Software Engineering in a Nutshell

- Development of software systems whose size/ complexity warrants a team or teams of engineers
  - multi-person construction of multi-version software [Parnas 1987]
- SCOPE: study of software process, development principles, techniques and notations
- GOAL: production of quality software, delivered on time, within budget, satisfying users' needs
- Software engineers must posses a broad range of skills
  - mathematics
  - computer science
  - economics
  - management
  - psychology

Software Programming ≠ Software Engineering

▪ single developer	▪ developer teams ▪ multiple roles
▪ "toy" applications	▪ complex systems
▪ short lifespan	▪ long, indefinite lifespan
▪ single or few stakeholders ❑ developer = user	▪ multiple stakeholders ❑ developer ≠ user ❑ user ≠ customer
▪ one-of-a-kind systems	▪ system families
▪ built from scratch	▪ reuse to amortize costs
▪ minimal maintenance	▪ maintenance is 60+% of total development costs

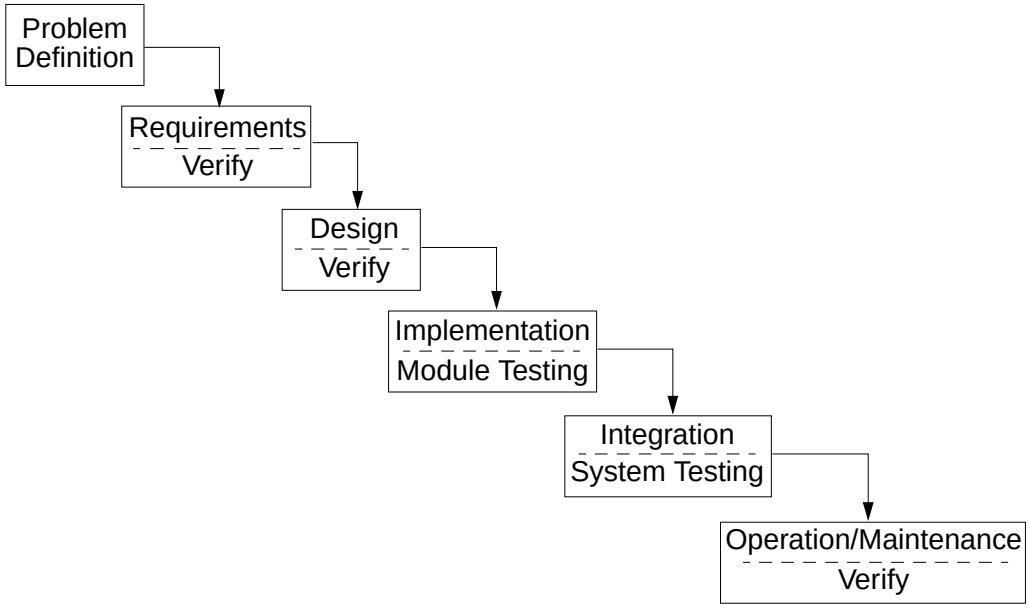
Software Engineering Is a Unique Brand of Engineering

- Software is malleable
- Software construction is human-intensive
- Software is intangible
- Software problems are unprecedentedly complex
- Software directly depends upon the hardware
  - it is at the top of the system engineering “food chain”
- Software solutions require unusual rigor
- Software has discontinuous operational nature

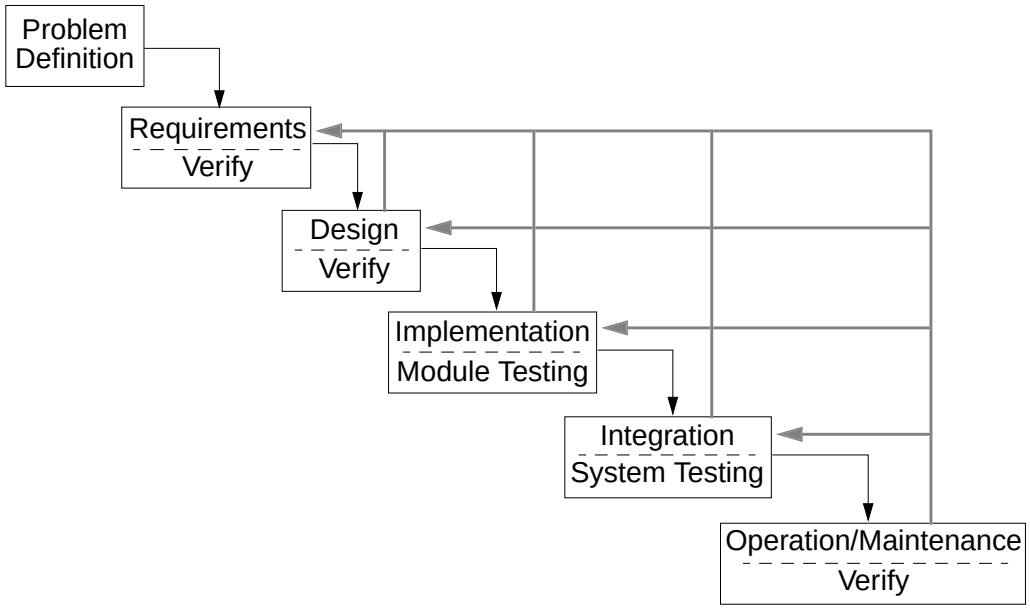
Economic and Management Aspects

- Software production = development + maintenance
- Maintenance costs > 60% of all development costs
  - 20% corrective
  - 30% adaptive
  - 50% perfective
- Quicker development is not always preferable
  - higher up-front costs may defray downstream costs
  - poorly designed/implemented software is a critical cost factor

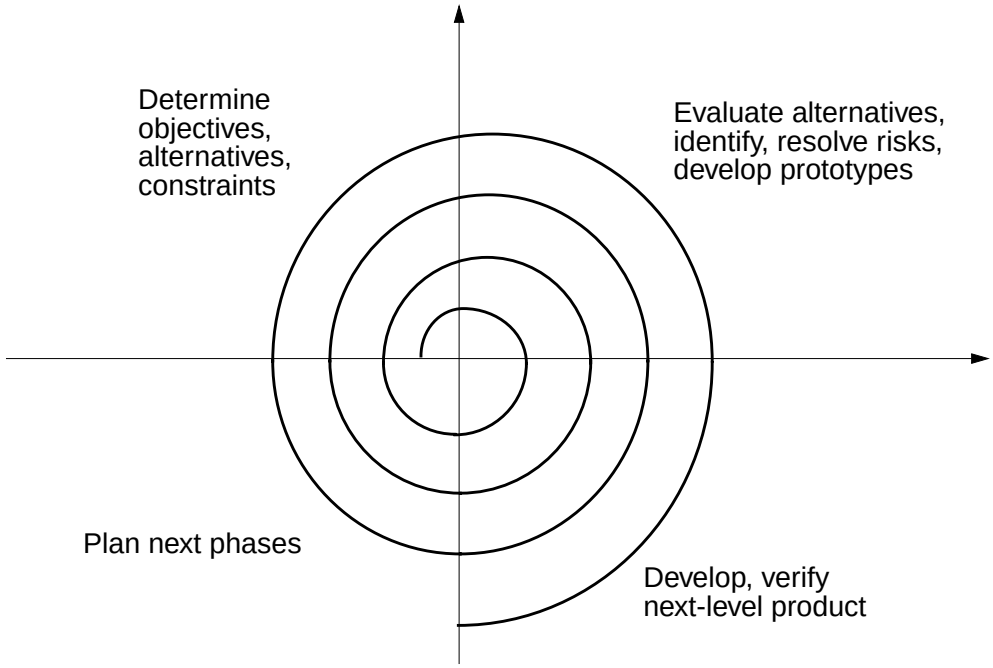
Software Development Lifecycle (1)



Software Development Lifecycle (2)



Software Development Lifecycle (3)



Requirements

- Problem Definition → Requirements Specification
  - determine exactly what the customer and user want
  - develop a contract with the customer
  - specifies *what* the software product is to do
- Difficulties
  - client asks for wrong product
  - client is computer/software illiterate
  - specifications are ambiguous, inconsistent, incomplete
- Verification
  - extensive specification reviews with the customer
  - identify ambiguity, inconsistency, incompleteness
  - ascertain feasibility, testability
  - develop system/acceptance test plan

Design

- Requirements Specification → Design
  - high-level: decompose software into modules with interfaces
  - detailed: develop module specifications (select algorithms and data types)
  - maintain a record of design decisions and traceability
  - specifies *how* the software product is to do its tasks
- Difficulties
  - miscommunication between module designers
  - design may be inconsistent, incomplete, ambiguous
- Verification
  - extensive design inspections to establish conformance to requirements
  - check module interactions
  - develop integration test plan

Implementation and Integration

- Design → Implementation
  - implement modules; verify that they meet their specifications
  - combine modules according to the design
  - specifies *how* the software product does its tasks
- Difficulties
  - module interaction errors
  - order of integration may influence quality and productivity
- Verification
  - extensive code reviews to establish conformance to requirements and design
  - check module interactions
  - develop unit test plan: focus on individual modules
  - test on unit, integration (focus on module interfaces) and acceptance (focus on requirements) test plans

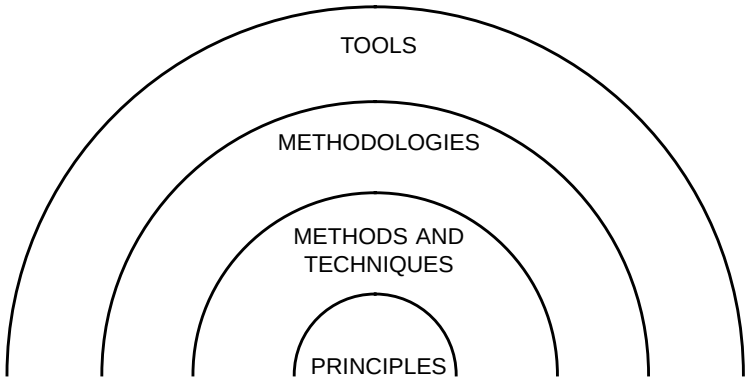
Operation and Maintenance

- Operation → Change
  - maintain software during/after user operation
  - determine whether the product still functions correctly
- Difficulties
  - rigid design
  - lack of documentation
  - personnel turnover
- Verification
  - extensive reviews to ensure correct changes and updates to documentation
  - test to determine that change is correctly implemented
  - regression testing: ensure that no inadvertent changes are made to compromise system functionality

Software Engineering Principles

- Rigor and formality
- Separation of concerns
  - modularity
  - abstraction
- Anticipation of change
- Generality
- Scalability
- Compositionality

From Principles to Tools



Software Qualities

- Qualities (a.k.a. “ilities”) are *goals* in the practice of software engineering
- External vs. Internal qualities
  - external qualities are visible to the user
    - reliability, efficiency, usability
  - Internal qualities are the concern of developers
    - they help developers achieve external qualities
    - verifiability, maintainability, extensibility, evolvability, adaptability
- Product vs. Process qualities
  - product qualities concern the developed artifacts
    - maintainability, understandability, performance
  - process qualities deal with the development activity
    - products are developed through process
    - maintainability, productivity, timeliness

Some Software Qualities (1)

- Correctness
  - ideal quality
  - established w.r.t. the requirements specification
  - absolute
- Reliability
  - statistical property
  - probability that software will operate as expected over a given period of time
  - relative
- Robustness
  - “reasonable” behavior in unforeseen circumstances
  - subjective
  - a specified requirement is an issue of correctness; an unspecified requirement is an issue of robustness

Some Software Qualities (2)

- Usability
  - ability of end-users to easily use software
  - extremely subjective
- Understandability
  - ability of developers to easily understand produced artifacts
  - internal product quality
  - subjective
- Verifiability
  - ease of establishing desired properties
  - performed by formal analysis or testing
  - internal quality

Some Software Qualities (3)

- Performance
  - equated with efficiency
  - assessable by measurement, analysis, and simulation
- Evolvability
  - ability to add or modify functionality
  - addresses adaptive and perfective maintenance
  - problem: evolution of implementation is ***too*** easy
  - evolution should start at requirements or design
- Reusability
  - ability to construct new software from existing pieces
  - must be planned for
  - occurs at all levels: from people to process, from requirements to code

Some Software Qualities (4)

- Interoperability
  - ability of software (sub)systems to cooperate with others
  - easily integratable into larger systems
  - common techniques include APIs, plug-in protocols, etc.
- Scalability
  - ability of a software system to grow in size while maintaining its properties and qualities
  - assumes maintainability and evolvability
  - goal of component-based development

Some Software Qualities (5)

- Heterogeneity
  - ability to compose a system from pieces developed in multiple programming languages, on multiple platforms, by multiple developers, etc.
  - necessitated by reuse
  - goal of component-based development
- Portability
  - ability to execute in new environments with minimal effort
  - may be planned for by isolating environment-dependent components
  - necessitated by the emergence of highly-distributed systems (e.g., the Internet)
  - an aspect of heterogeneity

Software Process Qualities

- Process is *reliable* if it consistently leads to high-quality products
- Process is *robust* if it can accommodate unanticipated changes in tools and the environments
- Process *performance* is productivity
- Process is *evolvable* if it can accommodate new management and organizational techniques
- Process is *reusable* if it can be applied across projects and organizations

Assessing Software Qualities

- Qualities must be measurable
- Measurement requires that qualities be precisely defined
- Improvement requires accurate measurement
- Currently most qualities are informally defined and are difficult to assess

Software Engineering “Axioms”

- Adding developers to a project will likely result in further delays and accumulated costs
- Basic tension of software engineering
  - better, cheaper, faster — pick any two!
  - functionality, scalability, performance — pick any two!
- The longer a fault exists in software
  - the more costly it is to detect and correct
  - the less likely it is to be properly corrected
- Up to 70% of all faults detected in large-scale software projects are introduced in requirements and design
  - detecting the causes of those faults early may reduce their resulting costs by a factor of 100 or more