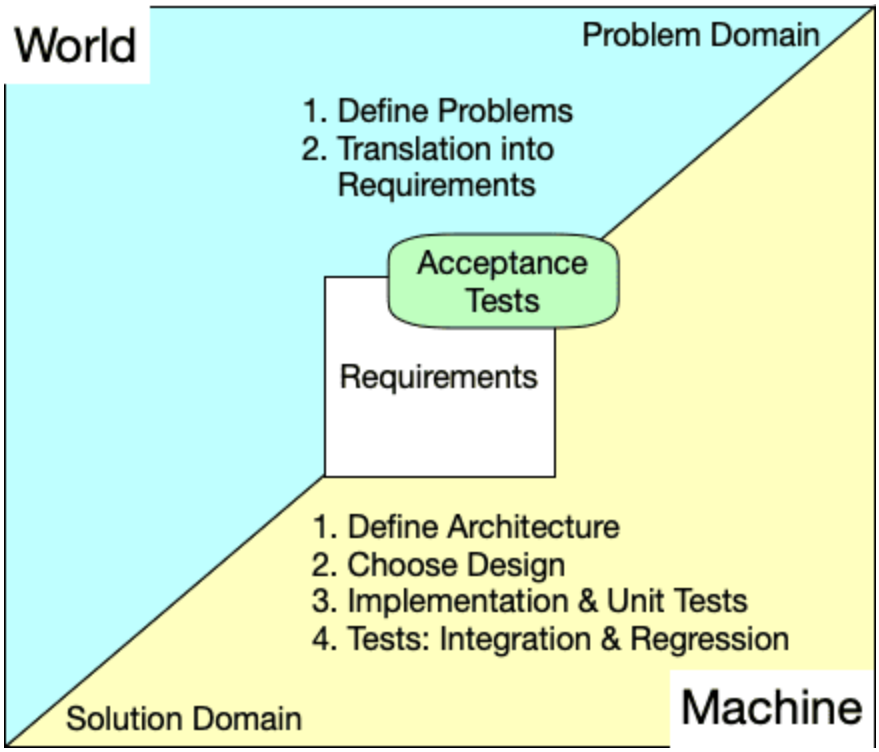
Prof. Cho

"Yes. In other words, unless we understand the client's problem domain, we cannot solve the problem correctly, let alone effectively."

Prof. Cho

"And, in the software engineering research, we use the terminology 'World' to describe the problem domain."

Prof. Cho shows a diagram:



Requirements Bridge World and Machine

Prof. Cho

"Requirements are the bridge between the client's problem (World) and your solution (Machine). Notice that when we flip 'W', it becomes 'M'."

Tim

"Haha, 'R' flips 'W' to 'M'."

Prof. Cho

"More like bridging between 'W' and 'M'. Without this bridge, you might build something technically perfect but completely useless to the client."

World vs Machine

Aspect	World (Problem Domain)	Machine (Solution Domain)
Focus	User's reality, problems, context	Software, databases, APIs, code
Language	Business terms, user goals	Technical terms, system operations

Concerns	What users need, why they need it	How to implement, architecture, design
Examples	• Students forget deadlines • Pizza orders get confused • Checkout is too slow	• Email notification service • Order management database • Performance optimization
Who owns it	Client/Users	Development team

Key Insight: The Translation Problem:

Clients speak in the language of the World domain: "Students forget deadlines."

Developers think in the Machine domain's language: "We need a cron job to send emails."

Requirements bridge the gap: "As a student, I want to receive email notifications N days before a deadline so that I don't forget assignments."

This connects:
- WHO (student - World)
- WHAT (email notification - Machine)
- WHY (don't forget - World benefits by Machine)

The Three Steps Approach

Prof. Cho writes down the three-step approach on the whiteboard.

Step 1: Understand Why & Define Problems ⟶ Explore the problem domain

Step 2: Translate into Features/Requirements ⟶ Bridge between World and Machine

Step 3: Provide High-Quality Solutions ⟶ Design, implement, test, deliver

Prof. Cho

"Let's say clients require us to build a to-do app. What is the first step?"

Tim

"It says 'Understand Why & Define Problems.'"

Jane

“Also, ‘Explore the problem domain.’”

Three Steps Visualized

Step	Activities	Artifacts	Time
1. Understand Why	• Interview users • Observe workflows • Identify pain points • Define problems	• Problem statements • User personas • Context diagrams	20% of project[17]
2. Define What	• List features • Write user stories • Prioritize requirements • Validate with users	• Feature list • User stories • Acceptance criteria • Use case diagrams	15% of the project

[17] Percentages are approximate and depend on project scope and complexity.

3. Build How	• Design architecture • Implement features • Write tests • Deploy & maintain	• Architecture diagrams • Code, tests, docs • Deployment pipeline	65% of the project

Critical Point: Don't Rush Step 1 & 2:

Notice that Steps 1 & 2 take 35% of the project time but involve almost no code. This feels wasteful to beginners who want to "get started."

However: Spending 35% of time on requirements saves 100% on building the wrong thing. The teams that skip to Step 3 immediately end up like Julie's team—throwing away weeks of work.

Step 1: Understand Why & Define Problems

Prof. Cho

"Yes. The first step is always asking 'Why?' to understand the problem domain."

Tim

"Can we just ask the clients what they want?"

Prof. Cho

"Yes, you can. But what about the situation where clients don't know what they really need? Or, even worse, they understand the problem wrong?"

Jane

"How can that happen? In this case, I think it's not my problem; It's their problem!"

Prof. Cho

"I understand, but who are we, software engineers? Are we coders who make software?"

Jane

"No. We are problem solvers by managing complexity."

Prof. Cho

"Yes. We solve clients' problems, not just write code for them. Our job is to make clients' lives easier with our software and give them the trust that we can solve their problems. Clients will pay us because of that."

Tim

"Wow. I can understand professionalism again. We make value by solving clients' problems, but amazingly, clients might not know exactly what they want or what their problems might be."

Prof. Cho

"Yes. That makes us problem solvers, not coders. And also, that's why we should start from asking why: 'why do clients want this Todo app?'"

Why Clients Often Don't Know What They Want

Scenario	What Client Says	What They Actually Need

Unfamiliar with tech	"Make it like Amazon"	Actually need: simple inventory tracking, not full e-commerce
Solving symptoms	"We need faster servers"	Actually need: database query optimization, not hardware
Copy competitors	"Add all features Competitor X has"	Actually need: specific features for their unique users
Unclear goals	"Make it user-friendly"	Actually need: specific usability requirements (e.g., "checkout in less than three clicks")

The Five Whys Technique:

To get to the root problem, keep asking "Why?":

1. "We need a todo app." → Why?
2. "Students forget deadlines" → Why do they forget?
3. "They have too many assignments." → Why is that a problem?
4. "No central tracking system" → Why don't they use existing tools?
5. "Existing tools don't integrate with Canvas" → **Root cause found!**[18]

Now you know: The real problem isn't "no todo app"—it's "**no Canvas-integrated deadline tracker**."

Prof. Cho writes down on the whiteboard.

> ***Observation:***
>
> *Students have multiple assignments with various deadlines, often forget them, and start too late.*

> ***Why It Matters:***
>
> *Students need a tool to manage time effectively and avoid missing deadlines.*

> ***Specific Problems:***
>
> - *P1: No proactive notifications for deadlines*
> - *P2: Limited browser-based accessibility*

[18]To make our storytelling simple, in the example, we intentionally built a simplistic web application in this story.

Prof. Cho

"We observe the problem domain, and then, we can see the problems. Then, we can define problems with our own words."

Jane

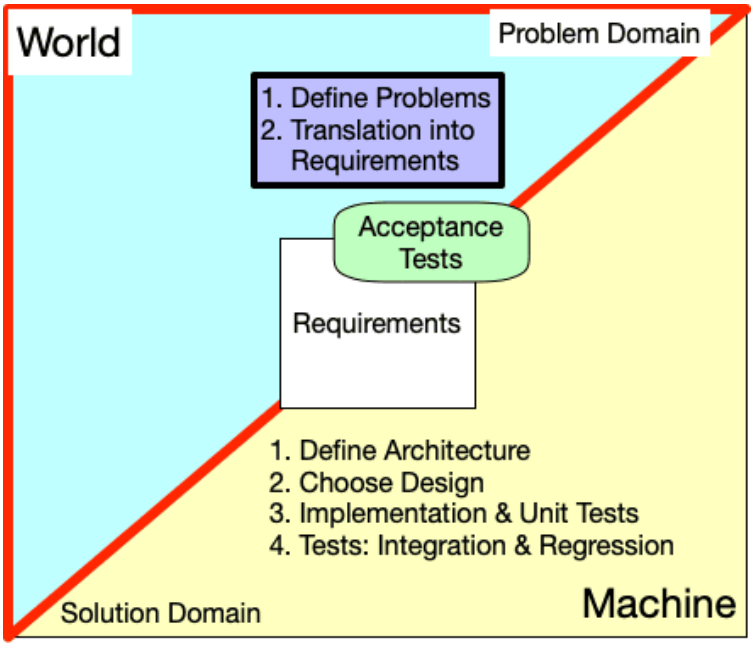
"Do we have to do that alone?"

Prof. Cho

"No. In most cases, we work with clients closely to come up with the problem definitions."

Prof. Cho

"But what is important is that we should **actively** understand the clients' problem domain in the real **World**, not passively waiting for clients to give us the problem list."



Tim

"I see why we need to understand the problem domain; otherwise, we may end up making useless[19] requirements."

Prof. Cho

"Yes, exactly. Let's talk about the second step."

[19] ineffective or not-correct

Step 2: From Problems to Features

Prof. Cho

"We have two problems, and each problem matches the feature that we are going to implement."

He keeps writing on the whiteboard.

Problem 1 ⟶ Feature 1: "App notifies the deadline N days before by email."

Problem 2 ⟶ Feature 2: "Students can do CRUD operations[20] via web"

Prof. Cho

"Features describe WHAT the system does, and they are from our understanding, and thus interpretation, of the **World.**"[21]

[20] Create, Read, Update, and Delete DB Operations

[21] In the real project, each member has to deal with at least four non-trivial features (2 features per sprint).

Problem-to-Feature Mapping

#	Problem (World)	Feature (Machine)
1	Students forget deadlines because no proactive reminders exist	Email notification system that alerts N days before the deadline
2	Students can't access assignments from anywhere (only on desktop at home)	Web-based CRUD interface accessible from any device

Good Feature Characteristics:

- **Addresses specific problem:** Each feature solves a defined problem
- **User-facing:** Described in terms users understand
- **Implementation-neutral:** Doesn't specify HOW (that's architecture/design)
- **Testable:** Clear success criteria

Example:

- ❌ Bad feature: "Use Node.js to send emails with cron jobs" (too technical, specifies HOW)
- ✅ Good feature: "App notifies students of deadlines via email" (what users experience)

From Features to Requirements

Prof. Cho

"Now we break features into specific user stories using the Actor-Goal format."

User Story Template:

- *As a [WHO - type of user]*
- *I want to [WHAT - action]*
- *So that [WHY - benefit]*

Prof. Cho

"This forces us to think from the user's perspective and makes requirements testable."

Jane

"I see the point. It is basically storytelling."

Prof. Cho

"Yes, and that's why it is called the **User Stories**."

Why User Story Format Works

Component	Purpose
As a [WHO]	• Identifies the actor/user type • Reminds us that different users have different needs • Example: "As a student" vs. "As an instructor" have different requirements
I want to [WHAT]	• Specifies the action/feature • Must be specific and actionable • Should be implementation-neutral (no HOW)
So that [WHY]	• Explains the value/benefit • Helps prioritize (if no benefit, why build it?) • Validates we're solving the right problem

Common Mistake #2: Writing Requirements From Developer Perspective:

❌ "The system shall use JWT tokens for authentication."

✅ "As a student, I want to stay logged in for 24 hours so that I don't have to re-enter my password every session."

Why? The first is a technical specification (HOW), not a requirement (WHAT). The second captures user value and can be tested, while remaining flexible about implementation.

Feature 1: Breaking Down Notification System

Prof. Cho

"We have two requirements from the first feature - notification system."

Feature 1: Email notification system

- *RQ1:* ***As*** *a student, I* ***want to*** *get a notification email from the app before N days for the schedule* ***so that*** *I do not miss any deadlines.*
- *RQ2:* ***As*** *a student, I* ***want to*** *click the checkmark of the schedule* ***so that*** *I can confirm that I am correctly notified and ready to finish the task.*

Prof. Cho

"The RQ1 (requirement 1) is broad, so we need to make sub-requirements from this 'Epic requirement'."

Tim

"So, when a requirement has sub-requirements, is it called the 'Epic requirement'?"

Prof. Cho

"Yes. The goal of requirements is to specify WHAT should be implemented and tested, so if necessary, we detail a requirement into multiple subrequirements whenever necessary."

*RQ1-1: **As** a student, I **want to** specify my email **so that** the system knows where to send notifications*

*RQ1-2: **As** a student, I **want to** specify N days **so that** I choose when to receive emails*

Prof. Cho

"Now, we have **3** requirements for the feature **1**."

Epic vs User Story vs Task

Level	Description	Example	Sprint
Epic	Large body of work, spans multiple sprints	"Notification System"	Multiple sprints
User Story	Single requirement implementable in 1 sprint	"Get email N days before"	1 sprint

Sub-story	Breakdown of complex story	"Specify email address"	< 1 sprint
Task	Implementation step (technical)	"Setup SMTP service"	Hours/ days

When to Break Down Requirements:

Break down a user story if:

- Estimated to take > 1 sprint[22]
- Multiple team members needed
- Unclear acceptance criteria
- Multiple technical components involved

RQ1 breakdown example:

- RQ1 (Epic): "Get notifications before deadlines"[23]
  - RQ1-1: "Specify email for notifications."
  - RQ1-2: "Choose how many days' advance notice."
  - RQ1-3: "Receive formatted email notification."
  - RQ1-4: "Unsubscribe from notifications."

Each sub-requirement is now:

- Small enough for one sprint
- Clear acceptance criteria
- Independently testable

[22]This is an example, as in this case, we can make multiple user stories instead to finish each user story in one sprint.

[23]For simplicity, requirements are not written in the Actor-goal format.

Feature 2: Breaking Down CRUD Operations

Prof. Cho

"Likewise, we can make **4** requirements for the feature **2** - web-based CRUD[24] operations."

Feature 2: Web-based CRUD operations

RQ1: As a student, I want to create schedules via form/JSON So that I can add schedules online

RQ2: As a student, I want to read schedules in JSON/HTML So that I can view schedules online

RQ3: As a student, I want to update schedules via form/JSON So that I can modify schedules online

RQ4: As a student, I want to delete schedules via form So that I can remove schedules online

Tim

"Then, overall, we have **7** requirements for two features."

[24] Create, Read, Update, and Delete

Prof. Cho

"Yes. Now, we translated the problem in the **World** into the problem in the **Machine**."

He writes down a note on the whiteboard and shows the diagram.

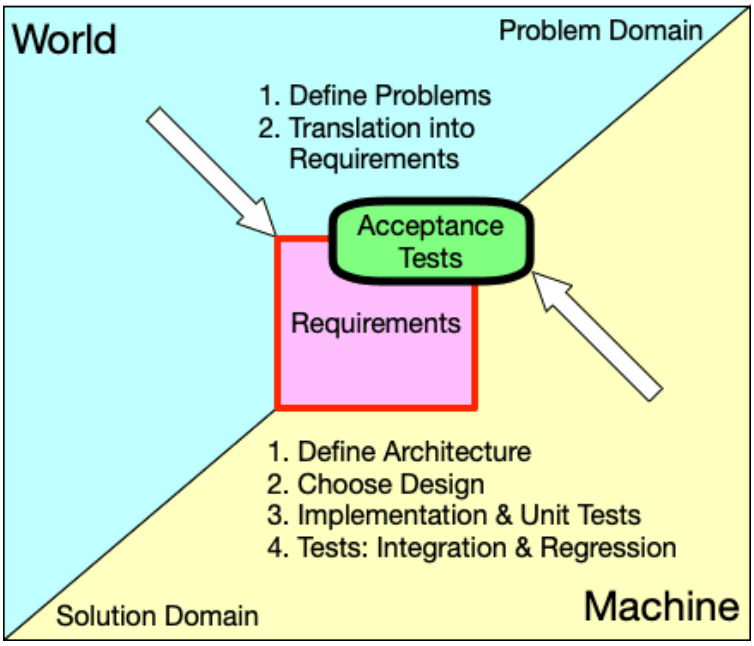
TodoApp: 2 Problems ⟶ 2 Features ⟶ 7 Requirements

Each requirement becomes an acceptance test object:

- *Act as the user (WHO)*
- *Perform the action (WHAT)*
- *Verify the benefit (WHY)*

If test passes ⟶ Requirement met ✓

If the test fails ⟶ Need more work. ✗



Requirements as Acceptance Tests

Converting Requirements to Tests:

Requirement: "As a student, I want to create schedules via a form so that I can add schedules online."

Acceptance Test:[25]

```
Given: User is logged in
When: User fills the schedule form with:
      - Title: "CSC Assignment"
      - Due date: "2024-12-15"
      - Description: "Final project."
      And clicks "Create."
Then:
```

[25]It is also called E2E (end to end) test.

```
      - Schedule appears in the schedule list
      - Success message shown
      - Database contains a new schedule
```

How to test:[26]

- Manual testing: QA person follows the steps
- Automated testing: Selenium/Cypress script
- Both verify the same acceptance criteria

[26]It is the team leader's role to come up with the test method with the team member who implements this feature.

INVEST: Checklist for Good Requirements

Prof. Cho

"When we make requirements, we have a simple rule 'INVEST'. Julie, do you have anything to add?"

- *I - Independent (can develop separately)*
- *N - Negotiable (details can be discussed)*
- *V - Valuable (provides clear user value)*
- *E - Estimable (team can estimate time)*
- *S - Small (fits in one sprint)*
- *T - Testable (can verify it works)*

INVEST Principle Explained

	Meaning	**Good Example**	**Bad Example**
I	Independent - can be developed without	"User can login"	"User can login AND reset

	depending on other stories		password”(2 stories)
N	Negotiable - details can be discussed, not a contract	“Display schedules in easy-to-read format”	“Display schedules in 12pt Arial font with 333 color” (over-specified)
V	Valuable - provides clear benefit to user/ business	“Receive deadline notifications”	“Refactor database schema” (technical, no user value)
E	Estimable - team can roughly estimate effort needed	“Create schedule via form”	“Implement AI to predict student stress” (too vague/ complex)
S	Small - completable within one sprint	“Add title field to schedule”	“Build entire notification system” (too large)
T	Testable - has clear pass/fail criteria	“Email sent 3 days before deadline”	“System is user-friendly” (subjective)

Julie

“Yes, I’d like to share the mistakes when students make requirements.”

She writes down the mistakes to avoid on the whiteboard.

❌ Skipping the "Why."

✓ "Students need quick ordering due to limited time."

❌ Vague requirements

✓ "Order confirmation within 2 seconds."

❌ Technical jargon

✓ "Students can create orders from mobile."

❌ Too large to implement

✓ Break into smaller pieces

Common Requirement Mistakes

Common Mistake #3: Vague or Ambiguous Requirements:

Vague (Bad)	Specific (Good)
"System should be fast"	"Page loads in < 2 seconds on 4G connection"
"App should be user-friendly"	"User can complete checkout in ≤ 3 clicks"

“Notifications should be timely”	“Email sent exactly 3 days before deadline at 8 AM user’s timezone”
“System should handle errors”	“On payment failure, user sees specific error message and retry option within 1 second”

Why? Vague requirements can’t be tested, estimated, or implemented consistently. Different developers will interpret them differently.

Common Mistake #4: Implementation in Requirements:

❌ “Use MongoDB to store user data with bcrypt for passwords.”

✅ “As a user, I want my password stored securely so that my account is protected.”

Why? Requirements describe WHAT and WHY, not HOW. Specifying implementation:

- Limits design options
- Might choose the wrong technology for the problem
- Makes requirements brittle (what if MongoDB doesn’t work?)
- Confuses clients (they don’t care about MongoDB)

Requirements Can Change

Prof. Cho

"Requirements DO change. That's normal in agile development. Don't be afraid of any changes; build the system that can embrace changes."

Manage Changes Professionally:

1. *Document the proposed change*
2. *Assess impact on timeline/budget*
3. *Prioritize (critical or can wait?)*
4. *Get approval from stakeholders*
5. *Update documentation and tests*

Keep requirements stable within each sprint!

Jane

"We need to manage complexity from changes!"

Why Requirements Change

Change Type	Example	Frequency
Better understanding	Client realizes they actually need X, not Y	Constantly (especially early)
Market shifts	Competitor launches similar feature, must differentiate	Monthly
Technology changes	New API available that enables better solution	Quarterly
User feedback	Beta users request different workflow	After each release
Business priorities	Company pivots, changes target users	Rarely but major impact

Agile Approach to Changing Requirements:

Traditional (Waterfall) mindset:

- Requirements frozen after initial phase
- Changes considered "scope creep" and resisted
- Result: Build what was requested 6 months ago (now outdated)

Agile mindset:

- Requirements expected to evolve
- Welcome changes that add value

- Freeze requirements **within** each sprint, but re-prioritize between sprints
- Result: Build what's needed now

Key difference: Agile embraces change as learning, not failure.

Managing Requirement Changes

Change Management Process:

1. **Request** - Stakeholder proposes change
   - Document: What changes? Why? Who requested?
2. **Analysis** - Team evaluates impact
   - How much work? Which components were affected?
   - Does it conflict with existing requirements?
3. **Prioritization** - Compare with current backlog
   - Must-have now? Or nice-to-have later?
   - What do we drop if we add this? (time is fixed)
4. **Approval** - Get stakeholder agreement
   - If we add Feature X, we'll delay Feature Y—okay?
   - Document decision
5. **Update** - Revise all artifacts
   - Update requirements document
   - Modify tests

- Communicate to all team members

Critical Rule: No changes mid-sprint. Changes queue for next sprint planning.

Looking Ahead

Prof. Cho

"Next, we'll explore architecture and design - **HOW** to build the machine that implements these requirements."

Julie

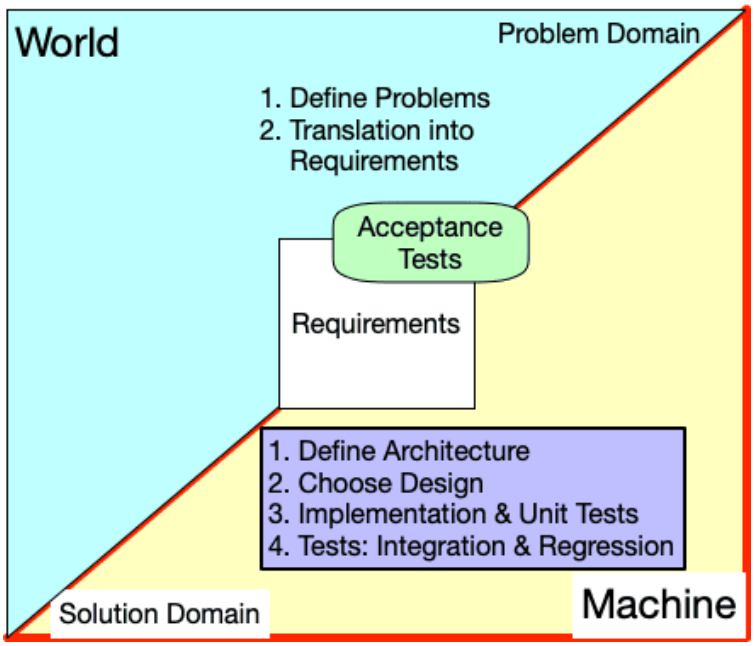
"I want to add that, now we solve problems in the 'Machine' domain."

Prof. Cho

"Yes. Now we need to answer the questions of WHAT and HOW. That is possible because we understood WHY"

Prof. Cho

"And the next step is software design. Software architecture is high-level design, and when we have software architecture, we can choose low-level design."



Ken

"Now I understand we can't design without requirements!"

Prof. Cho

"Exactly. You just mentioned one of the most important lessons: understand WHAT (requirements) before deciding HOW (architecture/implementation)."

The team leaves with a clear understanding: requirements are the foundation of the professional software development process.

Key Takeaways

- **Requirements bridge** World (problem) and Machine (solution)
- **Three-step approach:** Why → What → How (in that order!)
- **Professionals solve** client problems, not what they find interesting
- **User story format** ensures user-centric requirements
- **INVEST principles** validate requirement quality
- **Epic → Feature → User Story → Task** hierarchy manages complexity
- **Changes are normal** in agile; manage them professionally
- **Don't skip requirements** phase—every hour here saves 10 later

Chapter 5:
Software Architecture

With clear requirements in hand, the team is ready to start coding. But they quickly discover that knowing WHAT to build isn't enough - they need to know HOW to **organize** it for data model flow.

- Architecture provides the blueprint for software structure
- Three components: Elements, Interactions, and Constraints
- Use proven patterns like MVC, MVVM, and Client-Server

Quick Reference Guide

Core Concept

- Architecture = Blueprint for organizing code
- Data Model = The “main character” that flows through your system
- Three Components: EIC (Elements, Interactions, Constraints)

Why It Matters

- Without architecture: Integration nightmare
- With architecture: The Team can work independently
- Proven patterns save time & reduce errors

Three Components (EIC)

- **E**lements: Modules/ components
- **I**nteractions: How modules communicate
- **C**onstraints: Rules everyone follows

Common Patterns

- MVC: Model-View-Controller (web apps)
- MVVM: Model-View-ViewModel (complex UI)
- Client-Server: Separate presentation from logic

Notice

In this story, the team is expected to build a time management web application.

The team now understands the importance of requirements for bridging the World (problem) and the Machine (solution), and they have eight features in total (2 members x 2 features x 2 sprints). Out of the eight features, they have 24 requirements (3 per feature).

They have 24 acceptance tests: some are manual tests with instructions, and some are automated.

They need to implement four features (12 requirements and acceptance tests) and deliver them in sprint 1.

Another Chaos Begins

The team has requirements and is excited to start implementation.

Ken

"OK. Who is going to take what task?"

Tim

"I'll work on Feature 1 and its related requirements—mostly front-end work."[27]

Jane

"I'll handle Feature 2 on the client side!"

Ken

"As team leader, I'll manage the schedule and software quality. I'll focus on tests in the first sprint[28] and add documentation in the second sprint."[29]

[27] Simplified example for illustration.

[28] We also call it an iteration.

They dive in enthusiastically, each working independently.

Soon, confusion sets in.

Tim

"Wait, what information should my frontend send to Jane's CRUD server?"

Jane

"And I don't know what structure to use when connecting the server to the database?"

Ken

"What might be wrong? We just worked hard, and there is no problem in managing people!"

The team stares at each other in confusion.

The Integration Problem

Common Mistake #1: Coding Without Architecture:

[29] Team leaders don't implement features; instead, they manage the project and ensure software quality.

❌ "We have requirements, let's just start coding and figure it out as we go."

✅ "Let's spend a day[30] defining our architecture before anyone writes a single line of code."

Why? Without architecture:

- Tim creates a User object with fields: {name, email, id}
- Jane expects User with fields: {username, userEmail, userId}
- Integration fails—they're using different structures!
- Fix requires rewriting both implementations
- What should have been 1 hour becomes days of rework

Aspect	Without Architecture	With Architecture
Coding Speed	Fast initially (everyone codes own way)	Slower initially (must agree on structure)
Integration	Nightmare (nothing fits together)	Smooth (everything designed to connect)
Bugs	Many (interface mismatches)	Fewer (agreed contracts)
Total Time	2x-5x longer (rework)	On schedule (minimal rework)

[30] Or any time necessary.

Julie's Warning

Julie walks by and sees the chaos on their screens - different codebases with no apparent connection.

Julie

"Let me guess - you jumped straight into coding without an architecture?"

Ken

"We have requirements! So, we thought we could implement them independently."

Julie

"Requirements only tell you WHAT to build and test. Architecture tells you HOW to organize it for the data model to flow."

Tim

"Data model?"

The Leaning Tower Story

Julie pulls up a chair and shares her team's disaster.

Julie

"Last semester, our team built a social media app. We had great requirements, but no architecture."

She writes on the whiteboard.

- *3 developers coded independently*
- *Each used different patterns*
- *Nobody knew how the pieces fit together*
- *Integration was a nightmare*
- *Had to rebuild everything in Week 10*

Jane

"Ouch! So how do we avoid that?"

Real Integration Disasters

Julie's Team Without Architecture:

Week 1-8: Development

- Developer A: Built frontend expecting JSON responses
- Developer B: Built backend returning XML
- Developer C: Built a database with different field names than both

Week 9: Integration Attempt

- Frontend crashes (can't parse XML)
- Backend crashes (database fields don't match)
- Database queries fail (wrong field names)

Week 10-12: Emergency Rebuild

- Choose one approach (JSON)
- Rewrote all three components to match
- Missed two deadlines
- Final grade: C (late, buggy)

What They Should Have Done:

- Week 1: Define architecture (1 day)
- Week 1-8: Build to spec (integrated smoothly)
- Week 9-12: Polish & add features
- Final grade: A (on time, high quality)

What is Software Architecture?

Professor Cho joins the discussion.

> **Prof. Cho**
>
> "Architecture is like city planning. Imagine building a city without planning."

Without planning:

- *Houses built randomly*
- *Roads going nowhere*
- *No water/electricity plan*
- *Result: Total chaos!*

With city planning:

- *Zoned areas (residential, commercial, industrial)*
- *Connected road network*
- *Utilities planned first*
- *Result: Functional, livable city!*

Architecture Analogies

Analogy	Physical Architecture	Software Architecture
City Planning	• Zones (residential, commercial) • Roads connect zones • Utilities (water, power) follow rules	• Modules (UI, business logic, data) • APIs connect modules • Constraints (security, data formats)
House Blueprint	• Rooms with specific purposes • Doors/hallways connect rooms • Building codes (safety, electrical)	• Components with responsibilities • Interfaces connect components • Design patterns (proven solutions)
Restaurant	• Kitchen, dining room, storage • Wait staff move between areas • Health codes, fire safety	• Backend, frontend, database • APIs transfer data • Security, performance requirements

Architecture as Blueprints

Ken

"So, architecture is like blueprints for a house?"

Jane

"And it directs how a person can move from one room to the other."

Tim

"It sounds like real architecture!"

Prof. Cho

"Exactly! But in software architecture, it's not people that are moving, it's the data model."

Data Model

Prof. Cho

"To architect any software, we need to define the data model that we use for the software."

Jane

"What is the **data model**?"

Prof. Cho

"Data model is the data structure that is generated, transformed, stored, and shared in the software."

Tim

"So, it's like the main character of a movie."

Prof. Cho

"Yes. The data model is the main character that goes through the entire story of your application. Let me give you an example."

Data Model as Main Character

The Data Model Journey: Instagram Post

Stage	What Happens to the Post	Operation
Generation	User takes photo → Post object created	Create
Transformation	User adds filter, caption, tags → Post modified	Update
Storage	User hits "Share" → Post saved to database	Store
Retrieval	Friend opens app → Post fetched from database	Read

Sharing	Post appears in friend's feed	Display
Transformation	Friend likes post → Post likes incremented	Update
Deletion	User deletes post → Post removed	Delete

Key Insight: The Post data model is the thread that connects every part of Instagram. Frontend, backend, database, mobile app—all understand the same Post structure.

Prof. Cho

"Think about Instagram. What's the main character there?"

Jane

"A photo?"

Prof. Cho

"Close! It's actually a **Post**. A Post includes a photo, a caption, likes, comments, a timestamp, and who posted it. That whole package is the data model."

Tim

"So the Post travels through the app - from creation to storage to display?"

Prof. Cho

"Perfect! When you take a photo, Instagram **generates** a Post object. When you add filters and captions, it **transforms** the Post. When you hit share, it **stores** the Post in a database. When your friends scroll their feed, Instagram **shares** that Post with them."

Jane

"What about a shopping app like Amazon?"

Prof. Cho

"Good question! What do you think the main data model is?"

Tim

"A Product?"

Prof. Cho

"That's one of them! But Amazon actually has multiple important data models working together: Product, Cart, Order, User, Review. It's like a movie with multiple main characters."

Multiple Data Models Example: E-commerce

Amazon's Key Data Models:

```
// User Data Model
interface User {
  userId: string
  email: string
  name: string
  addresses: Address[]
  paymentMethods: PaymentMethod[]
}

// Product Data Model
interface Product {
  productId: string
  name: string
  price: number
  description: string
  images: string[]
  inventory: number
}

// Order Data Model
interface Order {
  orderId: string
```

```
  user: User
  items: OrderItem[]
  totalPrice: number
  status: 'pending' | 'shipped' | 'delivered'
  shippingAddress: Address
}
```

Why Multiple Models?

- Each represents a different concept
- They reference each other (Cart contains Products)
- Changes to one don't break others
- Easier to understand and maintain

Jane

"So you need to define all of these before building the app?"

Prof. Cho

"Exactly! Just like a movie needs a script before filming, we need to define our data models before architecture or design. Otherwise, every developer would imagine a different structure, and nothing would work together."

Tim

"What happens if we define it wrong?"

Prof. Cho

"Imagine filming a movie where the main character's name keeps changing, or they're a child in one scene and an adult in another. Chaos! Same with software - if your User model doesn't have an email field, but you try to send password reset emails, the whole system breaks."

Jane

"So the data model is like a contract that everyone agrees on?"

Prof. Cho

"Good analogy! Yes, it can be regarded as a contract. Once we agree that a Post has these specific fields, everyone - frontend, backend, database, mobile app - knows exactly what to expect."

Tim

"I see, no wonder I had no idea what to send from user input to Jane's CRUD server."

Jane

"Me too. I didn't have an idea what to store in my database."

Julie

"Yes, I was there too. But once my team figured out the data model for the team project, we could design how to flow the data model. In other words, we could design the **architecture**."

Jane

"I see, then for our project, the data model is simple. We need only Title and Date.[31]"

Jane writes down the data model on the whiteboard.

Data Model:

- *Title (string)*
- *Date (datetime)*

[31]For simplicity, we are only defining two fields for the data model in this example. In real applications, data models can be much more complex with many fields and relationships.

Tim

"That's simple, but good enough to describe what we need for this application."

Ken

"Yes, we need 'Date' for the notification with the 'Title', and that's enough for now.[32]"

[32] As we make progress, we can add more fields to the data model, but it's a good start for now.

Three Components of Architecture

Prof. Cho

"Software architecture has three key components: Elements, Interactions, and Constraints—we call this **EIC**."

EIC Framework Overview

Component	Description	Real-World Analogy	Software Example
Elements (E)	The building blocks/ modules	Buildings in a city	API Server, Database, UI
Interactions (I)	How elements communicate	Roads, utilities	REST API, function calls
Constraints (C)	Rules everyone follows	Building codes, zoning laws	Security policies, data formats

Component 1: Elements

> **Prof. Cho**
>
> "Elements are the building blocks - like the buildings in your city. We also call Elements **Modules**."

In Software, we use these terminologies to describe the idea of **Elements**[33]:

- *Modules*
- *Components*
- *Services*
- *Classes*
- *Units*

> **Tim**
>
> "So, elements are like the different parts of our app?"

> **Prof. Cho**
>
> "Exactly!"

[33] We use the terminology **modules** in software design.

Ken writes on the whiteboard:

Elements in Our System:

- *User Input Module*
- *Task Management Module*
- *Notification Module*
- *Database Module*
- *API Module*

Ken

"So each major piece is an element!"

Jane

"For me, I need the API Server module and DB module."

Jane

"The API server module receives API requests and processes them, and the DB module communicates with the remote database."

Tim

"I need the User Input module, API module, and Display module."

Tim

"Users give inputs to the User Input module, which uses the API module to make API calls to Jane's API server."

Defining Good Elements

Characteristics of Good Elements:

1. **Single Responsibility** - One clear purpose
   - ✅ "Authentication Module" handles login/logout
   - ❌ "User Module" handles login, profile, settings, notifications (too many responsibilities)
2. **High Cohesion** - Related functionality grouped together
   - ✅ Payment processing code all in Payment Module
   - ❌ Payment code scattered across User, Order, and Product modules
3. **Low Coupling** - Minimal dependencies on other elements

- ✅ Payment Module doesn't know about UI details
- ❌ Payment Module calls UI functions directly (tight coupling)

4. **Clear Boundary** - Well-defined inputs and outputs
   - ✅ "processPayment(order, paymentInfo) → PaymentResult"
   - ❌ "doStuff()" with unclear inputs/outputs

Common Mistake #2: 'God Module' Anti-Pattern:

❌ Creating one module that does everything:

```
class AppManager:
  - handles user login
  - processes payments
  - sends emails
  - manages database
  - renders UI
  [100+ methods, 5000+ lines]
```

✅ Separating into focused modules:

```
class AuthService: [handles authentication]
class PaymentService: [processes payments]
class EmailService: [sends emails]
class DatabaseService: [manages data]
class UIRenderer: [renders interface]
```

Why? God modules are impossible to maintain, test, or understand. Changes break everything.

Component 2: Interactions

> **Prof. Cho**
>
> "Interactions are like the roads and utilities in your city - how elements communicate. We also call Interactions **Interfaces**."

In Software, we use these terminologies to describe Interactions.[34]:

- *Interfaces*
- *API calls*
- *Database queries*
- *Event messages*
- *Function calls*

> **Jane**
>
> "So this is how our elements talk to each other?"

> **Prof. Cho**
>
> "Precisely!"

[34] We use the terminology **interfaces** in software design.

Tim

"Now I see how the pieces connect!"

Tim writes module interactions (interfaces).

Client Side Interactions:

- *User Input Module ⟶ API Module*
- *API Module ⟶ API Server Module*
- *API Server Module ⟶ Display Module*

Jane writes her interactions.

Server Side Interactions:

- *API Server Module ⟶ DB Module*
- *API Server Module ⟶ Display Module*
- *DB Module ⟷ Online DB*

Types of Interactions

Type	Description	Example	When to Use
Synchronous	Caller waits for response	Function call, REST API	Fast operations, need immediate result

Asynchronous	Caller doesn't wait	Message queue, event	Slow operations, can process later
Request-Response	Ask for data, get data back	HTTP request	Client needs specific data
Publish-Subscribe	Broadcast events, listeners react	WebSocket, event bus	Multiple components care about the same event
Database Query	Read/write to persistence	SQL, MongoDB query	Need to persist or retrieve data

Interaction Design Example: Email Notification:

Bad Interaction (Tight Coupling):

```
// UI Module directly talks to Email Service
class TaskForm {
  submitTask() {
    const task = this.getFormData()
    // UI knows about the database!
    // BAD!!!!
    database.save(task)
    emailService.send(task.user.email, "Task created")  // UI
knows about email!
  }
}
```

Problems:

- The UI module depends on the database and email services
- Can't change email service without changing UI
- Hard to test (must mock database and email)

Good Interaction (Loose Coupling):

```
// UI calls API, API orchestrates services
class TaskForm {
  submitTask() {
    const task = this.getFormData()
    // UI only knows about API
    api.createTask(task)
  }
}

class TaskAPI {
  async createTask(task) {
    await database.save(task)
    await emailService.notify('task_created', task)
    return task
  }
}
```

Benefits:

- UI only depends on API
- Can change email implementation freely
- Easy to test (mock just API)

Component 3: Constraints

> **Prof. Cho**
>
> "Constraints are the building codes and zoning laws - the rules everyone must follow."

In Software, we use these terminologies to describe Constraints:

- *Security rules*
- *Data formats*
- *Technology choices*
- *Performance requirements*
- *Coding standards*

> **Julie**
>
> "These are the rules everyone agrees to follow!"

Then she writes the project's constraints.

Our Constraints:

- *Delete API calls must be authenticated*
- *Use MongoDB for Online database (team decision)*
- *Use REST API for communication between client and server*
- *Follow REST principles for API design*
- *No direct database access from UI*
- *All API responses in JSON format*
- *Password must be hashed (bcrypt)*

Ken

"So constraints guide our implementation choices!"

Prof. Cho

"Yes. That is why we need the software architecture before diving into coding."

Types of Constraints

Type	Description	Example
Technology	What tools/ frameworks to use	React for frontend, Node.js for backend

Security	How to protect data and access	Password hashing, authentication methods
Performance	Speed and scalability requirements	API response time, concurrent users
Data Format	How data is structured and transmitted	JSON format, date formats
Coding Standards	How code should be written	Linting rules, test coverage

Common Mistake #3: Implicit Constraints:

❌ Team members use different approaches because constraints aren't documented:

- Tim uses snake_case: `user_name`
- Jane uses camelCase: `userName.`
- Result: Frontend and backend don't match!

✅ Document constraints explicitly:

```
# API Data Format Constraints
- Field names: camelCase
- Dates: ISO 8601 (YYYY-MM-DDTHH:mm:ssZ)
- Numbers: JSON number type (not strings)
- Boolean: true/false (not "true"/"false")
```

Why? Explicit constraints prevent integration bugs and wasted time.

Common Architecture Patterns

Ken

"So, how do we make good architecture?"

Prof. Cho

"My recommendation is to use proven patterns - templates that others have tested!"

Julie

"Like using a house plan instead of inventing your own!"

Prof. Cho

"You can invent your own software architecture, but until then, we can use well-known architecture patterns."

Why Use Patterns?

Aspect	Inventing Your Own	Using Proven Patterns
Time to Implement	Long (months of trial and error)	Short (ready-made solutions)
Risk of Failure	High (many unknowns)	Low (battle-tested)
Scalability	Uncertain (may not handle growth)	Known (used by millions)
Team Onboarding	Hard (new members confused)	Easy (standard names)
Maintenance Effort	Likely to need refactoring	Stable, well-documented

Inventing Your Own	Using Proven Patterns
• Months of trial and error • Make every beginner mistake • Uncertain if it scales • Hard to explain to new team members • Likely to need refactoring	• Start with a proven solution • Avoid common pitfalls • Know it scales (used by millions) • Standard names everyone understands • Stable, well-documented

Pattern 1: MVC (Model-View-Controller)

Prof. Cho

"MVC - Model-View-Controller - is like a restaurant..."

Model = Kitchen
- *Prepares food (data)*
- *Manages ingredients (business logic)*
- *Handles recipes (rules)*

View = Dining Room
- *What customers see (UI)*
- *Menu display (presentation)*
- *Table settings (styling)*

Controller = Waiter
- *Takes orders (user input)*
- *Coordinates between the kitchen and the customers*
- *Handles requests and responses*

Tim

"I remember I learned about this pattern when I learned web application development!"

Prof. Cho

"Yes. MVC pattern is one of the most widely used architecture patterns for web applications."

Jane

"One of the components is Model. Is it the same as the Data Model we discussed?"

Prof. Cho

"Good catch. Yes. The MVC architecture explains how the Data Model (managed by Model) is processed by the Controller and displayed by the View."

MVC in Detail

MVC Example: User Login:

```
// MODEL - Handles data and business logic
class User {
  async authenticate(email, password) {
    const user = await database.findUser(email)
    if (!user) throw new Error("User not found")

    const isValid = await bcrypt.compare(password,
user.hashedPassword)
```

```
    if (!isValid) throw new Error("Invalid password")

    return user
  }
}

// VIEW - Displays UI (React component)
function LoginView({ onSubmit, error }) {
  return (
    <form onSubmit={onSubmit}>
      <input name="email" type="email" />
      <input name="password" type="password" />
      <button>Login</button>
      {error && <div className="error">{error}</div>}
    </form>
  )
}

// CONTROLLER - Coordinates between Model and View
class LoginController {
  async handleLogin(email, password) {
    try {
      const user = await User.authenticate(email, password)
      this.view.showSuccess(user)
    } catch (error) {
      this.view.showError(error.message)
    }
  }
}
```

Data Flow:

1. User enters email/password in **View**
2. View calls **Controller**.handleLogin()
3. Controller calls **Model**.authenticate()
4. Model validates and returns the user

5. Controller updates **View** with result

Why MVC Helps

Jane

"So if I want to change how tasks look, I only touch the View?"

Prof. Cho

"Exactly! Each piece has one job. Model handles data and logic, View handles presentation, and Controller handles coordination."

Ken

"And they can change independently!"

Tim

"This is how we manage complexity!"

Prof. Cho

"Yes. That's how we understand software architecture as software engineers."

MVC Benefits[35]

Benefit	Why It Matters
Separation of Concerns	Model doesn't know about HTML, View doesn't know about database
Parallel Development	Designer works on View, backend dev works on Model, simultaneously
Testability	Can test Model logic without UI, test UI without real data
Reusability	Same Model can power web View, mobile View, API
Maintainability	Change database? Only touch the Model. Redesign UI? Only touch View.

Pattern 2: MVVM (Model-View-ViewModel)

> **Ken**
>
> "MVVM - Model-View-ViewModel - is similar to MVC but even better for complex UIs!"

[35] We discuss MVC architecture in more detail in the 200-level web development course.

Tim

"When do we use MVVM instead of MVC?"

Prof. Cho

"MVVM shines when you have complex UI interactions, real-time updates, and two-way data binding."

Prof. Cho

"You will learn how to use MVVM architecture to build mobile apps in the 400-level ASE course."

MVC vs MVVM

Aspect	MVC	MVVM
Controller/ViewModel	Controller: Handles user input, calls Model	ViewModel: Binds to View, automatically syncs
Data Flow	One-way: View → Controller → Model → View	Two-way: View ↔ ViewModel ↔ Model
Best For	Server-rendered web apps, REST APIs	Rich client apps, real-time UIs, mobile apps

Examples	Django, Ruby on Rails, Laravel	Flutter, SwiftUI, Vue.js, Angular
Complexity	Simpler, less boilerplate	More setup, but powerful for complex UI

MVVM Example: Flutter Counter App:

```
// MODEL
class CounterModel {
  int value = 0;

  void increment() => value++;
  void decrement() => value--;
}

// VIEWMODEL
class CounterViewModel extends ChangeNotifier {
  final CounterModel _model = CounterModel();

  int get count => _model.value;

  void increment() {
    _model.increment();
    // Automatically updates View!
    notifyListeners();
  }

// VIEW
class CounterView extends StatelessWidget {
  Widget build(BuildContext context) {
    return Consumer<CounterViewModel>(
      builder: (context, viewModel, child) {
        return Column(
          children: [
```

```
            Text('Count: ${viewModel.count}'), // Auto-
updates!
            ElevatedButton(
              onPressed: viewModel.increment,
              child: Text('+'),
            ),
          ],
        );
      },
    );
  }
}
```

Key Difference: When the count changes, the View automatically updates (data binding). No manual "update UI" code needed![36]

Pattern 3: Client-Server/3 Tier Architecture

Tim

"Wait, isn't our app already client-server?"

Prof. Cho

"Yes! Let me explain why..."

[36]Data binding is a powerful feature of MVVM that reduces boilerplate code and keeps UI in sync with data model automatically. Students will learn all the details in the 400-level ASE course.

He writes down on the whiteboard:

Client = Customer
- Orders coffee (sends requests)
- Waits for response
- Consumes product (displays data)

Server = Barista
- Receives orders
- Prepares coffee (processes requests)
- Serves customer (sends responses)

Communication = Order System
- Clear protocol (menu = API documentation)
- Standard process (order → payment → service)

Tim

"But our app has a database too. Does that change things?"

Prof. Cho

"Great observation! That makes it a **3-Tier Architecture**!"

He adds to the whiteboard:

3-Tier Architecture = Coffee Shop + Storage

Tier 1: Presentation (Client)

- Customer interface
- Mobile app, web browser
- Shows menu, takes orders

Tier 2: Application (Server)

- Barista (business logic)
- Processes orders
- Validates payments
- Manages workflow

Tier 3: Data (Database)

- Storage room
- Keeps recipes (product data)
- Stores customer history
- Tracks inventory

Prof. Cho

"Think about it: the barista doesn't memorize every recipe or customer order. They check the storage room (database) when needed!"

Tim

"So client-server is just 2-tier, but adding a database makes it 3-tier?"

Prof. Cho

"Exactly! Each tier has a specific job:

- Tier 1: **Show** information
- Tier 2: **Process** business logic
- Tier 3: **Store** data permanently"

Prof. Cho

"We can see this architecture pattern in real life pretty often. We separate the client and server, then we define communication between them."

Tim

"To manage complexity!"

Tim says proudly.

Our To-Do App: 3-Tier Architecture

Prof. Cho

"Haha, YES. Your project follows the 3-Tier pattern. And you can manage complexity using the 'divide and conquer' rule[37], with each tier handling its specific responsibility."

[37] Divide and Conquer is one of the most important software engineering principles.

Tier 1 - Presentation (Client):

- Web Browser (HTML/CSS/React)
- Displays UI and forms
- Sends HTTP requests
- Shows results to the user

Tier 2 - Application (Server):

- Node.js + Express
- Business logic and validation
- API endpoints (routes)
- Processes requests

Tier 3 - Data (Database):

- MongoDB
- Stores tasks permanently
- Manages data persistence
- Handles queries

Communication:

- Tier 1 ↔ Tier 2: REST API (HTTP)
- Tier 2 ↔ Tier 3: Database queries

Why 3-Tier Architecture?

Jane

"I see each tier can be developed, debugged, and tested independently. What are the other benefits of using a 3-Tier architecture?"

Prof. Cho writes down more:

Benefits:

1. *Separation of Concerns (each tier has one job)*
2. *Independent scaling (scale database separately from app server)*
3. *Multiple clients (web, mobile share the same tiers 2 & 3)*
4. *Security layers (database hidden from clients)*
5. *Easier maintenance (change one tier without affecting others)*
6. *Technology flexibility (can swap the database without changing the UI)*
7. *Centralized business logic (all rules in tier 2)*

Tim

"So the client never talks directly to the database?"

Prof. Cho

"Exactly! Please think of the coffee shop: customers (Tier 1) don't go into the storage room (Tier 3) themselves. They ask the barista (Tier 2), who knows the business rules and fetches from storage."

3-Tier in Practice

3-Tier Architecture Example: Creating a Task:

Tier 1 - Presentation (React):

```
// User clicks "Add Task" button
async function createTask(title, date) {
  // Tier 1 only knows about the API, not the database
  const response = await fetch('https://api.example.com/tasks', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ title, date })
  })

  if (!response.ok) throw new Error('Failed to create task')
  const task = await response.json()

  // Update UI to show new task
  displayTask(task)
}
```

Tier 2 - Application (Node.js/Express):

```
// API endpoint - handles business logic
app.post('/tasks', async (req, res) => {
  const { title, date } = req.body
```

```
  // Business logic: Validate input
  if (!title || !date) {
    return res.status(400).json({ error: 'Title and date required' })
  }

  // Business logic: Check if date is in the past
  if (new Date(date) < new Date()) {
    return res.status(400).json({ error: 'Date cannot be in the past' })
  }

  // Talk to Tier 3 (database)
  const task = await db.tasks.create({
    title,
    date,
    completed: false,
    createdAt: new Date()
  })

  // Send response back to Tier 1
  res.status(201).json(task)
})
```

Tier 3 - Data (MongoDB):

```
// Database schema definition
const taskSchema = new Schema({
  title: { type: String, required: true },
  date: { type: Date, required: true },
  completed: { type: Boolean, default: false },
  createdAt: { type: Date, default: Date.now }
})

// Tier 3 only handles data storage and retrieval
// No business logic here!
```

Key Points:

- Tier 1 (Client) doesn't know the database exists
- Tier 2 (Server) enforces ALL business rules
- Tier 3 (Database) only stores/retrieves data
- Each tier can be replaced independently

- Clear separation prevents spaghetti code

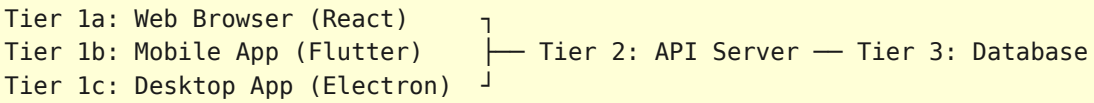
Ken

"What if we want to add a mobile app later?"

Prof. Cho

"Perfect question! You only need to build a new Tier 1 (mobile UI). Tiers 2 and 3 stay the same!"

Multiple Clients, Same Backend:



Each client uses the SAME API!

Our Architecture Decision

Ken

"Based on our discussion, our architecture should be 3-Tier with EIC[38]:"

Jane

"The data model has two components: Title and Date."

Tim

"I see how the **Data model** flows through all three tiers."

Jane

"I see the **elements** (three tiers) and **interactions** between them."

[38]Elements, Interactions, and Constraints

Ken

"Also **constraints** - like which tier can talk to which."

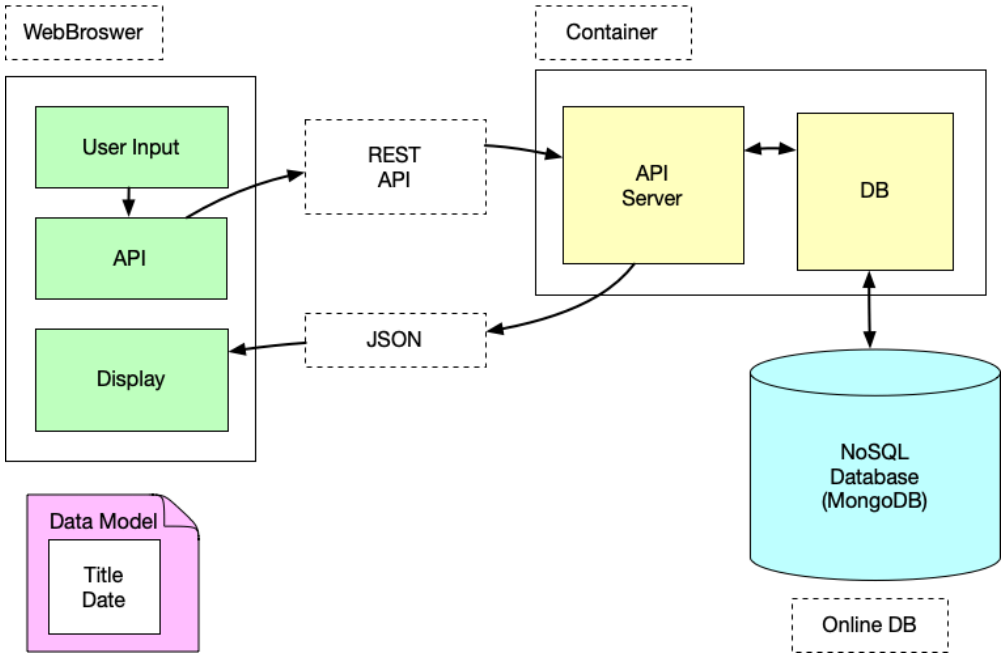
Prof. Cho

"Even though you need to fill in all the details from now on when you implement the detailed features, this is a good blueprint for all of you."

Ken

"Based on our discussion, this is our software architecture.[39] "

[39]This is a simplified architecture diagram for illustration purposes. In real applications, the architecture can be more complex with additional components and interactions.



Prof. Cho

"Great job, team! With this solid architecture, your project is set up for success. Remember, good architecture is the foundation of great software."

Summary: From Chaos to Structure

What the Team Learned:

Concept	Before 3-Tier	After 3-Tier
Data Model	Each developer imagined a different structure.	Single agreed structure: {title, date, completed}
Elements	Unclear modules, mixed responsibilities	Clear tiers: Presentation, Application, Data
Interactions	"I'll just call your functions somehow."	Tier 1 ↔ 2: REST API; Tier 2 ↔ 3: DB queries
Constraints	Business logic is scattered everywhere.	ALL logic in Tier 2, strict tier boundaries
Integration	Would have failed spectacularly	Smooth integration expected

Key Principles Learned:

1. **3-Tier Separation**
   - Tier 1: What users see (Presentation)
   - Tier 2: What rules to follow (Business Logic)

- Tier 3: Where data lives (Persistence)
- Never skip tiers!

2. **EIC Framework**
   - Elements: Three distinct tiers
   - Interactions: API between 1-2, queries between 2-3
   - Constraints: No Tier 1 to Tier 3 direct access
3. **Proven Patterns**
   - MVC for single applications
   - MVVM for complex UIs
   - 3-Tier for distributed systems
   - Don’t reinvent the wheel
4. **Architecture Before Code**
   - 2 hours of architecture saves weeks of rework
   - Enables parallel development
   - Makes integration smooth
   - Essential for team projects
5. **Documentation Matters**
   - Diagram the three tiers
   - Document interactions between tiers
   - Keep tier responsibilities clear
   - Update as the project evolves

Chapter 6:
Software Design

With requirements defined and architecture chosen, the team is eager to start coding. But there's one more crucial step: designing the exact structure of their code through modules and interfaces.

- Modules are self-contained units of functionality
- Interfaces are contracts defining how modules communicate
- Design patterns provide proven solutions to common problems

Quick Reference Guide

Design Fundamentals - Quick Summary:

What is Design?

- Detailed blueprints before coding
- Defines modules (units of functionality)
- Defines interfaces (communication contracts)
- Specifies exact structure & behavior

Why Design Matters

- Prevents coding chaos
- Enables testing
- Facilitates teamwork
- Supports maintainability

Good Module Characteristics

- Single Responsibility
- High Cohesion
- Low Coupling
- Clear Interface

Interface Benefits

- Contract between modules
- Multiple implementations possible
- Easy to test with mocks
- Program to interface, not implementation

Notice

This chapter explains core software design concepts.

1. Software Design is a significant topic with a dedicated course: ASE 420.
2. ASE 420 covers: OOP design (APIE, SOLID), UML diagrams, Design Patterns, Refactoring
3. This chapter focuses on core ideas: modules and interfaces
4. If you haven't taken ASE 420: think of design as creating modules with clear interfaces
5. This story simplifies the overall design process (in reality, prototyping often helps refine design), but the key concepts remain the same.

The Coding Trap, Again

Ken felt confident with their 3-Tier architecture in place.

Ken

"We have our software architecture. Let's start coding!"

Tim opened his editor enthusiastically.

Tim

"I'll make the User Input part!"

He started typing JavaScript code:

Jane

"Wait, what properties do you need? What methods?"

Tim

"Uh... I'll figure it out as I code?"

Common Mistake #1: Coding Without Design:

❌ “I’ll figure out the structure as I code.”

✅ “I’ll design modules and interfaces first, then code follows naturally.”

Why? Coding without design leads to unclear responsibilities, tangled dependencies, untestable code, and constant refactoring.[40]

Julie Intervenes Again

Julie

“Here’s what actually went wrong for us:”

[40] In the real world, prototyping helps when you’re uncertain about the design. However, you should refine the design afterward—and be prepared to throw away the prototype.

- *Week 4: "We already have architecture, let's code!"*
- *Week 5: One giant `Task` class doing everything*
- *Week 6: API calls return different shapes; nobody knows what depends on what*
- *Week 7: Unit tests impossible — components not isolated*
- *Week 8: Integration fails — modules assume different contracts*
- *Week 9: Circular dependencies between modules*
- *Week 10: Rewrite from scratch because **no one defined interfaces first***

Ken

"So architecture wasn't the problem?"

Julie

"Exactly. The architecture defined WHERE things go—but we never defined WHAT each module does or HOW they communicate through clean interfaces."

Team

"So design = modules + interfaces BEFORE code?"

Julie

“YES — design is the contract between modules so everyone builds compatible pieces.”

What is Software Design?

Professor Cho joins the discussion.

Prof. Cho

"Software design is about two things: Modules, which are self-contained units of functionality, and Interfaces, which are contracts defining how modules communicate."

Tim

"But we already discussed that architecture has Elements, Interactions, and Constraints. Are they different from design?"

Prof. Cho

"Good question! Think of design as the detailed blueprints within the architecture. Let me show you the difference:"

Architecture vs Design: Levels of Abstraction

Aspect	Architecture (Big Picture)	Design (Details)
Level	High-level structure	Low-level structure
Focus	WHAT & WHERE	HOW (specifically)
House Example	3 bedrooms, two baths, kitchen	Main bedroom: 12×14ft, door: 36in wide
Software Example	Client-server, REST API, PostgreSQL database	UserController class, getUserById(id: int) method
Output	Component diagrams, system overview	Class diagrams, interface definitions, method signatures
Diagram Tools	No unified architecture diagram	UML class diagrams, sequence diagrams

Architecture (Previous Lecture):

- The house has three bedrooms, two bathrooms
- Connected to water, electricity, sewage
- Garage attached to the east side

✅ Answers: **What** components exist? **Where** are they? **How** do they connect?

Design (This Lecture):

- Main bedroom: 12×14 feet, two windows (north wall)
- Master bath: Walk-in shower + soaking tub
- All doors: 36 inches wide, swing inward
- Windows: Double-pane, low-E coating

✅ Answers: **Exact specifications** for building each component

Prof. Cho

"Architecture defines the big structural decisions. Design fills in the precise details needed for implementation."

Prof. Cho

"Here's the key: each architectural element can be decomposed into multiple modules during design."

Architecture to Design: Decomposition

Architecture Element: "User Management System"

↓ (decomposed into)

Design Modules:

- UserController (handles requests)
- UserService (business logic)
- UserRepository (data access)
- UserValidator (validation rules)

Why Design Before Coding?

Tim

"Can't we just design as we code?"

Prof. Cho

"Let me show you a horror story..."

Code Without Design

Professor Cho projected the code on the screen:

JavaScript

```
function doStuff(data) {
  let x = data.email
  if (x.indexOf('@') > 0) {
    let y = x.split('@')
    if (y[1] === 'nku.edu') {
      // 50 more lines of mixed logic
      db.users.insert({email: x, password: data.pwd})
      sendEmail(x, 'Welcome!')
      logActivity('user_created', x)
      // validation + database + email + logging all mixed!
    }
  }
}
```

Prof. Cho

"You can see the problems with this approach without design. Right?"

He writes down on the whiteboard:

Problems Without Design:

1. **Unclear Purpose:** What does this function do? (Everything!)
2. **Untestable:** How do you test database, email, and validation together?
3. **Not Reusable:** Can't use email validation elsewhere
4. **Hard to Understand:** Others need hours to figure it out
5. **Fragile:** Any change breaks everything
6. **Tight Coupling:** Database, email, and validation are all tangled together

Code With Good Design

Prof. Cho

"Now watch this - with design:."

JavaScript

```
// Module 1: Validation (Single Responsibility)
class EmailValidator {
```

```
  isValid(email) {
    return email.includes('@') && email.includes('.')
  }

  isNKUEmail(email) {
    return email.endsWith('@nku.edu')
  }
}
```

JavaScript

```
// Module 2: User Service (Orchestrates the flow)
class UserService {
  constructor(validator, db, notifier, logger) {
    this.validator = validator  // Dependency injection
    this.db = db
    this.notifier = notifier
    this.logger = logger
  }
```

JavaScript

```
  async createUser(email, password) {
    // Step 1: Validate
    if (!this.validator.isValid(email)) {
      throw new Error('Invalid email')
    }

    // Step 2: Create user in database
    const user = await this.db.users.create({
      email,
      password: await this.hashPassword(password)
    })

    // Step 3: Send notification
    await this.notifier.sendWelcome(email)
```

```
    // Step 4: Log activity
    await this.logger.log('user_created', { email })

    return user
  }

  private async hashPassword(password) {
    // Password hashing logic isolated
    return bcrypt.hash(password, 10)
  }
}
```

Design Pattern: Dependency Injection:

Notice how `UserService` receives its dependencies (validator, db, notifier, logger) through the constructor rather than creating them internally.

Benefits:[41]

- Easy to test (inject mocks)
- Flexible (swap implementations)
- Clear dependencies (visible in the constructor)
- Reusable (not tied to specific implementations)

[41] Dependency Injection and many software design ideas will be discussed in ASE 420.

Benefits of Good Design

Benefits With Design:

1. ***Single Responsibility:*** *Each module has ONE clear purpose*
2. ***Testable:*** *Can test each part independently*
3. ***Reusable:*** *EmailValidator works anywhere*
4. ***Understandable:*** *Clear, self-documenting code*
5. ***Maintainable:*** *Changes are isolated*
6. ***Flexible:*** *Easy to swap implementations*

Prof. Cho

"We need to design software before implementation: modules and interfaces."

Common Mistake #2: God Objects:

❌ One class/module that does everything[42]

✅ Multiple focused modules, each with a single responsibility

Example:

[42] In software design, we call this 'the SRP(Single Responsibility Principle) violation').

- Bad: `TaskManager` handles database, validation, email, logging, UI
- Good: Separate modules for `TaskService`, `TaskValidator`, `TaskRepository`, `NotificationService.`

What Are Modules?

Jane

"Wow! The second version is so much clearer and easier to understand!"

Prof. Cho

"Think of modules as rooms in your house."

The Room Analogy

Aspect	House Room	Software Module
Purpose	Bedroom → sleeping	EmailValidator → validate emails
Boundaries	Walls separate rooms	Class/file boundaries
Internal	How furniture is arranged	Private methods, data structures
Interface	Door for entry	Public methods
Privacy	What happens inside is private	Implementation details hidden

Each Room (Module) Has:

1. Single Purpose

- Bedroom ⟶ Sleeping
- Kitchen ⟶ Cooking
- Bathroom ⟶ Personal care

2. Clear Boundaries

- Walls separate rooms
- Each room is independent

3. Internal Organization

- How furniture is arranged
- What items are stored
- Others don't need to know

Software Modules in Our App

> **Ken**
>
> "I see. We should design modules and interfaces for the architecture before making code."

Prof. Cho writes on the whiteboard:

User Input Module:

- **Purpose:** Handle user input, authenticate users, and pass valid requests to the server
- **Public Interface** (What others can use):
  1. login(username, password) ⟶ AuthToken
  2. validate(token) ⟶ boolean
  3. getUserProfile(token) ⟶ UserDTO
- **Private Implementation** (Internal details):
  1. hashPassword(password) ⟶ string
  2. compareHash(password, hash) ⟶ boolean
  3. generateJWT(userId) ⟶ string
  4. validateToken Expiration(token) ⟶ boolean

API Server Module:

- **Purpose:** Receive HTTP/REST requests, route to logic modules
- **Public Interface:**
  1. POST /login ⟶ calls UserInput.login()
  2. GET /profile ⟶ calls UserInput.getUserProfile()
  3. GET /tasks?userId=X ⟶ calls Database.getTasksByUser()
  4. Returns JSON in standard format: `{success, data, error}`
- **Private Implementation:**
  1. parseRequest(req) ⟶ RequestDTO
  2. validateInput(data) ⟶ boolean
  3. formatResponse(data) ⟶ JSON
  4. handleError(error) ⟶ ErrorResponse

Tim

"We define: (1) the goal of each module, (2) its public interface, and (3) what it does internally. This gives clear guidelines for implementation. I understand why we need software design!

Prof. Cho

"Yes. This is just an example—what we include in design documents can vary, but anyone should be able to implement from the design."

Module Characteristics: Good vs Bad

Prof. Cho

"We have a dedicated course on this, but here's a simple summary of key design principles we discuss in the class:[43]"

Principle	Good Practice ✅	Bad Practice ❌
Single Responsibility	Does ONE thing well	Does many unrelated things
High Cohesion	Everything inside relates to purpose	Random, unrelated methods
Low Coupling	Minimal dependencies	Knows too much about others
Clear Interface	Well-defined public methods	Confusing, inconsistent API

[43]This is just a simplistic overview. ASE 420 covers these principles in depth.

Encapsulation	Hides implementation details	Exposes everything as public

What Are Interfaces?

Ken

"If modules are rooms, what are interfaces?"

Prof. Cho

"Interfaces are doors and windows!"

The Door and Window Analogy

Aspect	Doors/Windows	Software Interface
Access	Define how to enter/exit	Define how to use module
Contract	Set rules (knock first)	Method signatures, parameters
Abstraction	Hide room's construction	Hide implementation details
Visibility	See what's inside	Documentation, type definitions

Doors and Windows:

- *Define HOW to enter/exit*
- *Show WHAT you can expect inside*
- *Set RULES for interaction*
- *Hide internal details*

You don't need to know:

- *How the room is built*
- *What's behind the walls*
- *Internal organization*

You only need to know:

- *Where's the door?*
- *What can I do inside?*
- *What are the rules?*

Prof. Cho

"The core idea is **abstraction**: we don't need to know module details, only how to use the module through its interface."

Software Interfaces

Tim

"I see! Interface = Contract between modules."

Prof. Cho

"Exactly! Here's a JavaScript interface example:"

JavaScript

```
// Interface: Payment Processor
// This is a CONTRACT - defines WHAT, not HOW

interface PaymentProcessor {
  // Process a payment
  // Parameters: amount in dollars
  // Returns: Promise that resolves to transaction ID
  processPayment(amount: number): Promise<string>

  // Refund a payment
  // Parameters: transaction ID from processPayment
  // Returns: Promise that resolves to a boolean (success/failure)
  refundPayment(transactionId: string): Promise<boolean>

  // Check payment status
  // Parameters: transaction ID
  // Returns: Promise resolving to status string
  // Possible values: 'pending', 'completed', 'failed',
'refunded.'
  getPaymentStatus(transactionId: string): Promise<string>
}
```

Prof. Cho

"Important: **Interface means contract** — not implementation! It states **what** functionality a module promises, not **how** it's done.[44]"

Jane

"Do we have to use a programming language's Interface feature when we design?"

Prof. Cho

"Not necessarily. Some languages don't have a built-in `interface` keyword. What matters is the **idea** of a clearly defined contract."

Ken

"In OOP languages, are public methods considered the interface?"

[44]Software design directs what to implement, but the actual code can vary widely as long as it fulfills the contract.

Prof. Cho

"Exactly! Any public method that other modules rely on is part of the **interface**. It's the agreed contract for how modules communicate."

Why Interfaces Matter

Tim

"I'm still not convinced. Why not just write the code using concrete classes? I see no problems as long as it works and implements the functionality specified by the architecture. It seems like making interfaces is just extra work."

Prof. Cho

"Let me show you the power of interfaces..."

Multiple Implementations

Professor Cho demonstrated:

Prof. Cho

"With a PaymentProcessor interface, we can implement it for CreditCard OR PayPal—same interface, different implementations."

JavaScript

```
// Implementation 1: Credit Card
class CreditCardProcessor implements PaymentProcessor {
  async processPayment(amount: number): Promise<string> {
    console.log('Charging $${amount} to credit card`)
    // Call credit card API...
    return "CC-12345"  // transaction ID
  }

// Implementation 2: PayPal
class PayPalProcessor implements PaymentProcessor {
  async processPayment(amount: number): Promise<string> {
    console.log('Processing $${amount} via PayPal`)
    // Call PayPal API...
    return "PP-67890"  // order ID
  }
```

Prof. Cho

"When we need to use a Credit card, we simply instantiate CreditCardProcessor. When we need PayPal, we instantiate PayPalProcessor. Both conform to the same PaymentProcessor interface!"

JavaScript

```
// Usage: Same interface, different implementations!
let processor: PaymentProcessor

processor = new CreditCardProcessor()
await processor.processPayment(100)  // Works!

processor = new PayPalProcessor()
await processor.processPayment(100)  // Also works!
```

The Magic of Interfaces: Polymorphism

Prof. Cho

"Your code can work with ANY PaymentProcessor without knowing which specific implementation!"

Jane

"Wow! We can swap payment methods without changing much code! I can see this is a powerful way of managing complexity."

Prof. Cho

"Yes, in this example, we manage complexity using the tool called **polymorphism** to harness the power of interfaces. Program to interfaces, not implementations!"

Tim

"Wow! I get it now. Interfaces enable flexibility, testability, and maintainability. If I just code to concrete classes, I lose all that power."

Common Mistake #3: Coding to Implementations:

❌ Hard-coded dependency on CreditCardProcessor

```
const service = new CheckoutService(new
CreditCardProcessor())
```

✅ Depends on interface, not implementation

```
const processor: PaymentProcessor = getProcessor()
const service = new CheckoutService(processor)
```

Why? Interfaces enable flexibility, testing, and future changes without modifying existing code.

Designing Our To-Do App

Ken

"Let's design our modules and interfaces!"

The team gathered around the whiteboard.

Data Model: The Foundation

Prof. Cho

"Before we design anything, we need to talk about the data model that we agreed on. Remember - this becomes the **language** all modules speak."

Jane

"From our requirements, the data model is simple. We need only the Title and Date to express the task.[45]"

[45]For simplicity, we are only defining two fields for the data model in this example. In real applications, data models can be much more complex with many more fields.

Jane writes down the data model on the whiteboard.

Data Model:

Task

- *title (string)*
- *date (datetime)*

Jane

"This Task data model will be the core structure that all elements use to create, read, update, and delete tasks. This is a TypeScript representation:[46]"

TypeScript

```
// Task Data Model
type Task = {
  title: string
  date: Date
}
```

[46]We use TypeScript because its type system allows us to explicitly define data structures and interfaces, making our design clearer and easier to understand. While we show TypeScript syntax, these same data modeling concepts apply to any programming language.

Tim

"That's simple, but good enough to describe what we need for this application."

Prof. Cho

"Perfect! This Task data model will appear in the interfaces of modules that handle tasks. Watch how it becomes the connector between all our modules."

He draws on the whiteboard:

*How the **Task Data Model** Flows Through the System:*

User Input ⟶ Task {title, date} ⟶ API Client ⟶ Server

Server ⟶ Task {title, date} ⟶ Database

Database ⟶ Task {title, date} ⟶ Display Module

Same structure everywhere = Easy communication!

Ken

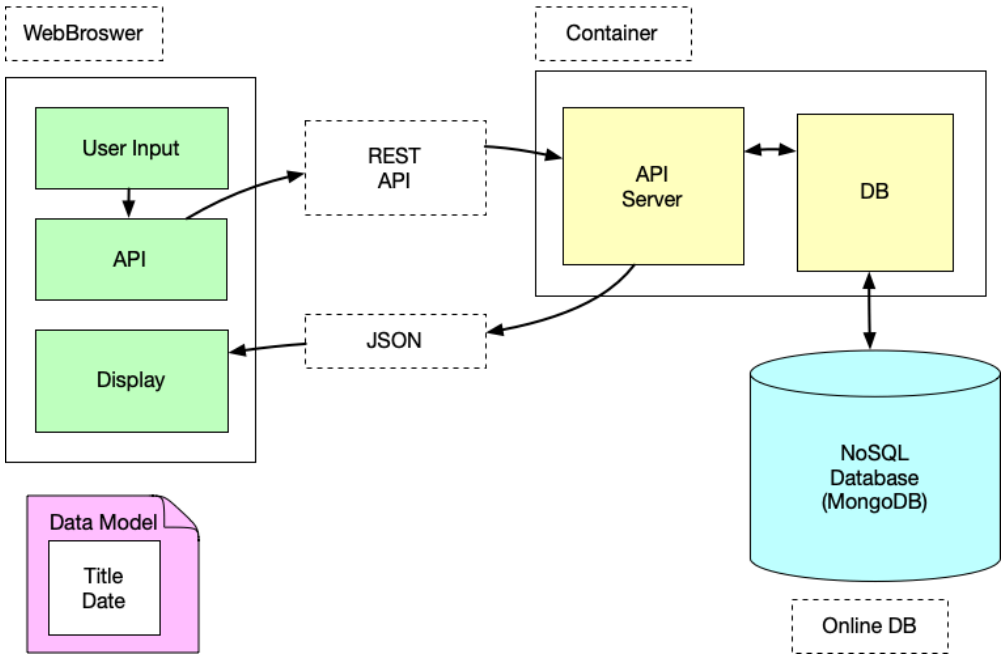
"Oh! So if every module uses the same data model—I mean, in this case, the Task structure—they can all work together seamlessly?"

Prof. Cho

"Exactly! That's why we define the data model FIRST. Now let's see how each module uses it."

Step 1: Identify Modules from Architecture

From their 3-tier architecture, they identified the needed modules:



Tim

"Based on our 3-tier architecture, I see we need modules in Tier 1 (Presentation)."

Tier 1 - Presentation Layer Modules:

1. **User Input Module**
   - *Purpose: Collect task data from the user*
   - *Uses: Task {title, date}*
2. **API Client Module**
   - *Purpose: Send Task to server, receive Task from server*
   - *Uses: Task {title, date}*
3. **Display Module**
   - *Purpose: Show the Task list to the user*
   - *Uses: Task {title, date}*

Jane

"And we need modules in Tier 2 (Application) and Tier 3 (Data)."

Tier 2 - Application Layer Modules:

1. **API Routes Module**
   - *Purpose: Receive HTTP requests with Task data*
   - *Uses: Task {title, date}*
2. **Task Service Module**
   - *Purpose: Business logic for Task operations*
   - *Uses: Task {title, date}*

Tier 3 - Data Layer Module:

1. *Database Module*
   - *Purpose: Store and retrieve the Task permanently*
   - *Uses: Task {title, date}*

Prof. Cho

"Notice anything? Every module works with the same Task structure!"

Step 2: Define Module Interfaces

Tim

"Now let's define what each module does with the Task data model."

Ken

"I'll start with User Input Module - it creates Task objects."

TypeScript

```
/**
 * User Input Module Interface
 * Collects data from the user and creates a Task object
```

```
 */
interface IUserInput {
  // Collects title and date from user form
  // Returns Task using our data model
  collectTask(): Promise<Task>

  // Validates input before creating Task
  validateInput(title: string, date: string): boolean
}
```

Prof. Cho

"Good! The interface returns a Task object. Now, who receives this Task?"

Jane

"The API Client Module! It sends the Task to the server."

TypeScript

```
/**
 * API Client Module Interface
 * Sends Task to server, receives Task from server
 */
interface IAPIClient {
  // Sends Task object to server via HTTP POST
  createTask(task: Task): Promise<Task>

  // Gets an array of Task objects from the server
  getAllTasks(): Promise<Task[]>

  // Updates existing Task on server
  updateTask(id: string, task: Task): Promise<Task>

  // Deletes Task from server
```

```
    deleteTask(id: string): Promise<boolean>
}
```

Tim

"I see the pattern! The server also uses the same Task structure."

TypeScript

```
/**
 * Task Service Module Interface (Tier 2)
 * Receives Task from API Routes, passes to the Database
 */
interface ITaskService {
  // Receives Task, stores in database, returns saved Task
  createTask(task: Task): Promise<Task>

  // Retrieves all Tasks from the database
  getAllTasks(): Promise<Task[]>

  // Updates Task in database
  updateTask(id: string, task: Task): Promise<Task>

  // Deletes Task from database
  deleteTask(id: string): Promise<boolean>
}
```

Ken

"And the Display Module receives the Task to show to users:"

JavaScript

```
/**
 * Display Module Interface
 * Receives Task objects and shows them to the user
 */
interface IDisplay {
  // Takes an array of Task and renders a list
  showTaskList(tasks: Task[]): void

  // Shows a single Task in detail
  showTaskDetail(task: Task): void

  // Formats Task.date for display
  formatDate(task: Task): string
}

// Same Task type in Display!
type Task = {
  title: string
  date: Date
}
```

Prof. Cho

“Excellent! Now let’s see the complete data flow:.”

Complete Task Flow:

1. User Input.collectTask() → creates Task {title: "Buy milk", date: "2024-03-15"}
2. APIClient.createTask(task) → sends the same Task to the server
3. TaskService.createTask(task) → validates and processes the same Task
4. Database.save(task) → stores the same Task permanently
5. Display.showTaskList([task]) → shows the same Task to the user

The same Task object flows through the entire system!

Why Data Model First?

Jane

"I get it now! If we hadn't agreed on the Task structure first, chaos would have happened!"

Prof. Cho

"Exactly. Let me show you what happens without an agreed data model:"

Without an Agreed Data Model:

User Input creates: {name: "Buy milk", deadline: "2024-03-15"} API Client expects: {title: string, dueDate: Date} Task Service expects: {taskName: string, date: string} Display expects: {description: string, when: Date}

Result: Nothing works! Everyone speaks a different language!

With Agreed Data Model (Task):

ALL modules use: {title: string, date: Date}

Result: Perfect communication! Everyone speaks the same language!

Tim

"So the data model is like a contract that everyone agrees to follow?"

Prof. Cho

"Exactly! And if we need to add a field later, we update ONE place."

TypeScript

```
// Original data model
type Task = {
  title: string
  date: Date
```

```
}

// Updated data model - add 'completed' field
type Task = {
  title: string
  date: Date
  completed: boolean  // NEW field
}

// Now ALL modules automatically know about 'completed.'
// because they all reference the same Task type!
```

Jane

"I also can see that even when we change the data model, the interfaces remain consistent across modules. This makes maintenance so much easier!"

Tim

"This makes our code future-proof! We can evolve the data model without breaking everything. In other words, we are building high-quality software!"

Step 3: Document Module Dependencies

Ken

"We should also map which modules depend on which:"

Module Dependency Flow (Top to Bottom):

```
Tier 1: User Input → API Client → Display
↓              ↓              ↑
Tier 2:          API Routes → Task Service
↓
Tier 3:                     Database
```

Rules:

- *Higher tiers depend on lower tiers*
- *Same-tier modules can communicate*
- *Never create circular dependencies*
- *Task flows down, responses flow up*

Tim

"What does 'higher tiers depend on lower tiers' mean exactly?"

Prof. Cho

"Good question! It means Tier 1 needs Tier 2 to do its job, and Tier 2 needs Tier 3. For example, the API Client in Tier 1 can't save a task by itself - it depends on the API Routes in Tier 2 to handle the request."

Tim

"So Tier 1 calls Tier 2, and Tier 2 calls Tier 3?"

Prof. Cho

"Exactly! But never the reverse. Tier 3 doesn't call Tier 2, and Tier 2 doesn't call Tier 1."

Jane

"I see the dependency direction, but what about 'Task flows down, responses flow up'?"

Prof. Cho

"Good catch! When a user creates a task, the Task data flows down from Tier 1 → Tier 2 → Tier 3. But the response - success or error - flows back up: Tier 3 → Tier 2 → Tier 1."

He adds to the whiteboard:

Data Flow Example (Creating a Task):

Down (Request):

```
Tier 1: User creates {title: "Buy milk", date: "2024-03-15"}
↓
Tier 2: Validates and processes the Task
↓
Tier 3: Saves Task to database
```

Up (Response):

```
Tier 3: Returns saved Task with ID
↑
Tier 2: Adds business logic (e.g., schedule reminder)
↑
Tier 1: Shows "Task created successfully!" to the user
```

Jane

"So Task data goes down the tiers, and the result comes back up?"

Prof. Cho

"Precisely! This is why we say 'Task flows down, responses flow up.' It's a two-way street, but the data and responses travel in opposite directions."

Common Mistake #4: Mismatched Data Structures:

❌ Each module defines its own version of Task

✅ Single Task definition used by ALL modules

Example of the problem:

```
// UserInput.ts
type Task = { name: string, when: Date }  // Different!

// TaskService.ts
type Task = { title: string, date: Date }  // Different!

// This causes TypeScript errors and runtime bugs!
```

Fix: Define Task ONCE in a shared location, import everywhere:

```
// models/Task.ts (shared file)
export type Task = {
  title: string
  date: Date
}

// UserInput.ts
import { Task } from './models/Task'

// TaskService.ts
import { Task } from './models/Task'

// Now everyone uses the SAME definition!
```

Step 4: See the Complete Picture

Prof. Cho

"Let's trace a complete example - creating a task:"

TypeScript

```
// User fills form and clicks "Add Task."

// Step 1: User Input Module collects data
const userInput = new UserInputModule()
const newTask = await userInput.collectTask()
// newTask = { title: "Buy milk", date: Date("2024-03-15") }

// Step 2: API Client sends to server
const apiClient = new APIClient()
const savedTask = await apiClient.createTask(newTask)
// Server receives the same structure, processes, and returns with the ID

// Step 3: Display Module shows result
const display = new DisplayModule()
display.showTaskDetail(savedTask)
// Displays: "Buy milk - Due: March 15, 2024."

// Same Task object, different modules, perfect harmony!
```

Jane

"This is so clear! The data model is the foundation, and interfaces show how modules pass it around."

Ken

"And because everyone agreed on the Task structure upfront, integration will be smooth!"

Tim

"Now coding will be straightforward - we know exactly what each module does!"

Prof. Cho

"More importantly, when you need to fix bugs or add features, you'll know exactly where to look and what to change. That's the power of good design!"

Prof. Cho

"Design is not just documentation - it's the blueprint that makes integration smooth and maintenance easy!"

Jane

"But we need the architecture for the good design, and the requirement for the good architecture! Everything builds on top of each other."

Prof. Cho

"Yes, software engineering is a layered process. Each phase builds on the previous one. Skipping steps leads to chaos!"

He writes down the key takeaways on the whiteboard to conclude:

Key Design Principles:

1. ***Data Model First:*** *Define the shared vocabulary before anything else*
2. ***Consistent Types:*** *Same Task structure in ALL module interfaces*
3. ***Clear Flow:*** *Task flows through the system in a predictable way*
4. ***Single Source of Truth:*** *One Task data model definition, imported everywhere*
5. ***Easy Changes:*** *Update the data model once, and all modules adapt*

Tim's Realization

Tim reflected on what he'd learned:

Tim thinks

"Design is like:

- *Detailed blueprints giving specifications before building*
- *A user manual telling you how to use each part*
- *LEGO instructions showing how pieces fit together*
- *An insurance policy protecting against chaos!*

Without design, we code in chaos. With design, implementation becomes straightforward."

Team

"Now we can code with confidence! And if we need to change something, we just update the design first. I think I know how to build high-quality software now!"

Jane

"I can clearly see that good design is the path to high-quality software!"

Success Indicators:

Your design is ready for implementation when:

- ✅ Every module has a single, clear purpose
- ✅ All interfaces are documented with types
- ✅ Dependencies flow in one direction (no cycles)
- ✅ Team members can explain any module's role
- ✅ You can write tests before writing the implementation
- ✅ Multiple people can work on different modules independently

Ready to Code

With requirements defined, architecture chosen, and design documented, the team was finally ready to start coding—but this time with confidence and clarity.

Ken

"Now we understand the full software engineering process."

1. **Requirements**: WHY solve this problem? WHAT to build?
2. **Architecture**: HOW to organize it at a high level? WHAT components?
3. **Design**: HOW to structure it in detail? WHAT modules and interfaces?
4. **Implementation**: Write code following the design and make tests to verify it works as intended!

Jane

"And we learned all this by doing, not just reading![47]"

Tim

"I can't wait to apply these principles to our project!"

Ken

"Remember: The design phase pays dividends during implementation.

- Good design → fast implementation
- No design → slow, chaotic implementation with constant refactoring

Invest time in design upfront, save time overall."

[47] Hands-on experience is the best way to internalize software engineering concepts. So, you should try building projects applying these principles.

Summary: Architecture vs Design

Aspect	Architecture	Design
Focus	System structure	Module structure
Question	What components? How do they relate?	What methods? What are parameters?
Abstraction	High-level	Detailed
Output	Component diagrams	Class diagrams, interfaces
Example	Client-Server, MVC pattern	UserService class with login() method
Changes	Expensive to change	Moderate to change

Chapter 7:
Coding & Testing

With their design complete, the team is ready to write code. But first, they need to understand how to verify that their code actually works correctly.

- Testing ensures code quality and prevents bugs
- Different types of tests serve different purposes
- Test-Driven Development (TDD) changes how we think about coding

Quick Reference Guide

The Testing Pyramid

```
      /\
     /E2E\     Few, slow, fragile
    /----\
   /Integ-\    Medium number
  /  ration\
 /----------\
/   Unit     \ Many, fast, focused
--------------
```

Test Distribution (70-20-10)

- 70% Unit Tests (fast feedback)
- 20% Integration Tests (modules work together)
- 10% E2E Tests (user workflows)
- Regression: Run all on every change

Unit Test Template

```
test('description', () => {
  // Arrange: setup
  // Act: call function
  // Assert: verify result
})
```

When to Write Tests

- ✅ Unit tests: with every function
- ✅ Integration: with feature completion
- ✅ E2E: for critical user flows
- ✅ Regression: automated on every commit

From Design to Code

Tim

"Once we have the design, implementation is almost automatic!"

Jane

"Yes! We know the modules; we have interfaces as contracts. We write code mechanically."

Why Coding Seems 'Automatic' After Design:

With good design, implementation becomes straightforward because:

1. **Module boundaries are clear**: You know exactly what to build
2. **Interfaces are defined**: You know the method signatures
3. **Dependencies are explicit**: You know what you need from other modules
4. **Test cases are obvious**: Interface contract suggests test scenarios

This is why we spend time on requirements, architecture, and design!

Team members begin implementing their modules in accordance with the design specifications.

Ken

"Remember: implementation is the shortest phase when you've designed well. With AI assistance, it becomes even faster![48]"

Modern Coding with AI Assistance:

Given a clear design document, AI tools can help generate:

- Boilerplate code from interface definitions
- Test stubs from method signatures
- Documentation from code
- Refactoring suggestions

However: AI doesn't replace design thinking—it accelerates implementation of well-designed systems.

[48] AI tools can help generate boilerplate code, test stubs, and documentation from well-defined designs, accelerating the coding process. However, good design is still essential for effective use of AI in software development.

The First Bug

Tim excitedly demonstrates his task creation function.

Tim

"Look! I finished the first function! It works perfectly!"

He clicks the "Add Task" button, and a new task appears in the list.

Jane

"That's great! Can I try?"

Jane types a task title and clicks submit. Nothing happens.

Jane

"Um... It's not working for me."

Tim

"That's weird! It worked for me. Try again?"

Jane tries again. Still nothing. Tim tries it. It works fine.

Tim

"I don't understand! It works on my computer!"

Ken

"Let me see your code..."

Ken reviews Tim's code and immediately finds the problem.

JavaScript

```
function createTask(title, dueDate) {
  // Bug: assumes title is always > five characters!
  if (title.length > 5) {
    const task = {
      id: generateId(),
      title: title,
      dueDate: dueDate,
      completed: false
    }
    return task
  }
  // Returns undefined if title <= 5 characters!
}
```

Ken

"Tim, your code only works when the task title has more than five characters. Jane's test task was 'Test'—only four characters."

Tim

"But... but I tested it! I created several tasks!"

Ken

"Yes, but all your test tasks had long titles like 'Finish homework assignment'. You never tested edge cases."

Common Mistake #1: Only Testing the Happy Path:

❌ Testing only with typical, expected inputs[49] leads to missed bugs.

✅ Testing edge cases, invalid inputs, boundary conditions

Edge cases to always test:

- Empty strings
- Very long strings
- Special characters
- Null/undefined
- Zero, negative numbers
- Boundary values (min/max)

[49] The "happy path" refers to the scenario where everything works as expected, without any errors or edge cases.

Julie's Testing Lesson

Julie, working on her senior project nearby, overheard the conversation.

Julie

"Welcome to the world of manual testing. Let me guess—you tested by clicking around?"

Tim

"Yeah… I made sure the feature worked before saying it was done."

Julie

"That's good! But manual testing has problems. Let me share what happened to my team last semester…"

The Testing Disaster Story

Julie

"We built a food delivery app. Every time we added a feature, we manually tested it."

Manual Testing Timeline:

- *Week 5: Test login feature (5 minutes)*
- *Week 6: Test login + order creation (10 minutes)*
- *Week 8: Test login + order + payment (20 minutes)*
- *Week 10: Test all features (45 minutes)*
- *Week 12: Test everything (90 minutes)*
- *Week 14: Test everything (3 hours!)*
- *Week 15: Still finding bugs missed in previous tests*

Julie

"By the end, testing took longer than coding. And we STILL missed bugs!"

Jane

"Why did you miss bugs if you tested everything?"

Julie

"Because humans are inconsistent. Sometimes I'd test with valid data, sometimes I'd forget to test edge cases. Sometimes I'd be tired and miss obvious issues."

Problems with Manual Testing

Aspect	Manual Testing	Problem
Time	5-180 minutes per full test	Time grows with features
Consistency	Human judgment	Different results each time
Coverage	Test what you remember	Forget edge cases
Repetition	Boring, mind-numbing	Errors from fatigue
Frequency	Before major releases	Bugs found too late
Documentation	Mental notes	No record of what was tested

What Are Automated Tests?

Professor Cho had joined the discussion.

Prof. Cho

"That's why professional software engineers write automated tests."

Prof. Cho

"Automated tests are code that tests your code. Instead of manually clicking buttons, you write test code that runs automatically."

Tim

"Why write more code for just testing? That sounds like extra work!"

Prof. Cho

"Let me show you why it's worth it..."

Manual vs Automated Testing Comparison

Aspect	Manual Testing	Automated Testing
Execution	Human clicks through UI	Computer runs test code
Speed	5-10 minutes per test	5-10 seconds per test
Consistency	Easy to make mistakes	The same every time
Repeatability	Boring and tedious	Runs while you sleep
Edge Cases	Hard to remember all	Easy to test all scenarios
Frequency	Before releases	On every code change
Cost	Human time	One-time writing cost
Documentation	None	Tests serve as documentation

Ken

"In professional teams, automated tests run on every commit. Before your code even goes to production, hundreds of tests verify it works."

Jane

"So it's like having a robot that tests everything for you?"

Prof. Cho

"Exactly! And the robot never gets tired or forgets to test something."

Example: Manual vs Automated

Prof. Cho

"Let's see the same test, manual vs automated:"

Manual Test Script:

1. Open browser
2. Navigate to http://localhost:3000
3. Click the "Add Task" button
4. Type "Buy groceries" in the title field
5. Select tomorrow's date in the date picker
6. Click "Submit."
7. Verify the task appears in the list
8. Verify task title is "Buy groceries."
9. Verify the task date is tomorrow
10. Repeat for edge cases (empty title, past date, etc.)

Time: 5 minutes per test run

JavaScript

```
// Automated Test
test('creates task with valid input', async () => {
  // Arrange
  const taskData = {
    title: 'Buy groceries',
    dueDate: new Date('2025-12-15')
  }

  // Act
  const result = await createTask(taskData)

  // Assert
  expect(result.title).toBe('Buy groceries')
  expect(result.dueDate).toEqual(new Date('2025-12-15'))
  expect(result.completed).toBe(false)
  expect(result.id).toBeDefined()
})

test('throws error for empty title', async () => {
  const taskData = { title: '', dueDate: new Date() }

  await expect(createTask(taskData)).rejects.toThrow('Title required')
})

test('throws error for past date', async () => {
  const pastDate = new Date('2020-01-01')
  const taskData = { title: 'Test', dueDate: pastDate }

  await expect(createTask(taskData)).rejects.toThrow('Due date must be in
future')
})

// Time: 0.05 seconds per test run (100x faster!)
```

The 3 A's of Test Structure: Arrange-Act-Assert:

Every test follows this pattern:

1. **Arrange**: Set up test data and conditions

   ```
   const taskData = { title: 'Test', dueDate: new
   Date() }
   ```

2. **Act**: Execute the function being tested

   ```
   const result = await createTask(taskData)
   ```

3. **Assert**: Verify the result

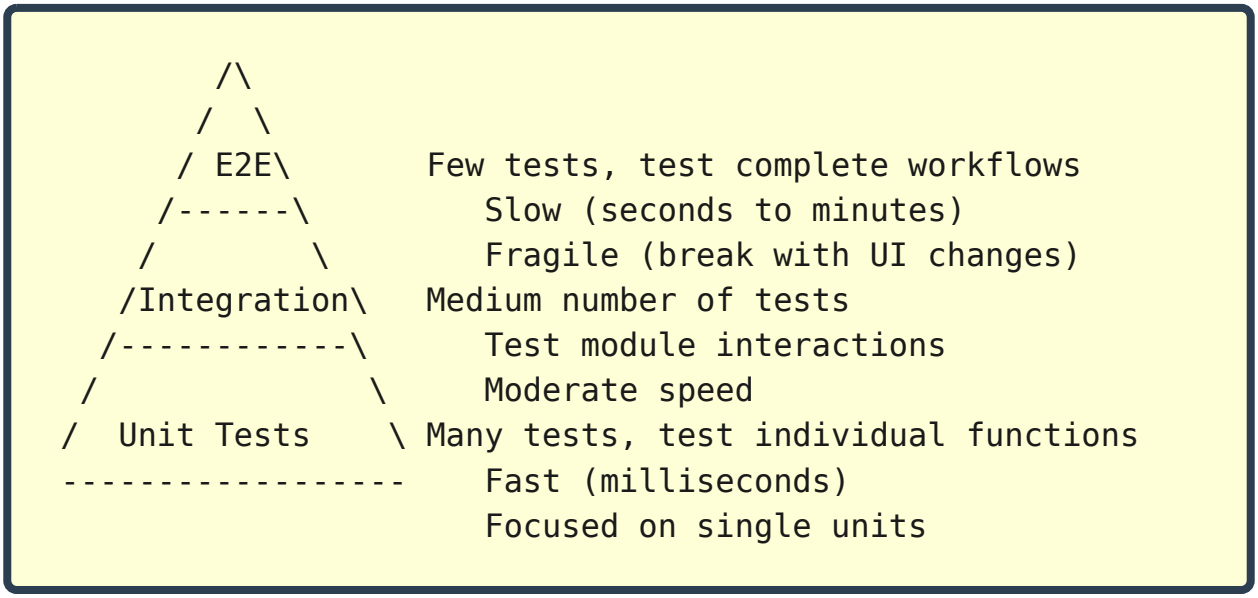
   ```
   expect(result.title).toBe('Test')
   ```

This pattern makes tests clear and consistent.

Types of Tests: The Testing Pyramid

Professor Cho explained the testing pyramid.

> **Prof. Cho**
>
> "There are three main types of tests, forming a pyramid:"



Test Type	Scope	Speed	Quantity	Purpose
Unit	Single function	Milliseconds	Many (70%)[50]	Verify module logic

[50] This is a guideline, the exact percentages vary by Project.

Integration	Multiple modules	Seconds	Medium (20%)	Verify interactions
E2E	Full system	Seconds-Minutes	Few (10%)	Verify user workflows

Tim

"I remember modules are also called units, so unit tests test modules, right?"

Prof. Cho

"Exactly! Unit tests verify that individual modules work as designed through their interfaces."

Jane

"In software architecture, we test elements. In design, we test modules. Same thing?"

Prof. Cho

"Yes. Architecture elements become design modules become code units[51] Unit tests verify the smallest testable pieces."

[51]This is a general idea, as one architecture element can become multiple design modules.

Jane

“Also, unittests are considered as a parf of coding, not testing?”

Prof. Cho

“Good point! Writing unit tests is part of coding because they are written alongside the code to verify its correctness from the start. Not as part of testing to verify after coding.”

Tim

“Then, integration tests check that multiple modules work together through their interfaces?”

Prof. Cho

“Perfect! Integration tests verify that when modules talk to each other, they communicate correctly.”

Jane

“And E2E is for acceptance tests—checking if requirements are satisfied?”

Prof. Cho

“Exactly. **End-to-End (E2E) tests** are like automated acceptance tests.

- They start from the **UI or API**
- Go through all layers (services, database, external APIs)
- Check if the **requirement is satisfied** from the user’s perspective.“

Tim

“I’m confused, We have discussed acceptance tests before, but now we have E2E tests. Are they different? “

Prof. Cho

“Great question! They are similar but not identical.

- **Acceptance tests** are defined based on requirements and can be manual or automated.
- **E2E tests** are a type of automated acceptance test that simulates real user workflows.

So, all E2E tests are acceptance tests, but not all acceptance tests are E2E tests.“

Tim

“Got it! So, we do our best to automate acceptance tests as E2E tests, but in some cases, it may not be possible. Is that correct?”

Prof. Cho

“Exactly! We aim to automate acceptance tests as E2E tests whenever feasible, but some scenarios may require manual testing due to complexity or resource constraints.”

Jane

“So:

- Unit tests: ‘Does this **single module** behave correctly?’
- Integration tests: ‘Do these **modules work together**?’
- E2E tests: ‘Are **user workflows** functioning end-to-end?’“
- Acceptance tests: ‘Does the **whole system** satisfy requirements?’“

Prof. Cho

"Perfect summary! Remember the **pyramid shape**:

- Many **unit tests** → fast, cheap, pinpoint bugs
- Fewer **integration tests** → check boundaries
- Few **E2E tests** → slow but closest to real usage."

Tim

"Even though E2E is like acceptance tests, we **still need** unit and integration tests because they find bugs earlier and tell us **exactly where** the problem is."

Julie

"Yes! If an E2E test fails: > 'Something is broken somewhere in the entire workflow.'

Unit & integration tests help **localize** the bug quickly. That's why we need **all three levels**."

Prof. Cho

"We also run all tests **automatically** when we make changes. We call this **regression testing**."

Jane

"Got it:

- Architecture: design **elements**, **interactions**, and **constraints**
- Design: **modules**, **interfaces**, and **dependencies**
- Testing: verify them **Unit → Integration → E2E**
- Regression: re-run all tests on every change

The pyramid makes sense now!"

Common Mistake #2: Inverted Test Pyramid:

❌ Many E2E tests, few unit tests (upside-down pyramid)

✅ Many unit tests, few E2E tests (proper pyramid)

Why the inverted pyramid fails:

- E2E tests are slow → long feedback cycles
- E2E tests are fragile → break with UI changes
- E2E tests don't pinpoint bugs → hard to debug
- Too expensive to maintain

Proper pyramid benefits:

- Fast feedback (unit tests run in seconds)
- Precise bug location (know exactly what broke)
- Cheap to maintain (unit tests rarely break)

Unit Tests: The Foundation

Prof. Cho

"Unit tests are the foundation. They test individual functions in isolation."

Characteristics of Good Unit Tests

Characteristic	Description
Fast	Thousands run in seconds
Isolated	No database, no network, no file system
Focused	One function, one behavior at a time
Independent	Can run in any order
Repeatable	Same result every time
Self-validating	Pass/fail, no manual checking

Prof. Cho

"This is an example of unit tests for an EmailValidator class, I mean module."

JavaScript

```
// Function to test
class EmailValidator {
  isValid(email) {
    if (!email || typeof email !== 'string') return false
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/
    return emailRegex.test(email)
  }
}

// Unit tests
describe('EmailValidator', () => {
  test('returns true for valid email', () => {
    const validator = new EmailValidator()
    expect(validator.isValid('tim@nku.edu')).toBe(true)
  })

  test('returns false for invalid email', () => {
    const validator = new EmailValidator()
    expect(validator.isValid('invalid')).toBe(false)
    expect(validator.isValid('')).toBe(false)
    expect(validator.isValid(null)).toBe(false)
  })
})
```

Test Organization with describe/test:

- **describe**: Groups related tests
- **test** (or **it**): Individual test case
- **beforeEach**: Runs before each test (setup)
- **afterEach**: Runs after each test (cleanup)

Benefits:

- Clear test organization
- Shared setup/teardown
- Easy to read test output
- Can run specific groups

Test Coverage

Prof. Cho

"Test coverage measures how much of your code is tested:"

Bash

```
Test Coverage Report for EmailValidator:

Function Coverage:   100% (2/2 functions)
Line Coverage:       100% (15/15 lines)
Branch Coverage:     100% (all if/else paths)

✅ All code paths tested
```

Tim

"Wow, that's a nice coverage report! How do we generate this?"

Prof. Cho

"Good question! You use a code coverage tool. For JavaScript/TypeScript projects, Jest has built-in coverage reporting."

Ken

“How do we run it?”

Prof. Cho

“Very simple - add the `--coverage` flag when running your tests:[52]”

Bash

```bash
# Run tests with coverage report
npm test -- --coverage

# Add it to package.json
"scripts": {
  "test": "jest",
  "test:coverage": "jest --coverage"
}
```

Prof. Cho

“In this example, npm test simply runs whatever command is defined under the test script in package.json. The standalone `--` is important: it tells npm, “Stop parsing options here. Everything after this goes directly to the underlying tool.“

[52]This is an example when we use jest in JavaScript/TypeScript.Most testing frameworks have built-in or plugin-based coverage tools. Check your framework’s documentation for details.

Jane

"So `--coverage` is not an npm option—it is passed straight through to Jest as a CLI flag."

Julie

"Yes. When Jest sees `--coverage`, it automatically instruments the source code, tracks which lines, branches, and functions are executed during the tests, and then generates a coverage report."

Prof. Cho

"That's why you don't need any extra configuration for basic usage—Jest already knows how to collect and output coverage by default."

Prof. Cho

"In short: npm just forwards the command, and Jest does all the real work."

Jane

"And it automatically shows which lines we haven't tested?"

Prof. Cho

"Exactly! It generates an HTML report you can open in your browser. Let me show you what you'll see:"

Coverage Type	What It Measures	Target
Line Coverage	% of lines executed by tests	>80%
Branch Coverage	% of if/else paths tested	>70%
Function Coverage	% of functions called	>90%
Statement Coverage	% of statements executed	>80%

Prof. Cho

"The HTML report highlights untested lines in red, so you know exactly what to test next."

Tim

"That's helpful! So we can see our weak spots immediately?"

Prof. Cho

"Yes! And most CI/CD pipelines can automatically fail builds if coverage drops below your target thresholds."

Warning: 100% Coverage ≠ Bug-Free:

You can have 100% line coverage but still have bugs:

```
function divide(a, b) {
  return a / b  // Bug: doesn't check for b === 0
}

test('divides two numbers', () => {
  expect(divide(10, 2)).toBe(5)  // 100% coverage, but misses
edge case!
})
```

Better test:

```
test('throws error when dividing by zero', () => {
  expect(() => divide(10, 0)).toThrow('Cannot divide by
zero')
})
```

Coverage is a tool, not a goal. Focus on testing **behaviors** and **edge cases**.

Jane

"Do we always have to aim for high coverage?"

Prof. Cho

“Great question! While high coverage is generally good, the goal is to test important behaviors and edge cases, not just to hit a number. Some code (like simple getters/setters) may not need extensive tests.”

Julie

“In other words, use coverage as a guide, but think critically about what to test. Our team normally sets targets like 80% line coverage, but focuses on critical logic.”

Prof. Cho

“Good point! Coverage targets should be balanced with practical considerations. As long as key functionalities are well-tested, slightly lower coverage in trivial code is acceptable.”

Integration Tests: The Middle Layer

Prof. Cho

"Integration tests verify that modules work together correctly. Let's talk about their characteristics."

Characteristic	Description
Scope	Multiple modules working together
Speed	Slower than unit tests (involves database, network)
Purpose	Verify module interactions and interfaces
Dependencies	May use real database, test database, or mocks[53]
Complexity	More setup required than unit tests

Prof. Cho

"Here's a simple example - testing Task Service with Database working together:"

[53]Mocks are fake versions of real components used for testing. A mock database pretends to save/retrieve data (usually in memory) without actually connecting to a real database, making tests faster but less realistic.

JavaScript

```
  // Integration test - TWO modules working together
  describe('TaskService with Database Integration', () => {
    let database
    let taskService

    beforeAll(async () => {
      // Setup test database
      database = await createTestDatabase()
      taskService = new TaskService(database)
    })

    afterAll(async () => {
      await database.close()
    })

    test('creates task and saves to database', async () => {
      // Arrange
      const taskData = {
        title: 'Buy milk',
        date: new Date('2024-12-31')
      }

      // Act - TaskService talks to Database
      const savedTask = await
taskService.createTask('user-123', taskData)

      // Assert - Check response
      expect(savedTask.id).toBeDefined()
      expect(savedTask.title).toBe('Buy milk')

      // Assert - Verify actually saved in database
      const fromDB = await
database.tasks.findById(savedTask.id)
      expect(fromDB.title).toBe('Buy milk')
    })
```

```
    test('retrieves tasks from database', async () => {
      // Arrange - Create some tasks first
      await taskService.createTask('user-123', {
        title: 'Task 1',
        date: new Date('2024-12-31')
      })
      await taskService.createTask('user-123', {
        title: 'Task 2',
        date: new Date('2024-12-31')
      })

      // Act - TaskService queries Database
      const tasks = await taskService.getTasks('user-123')

      // Assert
      expect(tasks.length).toBe(2)
      expect(tasks[0].title).toBe('Task 1')
      expect(tasks[1].title).toBe('Task 2')
    })
  })
```

Tim

"I see! Integration tests check if TaskService and Database actually work together?"

Julie

"Yes! This code shows that we need two modules for the integration test: TaskService and Database."

JavaScript

```
  beforeAll(async () => {
    // Setup test database
    database = await createTestDatabase()
    taskService = new TaskService(database)
  })
```

Julie

"Then, we check if two modules are working togehter without any issues."

JavaScript

```
test('creates task and saves to database', async () => {...})
test('retrieves tasks from database', async () => {...})
```

Prof. Cho

"Exactly! Integration tests ensure that when modules interact, they do so correctly according to their interfaces. This helps catch issues that unit tests alone might miss."

Jane

"I also can see that the database should be real, not mocked, to verify actual data storage. Is that right?"

Prof. Cho

“Good observation! Integration tests often use a real or test database to verify actual data storage and retrieval. Mocks can be used in unit tests, but integration tests should test real interactions whenever possible.”

End-to-End Tests: The Peak

Prof. Cho

"End-to-End (E2E) tests simulate real user workflows from start to finish. This is the peak of the testing pyramid."

Characteristic	Description
Scope	Entire system from UI to database
Speed	Slowest (seconds to minutes per test)
Purpose	Verify complete user workflows
Tools	Playwright, Cypress, Selenium
Environment	Real browser, real server, real database
Fragility	Break easily with UI changes

Prof. Cho

"End-to-End tests simulate real user interactions from start to finish[54]. Let's see a simple example using Playwright:[55]"

[54]This is a simplified example, in the real world, an E2E test would handle more scenarios and edge cases.

JavaScript

```
  // E2E test - simulates a real user clicking through the
app
  import { test, expect } from '@playwright/test'

  test('user can create and view a task', async ({ page }) =>
{
    // 1. Open the app
    await page.goto('http://localhost:3000')

    // 2. Login
    await page.fill('input[name="email"]', 'tim@nku.edu')
    await page.fill('input[name="password"]', 'password123')
    await page.click('button[type="submit"]')

    // 3. Create a new task
    await page.click('button:text("Add Task")')
    await page.fill('input[name="title"]', 'Buy milk')
    await page.fill('input[type="date"]', '2024-12-31')
    await page.click('button:text("Create")')

    // 4. Verify task appears in list
    const task = page.locator('.task-item', { hasText: 'Buy
milk' })
    await expect(task).toBeVisible()

    // 5. Complete the task
    await task.locator('input[type="checkbox"]').check()

    // 6. Verify the task is marked complete
    await expect(task).toHaveClass(/completed/)
  })
```

[55]Playwright is a modern E2E testing framework that automates real browsers for testing web applications. There are other popular tools like Cypress and Selenium as well.

Jane

"This test clicks through the entire app like a real user would?"

Prof. Cho

"Exactly! It opens a real browser, clicks buttons, fills forms, and checks what appears on screen. This tests everything working together - UI, API, and database."

Tim

"So E2E tests are the slowest but most realistic?"

Prof. Cho

"Yes! They catch problems that unit and integration tests might miss, but they take longer to run."

Ken

"E2E tests are like having a robot user clicking through your app—exactly like manual testing but automated!"

Prof. Cho

"Exactly! E2E tests are:"

- Slowest (simulate real browser)
- Most comprehensive (test entire workflows)
- Most fragile (breaks when UI changes)
- Fewest in number (only critical user journeys)

Common Mistake #3: Too Many E2E Tests:

❌ Write E2E tests for every feature and edge case

✅ Write E2E tests only for critical user workflows

Why?

- E2E tests are slow → long feedback cycles
- E2E tests are fragile → high maintenance cost
- E2E tests are hard to debug → when they fail, it's unclear why

Best practice:

- E2E: Critical happy paths only (login → create task → complete)
- Integration: Feature-level workflows
- Unit: All edge cases and error conditions

E2E Testing Tools: Playwright, Cypress, Selenium:

- **Playwright**: Modern, fast, supports multiple browsers
- **Cypress**: Developer-friendly, great DX, browser-based
- **Selenium**: Industry standard, mature, many language bindings

Choosing a tool:

- New projects: Playwright (fastest, most features)
- JavaScript-heavy: Cypress (best developer experience)
- Multi-language teams: Selenium (Java, Python, C# support)

> **Prof. Cho**
>
> "There are many E2E testing tools available. The best choice depends on your project's needs and team's expertise."

> **Prof. Cho**
>
> "Don't forget to make sure your E2E tests run in a clean environment, ideally using a test database and isolated server instance to avoid affecting real users."

Prof. Cho

“Finally, be sure to implement all the acceptance tests defined in your requirements as E2E tests to ensure the system meets user needs. If necessary, make some acceptance tests manual if they are too complex to automate.”

The Testing Strategy

The team gathered to discuss their testing strategy.

Tim

"Okay, I understand the types now. But how many tests should we write?"

Ken

"Good question! Let's think about it strategically..."

The 70-20-10 Rule

Ken

"In professional teams, a common ratio is:"

Test Distribution (70-20-10 Rule):

```
70% Unit Tests
- Test every function in every module
- Test all edge cases (empty, null, boundary values)
- Test error conditions
```

```
- Fast feedback (run in seconds)

20% Integration Tests
- Test module interactions
- Test database operations
- Test API endpoints
- Test service layer

10% E2E Tests
- Test critical user flows only
- Login → create task → complete
- Happy path of key features
- User acceptance scenarios
```

Jane

"Why so many unit tests compared to E2E tests?"

Ken

"Because unit tests give you the fastest, most precise feedback. If a unit test fails, you know exactly which function broke. If an E2E test fails, it could be anywhere in the system."

Julie

"Plus, E2E tests are slow. 1000 E2E tests might take hours. But 1000 unit tests? Maybe 30 seconds."

Test Type	1 Test	100 Tests	1000 Tests
Unit	5ms	0.5s	5s
Integration	100ms	10s	100s
E2E	5s	8min	1.5hrs

Ken

"Even though testing is one of the team leader's responsibilities, unit tests are part of implementation. So when you implement modules and functions, you should write corresponding unit tests."

Julie

"Yes, that's important. Team members should help the team leader create high-quality integration, E2E, and regression tests."

What to Test at Each Level

Prof. Cho

"Let me give you specific guidance on what to test at each level:"

Unit Tests - Test ALL:

- Happy path (valid inputs)
- Edge cases (empty, null, undefined, boundary values)
- Error conditions (invalid inputs, exceptions)
- Private functions (through public interface)
- All branches (if/else paths)

Integration Tests - Test:

- Module interactions (Service → Repository)
- Database operations (CRUD)
- API endpoints (request/response)
- Authentication/authorization flows
- External service calls (with mocks)

E2E Tests - Test ONLY:

- Critical user workflows (3-5 main flows)
- Happy paths of key features
- Core business requirements
- User authentication flow

Acceptance Tests - Test ALL:

- All defined acceptance criteria from requirements

Regression Testing

Prof. Cho

"One more type: **Regression testing** means running ALL tests on every code change."

Tim

"Why run ALL tests? That seems excessive."

Prof. Cho

"Because fixing one bug can create new bugs elsewhere. Regression tests ensure new changes don't break existing functionality."

Julie

"This is an example of CI/CD pipeline that runs all tests on every push for my project:"

YAML

```yaml
# Example: GitHub Actions CI/CD Pipeline
name: Run Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Install dependencies
        run: npm install
```

```
      - name: Run unit tests
        run: npm run test:unit

      - name: Run integration tests
        run: npm run test:integration

      - name: Run E2E tests
        run: npm run test:e2e

      - name: Generate coverage report
        run: npm run coverage

      - name: Fail if coverage < 80%
        run: npm run coverage:check
```

Jane

"So every time we push code, ALL tests run automatically?"

Ken

"Exactly! This is called Continuous Integration (CI). If any test fails, the push is rejected."

Benefit	Description
Catch bugs early	Before code reaches production
Confidence to refactor	Tests ensure changes don't break things
Documentation	Tests show how code should behave

Quality gates	Can't merge code with failing tests
Fast feedback	Know within minutes if something broke

Tim

"I'm worried because I have to many things to know. I'm not familiar with CI/CD pipelines and automated testing. I'm lost now."

Julie

"Don't worry! For now, just remember that regression testing means running all tests automatically on every change to catch bugs early. As you gain experience, you'll become more comfortable with the tools and processes involved."

Ken

"Yes. I agree. The most important thing is to have confidence that our code works correctly after every change. Automated regression testing gives us that confidence. I'll start with a simple Python script[56] to run all tests automatically. Once we have a fully working CI/CD pipeline, we can integrate it there. So, for now, focus on writing good tests at all three levels."

Tim

"That makes sense. Thanks, Ken! I feel better now."

Benefits of Automated Regression Testing:

1. **Catch bugs early**: Before code reaches production
2. **Confidence to refactor**: Tests ensure changes don't break things
3. **Documentation**: Tests show how code should behave
4. **Quality gates**: Can't merge code with failing tests
5. **Fast feedback**: Know within minutes if something broke

[56] Or JavaScript code or bash code; anything is OK as long as it runs all tests automatically.

Summary: Testing Pyramid Recap

Level	Scope	Speed	Quantity	Purpose
Unit	Single function	Milliseconds	Many (70%)	Verify logic
Integration	Multiple modules	Seconds	Some (20%)	Verify interactions
E2E	Full workflow	Minutes	Few (10%)	Verify workflow
Acceptance	Feature implementation	Minutes	Few (5%)	Verify requirements
Regression	All tests	Minutes	Every change	Prevent breaking changes

The team was now equipped with the knowledge and tools to build quality software with confidence.

Chapter 8:
Software Process

The team has learned all the technical skills—requirements, architecture, design, and testing. Now they need to understand the process that ties everything together.

- Software process provides structure for teamwork
- Agile methodology embraces change and delivers value early
- Scrum framework organizes work into sprints

Quick Reference Guide

Agile Core Values

1. Individuals & interactions > processes & tools
2. Working software > comprehensive docs
3. Customer collaboration > contract negotiation
4. Responding to change > following plan

Scrum Roles

- **Product Owner**: What to build
- **Scrum Master**: Facilitates process
- **Dev Team**: How to build

Sprint Structure (1-4 weeks)

- Planning: What will we build?
- Daily Standup: Progress & blockers
- Review: Demo to stakeholders
- Retrospective: How to improve?

Key Metrics

- **Velocity**: Story points per sprint
- **Burndown**: Work remaining over time
- **Sprint Goal**: What we commit to deliver

The Overwhelming Feeling

In the meeting room, the team sat together, looking at their backlog of features and feeling overwhelmed.

Tim

"We have eight features in total (2 members x 2 features x 2 sprints) to deliver. Out of the eight features, they have 24 requirements (3 per feature) to build. We also have various tests and documentation for high-quality software. How do we do all of this?"

Jane

"And what if we realize we need to change something after we've already built it?"

Ken

"Plus, we have deadlines! How do we make sure we deliver the promised features on time?"

Prof. Cho

"These are exactly the problems that software process solves. Let me explain..."

Common Mistake #1: No Process = Chaos:

❌ "Let's all just work on whatever we want!"

✅ "Let's use a structured process to coordinate our work."

Why? Without process:

- Duplicate work
- Unclear priorities
- Missed deadlines
- Nothing is ever "done."
- Team members don't know what others are doing

What is Process?

> **Prof. Cho**
>
> "Let's start with **process** in general. A process is simply a structured series of steps taken to achieve a goal. Here are some everyday activities that follow a process."

Activity	Without Process	With Process
Cooking	Throw ingredients together, hope for the best	Recipe: ingredients, steps, timing
Manufacturing	Each worker does random tasks	Assembly line: defined stages, handoffs
Travel	Wander	Itinerary: destinations, timeline, bookings
Surgery	Wing it	Protocol: pre-op, procedure steps, post-op

> **Prof. Cho**
>
> "Software development also follows a process—it organizes requirements, design, implementation, and testing into a coherent workflow."

Tim

“But we already know those activities. Why do we need a process?”

Prof. Cho

“Good question. The activities themselves don’t tell us:

- **When** each activity happens
- **Who** performs them
- **How** they connect and depend on each other
- **How** to handle changes
- **How** to track progress.“

Prof. Cho

“Without a process, these activities are just a checklist.

With a process, they become a **repeatable, predictable way** to deliver quality software on time and within budget.“

The No-Process Disaster

Julie

"In my first project, we had no process. Here's what happened:"

The Horror Story: No Process

- *Week 1: "Let's all work on whatever we want!" → No coordination, unclear priorities*
- *Week 3: "Wait, we're building duplicate features." → Tim and Jane both built login screens*
- *Week 5: "Who's working on what?" → Nobody knows the project status*
- *Week 8: "Nothing is finished!" → Started 10 features, completed 0*
- *Week 10: "Why doesn't anything work together?" → No integration, conflicting assumptions*
- *Week 12: "We're panicking and working random hours." → Burnout, low morale*
- *Week 14: "Everything is broken, and we're out of time." → Failed project*

Julie

"We failed because we had no process to coordinate our work."

Prof. Cho

"This is unfortunately common. Now let me show you what process could have prevented..."

What Went Wrong: Analysis

Problem	Root Cause	Process Solution
Duplicate work	No task assignment	Sprint planning assigns tasks
Unclear status	No communication	Daily[57] standup meetings
Nothing done	No definition of "done"	Acceptance criteria, sprint reviews
Integration fails	No checkpoints	Sprint demos force integration
Burnout	No sustainable pace	Time-boxed sprints, retrospectives

[57] Weekly for team project

The Waterfall Process

> **Prof. Cho**
>
> "Historically, there were two main approaches to software process..."

The Waterfall Approach

Waterfall Process (Traditional):

```
Months 1-2: Requirements (100% complete)
↓ (no going back)
Months 3-4: Design (100% complete)
↓ (no going back)
Months 5-8: Implementation (100% complete)
↓ (no going back)
Month 9: Testing (find all the bugs!)
↓
Month 10: Deployment (hope it works!)
```

Key Idea: Each phase must be 100% complete before the next phase. Like a waterfall—water only flows downward, never back up.

Characteristic	Description
Linear	Sequential phases, no overlap
Complete	Each phase 100% done before next

Document-Heavy	Extensive documentation at each stage
Rigid	Changes are expensive after phase complete
Late Feedback	No working software until the end

Problems with Waterfall

Ken

"Waterfall sounds logical and orderly, but I can see problems from the discussion we have had to make high-quality software."

Waterfall Problems:

1. *Late Feedback*
   - *See working software only at the end (month 10!)*
   - *If the client doesn't like it, it's too late to change*
   - *Example: "This isn't what we wanted" after 10 months of work*
2. *Changing Requirements*
   - *Requirements WILL change (markets shift, users learn)*
   - *Going back to month 1 is expensive*
   - *Example: Client says "We need mobile support" in month 8*
3. *Big Bang Integration*
   - *All modules are integrated at once at the end*
   - *High risk of integration failures*
   - *Example: Database schema doesn't match API expectations*
4. *Risk Management*

- *All risk at the end (what if it doesn't work?)*
- *No early validation*
- *Example: Performance issues discovered in month 9*

Prof. Cho

"Yes. Waterfall works well for projects with stable requirements and no uncertainty— like building a bridge. But software is different:

- Requirements change frequently
- Technology evolves rapidly
- User needs are discovered iteratively

That's why software engineers created the Agile approach."

Waterfall Process - Success & Failiure:

When Waterfall Works:

Waterfall is appropriate for:

- Projects with completely stable requirements
- Regulated industries (FDA, aviation) requiring extensive documentation
- Hardware/software integration where changes are costly
- Projects with no uncertainty

Why Waterfall Fails for Most Software:

- Requirements are rarely 100% known upfront
- Users don't know what they want until they see it
- Technology and markets change quickly
- Early feedback is valuable

The Agile Revolution

Prof. Cho

"To solve Waterfall's problems, software engineers created the Agile Manifesto in 2001."

The Agile Manifesto:

We value:

1. ***Individuals and interactions*** *OVER processes and tools*
2. ***Working software*** *Over-comprehensive documentation*
3. ***Customer collaboration*** *OVER contract negotiation*
4. ***Responding to change*** *OVER following a plan*

That is, while the items on the right have value, we value the items on the left MORE.

Tim

"So the real values are: **individuals**, **working software**, **collaboration**, and **change**."

Jane

"Not **processes**, **documentation**, **contracts**, and **plans**."

Prof. Cho

"Exactly! This doesn't mean we ignore processes or documentation— we just prioritize people and working software."

Tim

"I see. It's about being **agile** and adaptable, not rigidly following plans to deliver high-quality software by managing complexity in a team."

Agile vs Waterfall: Side-by-Side

Aspect	Waterfall	Agile
Delivery	One big delivery at the end (10 months)	Small deliveries every 4 weeks or so[58]
Requirements	Fixed upfront, hard to change	Evolve based on feedback

[58]Typically 1-4 weeks; for team project, two sprints, and each sprint is about 4 weeks

Changes	Expensive, require re-planning	Expected, embraced
Testing	At the end (big bang)	Continuously throughout
Risk	High (all at end)	Low (fail fast, adjust quickly)
Feedback	Late (after months)	Early and frequent
Documentation	Comprehensive upfront	Just enough, when needed

Tim

"So instead of one 10-month release, Agile delivers every 4 weeks or so?"

Prof. Cho

"Exactly! This means:

- Get feedback early (monthly feedback, not month 10)
- Adjust course quickly (reprioritize next 4 weeks)
- Deliver value incrementally (users get features sooner)
- Reduce risk (if something fails, only 4 weeks wasted)."

Common Mistake #2: 'Thinking Agile = No Planning':

❌ “Agile means we don’t plan, we just code!”

✅ “Agile means we plan iteratively and adapt.”

Truth:

- Agile still requires planning (sprint planning, backlog grooming)
- Difference: Plans are short-term (2 weeks) and flexible
- Long-term plans are directional, not fixed

Bad Agile: No planning, chaos **Good Agile:** Continuous planning, adaptation

Agile Principles: The Four Pillars

Prof. Cho

"The Manifesto has 12 principles, but I'll group them into four pillars for clarity:"

Prof. Cho

"You can memorize it as ICCE:"

1. **Iterative** Delivery
2. **Collaboration** & People
3. Embrace **Change**
4. Sustainable **Excellence**

Pillar 1: Iterative Delivery

Iterative Delivery Principles:

- *Deliver working software frequently (weeks, not months)*
- *Working software is the primary measure of progress*
- *Welcome changing requirements, even late in development*
- *Simplicity—maximize work NOT done—is essential*

Prof. Cho

"The core idea: Build in small increments, learn, adapt, repeat."

Tim

"Like building an MVP first to validate assumptions?"

Prof. Cho

"Exactly! Start with a minimum viable product, then enhance based on feedback."

Iterative Delivery Example:

```
Waterfall:
Month 1-10: Build a complete system
Month 10: Release everything
Feedback: Too late to change

Agile:
Week 1-2: Build MVP (login + basic task creation)
Week 3-4: Add reminders
Week 5-6: Add priority filtering
Every 2 weeks: Get feedback, adjust next iteration
```

Pillar 2: Collaboration & People

Collaboration Principles:

- Business people and developers work together daily
- Face-to-face conversation is the most effective communication
- Build projects around motivated individuals
- Self-organizing teams produce the best results
- Regular reflection and adjustment

Prof. Cho

"Software engineering is teamwork. Close collaboration reduces misunderstandings."

Jane

"Working with stakeholders daily ensures we solve **their** real problems."

Ken

"And self-organizing teams means WE decide how to do the work, not micromanagement."

Aspect	Traditional	Agile
Communication	Documentation, email	Face-to-face, daily standups
Stakeholder Involvement	Requirements phase only	Daily/weekly collaboration
Team Structure	Top-down management	Self-organizing
Decision Making	Manager decides	Team decides how

Pillar 3: Embrace Change

Embracing Change Principles:

- *Welcome changing requirements, even late*
- *Harness change for the customer's competitive advantage*
- *Agile processes promote sustainable development*
- *Continuous attention to technical excellence*

Prof. Cho

"Requirements WILL change—markets shift, users learn, priorities evolve. Agile treats change as normal, not as failure."

Tim

"So instead of resisting change, we build flexibility into our process?"

Prof. Cho

"Exactly. Short iterations make change cheap."

Example: Embracing Change:

Scenario: After Sprint 3, the client realizes the mobile app is more important than the desktop.

Waterfall Response:

- "Change order required."
- "Months of re-planning."
- "Expensive delay"

Finger-pointing and blame

Agile Response:

- "Let's reprioritize the backlog."
- "Next sprint: focus on mobile."
- "Desktop features move to later sprints."

Team adapts quickly

Result: Agile delivers what the client needs, when they need it.

Pillar 4: Sustainable Excellence

Sustainable Excellence Principles:

- Maintain a constant pace indefinitely
- Continuous attention to technical excellence
- Good design enhances agility
- Simplicity—maximize work NOT done

Prof. Cho

"Quality isn't just about the product—it's about how we work. If the team burns out or code becomes unmaintainable, we can't sustain delivery."

Jane

"So sustainable pace means no death marches or crunch time?"

Prof. Cho

"Right. Agile teams work at a consistent, maintainable pace— not sprint finish, not marathon exhaustion."

Tim

"And technical excellence means good architecture, testing, refactoring?"

Prof. Cho

"Exactly. Cut corners on quality, and you slow down later. Invest in quality, you speed up over time."

Practice	Short-Term Thinking	Sustainable Thinking
Code Quality	"Ship fast, fix later"	"Build right, move fast forever"
Testing	"No time for tests"	"Tests save time debugging"
Working Hours	"Crunch to meet deadline"	"Consistent 40 hours/ week"
Technical Debt	"Accumulate debt"	"Pay down debt continuously"

Ken

"I can see that these four pillars are actually to build high-quality software."

Tim

"Yes, we need to iteratively enhance quality while collaborating closely with clients. When change comes, we should embrace it, while maintaining excellence."

Prof. Cho

"Yes! Agile is not just about speed, it's about **sustainable high-quality delivery** through an iterative process and working with clients, I mean, real users."

Scrum Framework

Prof. Cho

"Scrum is the most popular Agile framework. Let me explain how it works..."

Jane

"Wait. We're talking about Agile, but what's a framework?"

Prof. Cho

"Good catch! Let me clarify the terminology."

Agile vs Scrum: Understanding the Relationship

Term	Definition
Agile	Philosophy, set of values and principles
Framework	Concrete implementation of philosophy
Scrum	One specific Agile framework
Kanban	Another Agile framework

XP	Yet another Agile framework (Extreme Programming)

Prof. Cho

"**Agile** is a philosophy—values and principles.

A **framework** implements those principles with specific roles, events, and practices.

Scrum is one framework that implements Agile principles."

Tim

"So Agile says 'collaborate closely,' and Scrum says 'have a daily standup meeting'?"

Jane

"Like OOP is a paradigm, and Java is one OOP language?"

Prof. Cho

"Perfect analogies! Agile = philosophy, Scrum = implementation."

Agile Frameworks Comparison:

Scrum:

- Fixed-length sprints (1-4 weeks)
- Defined roles (Product Owner, Scrum Master, Dev Team)
- Ceremonies (Planning, Standup, Review, Retrospective)
- Good for: Teams building products iteratively

Kanban:

- Continuous flow (no sprints)
- Visualize work on the board
- Limit work in progress
- Good for: Support teams, continuous delivery

XP (Extreme Programming):

- Pair programming
- Test-driven development
- Continuous integration
- Good for: High-quality code, technical teams

For this course, we focus on Scrum, the most widely used.

Scrum Overview: Sprints, Roles, Ceremonies

Prof. Cho

"Scrum has three key components: Sprints, Roles, and Ceremonies."

Component 1: Sprints (Iterations)

Sprints: Fixed Timeboxes for Work

- Duration: 1-4 weeks (most common: 2 weeks)
- Goal: Deliver **working software** each sprint
- The team selects work at the start of the sprint
- End with a potentially shippable product increment

Prof. Cho

"Why sprints? They create a **rhythm**—a predictable cycle where we plan, build, deliver, and reflect."

Tim

"So instead of planning for months, we only commit to weeks of work?[59]"

Prof. Cho

"Exactly. This reduces risk and allows quick adjustments."

Aspect	Long Planning (Waterfall)	Short Sprints (Scrum 2 week example)
Commitment	10 months of work	2 weeks of work
Risk	High (10 months wasted if wrong)	Low (2 weeks wasted if wrong)
Flexibility	Hard to change	Easy to reprioritize
Feedback	After 10 months	Every 2 weeks

[59]Typically 1-4 weeks; for team project, two sprints, and each sprint is about 4 weeks

Component 2: Roles

Scrum Roles:

1. **Product Owner**
   - Defines WHAT to build
   - Prioritizes features
   - Accepts or rejects work
   - Represents stakeholders/users
2. **Scrum Master**
   - Facilitates Scrum process
   - Removes blockers
   - Protects the team from distractions
   - Ensures the process followed
3. **Development Team**
   - Decides HOW to build
   - Self-organizing (no micromanagement)
   - Cross-functional (has all needed skills)
   - Commits to sprint goals
   - Delivers working software

Prof. Cho

"Notice the separation: Product Owner decides **what** to build, Dev Team decides **how** to build it."

Ken

"For the team project, team leaders are both Product Owner AND Scrum Master."

Prof. Cho

"Yes. In small teams, these roles often combine. In large companies, they're separate people."

Role	Team Project	Real Company
Product Owner	Team Leader	Product Manager
Scrum Master	Team Leader	Dedicated Scrum Master
Dev Team	Team Members	5-9 Developers

Component 3: Ceremonies (Events)

Prof. Cho

"Scrum defines four key ceremonies to structure the sprint:"

Scrum Ceremonies:

1. *Sprint Planning* (Start of sprint)

- What will we build this sprint?
- Team selects work from the product backlog
- Creates sprint backlog
- Defines the sprint goal

2. **Daily Standup** (Every day, 15 min)
   - What did I do yesterday?
   - What will I do today?
   - Any blockers?
3. **Sprint Review** (End of sprint)
   - Demo working software to stakeholders
   - Get feedback
   - Adapt product backlog
4. **Sprint Retrospective** (End of sprint)
   - What went well?
   - What could improve?
   - Action items for next sprint

Julie

"However, for the team project, we modify these ceremonies for the classroom context."

Prof. Cho

"Yes, we adapt Scrum to fit our academic schedule!"

He writes the modified version:

Modified Scrum Ceremonies for Class Project:

1. ***PPP (Project Plan Presentation)*** *(Start of project)*
   - *Present overall project plan*
   - *Define all sprint goals*
   - *Show initial backlog*
   - *One time only, not per sprint*
2. ***Weekly Standup*** *(Every week, not daily)*
   - *What did I do this week?*
   - *What will I do next week?*
   - *Any blockers?*
3. ***1st Sprint Review + 2nd Sprint Planning*** *(End of the 1st sprint)*
   - *Demo completed sprint work*
   - *Get professor/peer feedback*
   - *Plan next sprint immediately after*
   - *Combined into one ceremony*
4. ***Sprint Retrospective*** *(End of sprint)*
   - *What went well?*
   - *What could improve?*

- *Action items for next sprint*
- *Same as standard Scrum*

5. ***Final Project Presentation*** *(End of project)*
   - *Demo final product*
   - *Reflect on the overall project*
   - *Not part of Scrum, specific to the course*

Prof. Cho

"Notice how we adapted to our context: weekly instead of daily standups, and combined ceremonies to save time while keeping the core Scrum principles."

Ken

"Each ceremony serves a purpose:
- Planning aligns the team
- Standup maintains communication
- Review gets stakeholder feedback
- Retrospective drives continuous improvement."

Tim

"So these ceremonies embody the four columns of Agile principles—ICCE!"

Jane

"Yes! Iterative delivery (sprints), Collaboration (standups, reviews), Change (backlog adaptation), Excellence (retrospectives)."

Prof. Cho

"Exactly! Now you see how Scrum **implements** Agile philosophy."

Scrum Artifacts

Prof. Cho

"Scrum also defines three artifacts—tangible outputs that track progress."

Artifact 1: Product Backlog

Prof. Cho

"The Product Backlog is your master list - everything you plan to build for the entire project."

Product Backlog:

- *Complete list of ALL features/requirements for entire project*
- *Ordered list of everything needed in the product*
- *Owned by Product Owner*
- *Constantly evolving based on feedback*
- *Items prioritized by value to the customer*

Example:

1. *[High Priority] User can create tasks*

2. [High Priority] User can mark tasks complete
3. [Medium Priority] Email reminders 7 days before due
4. [Medium Priority] Filter tasks by priority
5. [Low Priority] Export tasks to CSV
6. [Low Priority] Dark mode UI

Tim

"So the Product Backlog contains everything we might build over all sprints?"

Prof. Cho

"Exactly! It's the big picture. Think of it as your project's wishlist, prioritized by importance."

Characteristic	Description
Ordered	Top items are highest priority
Dynamic	Changes based on feedback and learning
Detailed at top	High priority items well-defined, low priority items vague
Owned	Product Owner maintains
Scope	Entire project/semester

Tim

"But in the team project, we already have a defined set of features and requirements from the start. Do we have to make the Product Backlog again?"

Prof. Cho

"Great question! No, you don't create a completely new backlog. Your initial requirements document becomes the foundation of your Product Backlog."

Prof. Cho

"However, you should embrace change too!. As you make progress, you may discover new requirements or need to reprioritize existing ones based on feedback. So keep it updated throughout the project."

Jane

"So the Product Backlog is just a different format of our requirements, only that they can be updated throughout the project?"

Prof. Cho

"Exactly! It's your requirements expressed as user stories, prioritized, and managed in GitHub. As you work, you might discover new requirements or change priorities - that's why it's called a 'living document'."

Artifact 2: Sprint Backlog

Prof. Cho

"The Sprint Backlog is just what you commit to finishing in ONE sprint."

Sprint Backlog:

- *Subset of Product Backlog selected for THIS sprint only*
- *Team selects items they can realistically complete*
- *Owned by Development Team*
- *Can be updated based on progress during the sprint*
- *Can be broken down into specific tasks with time estimates (if necessary)*

Tim

“Can it be updated based on progress during sprint?”

Prof. Cho

“Good catch! Yes, the Sprint Backlog can be updated during the sprint as you learn more. For example, if a task takes longer than expected, you can adjust estimates. Or if you finish early, you might pull in another item.”

Ken

“But I thought the sprint was fixed? No changes allowed?”

Prof. Cho

“In a way, it is always a good idea to finish whatever is expected to be finished during the sprint. However, Scrum allows flexibility within the sprint. The idea is to commit to a goal, but adapt how you achieve it. And ‘No surprises.’”

Jane

"I see, instead of hiding, I mean not telling the team that something is taking longer than expected, we should communicate and adjust the Sprint Backlog accordingly."

Prof. Cho

"Perfect! And if you finish early or realize something takes longer, you can adjust the Sprint Backlog during the sprint. But we should let everyone know about these changes as quickly as possible."

Ken

"Where do we manage all these backlogs?"

Prof. Cho

"Great question! For this class, we use specific tools to manage our artifacts."

He writes on the whiteboard:

Our Artifact Management Tools:[60]

1. **GitHub** - Primary artifact management
   - Product Backlog ⟶ GitHub Issues (labeled: backlog)
   - Sprint Backlog ⟶ GitHub Project Board
   - Tasks ⟶ GitHub Issues (assigned to sprint)
   - Code ⟶ GitHub Repository
2. **Canvas Project Page** - Central information hub
   - Links to all GitHub artifacts
   - Project progress updates
   - Sprint review summaries (or links to the summaries)
   - All information is easily accessible from one place
3. **Individual Contribution Pages**
   - Each team member maintains their own page
   - Weekly progress updates required
   - Links to completed features/requirements
   - (Optional) Reflection on challenges and learnings

Jane

"So GitHub has the actual work, and Canvas connects everything together?"

[60] This is a guideline, so each team should adjust as needed

Prof. Cho

"Exactly! Canvas is your project dashboard - anyone can visit your project page and see current status, links to GitHub, and each member's contributions."

Jane

"Why do we need individual contribution pages?"

Prof. Cho

"Transparency and accountability. Each team member updates their page weekly showing what they accomplished, what they're working on, and any blockers. This helps me see individual contributions and helps teammates coordinate."

Tim

"Got it. So GitHub is where the work happens, and Canvas is where we report and reflect."

Jane

"Could you explain me about the Github issues?"

Prof. Cho

“Sure! GitHub Issues are where we track individual features, bugs, or tasks. Each issue represents a single piece of work. You can assign issues to team members, label them (e.g., ‘backlog’, ‘in progress’, ‘done’), and link them to milestones (sprints).”

Tim

“I’m not familiar with GitHub issues and some of the other tools that you told us before. Should I use all the tools you mentioned?”

Prof. Cho

“Not at all. Focus on solving problems and delivering value. Use the tools that help you do that effectively. GitHub Issues are great for tracking work, but if your team prefers another tool, that’s fine too.”

Prof. Cho

"The key is clear communication and transparency. It's even OK to use only Canvas if that works best for your team."

Jane

"Thanks for the clarification! I feel more confident about managing our work now."

Julie

"But don't forget that we need to use tools to manage complexity and solve problems, so at some point we need to use these tools and many others as a good professional software engineer effectively."

Artifact 3: Increment

Prof. Cho

"The Increment is what you actually deliver - working, tested software."

Increment:

- *Sum of all completed Product Backlog items*
- *Must be working, tested, and potentially shippable*
- *Definition of "Done" must be met*
- *Measurable progress toward the goal*

Definition of "Done" Example:

```
  A feature is "Done" when:
  ☑ Code written and merged to the main branch
  ☑ Unit tests written and passing (>80%
coverage)
  ☑ Integration tests written and passing
  ☑ Code reviewed by another team member
  ☑ Documentation updated
  ☑ Accepted by Product Owner
  ☑ Deployed to staging environment
```

Ken

"How do we measure our increment? How do we know if we're making good progress?"

Prof. Cho

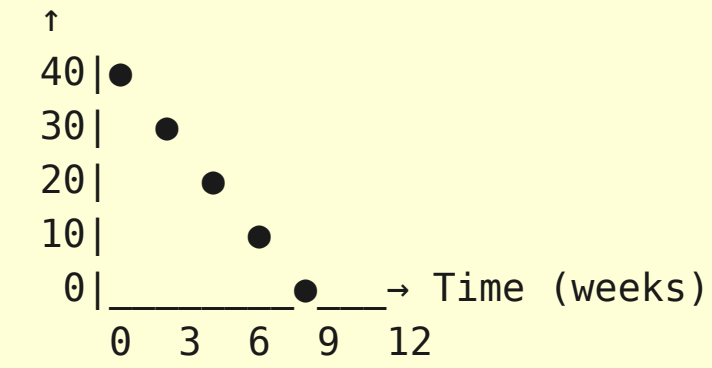
"Excellent question! We measure increment in three key ways:"

Measuring Increment:

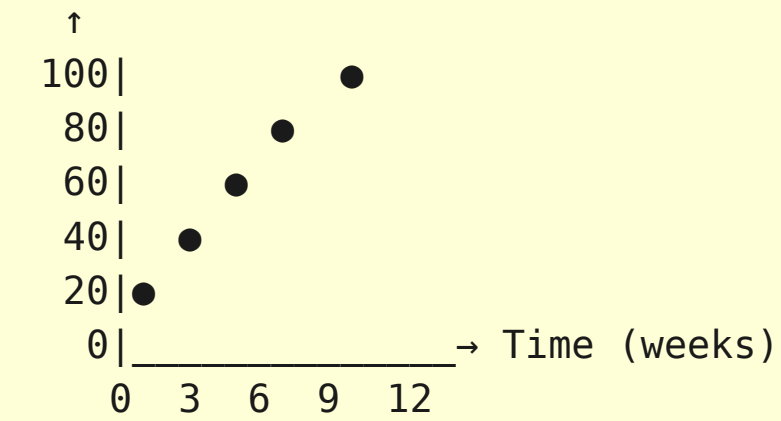
1. **Lines of Code (LoC)**
   - Track actual code written
   - Shows tangible output
   - Example: Sprint 1 = 500 LoC, Sprint 2 = 1200 LoC total
2. **Test Count**
   - Number of total unit tests
   - Number of total integration tests
   - Number of total E2E/acceptance tests
   - Indicates quality and coverage (more tests = better quality)
   - Example: Sprint 1 = 50 tests, Sprint 2 = 120 tests passing
3. **Burndown Chart (or Rate)**
   - Shows work remaining vs. time
   - Visualizes if the team is on track
   - Updates daily/weekly
   - Measure how many features/requirements are finished and how many are left to do

Example Burndown Chart:

Requirements Remaining



Burndown Rate (% Complete)



Ideal: Line goes down steadily Problem: Line stays flat (not making progress)

Jane

"So we can see if we're falling behind by looking at the burndown chart?"

Prof. Cho

"Exactly! If the line isn't going down, you need to address blockers or re-scope the sprint. These metrics help you stay honest about progress."

Tim

"I'm a little bit confused about the difference between user stories and tasks. Could you clarify it for us?"

Prof. Cho

"Sure! Understanding the difference between user stories and tasks is crucial for effective Agile development."

User Stories vs Tasks:

User Story: Feature from the user's perspective

- Format: "As a [user], I want [feature] so that [benefit]."
- Example: "As a student, I want to create tasks so that I can track assignments."
- Size: Completable in one sprint
- Lives in: Product Backlog → Sprint Backlog

Task: Technical step to implement the user story

- Format: Technical description + time estimate
- Example: "Create Task model with title, dueDate, priority fields (2 hours)."
- Size: Completable in hours, not days
- Lives in: Sprint Backlog only

Relationship:

- One user story → Multiple tasks (3-8 tasks typical)
- Sprint Backlog contains user stories
- Team breaks user stories into tasks during Sprint Planning
- Tasks are what you actually work on daily

Jane

"But, do we need to translate our requirements into user stories AND then break them into tasks again? That seems like extra work."

Prof. Cho

"Great question! Let me clarify the flow - it's not about duplicating work, but about progressive refinement."

He draws on the whiteboard:

From Requirements to Working Code:

Step 1: Requirements (Initial planning) "System shall allow users to create tasks."

↓ Convert to the user perspective

Step 2: User Story (Product Backlog) "As a student, I want to create tasks so I can track my assignments."

↓ Breakdown during Sprint Planning

Step 3: Tasks (Sprint Backlog)

- *Create Task model (4 hours)*
- *Build POST /tasks API (3 hours)*
- *Create task form UI (5 hours)*
- *Write tests (4 hours)*

↓ Daily work

Step 4: Actual coding and commits

Ken

"So the user story is WHAT we're building, and tasks are HOW we build it?"

Prof. Cho

"Perfect! User stories stay high-level and user-focused. Tasks are the nitty-gritty technical steps. You don't break into tasks until Sprint Planning when you're ready actually to work on it."

Tim

"Oh! So we don't need to create tasks for everything in the Product Backlog, only for what we're doing THIS sprint?"

Prof. Cho

"Exactly! That would be a waste of time. You only break user stories into tasks when you pull them into the Sprint Backlog. Why?"

Jane

"Because by then we know more details and can estimate better?"

Prof. Cho

"Yes! And requirements might change before you get to low-priority items, so detailed task planning would be wasted effort. This is called 'just-in-time planning.'"

When to Create What:

Product Backlog (at project start): ✅ *Convert requirements to user stories* ✅ *Prioritize user stories* ❌ *Don't create tasks yet*

Sprint Planning (start of each sprint): ✅ *Select user stories for this sprint* ✅ *Break selected user stories into tasks* ✅ *Estimate task hours* ❌ *Don't task-ify the entire backlog*

Daily Work: ✅ *Work on tasks* ✅ *Update task status* ✅ *Commit code with task references*

Ken

"That makes sense! So we save time by only planning in detail what we're about to build."

Prof. Cho

"Exactly! And in GitHub, this looks like: Product Backlog = Issues, Sprint Backlog = Issues with task checkboxes added."

He shows an example.

Evolution: From Product Backlog to Sprint Backlog (Task)

In Product Backlog (before sprint):

```
Issue #12: User can create tasks

Description:
As a student, I want to create tasks with
title and due date so I can track assignments.
```

Added to Sprint Backlog (during Sprint Planning):

```
Issue #12: User can create tasks

Description:
As a student, I want to create tasks with
title and due date so I can track assignments.

Tasks:
- [ ] Create Task model (4h) - @tim
- [ ] Build POST /tasks API (3h) - @jane
- [ ] Create UI form (5h) - @ken
- [ ] Write tests (4h) - @tim

Total: 16 hours
```

Jane

"Oh! So the same requirement just gets more detailed when we're ready to work on it?"

Prof. Cho

"Precisely! You're not creating new artifacts, just adding implementation details at the right time."

Tim

"But I think breaking into tasks might be unnecessary overhead for small user stories?"

Prof. Cho

"Good point! For small user stories that can be done in a few hours, you might skip a detailed task breakdown. Just track it as a single item. The key is balancing detail with efficiency."

Jane

"I got it, the goal is to manage complexity without unnecessary overhead."

Prof. Cho

"Yes. Use your judgment based on story size and team preference. Focus on solving problems, not bureaucracy. Use the tools only when they help you deliver value effectively."

Julie

“I see - if we’re adding complexity just to manage complexity, then the overhead defeats the purpose. The tools should simplify work, not complicate it.”

Prof. Cho

“Exactly! The goal is to manage complexity effectively, not to create more complexity. Always ask: Does this help us deliver value? If not, simplify.”

Understanding the Full Picture

With the software process explained, the team now understood how everything connected.

Ken

"Now I see the full software engineering picture:

1. **Requirements**: WHY solve this? WHAT to build?
2. **Architecture**: HOW to organize at a high level?
3. **Design**: HOW to structure in detail?
4. **Implementation**: Write the code
5. **Testing**: Verify quality
6. **Process**: Coordinate team, deliver incrementally."

Jane

"And Agile/Scrum provides the process to tie it all together!"

Tim

"I feel ready for the team project now. We have the tools and the process."

Prof. Cho

"Exactly! Software engineering isn't just about coding— it's about managing complexity through:

- Clear requirements
- Good architecture
- Thoughtful design
- Rigorous testing
- Effective process"

Process Enables Everything Else

Prof. Cho

"Remember: Process is not bureaucracy. Good process **enables** good work, it doesn't hinder it."

Prof. Cho

"Each technical skills area benefits from a solid process. Without process, even the best skills can lead to chaos."

Without Process	Technical Skill	With Process
Unclear priorities	Requirements	Prioritized backlog
Over-engineering	Architecture	MVP first, Focus on Data model
Premature optimization	Design	Simple, emergent design
Sloppy code	Implementation	Code reviews, standards
Skipped tests	Testing	TDD, CI/CD

Prof. Cho

"The result? Higher quality software, delivered faster, with happier teams."

Summary: The Agile Advantage

Principle	How It Helps
Iterative	Small increments, frequent feedback → catch problems early
Adaptive	Embrace change, adjust quickly → respond to real needs
Collaborative	Team + stakeholders work together → shared understanding
Transparent	Progress visible at all times → no surprises
Sustainable	Constant pace, technical excellence → avoid burnout
Empowering	Self-organizing teams → ownership and motivation

The journey from understanding software engineering concepts to applying them in a structured process had prepared the team for real-world software development.