Agile Recap - Answers/Summary

If stuck, reduce functionality and keep to the schedule, instead of trying to modify the schedule or add developers

- '9 pregnant women don't make a baby in a month'. Essentially, adding more developers rarely speeds up development in the short-term (can work long-term, but not relevant to this question)
- K eeping to deadlines in timeboxed iterations/sprints ensures a predictable delivery cycle and workload
- Lengthening development cycles reduces feedback from deliveries
- Reprioritization helps deliver on critical features while allowing those that appear less essential to be deferred
- Sustainable pace (No death marches or whatever)

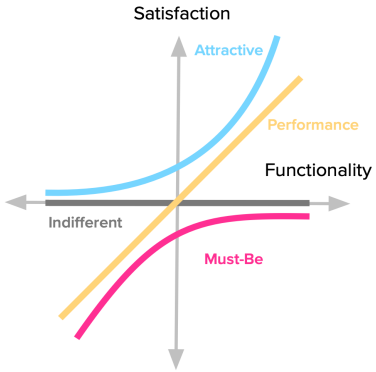
Big upfront requirements, why Agile opposes them and why some are still beneficial

- Agile seeks adaptability, feedback, delivering value early
  - Collecting big upfront requirements delays the start of development
  - Predefined requirements/product specifications hinder adaptability & customer feedback
  - Embrace change
  - Working software over comprehensive documentation
  - Incremental + iterative approach
  - Reduce risk + waste
- Some upfront planning is useful
  - Define a product vision + goals, tech stack, etc. (As we did in Sprint 0)
  - Set architectural foundations. Concerns over scalability, security, and integration should be considered early on in a project
  - External dependencies, risk analysis, legal requirements, budgeting, stakeholders cannot be 'discovered iteratively'

Thinking on the video of the ping pong ball, establish a flow before optimizing resources.
Importance of having customer relations

- XP has a customer representative role within teams (This isn't very viable, but still you get the point)
- Agile teams should have some role that interfaces with customers.
- Not all customer feedback is great - Customers don't really know what they want much of the time

Kano model:
Graph satisfaction on y-axis, level of fulfillment on x-axis.



Projects/developers should be executed at a sustainable pace, sprints should be planned with cushions of leftover time for the team members in case of unforeseen obstacles
Additive vs Multiplicative complexity

- Additive complexity refers to the linear growth of software complexity that simply results from the growth of a codebase/product
- Multiplicative complexity is often the result of a poorly designed architecture; increasing dependencies make it such that every additional feature multiplies the number of interactions, increasing complexity further

Accepting vs liking change

- I wouldn't dwell too much on this, the concept is self-explanatory
- PMs don't really enjoy change, but as part of Agile teams they should be willing to expect and accept it

Weighted-shortest-job-first feature prioritization

- I vaguely remember there being a slide and a short exercise on this, there will be an exam question about it, I will probably not have read the slide because I don't wanna search for it, assuming the concept is self-explanatory and then realize I probably should have at least skimmed the slide.

Iterative development

- Small incremental improvements. Break work down into iterations (Sprints in Scrum), each should produce a shippable product increment. Deliver working software in small steps rather than delivering all at once
- Frequent feedback and adaptation. Sprint review, demos, user testing are some of the feedback loops involved in iterations.
- Continuous learning & improvement. Retrospectives and learning on each cycle improve processes and decision-making
- Value prioritization. Features are delivered based on business value, backlog is refined to reflect changing priorities
- Reduced risk & early problem detection
- In Scrum, iterations are defined as sprints (1-4 weeks) where a set of backlog items are developed and reviewed
- In Kanban, Continuous iteration occurs through small batch delivery and WIP limits
- In XP, there are frequent releases (sometimes multiple per day) with constant refactoring and feedback

Requirements vs Scenarios

- Requirements cover the complete scope of functionalities, user stories cannot replace these, as you can't fully define a specification with them
- Sometimes requirements are necessary, think of legal requirements, architectural, performance, etc.

Scrum ceremonies

- Daily meeting
- Sprint demo
- Sprint planning
  - Planning Poker

Shared code ownership & cross-functional teams

- In shared code ownership, the entire team is responsible for all code, instead of having individual devs responsible for/'owning' sections
- Cross-functional teams refers to teams with knowledge from different areas. The professor speaks of wanting 'T-shaped' team members in Volvo Cars, with expertise in one specific area but at least surface-level knowledge on the rest of the areas relevant to the project

- Shared code ownership can improve code quality by bringing more eyes to each piece of code, speed up development by reducing bottlenecks on specific devs, encourages knowledge sharing and makes refactoring easier (no approval from a specific dev)
- Inconsistent coding styles, merge conflicts & overwrites, reduced sense of ownership/accountability and a need for strong communication are some drawbacks of shared code ownership
- Cross-functional teams can speed up delivery by reducing handoffs between teams, they improve collaboration and flexibility and are able to consider more aspects of a product, improving quality
- Cross-functional teams can face challenges with coordination, conflicting priorities, skill gaps and scaling (managing cross-functional teams in large organizations is complex)

Personal thoughts on pair programming

- Personal preferences and different work ethics: should not be forced onto all developers
- XP likes it

Test-Driven Development - Know 5 steps in detail, be able to provide examples (our own work is relevant, mention the Unit tests we wrote with JUnit for our backend, be aware that we wrote those tests after implementing the feature though so maybe google a better example and write it down here please uwu)

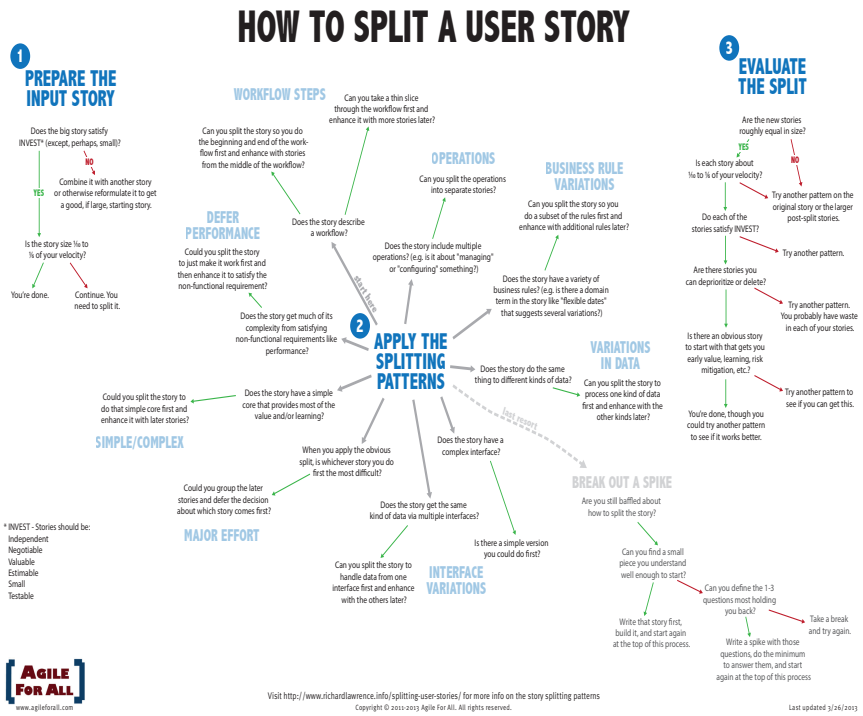
1. Quickly add a test
2. Run all tests, see the new one fail
3. Make a small change
4. Run all tests again, see them succeed
5. Refactor & remove duplication

Extreme Programming (XP)
Practices - Please list them down here if you have the slide with them on you :D. We need to be able to explain them listing them by heart is not necessary though
I am starting to believe that this subject is not worth the 3 credits I'll be getting at my home uni
Also, someone tell the student representatives that is the professor wants to forego the exam but also ensure that we actually learn the contents while keeping the primary focus on the project then having us answer the questions to the exam by sections as group assignments throughout the project would be a very effective way of achieving this while also being much less of a pain in the big O for all of us.

- Pair Programming
- System Metaphor
- Shared Code Ownership
- Small Releases
- Simple Design
- 40 Hour Week
- Planning Game
- Coding Standards
- Test first
- Continous Integration
- Customer On-site
- Refactoring
- Product Owner: customer representative, owns the product backlog, ensures team is building the right thing, prioritizes items in the product backlog, must also be able to negotiate and say "no" to the customer

- Scrum Master: Servant Leader ensures good working environment for the development team and removes impediments, coaches, initializes the agile ceremonies
- Development Team: self-organized cross-functional team, commits to delivering items in the sprint backlog, T-shaped team

Product backlog, why it should be detailed, what a DEEP backlog is, why we want ONE product backlog
Difference between project and product
User stories, INVEST format
Be able to break down a functionality into around 10 user stories:



Functional vs Non-functional requirements

- Functional: Define what the system should do Independent, Negotiable, Valuable, Estimatable, Small, Testable
- Non-functional: Define the constraints and how the system should behave Bounded, Independent, Negotiable, Testable