Eric Altenburg CS-555 February 9, 2021

Week 2

1. How much planning?
   - Depends on task's:
     - Domain
     - Are the requirement complete?
     - Will the requirements change over time?
     - Risk (reworking/budgeting for fixing mistakes)
     - Risk of doing the right thing too slow
     - Rewards?
2. Software dev life cycle
   - Specification: what functionality must we support?
   - Development: how do we create the software that delivers the functionality?
   - Validation: How do we verify that the software does what it's supposed to do?
   - Evolution: how does the software evolve to meet customer needs?
3. Waterfall model
   - Sequential
   - requirements -> design -> implementation -> verification -> maintenance
   - Can't go backwards to other steps, only forward
4. V model
   - Waterfall model with more rigorous validation/verification steps
5. Boehm's Spiral model
   - Incremental development and iterations
     1. objective setting
     2. Risk assessment and reduction
     3. Development and validation
     4. Planning for next iteration
   - After one iteration, you keep going back and starting over
6. Agile methods
   - Frequent iterations and deliverable
   - CLose collaboration between developers and customers
   - Support changing requirements
   - Frequent retrospection: learn and improvise from experience
   - Sprints (plan, design, build, test, and review)
     - Have retrospective meetings after each sprint to make changes in following sprints
7. Reason to use Agile Methods
   - Big upfront planning is not practical because of unstable changes and ambiguous requirements
   - Delivery through small baby steps through iterative and incremental development to reduce the chances of risk
   - Visibility with customers: customers are part of the term instead of being purely observers
   - Frequent reflections by the project team:
     - What are we doing well?
     - What can we improve?
8. Big Bang/Chaos (Ad-Hoc)
   - Little to no planning
   - Figure it out as you go
   - Typically used for very small projects (course projects, start ups, etc.)
   - Not highly recommended. . . .
9. Comparing software dev paradigms
   1. Lean
   2. Iterative
   3. Agile

    4. Ad-Hoc
    5. Traditional
10. Rational Unified Process (RUP)
    - 6 best practices of software engineering:
        1. develop iteratively
            - Solutions are too complex to get right first try
            - Use iterative approach and focus on high risk items in each pass
            - Customer involvement
            - Accommodate changes in requirements
        2. manage requirements
        3. use component-based architecture
            - Focus on early development and base lining of a robust architecture prior to full-scale development
            - Architecture should be flexible to accommodate changes
                - Modularization
        4. Model software visually (Unified modeling language UML)
        5. Continuously verify software quality
        6. Control changes
            - Change is inevitable
            - Manage the change request process
            - Control, track, and monitor changes
11. RUP phases
    - Inception: scope system for cost and budget, create basic use case models
    - Elaboration: mitigate risk by use case models
    - Construction: implement and test software
    - Transition: plan and execute delivery of system to customer
12. RUP disciplines
    - Used in each phase
    - Business modeling, requirements, analysis & design, implementation test, development, Configuration & change management, project management, and environment
13. Extreme Programming (XP)
    - An agile method
    - Combines best practices
14. 12 XP practices
    1. The planning game
        - Main planning process
        - Occurs once per iteration (once a week)
    2. Small releases
        - every release completely implements its new features
        - every release should contain most valuable business features
    3. Metaphor
        - Simple explanation of project
        - Agreed by members, simple for customers, and complete enough for architecture
    4. Simple design
        - Run all tests
        - No duplicated logic like parallel class hierarchies
        - States every intention important to the developers
        - Has the fewest possible classes and methods
    5. Testing
        - Developers continually write unit tests, which need to pass for development to continue
        - Customers write tests to verify features are implemented
        - Tests are automated so they are a part of system and continuously run to ensure the working of the system
    6. Refactoring

   - Devs reconstruct system without changing the behavior to remove problems with the code
   - How can we make the code simpler while still passing the tests?
7. Pair programming
   - Driver:
     - Thinks about the best way to implement
   - Navigator:
     - Viability of whole approach
     - Thinks of new tests
     - Thinks of simpler ways
     - Switch roles frequently
8. Collective ownership
   - Entire team takes ownership of whole system
   - Everyone knows a little bit of every part
   - If devs see opportunity to improve part of code, they do it
9. Continuous integration
   - Integrate and test every few hours, at least once per day
     - Don’t wait until end to integrate
   - All tests must pass
   - Easy to tell who broke code
     - Problem is likely to be in code most recently changed
10. Sustainable pace
11. Whole team
12. Coding standards