Managing the Risks of Teamwork

- teams have real advantages, but each advantage comes with a matching risk
- coordination overhead – reduce
  - cleanly split tasks along subsystem owner boundaries
  - ⇨ cooperative programming techniques when close collaboration is needed
- uneven contribution – prevent
  - ⇨ effort estimation
- communication overhead / misunderstandings – reduce
  - sync vs async – choose the right mode for the task, clear and specific messages
  - ⇨ code review – structured, routine communication that catches issues and spreads knowledge of the codebase
  - ⇨ documenting decisions – unwritten agreements drift in memory, even honestly
- groupthink – prevent
  - actively ask “does anyone see this differently?” rather than assuming silence means agreement

Managing the Risks of Teamwork

- conflict resolution – when prevention doesn’t work
  - understanding the causes of conflict
  - recognizing the conflict mode best suited to the problem
  - ⇨ collaborating well
- process infrastructure
  - ⇨ standups – find sticking points early
  - ⇨ shared task list – make ownership explicit rather than assumed

Cooperative Programming

Two ways to work together on code –

- *pair programming* – two people, one task, one keyboard
- *swarming* – the whole team temporarily comes together to works on one hard problem

Pair Programming Roles

- *driver* – at the keyboard – tactical
  - focused on the immediate line of code – ignores larger issues for the moment
  - talks through what they are doing, out loud, as they do it
- *navigator* – active observing – strategic
  - a second brain engaged with the problem ☆
  - reviews code on the go, watches the bigger picture, catches errors, thinks a step ahead
  - gives directions and shares thoughts – but don’t interrupt
  - asks “why?” when the driver’s approach isn’t obvious
- switch roles regularly (use a timer!)
  - to help prevent the driver from taking over and the navigator becoming a passive observer
  - to foster shared ownership, including both partners understanding the result

Example

- scenario: implementing `TableauColumn's canPlace(Card)`

step	what happens	who decides
1 – pick the goal	agree on one small, well-defined target – "make the `canPlace` tests pass" (test-early: tests are written but `canPlace` not yet implemented)	both, together, before typing starts
2 - drive	Alice types the method skeleton, narrating – "I'll check the empty-column case first..."	driver – moment-to-moment code
3 - navigate	Bob – "once we're past the empty check, we'll need the rank comparison, and don't forget the color check too"	navigator – direction, not syntax
4 – catch and pause	Alice implements the rank check, and completes the return in both cases Bob notices that the color check didn't get written, waits briefly Alice catches the problem herself while re-reading and adds the color check	5-second rule in action
5 – small goal done	all `canPlace` tests pass switch: Bob now drives the next goal	both – switch is scheduled, not arbitrary

Effective Pair Programming

Keeping pairing effective –

- give direction, not dictation – leave room for the driver to think
  - navigator guides towards the goal ("we'll need to handle the empty case next") rather than specifying exactly what to do ("now type System, dot, print, …", "now we need to create a new class called…", "press command shift O…")
- pause before pointing something out – 5-second rule
  - the driver may already see it, and a moment's patience avoids an unnecessary interruption
- share the keyboard
  - switch regularly so both people stay actively engaged
- stay present
  - set phones and email aside – a short, focused session is better than a long half-attentive one

Swarming

- the whole team works on one problem together
  - no fixed roles

Running swarming well –

- everyone can propose, everyone can push back
  - no automatic deference to whoever spoke first
  - don't assume silence means agreement
- state the actual question before diving into solutions
  - make sure everyone knows and agrees on what's being decided
- write down the resolution before ending
  - a decision that only lives in the room's memory will drift
- decide on a (rough) time limit at the start
  - swarming is expensive – if it is dragging past its purpose, stop and take steps to resolve the conflict

Cooperative Programming

When to use what –

- *pair programming* – when a mistakes would be costly or for knowledge transfer
  - simple-looking task + costly mistake – second set of eyes catches autopilot slips
  - complex task + costly mistake – benefits from two people's combined reasoning
  - knowledge transfer – one partner is less familiar with the area
    - they drive, the more experienced partner navigates – so the novice is an active participant, not passively watching the expert
- *swarming* – for a hard design decision, or something with a high risk of getting it wrong
  - typically design decisions rather than implementation – design becomes a contract that other code will be built against
- *splitting solo* – independent, well-specified subsystems where coordination cost would exceed the benefit
  - should be the norm – cooperative programming is expensive

The Cost of Mistakes

- how far would a mistake propagate?
  - does other code call on this, build on it, or assume its correctness?
  - e.g. errors in an invariant-maintaining helper or basic operations of a core data structure are costly
- how visible would the mistake be if it happened?
  - an immediate crash or obviously wrong on-screen result is easy to catch
  - subtlety wrong output can go on for a long time before being noticed
- how hard would it be to track down once discovered?
  - a bug which doesn't show up until many lines of code later – and in a different method – is hard to track down
- what does it touch that is expensive to undo?
  - corrupted persistent state or decisions baked into the structure is costly to undo once more has been built on top of the mistake
- is it deceptively simple?
  - autopilot trap – something can seem trivial but actually involves a complex condition that is tricky to get right

Code Review

What a code review is for –

- catch bugs before they ship
  - code review is the communication mechanism supporting the "built-in error catching" advantage of teams – structured and routine
- spread knowledge of the codebase across the team
  - nobody is the only one who understands a piece
  - ownership doesn't mean others are locked out
  - knowledge of the system is important for change-impact analysis
- enforce shared standards
- support mentoring in both directions
  - peers review each other
  - less experienced can catch things more experienced missed
    - especially "fresh eyes issues" such as unclear naming or untested edge case

Code Review

When it happens –

- on branch/pull request, before merging into the shared codebase
- ideally frequent small reviews throughout, not a big one-time gatekeeping at the end
  - faster to do well
  - catch problems while they are still cheap to fix
  - avoids time bottlenecks during crunch time

Code Review

What to look for –

- *correctness* – does the code actually do what it is supposed to?
- *standards* – naming, style, documentation present?
- *tests* – is new behavior covered?
- *readability* – could someone else (or you, in a few months) understand this?

Giving Feedback Well

Giving –

- *specific* – point to the exact line/issue, not a vague impression
- *actionable* – say what should change, not just what's wrong
  - bad: "this is confusing"
  - good: "on line 42, *n* isn't very descriptive – could this be *cardsToRemove* instead? also, I don't see a check for *n* being larger than the list size – is that handled somewhere else, or should we add it here?"
- *kind* – critique the code, not the person
  - "this method is confusing" vs "you wrote this confusingly"
- *separate* "must fix" from "consider" – not everything is equally important
  - a long list can feel overwhelming; an explicit labeling is more actionable

Receiving Feedback Well

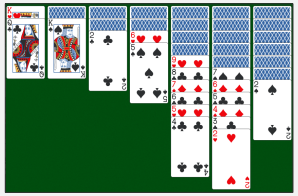
Receiving –

- don't take it personally
  - a review comment isn't an attack
  - the point is catching issues before they matter
- respond to each point
  - agree and fix or explain your reasoning – don't silently ignore
    - "good catch, fixed"
    - "I think this is fine because X, let me know if you disagree"
  - review should be genuine dialogue, not one-way inspection

Exercise

```java
/**
 * Checks if you can put a card down
 */
public boolean canPlace(Card c, TableauColumn col) {
    if (col.size() == 0) {
        if (c.getRank() == Rank.KING) {
            return true;
        } else {
            return false;
        }
    }
    Card t = col.top();
    if (c.getRank().getValue() == t.getRank().getValue() + 1) {
        if (c.getSuit().getColor() != t.getSuit().getColor()) {
            return true;
        }
    }
    return false;
}
```

tableau columns are built with alternating colors and decreasing rank



- (individually) annotate your printout
  - for each issue, also mark which review category (correctness, standards, tests, readability) and note whether it is must-fix or consider
- (pairs) compare annotations
  - pick your most important finding and share with your partner, phrasing it as a review comment (specific, actionable, kind)
  - compare the rest – did you both catch the same things?

Solutions

issue	category	severity	model review comment
the rank comparison is backwards – checks whether card being placed is one rank higher rather than one rank lower	correctness	must fix	"On this line, I think the comparison is inverted – shouldn't the placed card need to be lower than the top card rather than higher? As written, this would accept putting a king on a queen rather than the reverse."
non-descriptive parameter and variable names (`c`, `col`, `t`)	standards	consider (worth fixing but not outright broken)	"Could we rename `c`/`col`/`t` to something like `card`/`column`/`topCard`? It'd make this a lot easier to read at a glance."
no tests accompany this method	tests	must fix	"This has at least three distinct branches (empty column requires a king, correct rank, correct color) – could we get a few tests covering each, especially since this is exactly the kind of logic that's easy to get subtly wrong?"
javadoc is vague and doesn't describe the actual contract	readability / standards	consider (worth fixing but not outright broken)	"Could the doc comment spell out the actual rule – empty column needs a king, otherwise one rank lower and opposite color – and use `@param`/`@return`? Right now it doesn't tell a reader what 'can place' actually means."