

Lab 3 Debrief

- practice core processes enough so that you can do them efficiently and correctly
 - setup: creating a new project (in the right place), importing files, creating a git repository (in the right place), .gitignore, initial commit
 - routine: the commit process (stage first!)

Pair Programming – Common Sticking Points

- different cognitive styles
 - some people think best out loud in real time, others need to work through an idea on their own first
 - fix: make space for it – “give me a minute to think” is a normal, nameable request that either party can make
- navigator solves the problem on their own, driver is just a typist
 - fix: navigator speaks up before the solution is complete – identify the problem and invite the driver back in rather than figuring it all out first
 - e.g. “I think we need to handle X, but I’m not sure how” or “I think we’re comparing the wrong two cards” rather than “change line 3 to i+1”
- skill-gap micro-management
 - navigator gives a higher-level direction but driver doesn’t know how to translate it into code
 - fix: driver asks a clarifying question instead of waiting for the navigator to get more specific
 - shift to knowledge-transfer pairing if this is more than a specific point
- key fix – avoid either party becoming only a passive participant

Swarming – Common Sticking Points

- no real weighing of tradeoffs
 - a design is adopted because it was proposed first or confidently, not because alternatives were considered
 - fix: before finalizing a decision, ask “what’s a different way we could do this, and why is our choice better?”
- a proposal never acknowledged – or never heard
 - fast-moving discussion can sweep past an idea without anyone consciously overlooking it
 - fix: check that every idea actually got heard – “hold on, did we catch what Alice said a minute ago?” – and make sure that an outcome – adopt, reject, or incorporate – is decided for each idea
- quiet members left out
 - silence can mean social hesitance, needing more time to think, lacking the background to contribute, coasting, ...
 - a fast-moving group may finish before a careful thinker has time to form a view
 - fix: include brief silent-write steps before discussion starts; when discussion is wrapping up, directly ask someone who hasn’t spoken for their thoughts
- key fix – build in moments to slow down

Effort Estimation

- uneven contribution is best prevented, not just resolved after the fact

Why estimate effort –

- helps split work fairly between teammates
- early flag if a task is bigger than expected
 - avoids discovery two days before the deadline
- absolute time estimates (hours) are notoriously unreliable
 - everyone underestimates – *planning fallacy*
 - absolute estimates require correctly predicting unknowns – “will I run into an unexpected problem?”

Effort Estimation

Relative estimation –

- compare tasks to each other – is this task bigger or smaller than another?
 - use T-shirt sizes (S/M/L)
- easier to be roughly right about relative sizes than absolute time

Planning poker –

- everyone estimates a task privately at the same time
- reveal simultaneously
- if estimates differ a lot, discuss why
 - the discussion often reveals a hidden assumption or misunderstanding – the most valuable part

Exercise

You are adding to the Klondike game –

- implement Card's `isOppositeColor(Card other)`
- implement TableauColumn's `canPlace(Card card)`
- implement a method that prints the entire board state (stock, waste, all tableau columns, all foundation piles) [text, to the console]
- implement GameEngine's `checkWinCondition()`
 - all foundation piles are complete
- write a JUnit test suite covering `canPlace`'s normal, edge, and error cases

- (individually) assign each task a T-shirt size: S, M, or L
- (at once) all reveal (show of hands)
- (whole class) discuss tasks with a spread – why? what were you picturing when you sized this?

Discussion

A	a single comparison, low ambiguity as to scope
B	considering just rank/color check, or also thinking about the empty-column case?
C	a scope question rather than difficulty – how polished should the formatting be? are there already <code>toString()</code> methods or do they need to be written?
D	sounds larger (big important task) but actually only involves reading/checking state (and those methods should already exist)
E	considering just the normal paths or also edge and error cases? “write tests for X” is not well-scoped on its own – depends on what is considered adequate coverage

- common reasons for different estimates
 - scope ambiguity – differences as to what counts as “done”
 - infrastructure assumption gap – overlooked whether supporting groundwork already exists or needs to be built
- revealing these differences / gaps early is the most valuable part of the estimation exercises

Meeting Structure

Meetings need structure –

- have an agenda rather than an open-ended conversation
 - “so, how’s it going?” tends to run long without a clear outcome and/or stay too vague to turn up anything actionable
- lightweight “standup” format
 - cover: what I did, what I will do next, what’s blocking me
 - the goal is a fast check in to turn up problems rather than a status report to justify your time

Keeping Track of Work

- shared task list/board beats “I thought you were doing that”
 - e.g. shared document with a simple checklist
 - key is the habit of writing it down where the whole team can see

Documenting Decisions

- write down what you agreed on, especially design decisions
 - memory of verbal agreements fades and diverges

Collaborating Well

- resolution process
 - name the issue concretely and early
 - state your interest, not just your position
 - listen for their underlying interest too
 - look for an option addressing both
 - if stuck, bring in a third party
- what not to do
 - silently resent a partner instead of raising the issue
 - escalate to accusation instead of describing behavior
 - wait until the deadline to bring up a problem that’s been building for a week
 - choose avoiding or accommodating out of habit, not because it actually fits

Example

- scenario: partway through a project, Alice notices Bob hasn't committed anything on his part of the task

step	what it looks like	what it doesn't look like
1 – name the issue concretely and early	Alice: "I checked the repository this morning and I don't see any commits on the UI or tests yet."	"Why haven't you done anything?" – an accusation instead of an observable fact, sets up for a defensive response
2 – state your interest, not your position	Alice: "I'm worried we won't have time to integrate and test everything before the deadline."	"You need to commit code today," – a demand, which may not even solve the real problem
3 – listen for their underlying interest	Bob: "I got stuck on how to handle invalid input without the whole thing crashing and I didn't want to bring it up."	Alice keeps talking instead of finding out that Bob is stuck rather than lazy or careless
4 – look for an option addressing both	Alice: "Want to pair on the input-validation part right now? I can drive if that helps, and you can pick the UI back up once that's done."	Alice insists Bob just figure it out alone – solves nothing since the actual problem is not addressed
5 – escalate if stuck	if that doesn't resolve it or it keeps happening, bring it to office hours – not something to just let go	having the same conversation again next week, hoping it works better this time

Working With Existing Code

- most software development time is reading and modifying existing code rather than starting from scratch
- the central question: how to understand a codebase?
- the trap – reading code like it's a novel
 - don't start at line 1 of the first file and read straight through
 - unfamiliar code is understood by *finding structure* first

Avoiding the Trap

- start from an entry point
 - find where execution – of the whole program, of a particular operation – begins or the top-level class orchestrating everything and go from there
- the forest, not the trees – at first
 - ask what does this program/method/chunk of code *do* – at the highest level – before getting into the *how*

Exercise

```
public boolean chk(Card c, Column col) {
    if (col.sz() == 0) {
        if (c.getRank() == Rank.KING) {
            return true;
        }
        return false;
    }
    Card t = col.top();
    if (c.getRank().getValue() == t.getRank().getValue() - 1) {
        return true;
    }
    return false;
}
```

git history: a single commit with the message "stuff"

- what does this method do?
- is there a bug? if so, describe it
 - doesn't check the color part of the rule, only rank
- how long did it take to understand this method? what helped (or hindered)?

```

/**
 * Removes and returns the top n cards of this column, in
 * bottom-to-top order. If a card remains after the removal and it
 * is face-down, it is turned face-up automatically.
 *
 * @param n how many cards to remove
 * @return the removed cards, bottom-to-top
 */
public List<Card> pickUp(int n) {
    List<Card> removed =
        new ArrayList<>(cards.subList(cards.size() - n, cards.size()));
    for (int i = 0; i < n; i++) {
        cards.remove(cards.size() - 1);
    }
    return removed;
}

```

```

@Test
void pickUpFlipsNewlyExposedCard() {
    TableauColumn column = new TableauColumn();
    column.place(faceDown(Rank.NINE, Suit.CLUBS));
    column.place(faceUp(Rank.KING, Suit.HEARTS));

    column.pickUp(1);

    assertTrue(column.top().isFaceUp());
}

```

Git history (in order):

- "Add TableauColumn: isEmpty, place, top"
- "Add pickUp(n): remove top n cards"
- "Add Javadoc and test for pickUp's flip-on-expose contract"

- what does this method do?
- is there a bug? if so, describe it
 - doesn't flip the new top card face up
- how long did it take to understand this method? what helped (or hindered)?

62

Observations

- key to working with existing code is understanding what the code is supposed to do
- this is a whole lot harder if you have to reconstruct the entire intended behavior of the method purely from the code itself

CPSC 329: Software Development • Fall 2026

64

Avoiding the Trap

- start from an entry point
 - find where execution – of the whole program, of a particular operation – begins or the top-level class orchestrating everything and go from there
- the forest, not the trees – at first
 - ask what does this program/method/chunk of code *do* – at the highest level – before getting into the *how*
- use whatever trail the author left
 - the commit history tells how the program was built
 - often mirrors dependency order – foundations before task logic
 - documentation in the code and test names describe intended behavior
 - test names can provide faster insight than the test body or program code itself
 - design documents, test plan are additional resources
 - can provide additional rationale and insight about decisions and intentions

CPSC 329: Software Development • Fall 2026

65

Habits to Develop

- run it before you read it
 - reading the code becomes "ah, this is what does that" rather than "what does this do?!"
- change the smallest thing first
 - make one small low-risk change and confirm that you understand what happened before building a big new feature
 - a tiny, deliberate experiment is safer than a big, confident guess
- ask, don't guess
 - if something is unclear, ask the code's owner
 - building on top of a wrong guess only makes the mistake more expensive to fix

CPSC 329: Software Development • Fall 2026

66