Managing Risk and Technical Debt

17-313 Fall 2026

Foundations of Software Engineering

https://cmu-17313q.github.io

Eduardo Feo Flushing



Administrivia

- P4A deadline: Tuesday, end of day. Plan your interviews.
- Gradebook is online (since October 19; check the Slack announcement).
- P4B posted soon.
- Recitation tomorrow.

Learning Goals

- Prioritize project risks and choose appropriate controls.
- Identify technical debt and explain its future cost.
- Justify a decision to accept or reduce a specific debt.

Outline

- What could prevent our project from succeeding?
- How do we assess and control risk?
- How do today's choices affect tomorrow's work?
- How do we decide what debt to accept or repay?

What Is Software Risk?

- A risk is a measure of the potential inability to achieve overall program objectives within defined cost, schedule, and technical constraints.
- A risk is about the future: a possible event that has not happened yet.
- This is different from an **issue**: a problem that has already occurred.



Conrow, E. 2003. Effective Risk Management: Some Keys to Success, 2nd ed. Reston, VA, USA: American Institute of Aeronautics and Astronautics (AIAA).

Likelihood and Impact



Probability (likelihood) of failing to achieve a particular outcome.



Impact (consequences) of failing to achieve that outcome.

Likelihood alone cannot determine priority.

NOTES

Compare two project risks: a likely-but-minor one can matter less than a rare-but-severe one.

Internal and External Risks

Originates Within the Project



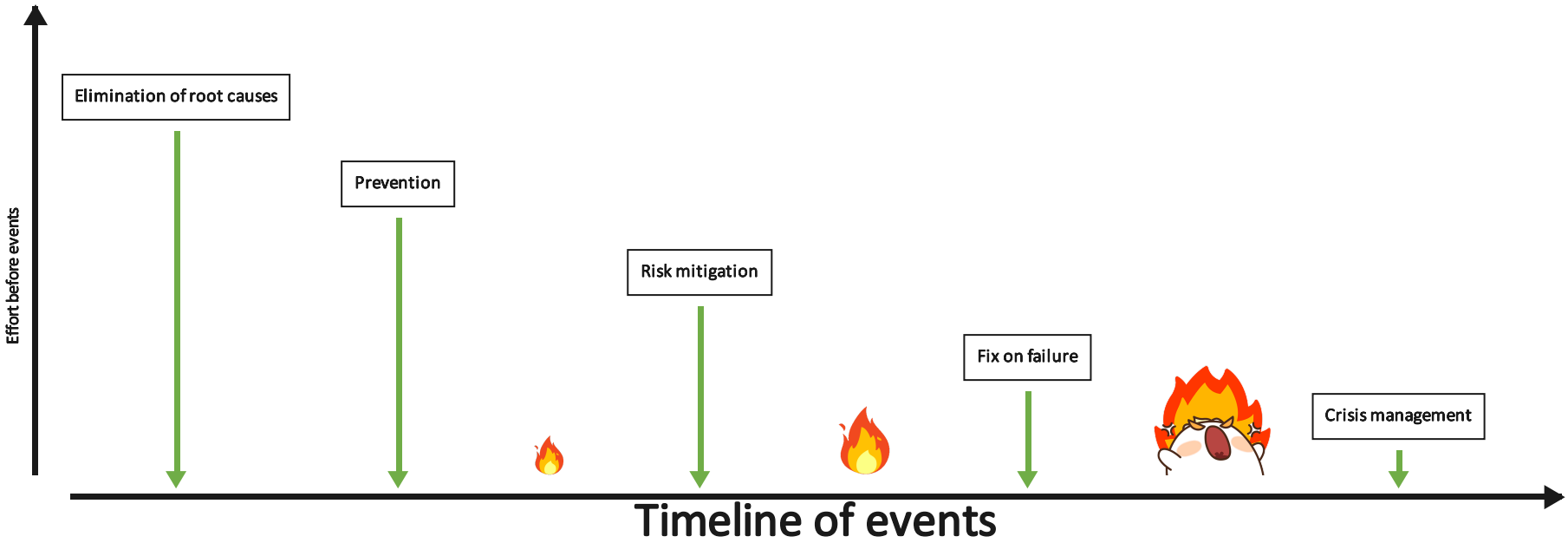
Scope, schedule, staffing, design decisions.

Originates Outside the Project



Vendor changes, market shifts, natural events.

Levels of Risk Management



McConnell, S. Rapid Development: Taming Wild Software Schedules. 1996.

Levels of Risk Management

1. **Elimination of root causes:**
   - Identify and eliminate factors that make it possible for risks to exist at all.
2. **Prevention:**
   - Implement and execute a plan to identify risks and prevent them from becoming problems.
3. **Risk mitigation:**
   - Plan ahead to provide resources to cover risks if they occur, but do nothing to eliminate them.
4. **Fix on failure:**
   - Detect and react to risks quickly, but only after they have occurred.
5. **Crisis management:**
   - Fire fighting; address risks only after they become problems.

Not covered in this class

NOTES

In this lecture, prevention reduces the likelihood of failure, while mitigation limits its consequences. Elsewhere, mitigation can refer to both. Root-cause elimination addresses a specific source of risk; other sources may remain.

McConnell, S. Rapid Development: Taming Wild Software Schedules. 1996.

Outline

- What could prevent our project from succeeding?
- **How do we assess and control risk?**
- How do today's choices affect tomorrow's work?
- How do we decide what debt to accept or repay?

From Levels to Tasks

To eliminate causes, prevent failures, or prepare for their consequences, we need to:

- **Assess risks:** identify what could go wrong, analyze causes and consequences, and prioritize.
- **Control risks:** plan and implement responses, then monitor their effectiveness.

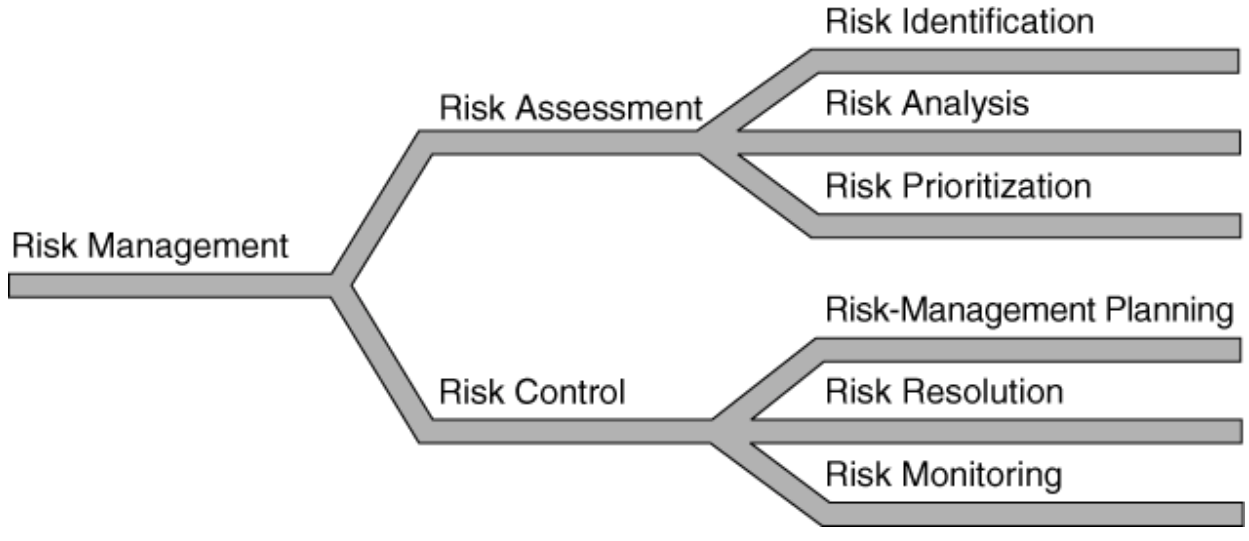
Example: duplicated validation rules may become inconsistent.

Approach	What assessment helps us understand	What control puts into practice
Root-cause elimination	Multiple copies require manual synchronization.	Replace the copies with one shared implementation.
Prevention	Developers may overlook copies when making changes.	Add consistency tests and review checks.
Mitigation	Inconsistent validation could disrupt users.	Prepare a rollback procedure to limit disruption.

NOTES

Consider an app with separate copies of username-validation code in its registration and profile-edit forms. Updating one copy could make the forms enforce different rules. The levels describe how proactively we address this risk. Assessment and control provide the tasks for choosing, implementing, and monitoring a response.

Tasks of Risk Management



Assessment identifies and prioritizes risks. Control plans, implements, and monitors the chosen responses.

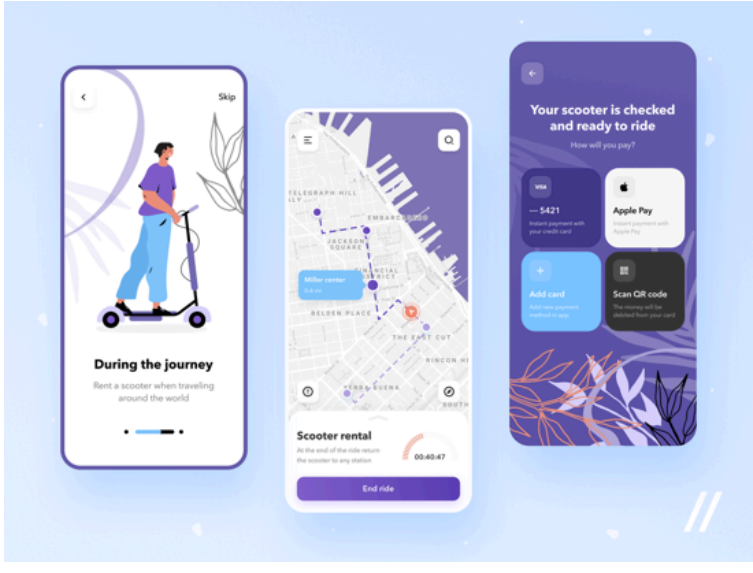
McConnell, S. Rapid Development: Taming Wild Software Schedules. 1996.

Activity 1a: Risk Identification

Your team is building a scooter-rental app: riders find a nearby scooter on a map, unlock it, ride, and end the rental from their phone.

What risks exist for the scooter app?

Identify three plausible risks, including at least one beyond a code defect.



2 min individually · 2 min discussing

Qualitative Risk Assessment

TABLE III. Risk assessment matrix

RISK ASSESSMENT MATRIX				
SEVERITY / PROBABILITY	Catastrophic (1)	Critical (2)	Marginal (3)	Negligible (4)
Frequent (A)	High	High	Serious	Medium
Probable (B)	High	High	Serious	Medium
Occasional (C)	High	Serious	Medium	Low
Remote (D)	Serious	Medium	Medium	Low
Improbable (E)	Medium	Medium	Medium	Low
Eliminated (F)	Eliminated			

Severity × probability: a contextual low/medium/high rating is often enough.

NOTES

Qualitative assessment combines likelihood and impact categories.

Risk Exposure and Prioritization

Risk	Probability	Size of Loss (weeks)	Risk Exposure (weeks)
Overly optimistic schedule	50%	5	**2.5**
Additional features added by marketing	35%	8	**2.8**

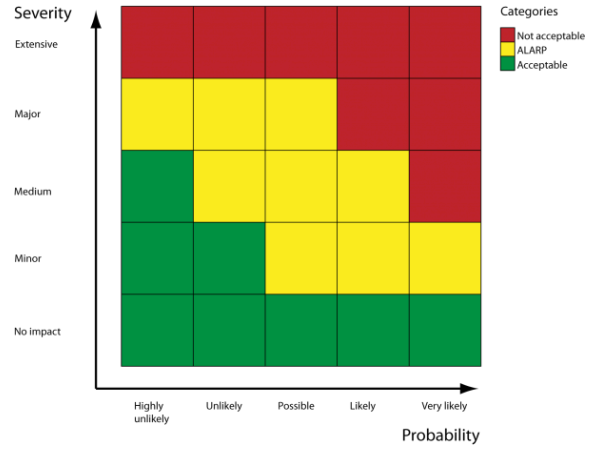
NOTES

Risk exposure = probability times loss. Focus on the highest-exposure risks first. Numerical exposure requires an estimated probability and a loss expressed in a meaningful unit. A 50% probability of losing five weeks produces an expected loss of 2.5 weeks. The actual outcome depends on whether the event occurs.

Activity 1b: Risk Analysis

For the three risks from Activity 1a, rate likelihood and impact, and state your assumptions.

Select your two most important risks to carry forward.



2 min individually · 2 min discussing

Numbers Need Assumptions

- Risk analysis can become a numbers game, used to justify a decision that was already made.
- A number is only as good as the assumptions behind it.

- Record the assumptions behind each estimate.
- Acknowledge the uncertainty in the estimate itself.
- Revise estimates as you learn more.

Risk Control and Ownership

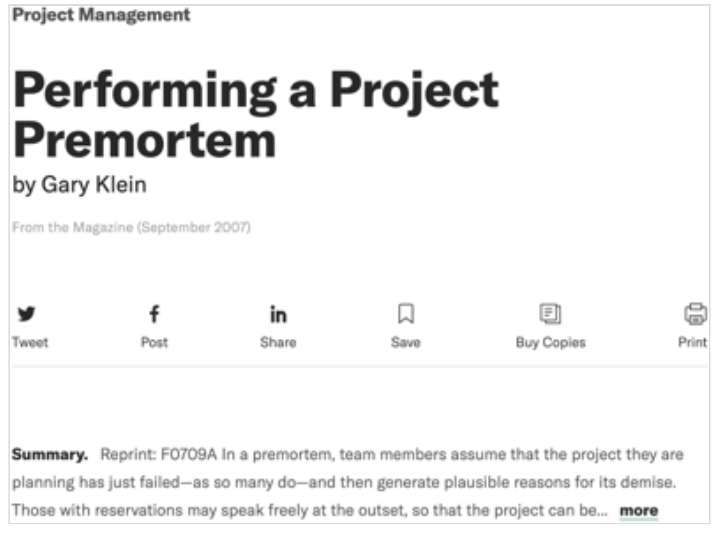
- What steps can be taken to avoid or mitigate the risk?
- Can you better understand or eliminate the cause?
- Who is responsible if this risk materializes?

NOTES

A useful risk control has four parts: a risk statement, a response, an owner, and a review trigger.

Premortems

- Imagine the project has already failed.
- Ask: "What happened?"
- Unlike a typical critique, a premortem assumes failure up front, so people speak freely instead of defending the plan.



Project Management

Performing a Project Premortem

by Gary Klein

From the Magazine (September 2007)

Tweet Post Share Save Buy Copies Print

Summary. Reprint: F0709A In a premortem, team members assume that the project they are planning has just failed—as so many do—and then generate plausible reasons for its demise. Those with reservations may speak freely at the outset, so that the project can be... **more**

This is the technique behind Activity 1a: imagining failure surfaces risks people might otherwise hold back.

Klein, G. Performing a Project Premortem. Harvard Business Review, 2007.

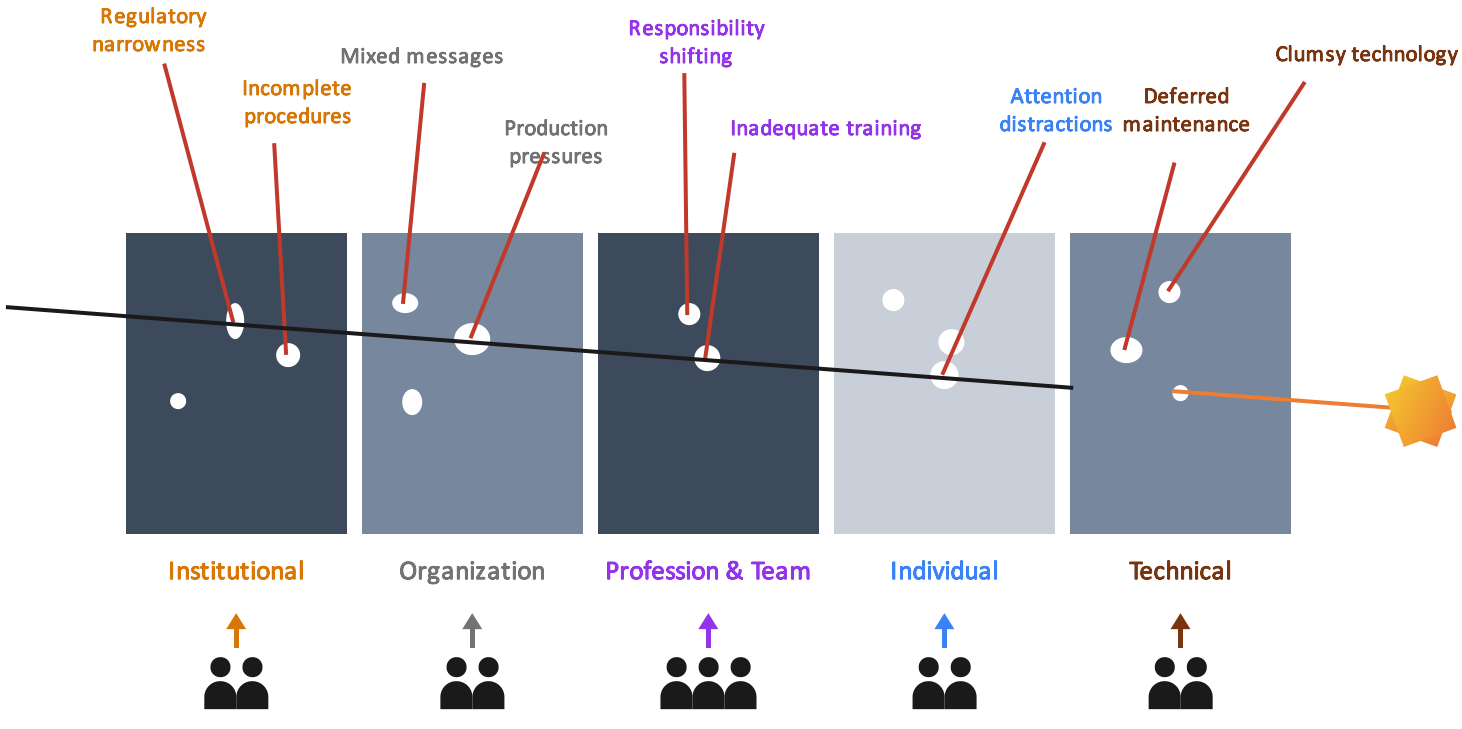
Activity 1c: Risk Elimination and Mitigation

For your highest-priority risk, how would you eliminate or mitigate it?

1. Decide: can the cause be eliminated, or only its likelihood or impact reduced?
2. Specify a response, an owner, and a review trigger.

2 min completing the response · 2 min reporting and debrief

Swiss Cheese Model



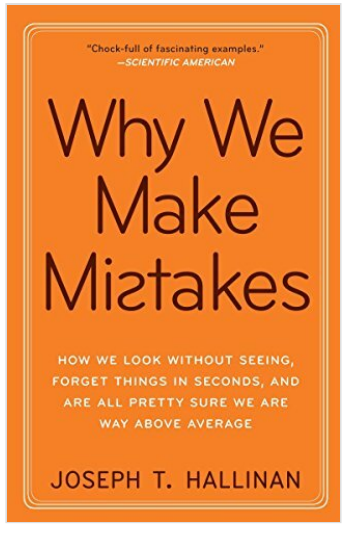
NOTES

Each layer has gaps. A failure gets through only when the gaps line up.

Modified from Reason, 1999, by R.I. Crook.

Why Human Error Persists

- We cannot reliably find our own typos: it is hard to review attention to our own, familiar work.
- Boredom and repetitive tasks increase error rates.
- We cannot remove human error, but we can design processes that prevent and detect mistakes.



Automation and Human Review

Automate what we can. Review what we cannot.

- Automated checks are fast, consistent, and catch what they were built to catch.
- For everything else, a human reviewer brings judgment and context an automated check does not have.

Code Review: What Deserves Attention?

- Understand the author's intent before judging the implementation.
- Focus on design and failure cases, not style.
- Automate formatting and other mechanical checks; don't spend review time on them.
- Give feedback respectfully: a reviewer critiques code, not the person who wrote it.

NOTES

A superficial review (style nits, rubber-stamp approval) misses the defects that matter most.

A Short Checklist

- A checklist does not replace judgment; it protects against forgetting something you already know matters.
- Checklists help manage complex processes where a missed step is easy and costly.

Before merging a change, ask:

- Does this change have a test for the behavior it adds or fixes?
- Does it handle the failure cases, not just the happy path?
- Would someone unfamiliar with this code understand it in six months?

Part II: Technical Debt and Future Project Risk

Technical Debt: Example

Technical debt: a shortcut now that makes future changes more costly.

To meet the sprint deadline, a team copies the same discount-calculation code into checkout and invoice generation.

Later, the discount rules change.



Concept	Shared example
The debt	Two separate implementations of the same calculation.
The interest	Every rule change requires updating and testing both copies.
The related risk	Someone updates only one copy, so the checkout total and invoice disagree.

NOTES

The initial shortcut saves effort; maintaining it creates additional work.

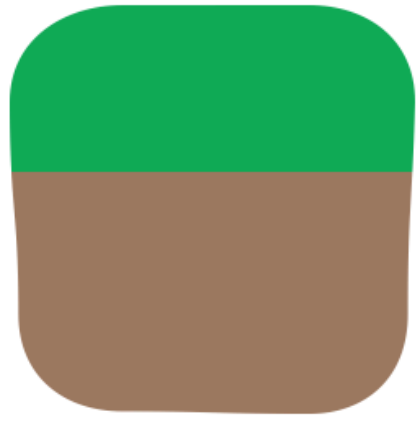
Outline

- What could prevent our project from succeeding?
- How do we assess and control risk?
- **How do today's choices affect tomorrow's work?**
- How do we decide what debt to accept or repay?

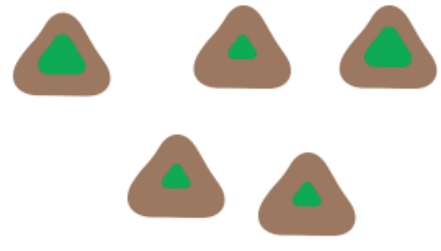
The Technical Debt Metaphor

*Any software system has a certain amount of **essential** complexity required to do its job...*

*... but most systems contain **cruft** that makes it harder to understand.*



*Cruft causes changes to take **more effort***



The technical debt metaphor treats the cruft as a debt, whose interest payments are the extra effort these changes require.

martinfowler.com/bliki/TechnicalDebt.html

What Counts as Technical Debt?

"In software-intensive systems, technical debt consists of design or implementation constructs that are expedient in the short term but set up a technical context that can make future change more costly or impossible."

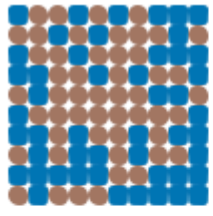
Avram, Kruchten, Nord, Ozkaya. Reasoning about Technical Debt.

NOTES

Technical debt needs an identifiable element and an explanation of its future cost, not just something you dislike about the code.

Internal Quality and the Cost of Change

If we compare one system with a lot of cruft...



the cruft means new features take longer to build



this extra time and effort is the cost of the cruft, paid with each new feature

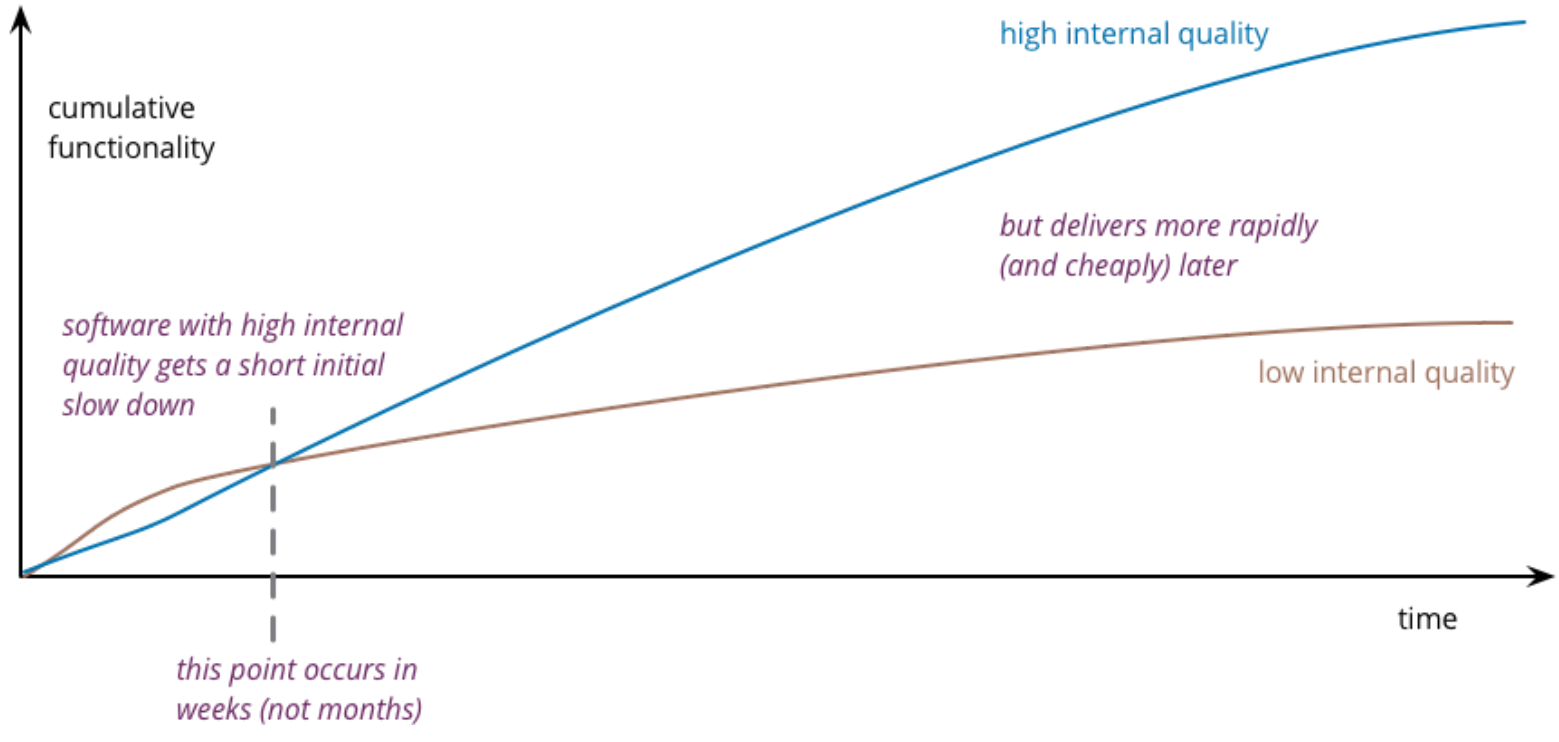
...to an equivalent one without





free of cruft, features can be added more quickly

Internal Quality as an Investment



Where Debt Comes From

- Tightly-coupled components.
- Poorly-specified requirements.
- Time pressure, with no time allotted to pay it back.
- Lack of knowledge: the team didn’t yet know a better way.
- Lack of ownership: no one is clearly responsible for noticing or fixing it.

NOTES

Distinguish causes (like time pressure) from concrete debt items (like duplicated handlers).

All Software Comes With Debt

Bit rot (also known as software decay or environmental drift) is the gradual, natural deterioration of a software system's performance, reliability, and security over time, even if the software itself doesn't change.



Dependencies, APIs, and environments keep changing, even when your code does not.

The Consequences of Excessive Debt

- Bad code can be demoralizing.
- Conversations with the client become awkward.
- Team infighting.
- Turnover and attrition.
- Development speed drops.



NOTES

Repeated extra work compounds into delivery uncertainty: estimates stop being reliable.

Different Kinds of Technical Debt

A reasonable call at the time?

Knowingly taken on?

	Reckless	Prudent
Deliberate	*"We don't have time for design."*	*"We must ship now and deal with consequences later."*
Inadvertent	*"What's layering?"*	*"Now we know how we should have done it."*

NOTES

Prudent, inadvertent debt can emerge from a reasonable design based on the knowledge available at the time. Later experience reveals limitations and suggests a better approach. Assess the original decision using the information and constraints the team faced when making it.

martinfowler.com/bliki/TechnicalDebtQuadrant.html

Activity 2: Technical Debt in CoolCamel

Describe two plausible examples of technical debt in the CoolCamel scenario from exam 1:

1. One deliberate, prudent example.
2. One reckless, inadvertent example.

For each: discuss the reason for incurring the debt (did it add value?) and a debt payback strategy.

NOTES

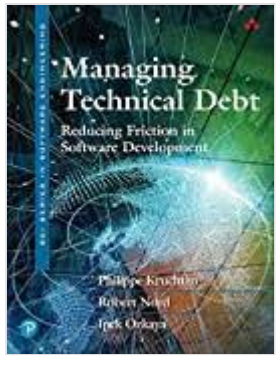
Possible deliberate, prudent example: hardcoding heat-index thresholds for Doha's climate instead of building a configurable, region-agnostic system, to ship the MVP faster. Reasonable if the team documents the assumption and plans to generalize before expanding to another city.

Possible reckless, inadvertent example: letting wearable-data parsing and GPS-dispatch logic become tightly coupled, without recognizing the coupling or seeking review, so a wearable firmware update silently breaks dispatch.

Outline

- What could prevent our project from succeeding?
- How do we assess and control risk?
- How do today's choices affect tomorrow's work?
- **How do we decide what debt to accept or repay?**

How Do We Manage Technical Debt?



Kruchten, Nord, Ozkaya. Managing Technical Debt: Reducing Friction in Software Development.

Principles: Making a Useful Debt Claim

- A debt claim is a dialogue: name the debt, and be ready to defend why it's debt.
- Interest can be a credible future cost, even before it has appeared yet.
- Debt is often widespread: one shortcut rarely stays isolated to one place.
- Trace the claim to a specific element of the system, not a general feeling.

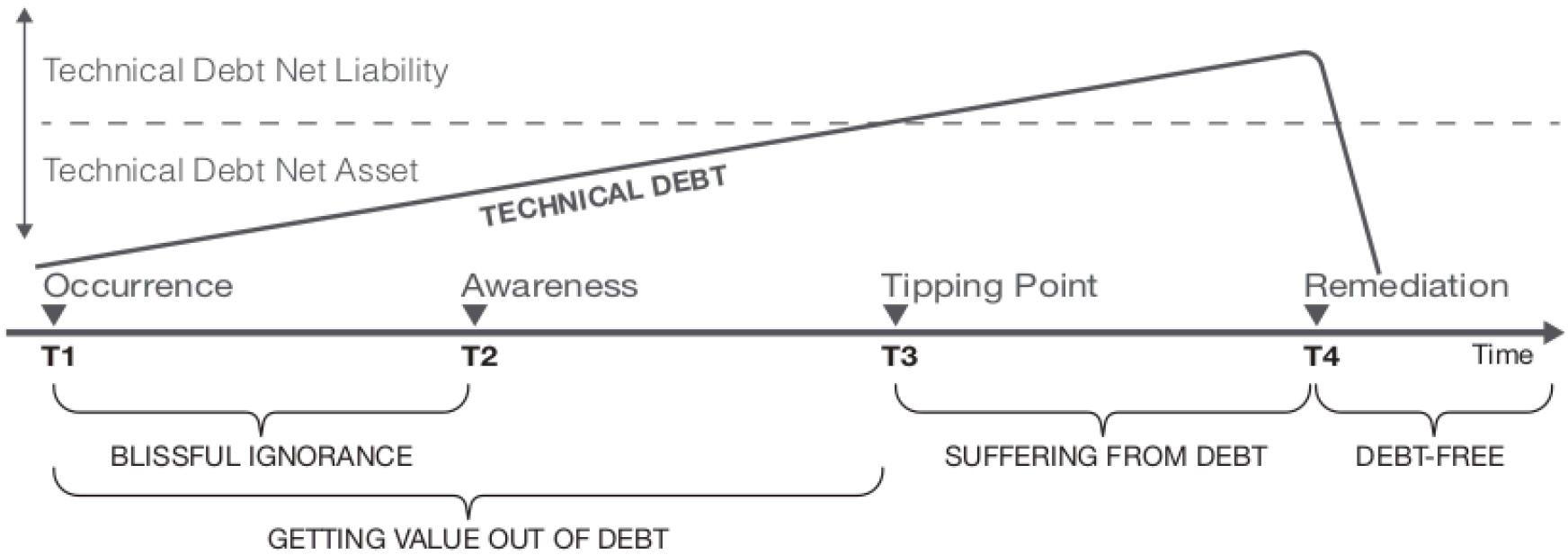
Principles: Context Determines Cost

A shortcut can be a reasonable tradeoff in one context and create substantial future costs in another.

Questions to ask:

- How long will this system remain in use?
- How often will the affected functionality change?
- How many other parts depend on it?

When Should We Reduce Debt?



NOTES

Compare future interest and risk against repayment effort, remaining lifetime, and competing work. The curve illustrates one possible evolution of technical debt. Its cost depends on future changes, dependencies, and the system's remaining lifetime. Teams should revisit these assumptions and compare expected future costs with repayment effort. Accepting some debt until retirement can be a reasonable decision.

Managing Technical Debt Continuously

- **Recognize** it: name the debt and the element it affects.
- **Make it visible**: track it where the team will actually see it.
- **Decide**: repay, schedule, monitor, or accept.
- **Live with it**: a decision not to repay yet is still a decision, made on purpose.

This is the same cycle as the risk-management process from Part I: identify, assess, decide, and track.

Example: A CoolCamel Debt Item

- **Affected code:** the server address is hardcoded in several app files.
- **Short-term benefit:** the team got the prototype working quickly.
- **Interest:** changing servers requires finding and updating every copy of the address.
- **Related risk:** a missed copy causes some requests to go to the old server.
- **Repayment:** store the address in one configuration setting that all requests use.
- **Owner and review trigger:** app maintainer, before the next server migration.

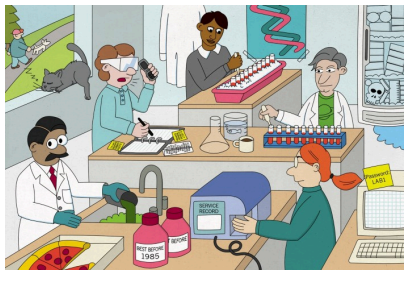
Reducing Inadvertent Debt

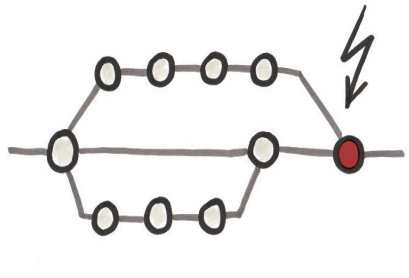
How can we avoid debt we didn't mean to take on?

- Focused code review, aimed at design and failure cases.
- Shared understanding: more than one person familiar with each part of the system.
- Clear ownership, so gaps don’t fall through the cracks.
- Checking assumptions before they spread through the codebase.

Process Choices Can Create Extra Work

- Missing checks: no QA process, or one nobody follows.
- Problematic integration and review practices: branches that live too long, reviews that rubber-stamp.
- Obstructive processes: steps so painful that people route around them.
- Repetitive manual work: the same manual step, every single time, instead of automated.









AI-Generated Code and Technical Debt

Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild

YUE LIU, Adelaide University, Australia
RATNADIRA WIDYASARI, University of Victoria, Canada
YANJIE ZHAO, Huazhong University of Science and Technology, China
IVANA CLAIRINE IRSAN, Singapore Management University, Singapore
JUNKAI CHEN, Singapore Management University, Singapore
DAVID LO, Singapore Management University, Singapore

AI coding assistants are now widely used in software development. Software developers increasingly integrate AI-generated code into their codebases to improve productivity. Prior studies have shown that AI-generated code may contain code quality issues under controlled settings. However, we still know little about the real-world impact of AI-generated code on software quality and maintenance after it is introduced into production repositories. In other words, it remains unclear whether such issues are quickly fixed or persist and accumulate over time as technical debt. In this paper, we conduct a large-scale empirical study on the code quality issues introduced by AI coding assistants in the wild. To achieve that, we built a dataset of 302.6k verified AI-authored commits from 6,299 GitHub repositories, covering five widely used AI coding assistants. For each commit, we run static analysis before and after the change to precisely attribute which code smells, correctness issues, and security issues the AI introduced. We then track each introduced issue from the introducing commit to the latest repository revision to study its lifecycle. Our results show that we identified 484,366 distinct issues, and that code smells are by far the most common type, accounting for 89.3% of all issues. Between 7.5% and 14.8% of each assistant's commits introduce at least one issue. More importantly, 22.7% of tracked AI-introduced issues still survive at the latest version of the repository. These findings show that AI-generated code can introduce long-term maintenance costs into real software projects and highlight the need for stronger quality assurance in AI-assisted development.

Liu, Widyasari, Zhao, Irsan, Chen, Lo. *Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild.* · arxiv.org/abs/2603.28592

AI-Generated Code Can Introduce Long-Term Maintenance Costs

- 302.6k commits from five popular AI coding assistants, tracked over time: issues attributed to the AI are checked again later to see if they get fixed or persist.
- **7.5% to 14.8% of each assistant's commits** introduced at least one issue.
- **22.7% of those issues still survive** at the latest version of the repository: AI-introduced debt doesn't get paid down on its own.

NOTES

The 22.7% figure refers to tracked issues that remained at the end of the observation period. Their persistence suggests continuing maintenance obligations. The effort eventually required will depend on how the affected code is used and changed.

https://arxiv.org/abs/2603.28592

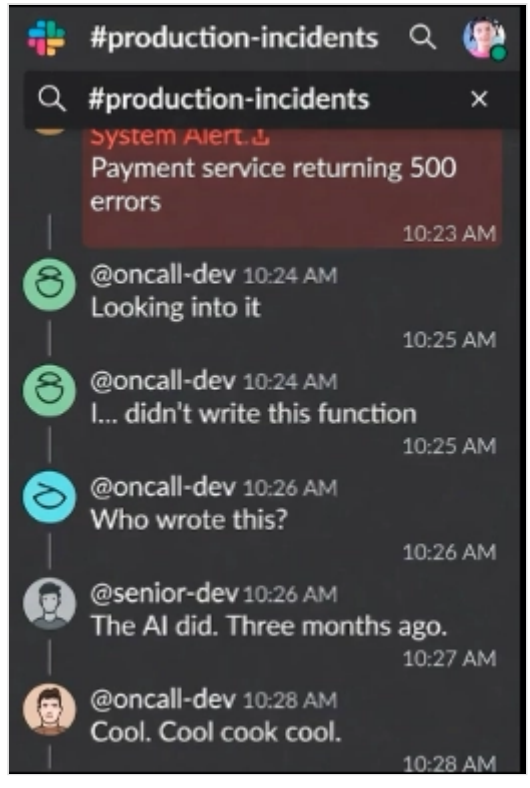
Discussion: Has Code Review Helped Your Team?

Reflect on your 17-313 experience, as both an author and a reviewer. Do you agree or disagree with each claim? Support your answer with a specific PR or review.

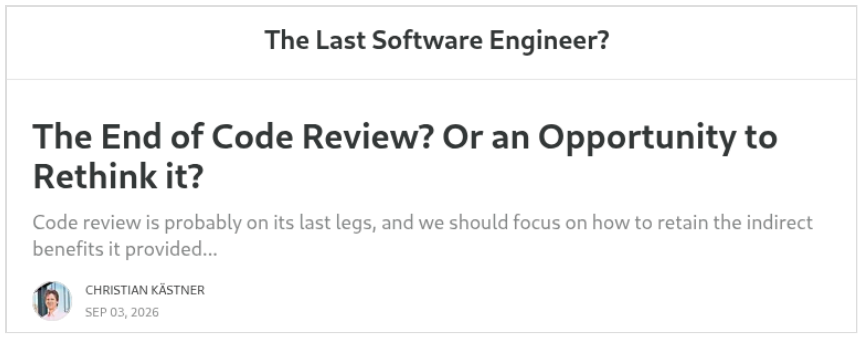
- "Code review has been worth the time we spend on it."
- "I have benefited from review beyond having bugs caught." What did you gain: learning, better design, understanding teammates' code, or something else?
- "An automated reviewer could provide the same benefits I've received from human review." Which benefits would be hardest to preserve?

3-4 minutes in pairs

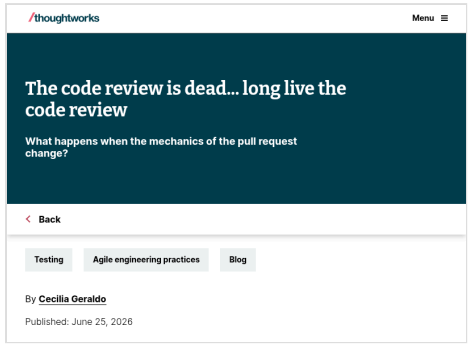
Who Wrote This?



Is Code Review as We Know It Dying?



thelastsoftwareengineer.substack.com



thoughtworks.com

AI accelerates code production, pressuring manual review. Automating it away risks losing:

- **Shared understanding & ownership**
  - design discussions, walkthroughs, team updates
- **Learning & mentorship**
  - explicit coaching, pairing, feedback
- **Technical excellence & risk management**
  - automate routine checks; humans focus on architecture, test design, safeguards

NOTES

Where does human involvement contribute most to understanding the system and changing it safely?

Takeaways

- Risk management and technical debt are the same underlying skill: name what could go wrong, decide what to do about it now versus later, and make sure someone owns the decision.
- Prioritize by exposure (probability × impact), not just likelihood.
- A debt claim needs a specific element and an explanation of its future cost, not just a bad feeling about the code.
- Explicit consequences, a named owner, and a review trigger turn a vague worry into a decision the team can actually act on.
- Code review's value goes beyond catching bugs: shared understanding, learning and mentorship, and technical excellence are benefits worth preserving deliberately, not just a side effect of approving pull requests.