Feedback comment:
Learning experiences that emulate the real world are indeed important for teaching students about the challenges of scaling up people or time. But even if instructors had all the time and resources to essentially recreate a real-world software engineering experience, this would likely not serve our students’ learning goals well. Some degree of curation is necessary to ensure that students can focus on the important learning goals, rather than on issues in real-world software engineering that may spontaneously arise.

How incorporated:
Weakened the strength of the triggering statement: “many topics in software engineering” reduced to “several”. Considered adding additional sentence to add nuance, but the next paragraph is already that.

Why not incorporated:

Date considered: 2022-07-27

Feedback comment:
While the previous paragraphs articulate well why a larger focus on workflow and participation are important, they seem to leave two important questions regarding changes from CS2013. First, why is there a shift away from leadership and project management – wouldn’t this become more important as the number of people scales up? Second, in what ways are the increased focus on team participation, communication, and collaboration different from the actual software workflow where these skills are often used (e.g., code review, design discussions)?

How incorporated:
Added explanation:
The common reasoning behind most of these changes is to focus on material that learners would not pick up elsewhere in the curriculum, and that will be relevant immediately upon graduation, rather than at some future point in their careers.

Why not incorporated:

Date considered: 2022-07-27
It’s unclear what “realities of teams” refers to - as it's written here, it seems to be a softer way of saying “drawbacks”.

**How incorporated:**

Replaced that bullet with two:

- Advantages of teamwork
- Risks and complexity of such collaboration

**Why not incorporated:**

**Date considered:** 2022-07-27

What scale of team does this refer to? The "necessary roles and responsibilities" will depend on the size and scope of the team.

**How incorporated:**

**Why not incorporated:**

“team” is fairly generic in terms of size, but regardless of size/scope the objective is relevant.

**Date considered:** 2022-07-27

Based on how the rest of this item is written, it seems like "team building" means the establishment of team culture rather than the fostering of an atmosphere of trust. If this isn't the case, this phrasing should be clarified to indicate team culture explicitly.

**How incorporated:**

Added “establishing healthy team culture,” to the list to explicitly incorporate both aspects.

**Why not incorporated:**

**Date considered:** 2022-07-27

Feedback comment:
In addition to branching strategies, we should also consider covering history management or merging/rebasing strategies in this area.

**How incorporated:**

Added bullet, “Merging/rebasing strategies, when relevant.”

**Why not incorporated:**

**Date considered:** 2022-07-27

Feedback comment:
Debugging doesn't seem like a modern IDE facility, unless the intent here is to specifically point out the integration of debugging tools into modern IDEs.

**How incorporated:**

**Why not incorporated:**
This is indeed about debugging tools, in parallel to refactoring, searching, etc. tools.

**Date considered:** 2022-07-27

Feedback comment:
As worded, it's unclear whether this outcome means "given the task of designing a simple software system, select and use a design paradigm for the design of said system" or "given the task of designing a simple software system and a specific design paradigm, apply the given paradigm in designing said system".

**How incorporated:**
Revised sentence, “Select and use an appropriate design paradigm to design a simple software system and explain how system design principles have been applied in this design.”

**Why not incorporated:**

**Date considered:** 2022-07-27

Feedback comment:
This item is also listed in SE/Tools and Environments, and seems to be better suited for there rather than here.

**How incorporated:**

**Why not incorporated:**
It’s relevant in both places.

**Date considered:** 2022-07-27

---

**Feedback comment:**
As written, this disposition could be read as overlapping somewhat with being Collaborative. Framing this explicitly as respectful day-to-day behavior outside of one’s team (e.g., in one's company, in public, on social media) would make clearer the connection to professionalism.

**How incorporated:**
Expanded with “and commitment to respectful day-to-day behavior outside of one’s team”

**Why not incorporated:**

**Date considered:** 2022-07-27

---

**Feedback comment:**
Software engineering is distinct in two dimensions: team size, and time (or however you characterize those two dimensions)

**How incorporated:**
Added a closing, “: time and people.”

**Why not incorporated:**

**Date considered:** 2022-07-27

---

**Feedback comment:**
presented in the abstract but with grounding in real examples. There’s no reason to pin people to a sense that a lecture must be thoroughly theoretical.
Impressing upon students the risks of failure in terms they can understand is also an option. For instance, getting students to work together to experience the hell of merge conflicts (or not quite merge conflicts — just divergent changes in a unit) to motivate the need for refactoring and principled design. Then when teaching design principles, harken back to that experience so they can relate to the need for principles.

Step 1: PRODUCTIVE FAILURE. get students to do a project where (due to inexperience) they fail to modularise properly and have huge issues because of it (bugs are introduced, headaches adding new functionality).

Step 2: teach them design principles/refactoring/etc motivating all the topics from within the experience you know they had.

How incorporated:

Why not incorporated:

Valid points, but better suited to the companion volume / suggested lesson plans / good exercise material.

Date considered: 2022-07-27

Feedback comment:

These are good shifts.
N/A, but the feedback is appreciated.

Why not incorporated:

Date considered: 2022-07-27

Feedback comment:
Extra line

How incorporated:
Removed extraneous line.

Why not incorporated:

Date considered: 2022-07-27

Feedback comment:
This parenthetical stands out because it's the only one of its kind. Is it possible to re-word this sentence so you don't need the parenthetical?

How incorporated:
Revised this phrase to, “especially focusing on the value of disagreeing about ideas or proposals without insulting people.”

Why not incorporated:

Date considered: 2022-07-27

Feedback comment:
Requirements have changed significantly over time because of the forces on them (technological concerns such as time to compile and build, the provision of IDE-based refactoring support, the ever increasing speed of release opportunities — from building a machine in situ, to sending large disks, sending smaller disks, sending CDs, something downloadable, and then finally web applications that can be released daily). This context is vitally important for students understanding the present state of requirements, and even projecting into the future of requirements. Students shouldn't be given the impression that the current state is THE WAY things are done, because it promotes a fixed mindset, and
doesn’t provide them with the agility to change in the face of new opportunities. Threading historical context throughout the entire SE curriculum is very important, but especially so with respect to software requirements.

How incorporated:

Why not incorporated:

We agree with the thrust of the comment, but don’t believe edits are needed. This is (lightly) covered in “Product evolution” already. More broadly, SEP has a section on the importance of computing history.

Date considered: 2022-07-27

Feedback comment:

Strong preference here for user stories - even large development shops are moving to that model.

Also they have a strong set of design principles (INVEST) which is important to understand for new developers.

Teaching “Estimation” can be reduced to teaching a matrix of “locations of the change” and “complexity of the change”. So basically four estimates — easy single location, complex single location, easy multiple location, complex multiple location. And then associate likely durations with each of those, and perhaps moving “complex multiple location” to its own spike or research ticket.

How incorporated:

Why not incorporated:

Broadly agreed, but the details are best suited to companion volume material / best practices in presenting the material identified.

Date considered: 2022-07-27

Feedback comment:

I might suggest putting code smells here, or “flaws” here, so that you can juxtapose flawed code against principled code. Additionally integrating the teaching of refactoring into the teaching of design principles is perhaps a better approach, since developers identify principle violations as code smells or flaws, then apply refactorings to achieve principled code. At the very least, refactoring should be taught prior to design.
How incorporated:
Added a bullet on “Code Smells” to the CS Core material.

Why not incorporated:

Date considered: 2022-07-27

Feedback comment:
Differentiate between tests of the specification (black box) and tests of the implementation (glass box).

How incorporated:
Already implied in the section on Test Doubles (stubs, mocks, fakes). Expanded the text there to include “Especially consider the appropriate use of these techniques for testing error handling, and the awareness of how these can become brittle verification of an implementation rather than a test of the contract.”

Why not incorporated:

Date considered: 2022-07-27

Feedback comment:
Comments likely shouldn’t be used at all except to state specification of methods. I disagree with comments to provide comments to clarify tricky pieces of code. Instead, that code should be refactored into well named methods that clarify its purpose. TODO comments are fine.

How incorporated:
Rephrased, “Identify subtle/tricky pieces of code and refactor to be self-explanatory if possible, or provide appropriate comments to clarify.”

Why not incorporated:

Date considered: 2022-07-27

Feedback comment:
Again, fundamentally disagree with the use of implementation comments. At least they should only be taught with big warnings around them and the recognition that the state of the practice considers them a code smell.
How incorporated:

Weakened the suggestion, “Write appropriate interface and (if needed) implementation comments for a small component.”

We’re somewhat concerned with throwing away the notion of implementation comments entirely. Although we strongly agree that it’s rarely essential, there is some amount of exposure likely needed before learners can decide when this is the right tool vs. refactoring the code directly.

Why not incorporated:

Date considered: 2022-07-27

Feedback comment:

In TDD, black box tests are written prior to code. So bringing up code coverage as a measure of testing, and conflating it with test planning and generation, is confusing.

First step: Write black box tests against the specifications. Discuss generation of black box test (boundaries, equivalence classes, formal models, etc).

Second: write the code, and re-run the tests, checking for coverage. Write new IMPLEMENTATION specific tests (glass box tests) to improve coverage. Also perform manual mutations to check test suite health, and write tests that don't catch mutants.

My new suggested hierarchy:

TEST DRIVEN DEVELOPMENT:

1. BEFORE YOU HAVE CODE:
   - test generation from formal methods and specification.
   - equivalence classes, boundary conditions, etc.

2. AFTER YOU HAVE CODE
   - test matrices,
   - code coverage
- mocking, OO testing, dependency injection
- test tooling, analysis techniques
- etc

3. VERIFICATION AND VALIDATION... (continue on)

**How incorporated:**

**Why not incorporated:**

We are not currently suggesting that TDD is presented as fait accompli, although we’d broadly like to see both education and industry move in that direction. The topic breakdown as presented is not a progression as much as a set of topics that deserve to be discussed individually: how do we plan a strategy for testing? How do we generate tests? When do we write tests? etc.

The topic progression suggested by the reviewer seems broadly healthy, but is likely better presented in the companion volume.

**Date considered: 2022-07-27**

Feedback comment:

Missing item: understanding the difference between pre-code (specification only, black box) tests and with-code tests (implementation specific, glass box).

**How incorporated:**

Expanded the description of mocking and dependency injection (the most “glass box” forms of testing) with “and the potential for glass-box brittleness when implementation / mockist tests are overused.”

**Why not incorporated:**

**Date considered: 2022-07-27**

Feedback comment:

This topic should appear BEFORE principles, since understanding code flaws is prerequisite to understanding design principles.

**How incorporated:**
Why not incorporated:
There is not a suggested nor required ordering of these requirements.

Date considered: 2022-07-27

Feedback comment:
Include switch on type code smell fix, divergent changes, shotgun surgery fixes.

How incorporated:

Why not incorporated:
The specific fixes for specific problems are not covered here. General fixes (refactoring patterns) and general problems (static analysis surfacing code smells) seem to cover both sides of the problem. Those are both already surfaced.

Date considered: 2022-07-27