Software Engineering Introduction & Background

Department of Computer Science
Kent State University

Complaints

- Software production is often done by amateurs
- Software development is done by tinkering or by the “million monkey” approach
- Software is unreliable and needs permanent maintenance
- Software is messy, lacks transparency, prevents improvement or building on (or costs too much to do so)

General Problems

- 50% of all software projects fail
  - Never delivered/completed
  - Do not meet requirements or user needs
  - Excessive failures (bugs)
  - Excessively over budget or late
- Quality and reliability of many software systems can not be formally assessed

Problems with Software Production

- Complexity
- Conformity - conform to the existing process or have the process conform to the software
- Changeability - Software can "easily" be changed, but a bridge is almost impossible to move
- Invisibility - software is very hard to visualize
- Brook's "No Silver Bullet" [IEEE Computer 9(4), 1987]
  - Software is very difficult to develop, and most likely will not get easier.
  - Reuse is one solution suggested.
  - In 20 years, 6% per year production improvement.

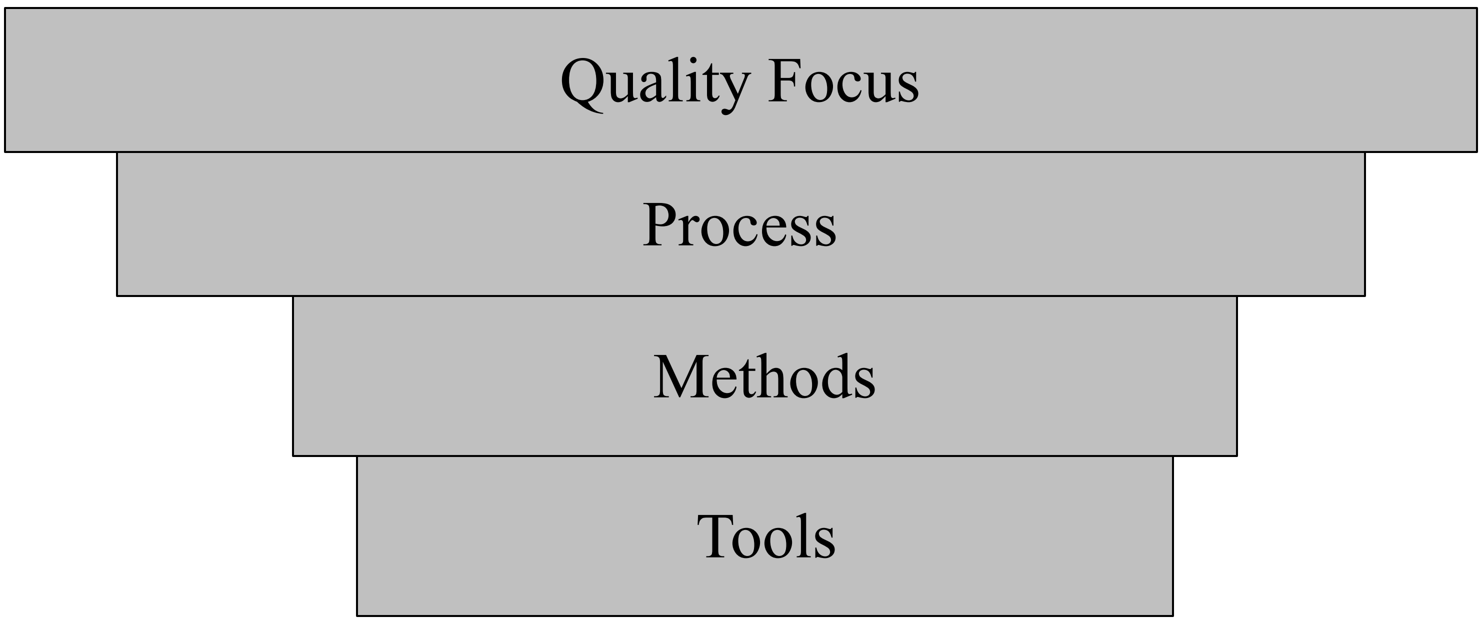
Questions

- Why does it take so long to get software completed?
- Why are costs so high?
- Why can’t all errors be found before the software is put into production?
- Why is it difficult to measure the progress at which software is being developed?

Some Facts

- Software is developed not manufactured.
- Software does not wear out.
- Most software is custom built rather than assembled from existing components.

Software Engineering



A Layered Approach

- Focus on quality
  - Power plant vs. Word processor
- Process layer that enables rational and timely development of software (e.g., Agile)
  - Key process areas must be established for effective delivery of software technology
- Methods provide support for process (OO)
- Tools provide support for methods (.net)

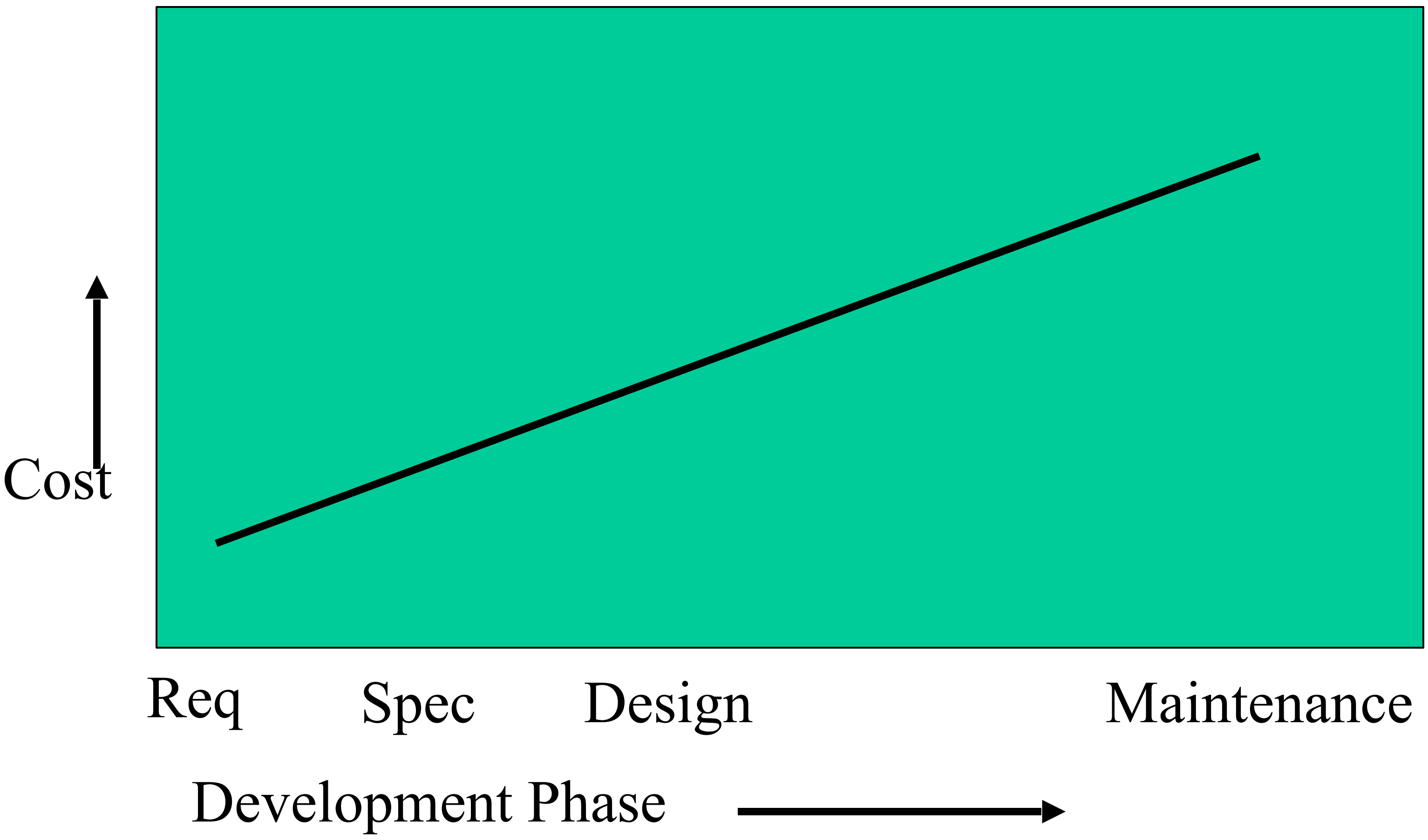
Building Software

- What is the problem to be solved?
- What characteristics of the entity are used to solve the problem?
- How will the entity (and solution) be realized?
- How will the entity be constructed?
- What approach will be used to uncover errors?
- How will it be supported over the long term?

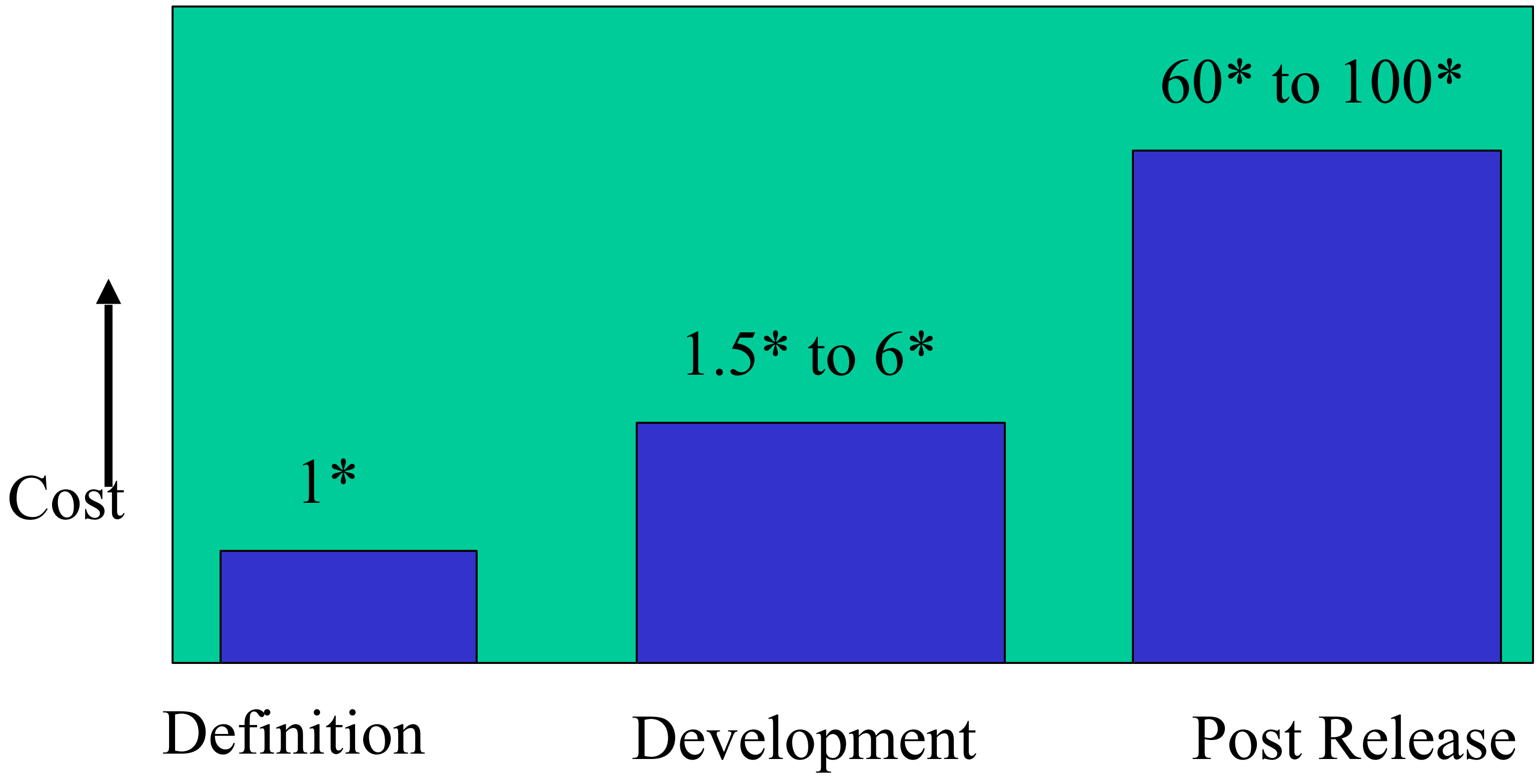
Phases of Software Life Cycle

- Definition Phase - behavior of the system
- Development Phase - How to obtain the desired behavior
- Maintenance - change the behavior
  - Corrective - fix uncovered defect
  - Adaptive - Platform change
  - Enhancement - Perfective, additional functionality
  - Preventive - re-engineering, make system more maintainable

Cost to fix faults



Cost to fix faults



Software Applications

- System Software
- Real time
- Business
- Engineering and Scientific
- Embedded systems
- Personal Computing

What types of Development Models fit for which applications?

Software Process Models

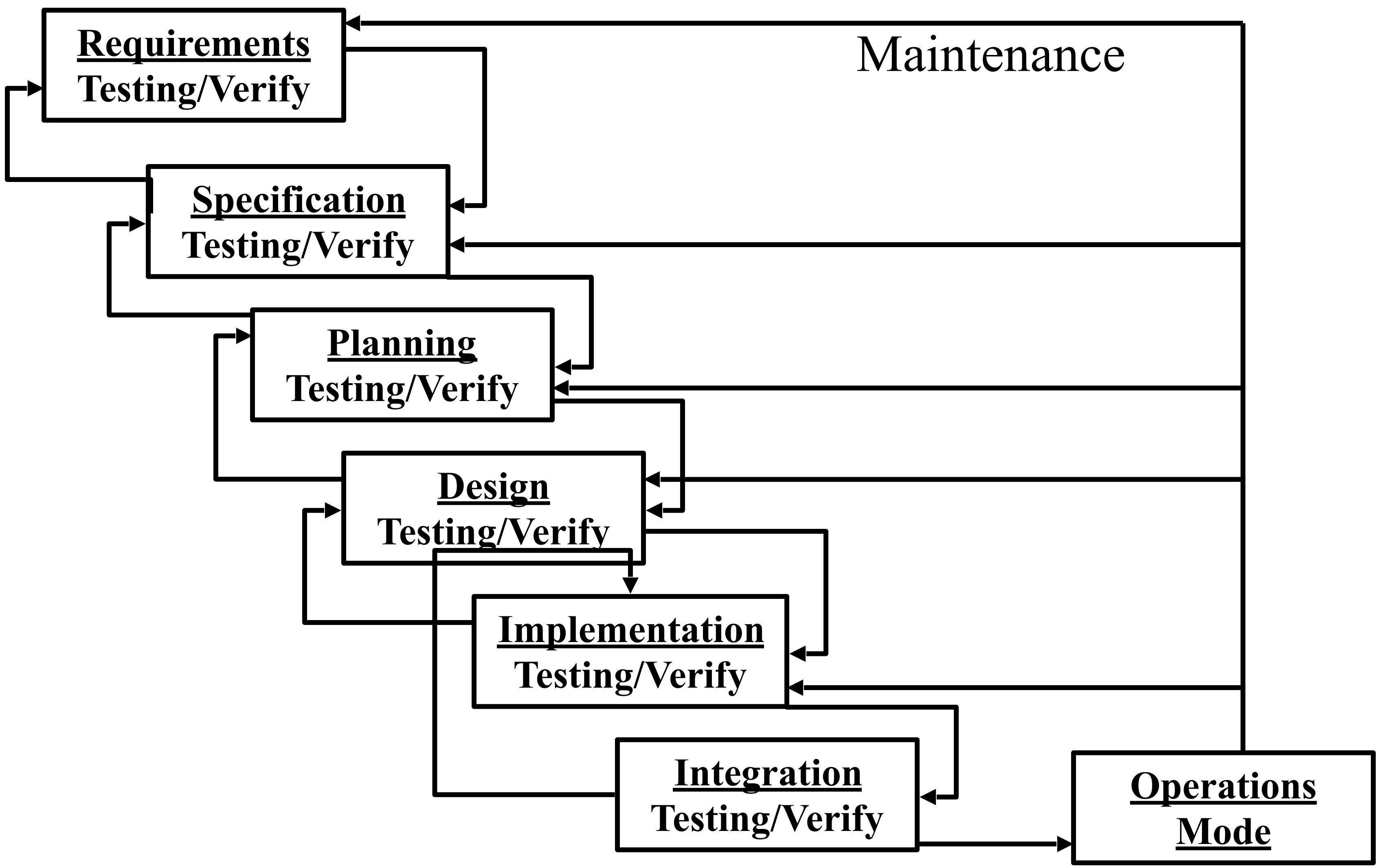
- Classical Process Models
  - Waterfall
  - Linear Sequential
  - Prototyping Model
  - Rapid Application Development

- Evolutionary Process Models
  - Incremental Model
  - Spiral Model
  - Component Assembly Model
  - Concurrent Development Model

Classical Lifecycle Model aka Waterfall

- Requirements Phase
- Specification Phase (Analysis)
- Planning Phase
- Design Phase
- Implementation Phase
- Integration and Testing
- Maintenance
- Retirement

Sequential Model



Sequential Model

- Feedback loops to correct uncovered faults
- Testing and Verification at each phase
- Documentation at each phase
- Each phase is completed before next phase can begin

Sequential Model: Problems

- Real projects don’t often run sequentially
- Customers must have patience
- Development is often delayed, i.e., “blocking states”
- Specifications may not reflect client expectations
- Staffing problems, e.g., “tall, narrow” developers versus “short, wide” developers

Prototyping

- Modified Sequential Model

- A prototype is constructed to determine system requirements and specifications
- Prototype is used as a tool to determine clients needs
- Numerous problems can be uncovered during prototype development and evaluation

Prototyping： Problems

- Prototype is viewed by the customer and management as a completed system
- Design decisions, e.g., language, platform, API, etc., chosen for prototype are difficult to have changed, but may be inappropriate for completed system
- Small, visible changes between prototype and finished system are easily perceived by the customer

Rapid Application Development

- High-speed modification of linear sequential mode.
- Component-based construction of system
- Very short time frame
- Typically used for information systems
- Difficult for applications in which the parts are not already components
- Unsuited for projects with high technical risk

Software Evolution

- All software evolves (changes) over time
- Requirements change over the lifetime of the project
- Time to market means we cannot wait until the very end of the project for a solution
- Must make efficient use of team members

- Iterative model
- Develop increasingly more complex versions of the software

Incremental Model

- Combines linear sequential model with prototyping

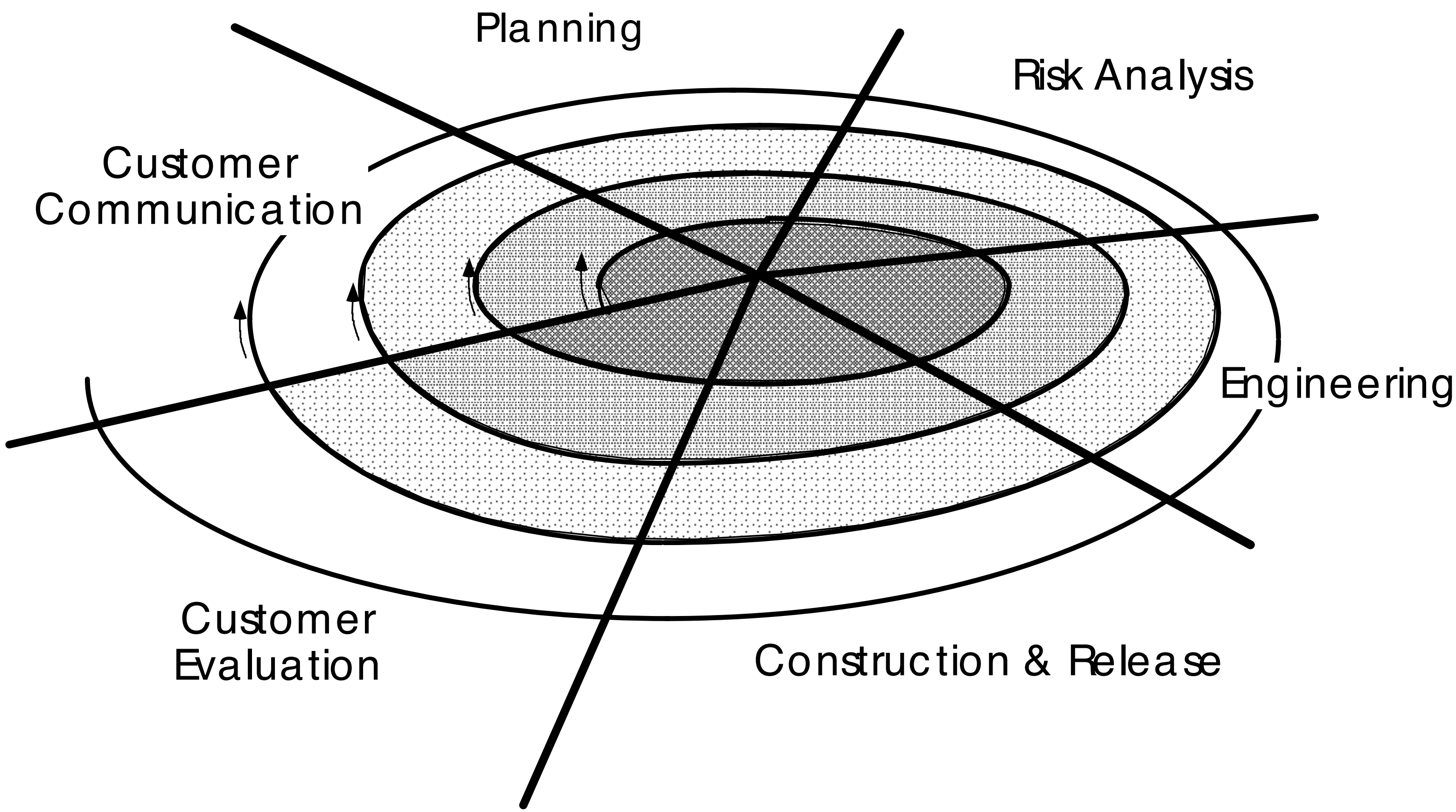
- Produces increments of a system.
- First produce the core product
- A set of new functionality is added in each new increment
- The first increment can be viewed as a prototype that is used by the client
- Overlapping sequences of process stages
- Focus on a set of deliverables
- Allows workers dedicated to a particular stage, e.g., “short, wide” developers

Spiral Model

- Software is developed in a set of incremental releases
- Early iterations may be prototypes or paper models
- Later iterations are increasingly more complex versions of the software

- Divided into a number of framework activities or task regions (typically between 3 and 6)
- Allows for efficient use of resources

Spiral Model



Component Assembly Model

- Use a set of pre-existing components to construct a new system
- Need a library of existing component
- Need a method of indexing these components

- Narrow domain
- Subset of system uses existing components

Which process to use?

- Based on needs and goal of the organization
- Problem domain
- Application area
- Composition of development team

- Customized process to fit the organization
- It’s not a process unless it’s written down.
- Define:
  - Goals, processes, methods, tools

Methods: OO Analysis and Design

- Object Oriented Analysis - Method of analysis which examines requirements from a perspective of the classes and objects found in the vocabulary of the problem domain.

- Object Oriented Design - Method of Design encompassing the process of object oriented decomposition. Logical and physical as well as static and dynamic models are depicted.

Software Testing

- Verification - whether something has been correctly carried out. Are we building the product right?

- Validation - whether something satisfies its specification. Are we building the right product?

- Software testing process:
  - Software Quality Assurance (SQA)
  - Independent Verification and Validation (IV&V)

SQA Activities

- Evaluations to be performed
- Audits and reviews to be performed
- Standards that are applicable to the project
- Procedures for error reporting and tracking
- Documents to be produced by SQA group
- Amount of feedback provided to software project team

Types of Testing

- Execution based testing
- Non-execution based testing

- Non-execution based testing:
  - Walkthroughs
  - Inspections

Walkthroughs

- Informal examination of a product (document)

- Made up of:
  - developers
  - client
  - next phase developers
  - SQA leader

- Produces:
  - list of items not understood
  - list of items thought to be incorrect

Inspections

- Formalized examination of a product (document)

- Formal steps:
  - Overview
  - Preparations
  - Inspection
  - Rework
  - Follow-up

Inspections

- Overview - of the document is made
- Preparation - participants understand the product in detail
- Inspection - a complete walk through is made, covering every branch of the product. Fault finding is done
- Rework - faults are fixed
- Follow - up check fixed faults. If more than say 5% of product is reworked then a complete inspection is done again.

- Statistics are kept: *fault density*

Execution Based Testing

"Program testing can be a very effective way to show the presents of bugs but is hopelessly inadequate for showing their absence" [Dijkstra]

- Fault: "bug" incorrect piece of code
- Failure: result of a fault
- Error: mistake made by the programmer/developer

Behavioral Properties

- **Correctness** - does it satisfy its output specification?
- **Utility** - are the user's needs met
- **Reliability** - frequency of the product failure.
  - How long to repair it?
  - How lone to repair results of failure?
- **Robustness** - How crash proof in an alien environment?
  - Does it inform the user what is wrong?
- **Performance -** response time, memory usage, run time, etc.

Methods of Testing

- Test to specification:
  - Black box,
  - Data driven
  - Functional testing
  - Code is ignored: only use specification document to develop test cases

- Test to code:
  - Glass box/White box
  - Logic driven testing
  - Ignore specification and only examine the code.

Feasibility

- Pure black box testing (specification) is realistically impossible because there is (in general) too many test cases to consider.
- Pure testing to code requires a test of every possible path in a flow chart. This is also (in general) infeasible. Also every path does not guarantee correctness.
- Normally, a combination of Black box and Glass box testing is done.

Can you Guarantee a Program is Correct?

- This is called the Halting Problem (Theory of Computer Science stuff).

- Write a program to test if any given program is correct. The output is correct or incorrect.
- Test this program on itself.
- If output is incorrect, then how do you know the output is correct?

- Conundrum, Dilemma, or Contradiction?

Development of Test Cases

- Test cases and test scenarios comprise much of a software systems *testware*.
- Testware is all the “wares” that go with testing.

- Black box test cases are developed by domain analysis and examination of the system requirements and specification.

- Glass box test cases are developed by examining the behavior of the source code.

Pairing down test cases

- Use methods that take advantage of symmetries, data equivalencies, and independencies to reduce the number of necessary test cases.

- Equivalence Testing
- Boundary Value Analysis

- Determine the ranges of working system
- Develop equivalence classes of test cases
- Examine the boundaries of these classes carefully

Equivalence Testing

- Example: sort(lst, n)
  - Sort a list of numbers
  - The list is between 2 and 1000 elements
- Domains:
  - The list has some item type (of little concern)
  - n is an integer value (subrange)
- Equivalence classes;
  - n<2
  - n>1000
  - 2 <= n <= 1000

Equivalence Testing (example)

- What do you test?
- Not all cases of integers
- Not all cases of positive integers
- Not all cases between 1 and 1001

- Highest payoff for detecting faults is to test around the boundaries of equivalence classes.

- Test n=1, n=2, n=1000, n=1001, and say n= 10
- Five tests versus 1000.

Structural Testing

- Statement coverage -
  - Test cases which will execute every statement at least once.
  - Tools exist for help
  - No guarantee that all branches are properly tested. Loop exit?
- Branch coverage
  - All branches are tested once
- Path coverage - Restriction of type of paths:
  - Linear code sequences
  - Definition/Use checking (all definition/use paths)
  - Can locate dead code

Proofs of Correctness

- Mathematical proofs (as complex and error prone as coding)

- Leavenworth ‘70 did an informal proof of correctness of a simple text justification program. (Claims it’s correct!)
- London ‘71 found four faults, then did a formal proof. (Claims it’s now correct!)
- Goodenough and Gerhar ‘75 found three more faults.

- Testing would have found these errors with much difficulty.

Software Metrics

- Measure - quantitative indication of extent, amount, dimension, capacity, or size of some attribute of a product or process.

- Metric - quantitative measure of degree to which a system, component or process possesses a given attribute.

- Number of errors
- Number of errors found per person hours expended

- Metric: A handle or guess about a give attribute.

Process and Product Metrics

- Process -
  - Insights of process paradigm, software engineering tasks, work product, or milestones.
  - Lead to long term process improvement.

- Product -
  - Assesses the state of the project
  - Track potential risks
  - Uncover problem areas
  - Adjust workflow or tasks
  - Evaluate teams ability to control quality

Some Metrics

- Defects rates
- Errors rates

- Measured by:
  - individual
  - module
  - during development

- Errors should be categorized by origin, type, cost

Some Metrics

- Direct measures - cost, effort, LOC, etc.
- Indirect Measures - functionality, quality, complexity, reliability, maintainability

- Size Oriented:
  - Lines of code - LOC
  - Effort - person months
  - errors/KLOC
  - defects/KLOC
  - cost/KLOC

Complexity Metrics

- LOC - a function of complexity
- language dependent

- Halstead's Software Science (entropy measures)
  - n_1 - number of distinct operators
  - n_2 - number of distinct operands
  - N_1 - total number of operators
  - N_2 - total number of operands

Halstead’s Metrics

- Length: $N = N_1 + N_2$
- Vocabulary: $n = n_1 + n_2$

- Estimated length: $N' = n_1 \log2 n_2 + n_1 \log2 n_2$

- Volume: $V = N \log2 n$

- Number of bits to provide a unique designator for each of the n items in the program vocabulary.

Estimating Software Size

- Standard Component Method
- Function Point
- Proxy Based Estimation

Standard Component Method

- Gather data about various level of program abstraction, subsystems, modules, reports, screens.

- Compare these to what is predicted in the system

- Estimate= $\left(\textbf{Smallest value estimate} + 4^{*} \textbf{Most likely or common estimate} + \textbf{Largest value estimate} \right)$

Function Point Method

- Functions:

- Inputs: screens, forms (UI) or other programs which add data to the system. Inputs that require unique processing
- Outputs: Screens, reports, etc
- Inquiries: Screens which allow users to interrogate or ask for assistance or information
- Data files: logical collections of records, tables in a DB
- Interfaces: Shared files, DB, parameters lists

Function Point Method

- Review requirements
- Count number of each function point type
- Use historical data on each function point type to determine estimate

- Function point does not map to physical part of source.
- Can not measure FP in a given system (automatically)