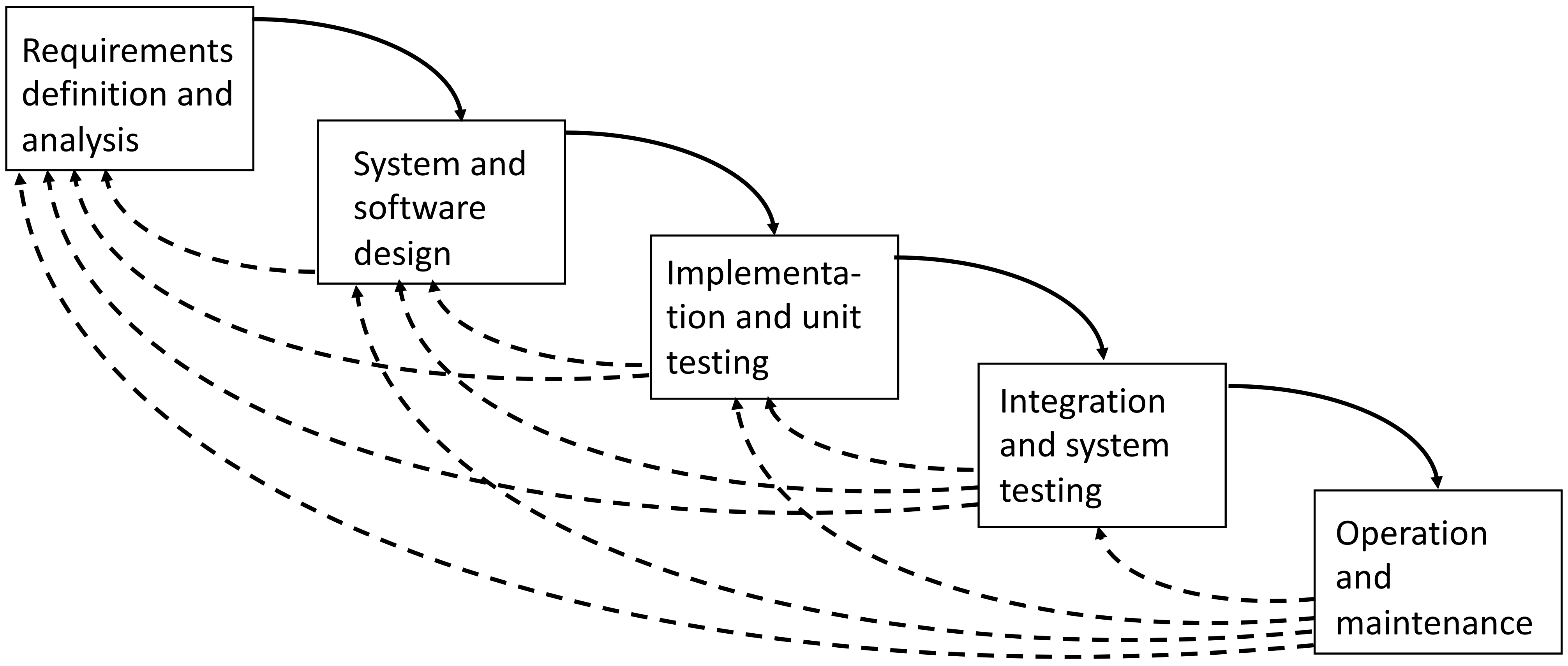
ECS 235B, Lecture 11

February 1, 2019

Waterfall Life Cycle Model

- Requirements definition and analysis
  - Functional and non-functional
  - General (for customer), specifications
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance

Relationship of Stages



Agile Software Development

- Software development is creative process, always changing, never really completed
- Leads to agile methodologies
  - Focuses on working together
  - Agile team efficiently works together in their environment
  - Team engages customer as a member of the team, developing requirements and scoping of the project
  - Accept, adapt to rapidly changing requirements
    - Allows for continuous improvement

Agile Methodologies

Term “Agile software development” used to describe several Agile methodologies

- Scrum
- Kanban
- Extreme Programming (XP)
- Others
  - Feature-Driven Development (FDD), Dynamic Systems Development Method (DSDM), Pragmatic Programming

In all, evidence of trustworthiness for assurance adduced *after* development

Scrum

- Split project into small parts that can be done in a short timeframe (called a *sprint*)
  - This *product backlog* created by product owner, who represents customer, product stakeholders
- Scrum team agrees on a small subset from top of backlog, decides how to design, implement it
  - Goal: complete this within the sprint
- Every day, team meets to evaluate progress, adjust as needed to get a workable solution within each sprint
  - At the end, work completed should be ready to ship, demo, or put back into backlog if not complete
- Iterate until product complete

Kanban

- Identify lanes of work: to be done, in progress, completed, deployed
- Each lane except the last has limit on how many items can be in that lane
  - Based on staff available to perform the work
- Teams take item off to be done lane, work on it until completed
  - When implemented correctly, team is completing work on top item in lane when another item arrives
- Goal: deliver product to customer within expected timeline
  - Methodology originated at Toyota

Extreme Programming

- Rapid prototyping and “best practices”
- Project driven by business decisions
- Requirements open until project complete
- Programmers work in teams
- Components tested, integrated several times a day
- Objective is to get system into production as quickly as possible, then enhance it

Models

- Exploratory programming
  - Develop working system quickly
  - Used when detailed requirements specification cannot be formulated in advance, and adequacy is goal
  - No requirements or design specification, so low assurance
- Prototyping
  - Objective is to establish system requirements
  - Future iterations (after first) allow assurance techniques

Models

- Formal transformation
  - Create formal specification
  - Translate it into program using correctness-preserving transformations
  - Very conducive to assurance methods
- System assembly from reusable components
  - Depends on whether components are trusted
  - Must assure connections, composition as well
  - Very complex, difficult to assure

Key Points

- Assurance critical for determining trustworthiness of systems
- Different levels of assurance, from informal evidence to rigorous mathematical evidence
- Assurance needed at all stages of system life cycle

Threats and Goals

- *Threat* is a danger that can lead to undesirable consequences
- *Vulnerability* is a weakness allowing a threat to occur
- Each identified threat requires countermeasure
  - Unauthorized people using system mitigated by requiring identification and authentication
- Often single countermeasure addresses multiple threats

Architecture

- Where do security enforcement mechanisms go?
  - Focus of control on operations or data?
    - Operating system: typically on data
    - Applications: typically on operations
  - Centralized or distributed enforcement mechanisms?
    - Centralized: called by routines
    - Distributed: spread across several routines

Layered Architecture

- Security mechanisms at any layer
  - Example: 4 layers in architecture
    - *Application layer*: user tasks
    - *Services layer*: services in support of applications
    - *Operating system layer*: the kernel
    - *Hardware layer*: firmware and hardware proper
- Where to put security services?
  - Early decision: which layer to put security service in

Security Services in Layers

- Choose best layer
  - User actions: probably at applications layer
  - Erasing data in freed disk blocks: OS layer
- Determine supporting services at lower layers
  - Security mechanism at application layer needs support in all 3 lower layers
- May not be possible
  - Application may require new service at OS layer; but OS layer services may be set up and no new ones can be added

Security: Built In or Add On?

- Think of security as you do performance
  - You don’t build a system, then add in performance later
    - Can “tweak” system to improve performance a little
    - Much more effective to change fundamental algorithms, design
- You need to design it in
  - Otherwise, system lacks fundamental and structural concepts for high assurance

Reference Validation Mechanism

- *Reference monitor* is access control concept of an abstract machine that mediates all accesses to objects by subjects
- *Reference validation mechanism* (RVM) is an implementation of the reference monitor concept.
  - Tamperproof
  - Complete (always invoked and can never be bypassed)
  - Simple (small enough to be subject to analysis and testing, the completeness of which can be assured)
    - Last engenders trust by providing evidence of correctness

Examples

- *Security kernel* combines hardware and software to implement reference monitor
- *Trusted computing base* (*TCB*) consists of all protection mechanisms within a system responsible for enforcing security policy
  - Includes hardware and software
  - Generalizes notion of security kernel

Adding On Security

- Key to problem: analysis and testing
- Designing in mechanisms allow assurance at all levels
  - Too many features adds complexity, complicates analysis
- Adding in mechanisms makes assurance hard
  - Gap in abstraction from requirements to design may prevent complete requirements testing
  - May be spread throughout system (analysis hard)
  - Assurance may be limited to test results

Example

- 2 AT&T products with same goal of adding mandatory controls to UNIX system
  - SV/MLS: add MAC to UNIX System V Release 3.2
  - SVR4.1ES: re-architect UNIX system to support MAC

Comparison

- Architecting of System
  - SV/MLS: used existing kernel modular structure; no implementation of least privilege
  - SVR4.1ES: restructured kernel to make it highly modular and incorporated least privilege

Comparison

- File Attributes (*inodes*)
  - SV/MLS added separate table for MAC labels, DAC permissions
    - UNIX inodes have no space for labels; pointer to table added
    - Problem: 2 accesses needed to check permissions
    - Problem: possible inconsistency when permissions changed
    - Corrupted table causes corrupted permissions
  - SVR4.1ES defined new inode structure
    - Included MAC labels, DAC attributes
    - Only 1 access needed to check permissions

Requirements Assurance

- *Specification* describes of characteristics of computer system or program
- *Security specification* specifies desired security properties
- Must be clear, complete, unambiguous
  - Something like "meets C2 security requirements" not good: what *are* those requirements (actually, 34 of them!)

Example

- “Users of the system must be identified and authenticated” is ambiguous
  - Type of ID required—driver’s license, token?
  - What is to be authenticated—user, representation of identity, system?
  - Who is to do the authentication—system, guard?
- “Users of the system must be identified to the system and must have that identification authenticated by the system” is less ambiguous
  - Under what conditions must the user be identified to the system—at login, time of day, or something else?

Example

- "Users of the system must be identified to the system and must have that identification authenticated by the system before the system performs any functions on behalf of that identity"
  - Type of identification is user name
  - User identification (name) to be authenticated
  - System to authenticate
  - Authentication to be done at login (before system performs any action on behalf of user)

Methods of Definition

- Extract applicable requirements from existing security standards
  - Tend to be semiformal
- Combine results of threat analysis with components of existing policies to create a new policy
- Map the system to existing model
  - If model appropriate, creating a mapping from model to system may be cheaper than requirements analysis