CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Software Process Models

Department of Computer Science
University of Maryland, College Park
Software Process Models

- Software methodology
  - Codified set of practices
  - Repeatable process for producing quality software
- Software process model
  - Methodology for organizing software life cycle
- Major approaches
  - Waterfall model
  - Iterative development
    - Unified model
    - Agile software development
      - Extreme programming (XP) (prominent example)
  - Formal methods
Waterfall Model

- **Approach**
  - Perform steps in order
  - Begin new step only when previous step is complete
  - Result of each step flow into next step

```
Problem specification
↓
Program design
↓
Selection of algorithms and data structures
↓
Coding and debugging
↓
Testing and verification
↓
Documentation and support
↓
Maintenance
```
Waterfall Model

• Advantages
  • Simple
  • Predictable results (emphasizes predictability)
    • Software follows specifications
  • Reasonable for small projects

• Problems
  • In real life
    • May need to return to previous step
    • Steps may be more integrated
    • Steps may occur at same time
  • Unworkable for large projects
Iterative Software Development

• Approach
  • Iteratively add incremental improvements
  • Take advantage of what was learned from earlier versions of the system
  • Use working prototypes to refine specifications
Iterative Software Development

• Goals
  • Emphasize *adaptability* instead of predictability
  • Respond to changes in customer requirements

• Examples
  • Unified model
  • Agile software development
    • Extreme programming (XP)
Unified Model

• Development divided into phases (iterations)
  • Inception
  • Elaboration
  • Construction
  • Transition
• During each phase
  • Multiple iterations of software development
  • Development treated as mini-waterfalls
  • Emphasis gradually shifts from specification to testing
Unified Software Life Cycle Model

- Inception
  - Planning
  - Analysis
  - Architecture
  - Design
  - Implementation
  - Integration
  - Test/Assessment

- Elaboration

- Construction
  - Iteration #1
  - Iteration #2...
  - Iteration #n+1
  - Iteration #...
  - Iteration #m
  - Iteration #m+1
  - Iteration #m+2...

- Transition
Agile Software Development

- Agile approach
  - Based on iterative development
    - Short iterations (timeboxes) lasting 1-4 weeks
  - Working software as principal measure of progress
    - Produced at end of each iteration
  - Adds a more people-centric viewpoint
    - Face-to-face communication preferred
    - Co-locate programmers, testers, “customers”
  - Relies on adapting to feedback rather than planning as the primary control mechanism
    - Less specification & documentation
Extreme Programming (XP)

- Prominent example of Agile methodology
  - Iterative, adaptive software development
- Describes set of day-to-day practices
  - Followed by managers & programmers
  - Intended to encourage a set of values
- Appropriate for environments with
  - Small teams
  - Rapidly-changing requirements
Extreme Programming Values

• Communication
  • Rapidly building & disseminating institutional knowledge among programming team

• Simplicity
  • Implement simplest code needed by customer without emphasis on future versions

• Feedback
  • From testing, team members, customers

• Courage
  • Willingness to rewrite / refactor software to add or change features
Extreme Programming Practices

• Pair programming
  • Pairs of programmers combine software development efforts at one computer
  • Especially useful for novice programmers
• Test-driven development
  • Tests are designed first, before writing software
• Continuous integration
  • Tests performed throughout development process
• On-site customer
  • Customer available at all times to answer questions
Formal Methods

- Mathematically-based techniques for
  - Specification, development, and verification
  - Software and hardware systems
- Intended for high-integrity systems
  - Safety
  - Security
- Levels
  - 0 – Informal implementation of formal specifications
  - 1 – Formal code development & verification
  - 2 – Theorem prover to ensure correctness
Choosing A Software Model

• Which software process model is appropriate?
• For class programming projects
  • Code and test probably suffices
  • But software in real world not like class projects
• Some big questions
  • Do you understand what you are trying to build?
  • What is the cost of change?
  • How many people have to interact with the design?
  • How easy is it to get the entire thing in your head?
Do You Understand The Problem?

- In many cases, the things we want software to do are not well understood
  - Examples
    - Provide a web interface for student applications
    - Allow users to view and manipulate photographs
    - Build a better search engine
  - Hard to understand constraints / interactions
  - May have to build prototype
    - To understand how users can effectively use it
What Is The Cost Of Change?

- Possible situation
  - Most coding already complete
  - Realize need to change something in the design or even the requirements
- How expensive is that?
  - If hugely expensive better get requirements & design right before completing too much code
- Some people believe recent software development techniques have substantially reduced cost of change
  - Possible reasons
    - Safer programming languages
      - E.g., not C/C++/assembly language
    - Object-oriented design & programming
    - Test-driven development
Rapid Prototyping

- Goal → explore requirements
  - Without building the complete system
- Start with part of the functionality
  - That will yield significant insight
- Build a prototype
  - Focus on core functionality
- Use the prototype to refine the requirements
- Repeat the process, expanding functionality