Software Specifications

17-313 Fall 2026

Foundations of Software Engineering

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

Eduardo Feo-Flushing

S3D Software and Societa Systems Department

Carnegie Mellon University

Administrivia

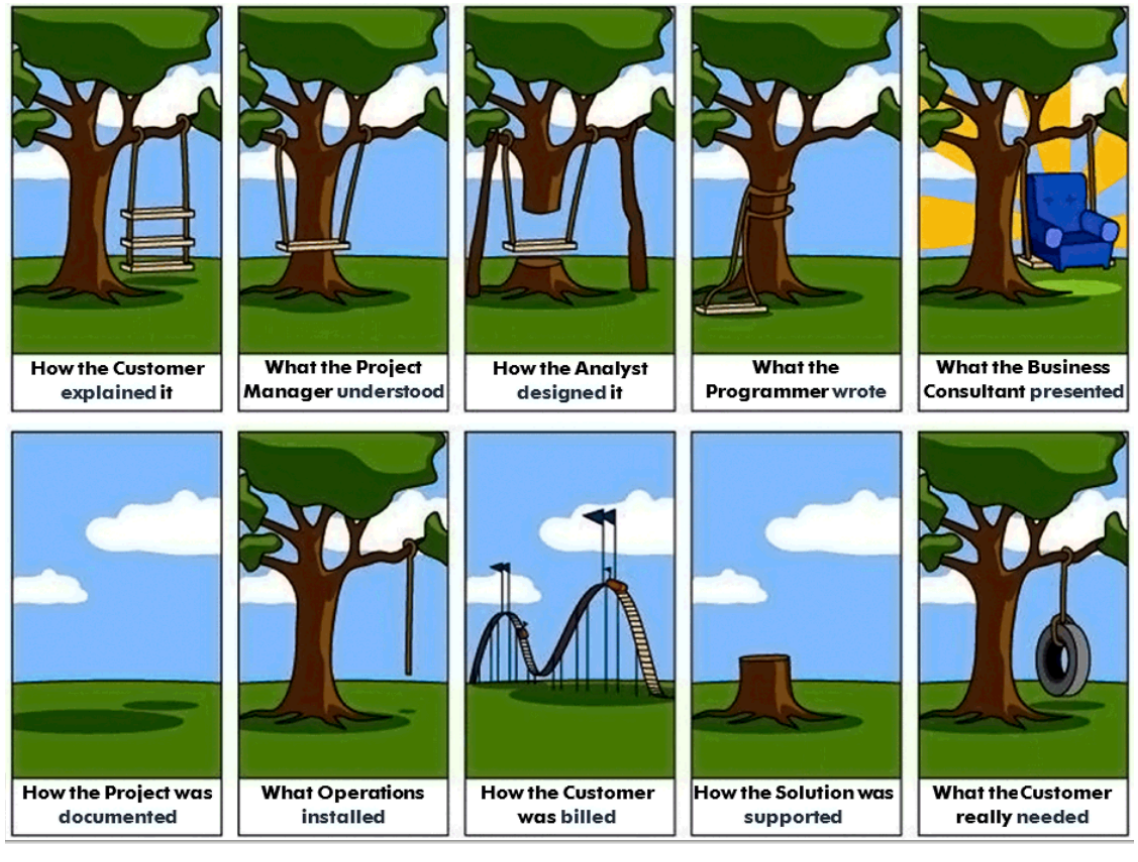
- **P2B**: Slides are due Sunday; all remaining deliverables are due Monday.
- Tomorrow's recitation: 2:30 p.m. in **Room 1030** (room change)
- Next quiz: After fall break.
- **P2B Extra Credit** for Fun: It was great to see everyone enjoy the last one.
  - Try something different this time!

Software Specifications

- Explain why specifications matter.
- Identify ambiguity and missing decisions.
- Refine a specification.
- Connect specifications to AI-assisted development and specifications.

NOTES

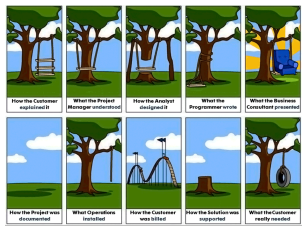
Today we'll focus on a deceptively simple question: how precisely must we describe software before someone else, or an AI agent, can build what we intended?



NOTES

This is known as the Tree Swing Cartoon (also titled "How IT Projects Really Work"). It is a legendary corporate meme dating back to the 1970s. It satirizes common communication breakdowns, misaligned expectations, and organizational disconnects across different departments during software development and corporate projects.

What Went Wrong?



Did the developers build the wrong thing, or did the team never establish a shared understanding of the right thing?

Many software failures begin before code is written: different people understand the desired behavior differently.

NOTES

Many software failures begin before code is written: different people understand the desired behavior differently.

What Is a Specification?

"A software specification describes the required behavior and constraints of a software system or component."

Specification	Design
What must happen	How it will be achieved
Required behavior and constraints	Components, structures, and technical decisions

NOTES

A specification constrains a design, but multiple designs may satisfy the same specification.

Why Write Specifications?

Agree: Establish shared expectations.

Decide: Expose unresolved questions while changes are inexpensive.

Build: Guide developers and AI coding agents.

Evaluate and evolve: Preserve intent for review and future changes.

Clarifying required behavior also makes scope, risks, and effort easier to estimate.

Further reading: Joel Spolsky, "Painless Functional Specifications"

NOTES

In the Pre-Lecture Reading, Spolsky argues that specifications let teams explore alternatives cheaply, save communication time, and support scheduling.

What Makes a Useful Specification?

- Clear
- Consistent
- Sufficiently complete
- Explicit about assumptions and constraints
- Precise about important cases
- Written at an appropriate level of detail

NOTES

"Sufficiently complete" is doing a lot of work here: not perfectly complete. A specification should express decisions that affect required behavior without prescribing every implementation detail.

Activity: WordMatch

An unfamiliar word-guessing game. Assume you've never played anything like it.

Green: the letter is in the secret word and in the correct position.

Yellow: the letter is in the secret word but in the wrong position.

Black: the letter is not in the secret word.

Assume this is an unfamiliar game. Use only the rules shown here.

NOTES

We're deliberately not calling this Wordle. If someone recognizes the game, they'll import rules we haven't stated; that's exactly the failure mode this activity studies.

Part 1: Interpret the Specification

In small groups (2 or 3 people), determine the expected behavior for:

1. Secret: **APPLE**; Guess: **ALLEY**

A	L	L	E	Y

2. Secret: **APPLE**; Guess: **PUPPY**

P	U	P	P	Y

3. A guess containing only four letters
4. A five-letter string not found in the dictionary

Record your answers and every assumption you had to make. There may not be a single correct answer under the current specification.

NOTES

The goal is not to reproduce official Wordle behavior. It is to discover what the supplied rules fail to determine.

Part 2: Ask an LLM

> *"You are interpreting the specification of an unfamiliar word game. Use only the rules below and do not rely on your knowledge of Wordle. Determine the expected behavior for each scenario. Identify any decisions that the rules leave unclear. Do not generate code."*

Compare:

- Did the LLM reach the same decisions?
- Did it explicitly identify ambiguities?
- Did it silently add assumptions?
- Did different groups receive different interpretations?

Part 3: Refine the Specification

Add as few rules as possible to resolve the ambiguities your group found.

You are designing the behavior of WordMatch. There is no single correct answer.

What Did the Activity Reveal?

- The original rules appeared clearer than they were.
- Familiarity caused people to import unstated rules.
- Different people made different reasonable assumptions.
- The LLM also filled gaps instead of consistently requesting clarification.
- Concrete scenarios exposed ambiguity.
- Refinement required human decisions about intended behavior.
- Two different specifications could define two different but reasonable games.

When a specification is vague, humans and AI do not stop: they fill the gaps with assumptions.

NOTES

This is the central takeaway of the whole lecture

Specs Have Been Around for a Long Time

- **Design by Contract:** preconditions, postconditions, and invariants
- **Model-driven development:** models used to generate code structures
- **BDD:** Given/When/Then scenarios describing expected behavior
- **API-first development:** OpenAPI schemas generating documentation, clients, and mocks

The goal remains the same: make intent explicit and use the specification as a shared source of truth.

NOTES

The underlying idea is not new: engineers have long written descriptions that constrain or generate other artifacts. Earlier automated approaches generally required rigid, machine-readable formats and deterministic tools. Generative AI can interpret natural-language specifications and produce much more of the implementation. This reduces the need for rigid syntax but increases the need for human review.

Example: BDD Specification for WordMatch

```
Feature: Evaluate a WordMatch guess

  As a player
  I want each letter in my guess to receive a color
  So that I can determine how closely it matches the secret word

  Rule: Exact matches are assigned first, and each occurrence
        in the secret word can match at most one guessed letter.

  Scenario: Guess contains more occurrences of a letter than the secret
    Given the secret word is "APPLE"
    When the player guesses "PUPPY"
    Then the colors should be:
      | Position | Letter | Color  |
      | 1        | P      | Yellow |
      | 2        | U      | Black  |
      | 3        | P      | Green  |
      | 4        | P      | Black  |
      | 5        | Y      | Black  |
```

This is one possible answer to the **APPLE/PUPPY** scenario from Part 1: it makes two refinement decisions executable: exact matches are assigned first, and each secret letter can only be matched once.

NOTES

BDD scenarios like this one make a specification's decisions testable and unambiguous, but writing one still requires the human decisions this lecture's activity was about. The framework syntax doesn't resolve ambiguity by itself; someone still has to decide the rule.

Example: Design by Contract for WordMatch

```
evaluateGuess(secret, guess) → colors

Preconditions
  - secret and guess each contain exactly five letters.
  - Both words belong to the supplied dictionary.

Postconditions
  - colors contains exactly five values, each GREEN, YELLOW, or BLACK.
  - GREEN: the letter matches the secret at that position.
  - Exact matches are assigned before misplaced matches.
  - Each secret occurrence can match at most one guess occurrence.
  - Remaining letters, left to right: YELLOW if an unmatched
    occurrence remains in the secret, otherwise BLACK.

Invariant
  - GREEN + YELLOW count for a letter never exceeds its count
    in the secret.
```

secret = "APPLE", guess = "PUPPY" → 

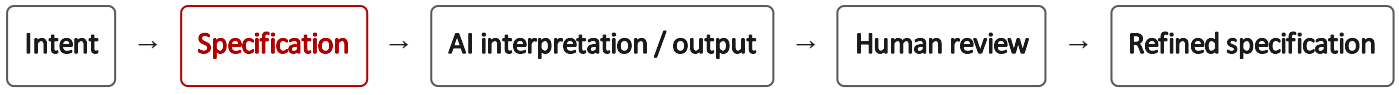
Design by Contract expresses the assumptions that must hold before an operation and the guarantees that must hold afterward.

Unlike the BDD scenario, which illustrates one concrete case, this contract describes constraints that apply to every valid guess.

NOTES

The precondition/postcondition/invariant structure is the same idea as the BDD scenario.

Specifications in AI-Assisted Development



- The specification provides a shared source of truth for humans and AI.
- AI can propose interpretations and identify possible gaps.
- Generated design or code is only one interpretation of the specification.
- Humans remain responsible for deciding the intended behavior.
- When an AI makes the wrong assumption, clarify the specification, not only the generated output.

A prompt asks an agent to do something. A specification states what must be true of the result.

Reference: Martin Fowler, on the evolving meaning of Spec-Driven Development

---

NOTES

The exact meaning of "Spec-Driven Development" (or SDD) is still being actively debated. The point isn't the label; it's the discipline of writing down what must be true before generating code.

From Machine-Parsed to AI-Interpreted Specifications

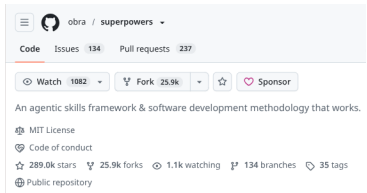
Traditional automated specifications	AI-era specifications
Schemas and domain-specific languages	Structured natural language and Markdown
Strict, machine-readable syntax	Flexible syntax interpreted in context
Deterministic transformation	Probabilistic generation
Generate checks, documentation, clients, mocks, or code skeletons	Generate designs, plans, tasks, and implementations
Syntax errors usually stop the tool	Ambiguity may be resolved silently by the AI

The format became more flexible, but the need for precision did not.

NOTES

This is not a complete replacement of older formats. Modern "Specification-Driven" approaches can combine natural-language specifications with schemas, contracts, and other structured artifacts. Earlier tools usually rejected invalid syntax; an LLM may instead accept an ambiguous specification and choose a plausible interpretation without telling us. AI can generate substantially more of the implementation, but the result remains an interpretation of the specification.

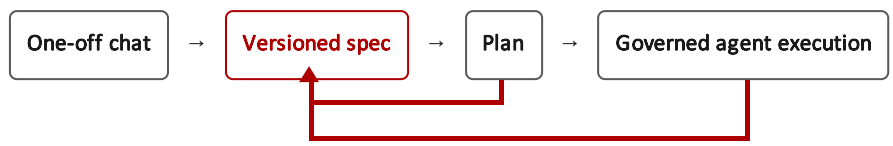
Recent Progress: Specs Become Agent Workflows



- Maintains versioned artifacts: constitution → spec → plan → tasks
- Agent commands per stage: specify → clarify → plan → analyze → implement → converge
- Adds quality gates for ambiguity, consistency, and missing work.

- Refines requirements through questions before implementation.
- Requires human approval of the design and implementation plan.
- Breaks plans into small tasks delegated to fresh subagents.
- Enforces TDD and staged code review during implementation.

Emerging trend: specifications are becoming the control plane for coding agents: persistent artifacts that coordinate planning, implementation, delegation, and review.



↺ Specs also evolve: what's learned during execution feeds back into the versioned spec; it's not a one-shot artifact.

Tradeoff: more upfront artifacts and agent work may reduce ambiguity and context drift, but better outcomes are not automatic.

NOTES

New AI tools are turning specifications into workflows with explicit stages, persistent artifacts, human approval gates, and consistency checks. Spec Kit emphasizes repository-based artifacts and commands. Superpowers emphasizes agent skills and increased human approval during all stages. Some practitioners also stress that the spec itself keeps evolving: execution surfaces gaps that get folded back into the versioned spec, closing the loop rather than treating it as a one-time handoff.

Takeaways

Specifications make intended behavior explicit.

Ambiguity becomes visible when someone tries to use the specification.

AI makes specification skills more important, not less important.