Agility

UNC COMP 523

October 5, 2020

Jeff Terrell

Announcements

- music: *Paranoid Android* by Radiohead, who exhibit great musical agility IMO
- please join our class on OpenClass.ai *before Q4 on Wednesday*—see Piazza for a link
- Q1 correction forthcoming
- pivoted from talking about testing—agility should come first

Assignment update

- A8: Walking Skeleton ☕☕☕☕☕ due next week!
- Also, you'll be presenting A9: Midterm Presentation ☕☕ in class next week.
- Preferences for A10: Tech Talk ☕☕☕☕ due week after next.
- Tech talk dates and topics are first-come, first-served, and you're welcome to submit preferences early.

Outline

- Interpersonal effectiveness, continued
- Why agility?
- Introducing extreme programming (XP)
- XP practices

Outline

- **Interpersonal effectiveness: relationship skills (GIVE) and self-respect skills (FAST)**
- Why agility?
- Introducing extreme programming (XP)
- XP practices

Relationship skills: GIVE

- Typically this goal is not at 0—do you truly not care whether a person likes you or hates you after the interaction is over?
- If relationship is always your primary goal, you might burn out in the long term
- G: (be) Gentle: no attacks, no threats, no judging, no disrespect
- I: (act) Interested: listen, focus, eye contact, lean in, no interruptions
- V: Validate: using words and actions, show that you understand the other person's feelings and thoughts
- E: (use an) Easy manner: smile, use humor, be light-hearted

Validation

- There are reasons behind a person's desires and preferences; showing that you understand matters
- You can validate underlying reasons even without validating particular behaviors
- Story: Johnny, trying to pay attention in class
- "It makes sense that you..."
- "I'd want the same thing in your position."
- What kind of counselor would you prefer: a validating one or a judgmental one?
- Interpersonally, validation can unlock deeper insight, appreciation, and trust
- Inwardly, validation can have the same effect

Levels of validation

1. pay attention
2. reflect back
3. "read minds"
4. understand (how does their position make sense?)
5. acknowledge the valid
6. show equality: treat the other person as an equal, not greater or lesser than you, not fragile, not incompetent

Self-respect skills: FAST

- F: (be) Fair, to yourself and the other person; validate your own feelings and wishes too
- A: (no) Apologies: don't apologize for what you want
- S: Stick to values: don't compromise on your own values (unless it's very important)
- T: (be) Truthful: don't lie or exaggerate. Don't act helpless when you're not.

Summary of Interpersonal Effectiveness

- You'll be dealing with people, so it's worth learning how
- The dialectic involves two sides working together to find the truth through dialogue
- The DBT world view: everything is connected; reality consists of internal opposing forces; and the nature of reality is change
- Core practices of DBT: mindfulness and validation
- 3 goals of interpersonal effectiveness: objective, relationship, and self-respect
- Use DEAR MAN skills for objectives goals/assertiveness
- Use GIVE skills for relational goals
- Use FAST skills for self-respect goals

Outline

- ~~Interpersonal effectiveness: relationship skills (GIVE) and self-respect skills (FAST)~~
- **Why agility?**
- Introducing extreme programming (XP)
- XP practices

Think about it

Read the following story:

> TLA Corp. engages Acme Software to build an app. Acme reads the project description and quotes $250k over 3 months. They go build it, and deliver it just in time. TLA reviews it and says it's not what they asked for. TLA has people waiting on the software, and Acme has another project lined up that they need to start on. How can they resolve this disagreement?

What went wrong?

What went wrong?

- Reliance on a written document over in-person,collaborative explorations
- Predicting relatively far out into the future, and committing to that
- No opportunity for iterative feedback from the client!
- A locked-in timeframe and budget doesn’t mix with changing requirements

Why do it this way?

Let's approach the failed project with validation, not judgment.

- TLA Corp. needs to decide whether to build this software.
- TLA Corp. needs to decide whether to engage Acme Software to build this software.
- Acme Software needs to decide whether to engage this project.
- TLA Corp. needs to line up marketing releases and sales efforts for when the software is finished.
- Acme Software needs to keep developers busy and tell other clients when they can start work on their project.
- Insight: *A fixed budget and timeframe are key to inform these decisions and plans.*

The problem

- I've never met a client who can imagine in full detail what they want.
- Even if they could, how can they (efficiently) get the details of that vision into my head?
- Even if they did, how can I have confidence about my estimates?
- Predicting the future with any accuracy is hard!
- (Arguably harder in our field given the pace of change.)
- With everything nailed down, *change is the enemy*.
- In practice, you hedge your bets: inflate your estimates to allow for something to go wrong.
- A fixed term and budget typically means that one side or the other loses.

Introducing agility

- In general, *agility* is useful in a changing environment
- Key idea: **expect change, and handle it gracefully**
- Deemphasize planning and predicting the future
- Instead, emphasize **iteration**, taking things as they come (not before), and responding to new information smoothly

The solution

- TLA Corp. sees the work in progress and realizes that it's not what they want.
- Early feedback means things can change when they're fresh: the details are (more) in-mind for developers, and other code hasn't gotten piled on top yet.
- Acme Software is paid by the developer hour, so doesn't have an incentive to resist change.
- TLA Corp. can see the work as it unfolds, so has confidence that Acme Software is working at a good pace.
- With communication and iteration comes **trust**, a *very* valuable resource when navigating conflicts (and avoiding them altogether).

Outline

- ~~Interpersonal effectiveness: relationship skills (GIVE) and self-respect skills (FAST)~~
- ~~Why agility?~~
- **Introducing extreme programming (XP)**
- XP practices

Extreme programming (XP)

- We've seen why agility matters.
- Extreme programming (XP) details *how* to do agility.
- Comes from Kent Beck in the late 90's; see Wikipedia article
- Goal: produce higher-quality software faster
- 4 activities
- 5 values
- 12 practices

XP activities

- coding
- testing (this is central, which is part of why XP was considered extreme)
- listening (to client)
- designing (esp. when things get complex)

XP values

- communication
- simplicity
- feedback
- courage
- respect

XP value #1: communication

- general need for software projects: communicating system requirements to developers
- classically accomplished by documentation
- documentation is useful because written words don't change like memories do
- but documentation is (or tends to be) *static*
- things change (see previous section), and documentation makes responding to change harder
- XP techniques build and share institutional knowledge among dev team
- goal: give all devs a shared view of the system that matches the view held by users
- consequently, XP favors simple designs, common metaphors, collaboration between users and programmers, frequent verbal communication, and feedback

Client/developer communication

- developer's goal: try to see the world through the client's eyes and understand how they think
- client brings the *what*: idea, vision, goals
- developer brings the *how*: software development skills
- collaboration is essential
- like using your phone to navigate through an unknown city
- as developers, your job is to be the *executive* of the client's will: make it so

XP value #2: simplicity

- only code for the needs of *today*, not tomorrow
- helps maintain focus, because you limit what's currently in view
- don't overengineer your solution: YAGNI
- simple code and designs aid communication because they're more easily understood
- "But what if I know I'll need something tomorrow, and I'll need to undo this code then?" Be reasonable, but might still be worth waiting on it.

XP value #3: feedback

- get feedback from the system (i.e. your running code) by running tests
- get feedback from the client every 2-3 weeks when showing them the system
- get feedback from the team during retrospectives
- example: a feature is 80% done. Client can decide whether remaining 20% is higher priority than the next feature.
- iteration means less time planning, more time doing
- "Optimism is an occupational hazard of programming. Feedback is the treatment." - Kent Beck

XP value #4: courage

- trust that you'll be able to find out what you need to know tomorrow, so don't worry about it now; stay in the present
- be OK making mistakes (even in front of others; see pair programming practice below), because you can fix them later
- be bold making changes, because tests should tell you if you broke something else
- be OK with messy code (initially); tests also give confidence when cleaning it up (but clean it up; see simplicity above)
- throw away old code! you can find it later with git if you need to

XP value #5: respect

- respect the time of teammates: don't commit broken code
- respect your own work: take the time to polish the structure of what you wrote
- other values tend to reinforce respect, which increases developer satisfaction and loyalty

Poll

Outline

- ~~Interpersonal effectiveness: relationship skills (GIVE) and self-respect skills (FAST)~~
- ~~Why agility?~~
- ~~Introducing extreme programming (XP)~~
- **XP practices**

XP practices

Fine-scale feedback

- Pair programming
- Planning game
- Test-driven development
- Whole team

Continuous process

- Continuous integration
- Refactoring
- Small releases

Shared understanding

- Coding standards
- Collective code ownership
- Simple design
- System metaphor

Programmer welfare

- Sustainable pace

XP practice #1: Pair programming

- What is it? Two people working as a single unit to write code
- One computer (screen, keyboard, mouse), but two people
- Two roles: "driver" uses the computer and focuses on details
- "Navigator" focuses on the big picture and reviewing the code being written
- Roles should switch frequently (ideally, pairs do too)
- Very efficient way to share knowledge! (Remember value #1: communication)

Pair programming efficiency

But isn't that wasteful? Actually, no.

- you maintain higher focus, so get done sooner
- (be honest: how focused are you when working on your project? Might you be less distracted when pair programming?)
- the code quality is higher, so less time spent chasing bugs
- help debugging, and diversity of experience to bring to a task
- studies tend to show that PP is at most 25% slower, but often on pair with or even faster than solo programming

Pair programming difficulties

- verbalizing your thought processes
- exhausting your verbal circuits
- coordinating working times
- showing somebody your deficiencies (emotional resistance)

Pair programming benefits

- each dev knows more about the total codebase, and fewer areas of the codebase have a "truck number" of 1
- research shows that most people end up really enjoying this
- you learn and grow faster as a developer (and as a teacher/communicator)
- you have a way to learn ancillary things, e.g. tools and tricks with git, your editor, or the CLI

XP practice #2: Planning game

- there's a lot to this that I don't want to spend time on
- essentially, client brings desiderata and can sort things by business value
- dev team brings know-how and can sort things (approximately) by implementation time
- together, decide what priorities are for next iteration
- as iteration unfolds, can "steer" things as surprises arise
- after the iteration, review how the last iteration went, and do it again for the next iteration

XP practice #3: Test-driven development

- recall that testing is a core activity of XP, as important as coding
- tests are an important source of feedback for developers: is the code correct?
- test-driven development (TDD) says to write the tests *first*, before the code
- 3 phases: "red, green, refactor": i.e. write failing test, make it pass, then clean up your code
- with a fast feedback loop, this is fun! Helps establish a creative "flow state".

XP practice #4: Whole team

- this basically means that the client should be available to answer questions as though they were on the dev team
- in other words, the client doesn't just pay the bill

XP practice #5: Continuous integration

- principle: the development team should always be working on the latest version of the software
- why? Reduce the time gap, and notice merge conflicts or test failures while the both sides of the work are fresh on devs' minds.
- with trunk-based development, as devs merge new commits into the trunk, feature branches should be rebased
- rule of thumb: try to rebase your feature branch atop the latest master every day before you start work

XP practice #6: Refactoring

- remember simplicity and YAGNI?
- a consequence is that you must re-design an existing approach in light of new requirements
- this is called refactoring: changing the implementation of an approach without changing its behaviors
- with feedback from tests (esp. fast unit tests), this is typically not difficult (see value #4: courage)
- you might have to change the unit tests as well as the code

XP practice #7: Small releases

- to get frequent feedback from client, you need to release (deploy) frequently
- this also helps build trust between dev team and client
- related idea in devops: continuous deployment

XP practice #8: Coding standards

- having quirky coding styles can hinder collective code ownership
- with agreed-upon standards, it's easier to understand a module you didn't write (see value #1: communication)

XP practice #9: Collective code ownership

- everyone is responsible for all the code
- each developer has the authority to change any part of the code (see courage and respect values)
- unit tests help give courage to change somebody else's code: you'll know if you break it

XP practice #10: Simple design

- is this code I wrote as simple as I can make it? If not, refactor.
- helps achieve Simplicity, in service to Communication

XP practice #11: System metaphor

- dev team, managers, and clients should be able to tell how the system works
- primary implication: names of major components should be clear
- a developer should be able to guess the functionality of a class, method, or function from its name

XP practice #12: Sustainable pace

- don't work more than 40 hours per week
- if that's really needed one week, take time off the next week
- belief: people perform best and most creatively when they're well rested
- pair programming and other practices might take it out of you more than non-XP practices
- emphasis on tests also reduces the need for pager duty on nights and weekends

XP practice #13: Retrospectives

- not actually one of the core 12 XP practices, but IMO pretty important
- recommended cadence: at the end of each iteration (say every other week)
- sit down with team and ask, "what went well?"
- then ask, "what didn't go well?"
- then vote on 1-3 topics from each category to dissect and discuss
- for good things, dissecting can help reproduce the success
- for bad things, dissecting can help avoid the problem
- can do this with client or just among dev team (or both)

Setting client expectations

- projects with fixed timeframe and budget feel more definite to clients
- in practice, they often aren't—but the initial perception matters
- convincing client to engage your team is difficult anyway, and esp. so if you won't commit to timeframe or budget
- recommendation: convey value to client ASAP!
- ideally, get a clickable prototype after a week or so
- then walking skeleton ASAP after that
- (this is one reason I'm requiring walking skeletons: knowing the strategy can majorly help build trust with clients
- for bigger projects, do a "planning & exploration" phase first to identify biggest risks and provide rough estimates, which inform budget for main project phase

Kanban

Scrum