Software Development

Software Development Activities

- Problem Definition
- Requirements Analysis
- Implementation Planning
- High-level Design (or Architecture)
- Detailed Design
- Coding and Unit Testing (Debugging)
- Integration and System Testing
- Deployment and Maintenance
- Documentation, Verification, Management

Problem Definition

- A clear statement of the problem
- Defines problem without any reference to solution
- Should be in user's language
  - Not in technical terms

Failure to define problem may cause you to solve the wrong problem

Requirements Analysis

- This is the *what* is to be solved
- Helps ensure that the user rather than the programmer drives system's functionality
- Helps avoid arguments
- Minimizes changes to the system after development begins
- Change is inevitable
  - Set up change-control procedure
  - Use development approaches that accommodate changes

Specifying requirements adequately is a key to project success

Implementation Planning

- Defines how the product will be implemented
- Establishes development schedule
- Identifies resources required
  - Labor hours and people
  - Other direct costs
- Estimated budget
- Create Work Breakdown Structure (WBS)

This is the roadmap that will be used throughout the development process. Without a roadmap you don't know where you are going and you won't know that you arrived.

High-level Design
(or Architecture)

- This is *how* to solve the problem
  - (recall that requirements is the *what* is to be solved)
- Defines the overall organization of the system in terms of *high-level components and interactions*
- All design levels are documented in specification documents that keep track of design decisions

High-level and detail-level are usually not separated

Detailed Design

- Components are decomposed into lower level modules
- Precisely defined interactions
- Interfaces are defined
- Constraints are identified

The purpose of the design phase is to specify a particular software system that will meet the stated requirements

Coding and Unit Testing
(Debugging)

- Produces the actual code that will be delivered to the customer
- Typically develop modules that are independently tested

Results in an implemented and tested collection of modules

Integration and System Testing

- All the modules that have been developed and tested individually are put together – integrated – and are tested as a whole system
- Integrated and tested progressively (on larger sets of modules)
- Some coding may be necessary to complete the integration

The final stage is “actual” use or *alpha testing*.

Deployment and Maintenance

- Deployment: defines the physical run-time architecture of the system
  - Set proper system parameters
  - Install applications
- Maintenance: long-term development
  - 60% of total cost of software is maintenance
  - Cost of maintenance:
    - 40% to changes in user’s requirements
    - 17% to changes in data formats
    - 12% to emergency fixes
    - 9% to routine debugging
    - 6% to hardware changes
    - 5% to improvements in documentation
    - 4% to improvements in efficiency
    - 7% to other sources

Documentation, Verification, Management

- Common to all the other activities
- Documentation
  - Main result of any activity – each activity products at least one document
- Verification and Validation
  - Verification: Assessment of the internal correctness of process
  - Validation: how the product responds to needs of customer
  - Performed as quality control as reviews, walk-througs, and inspections
  - Discovery and removal of errors as early as possible
- Management
  - Budget, schedule, resources

Development Lifecycle Models

- The documented collection of policies, processes and procedures
  - used by a development team or organization to practice software engineering
  - is called its software development methodology (SDM) or system development life cycle (SDLC).

Development Lifecycle Models

- The challenge in selecting and following a methodology is to do it wisely
  - to provide sufficient process disciplines to deliver the quality required for business success,
  - while avoiding steps that waste time, squander productivity, demoralize developers, and create useless administrivia.
- The best approach for applying a methodology is to consider it as a means to manage risk.
  - You can identify risks by looking at past projects.

Development Lifecycle Models

- Waterfall processes
  - The best-known and oldest process is the waterfall model, where developers (roughly) follow the steps in order.
  - After each step is finished, the process proceeds to the next step, just as builders don't revise the foundation of a house after the framing has been erected.

Development Lifecycle Models

- Iterative processes
  - prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster.
  - Agile software development processes are built on the foundation of iterative development. To that foundation they add a lighter, more people-centric viewpoint than traditional approaches.
  - Extreme Programming, XP, is the best-known agile process. In XP, the phases are carried out in extremely small (or "continuous") steps compared to the older, "batch" processes.

Development Lifecycle Models

- Formal methods
  - mathematical approaches to solving software and hardware problems at the requirements, specification and design levels. Examples include the B-Method, Petri nets, RAISE and VDM.
  - Finite state machine based methodologies allow executable software specification and by-passing of conventional coding; e.g. virtual finite state machine or event driven finite state machine.

Development Lifecycle Models

- Examples:
  - Waterfall (Pure Waterfall)
  - Spiral (Spiral)
  - Evolutionary (Evolutionary Prototype)
  - Incremental (Staged Delivery)
  - ...
- Also
  - Capability Maturity Model® - Integration (CMMI)
  - Rational Unified Process

Development Lifecycle Models – Selection Criteria

- Does the system have a precedent (I.E., Have similar systems been built before)?
- Is the technology understood and stable?
- Are the requirements understood and stable?
- Are suitable COTS products available and ready to be used in end products?
- Is this a large or complex project or product?
- Is the project fully funded at startup?
- Is the project cost or schedule constrained and requirements cannot be reduced?
- Is there a need for engineering subprojects driven by risk identification and mitigation?
- Is the existing system's maintenance cost too high/is her a need to facilitate future system enhancements?

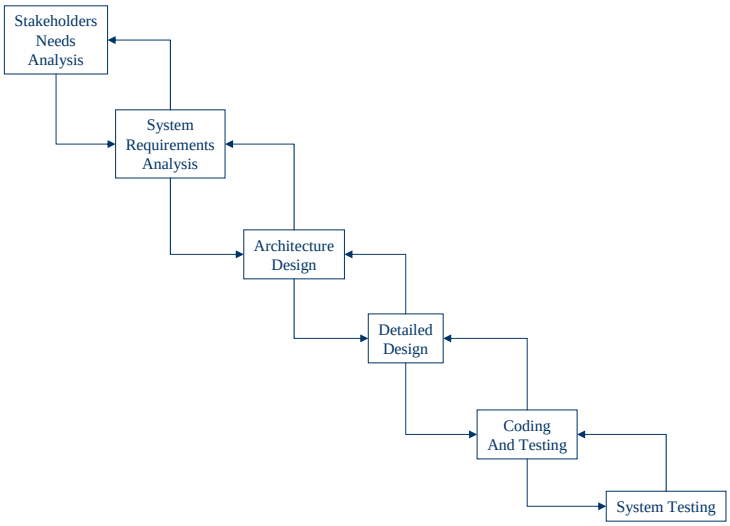
Waterfall

- Description
  - An orderly sequence of steps from the initial software concept through system testing
  - A review at the end of each phase
  - Document driven
  - Phases are discontinuous

Waterfall

- Advantages
  - Helps minimize planning overhead
  - Works well for projects that are well understood buy complex
  - Works well when quality requirements dominate cost and schedule
  - Works well even if your technically staff is not expert
- Disadvantages
  - Have to fully specify requirements at beginning of project
  - Waterfall model isn’t flexible
  - Generates few visible signs of progress until the very end
  - Excessive amount of documentation

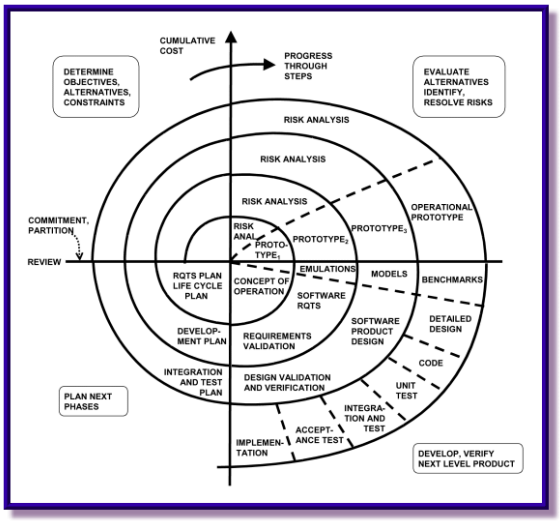
Waterfall Model



Spiral

- Description
  - A metamodel that can accommodate any process development model
  - A particular model is chosen based on level of risk
  - Spiral model is cyclic
  - Four stages
    - Objectives identified
    - Alternatives evaluated and risk areas identified
    - Develop and verify next level of product
    - Review and plan for next iteration

Spiral



http://www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html

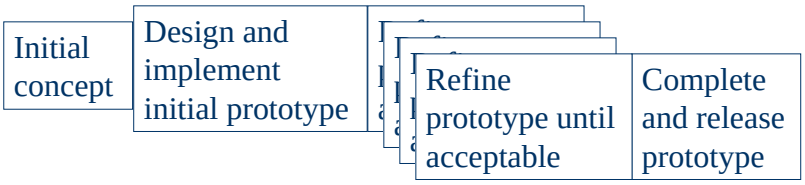
Spiral

- Advantages
  - Requirements not well understood
  - Risks not known
  - As costs increase risks decrease
  - Provides management insight
- Disadvantages
  - Difficult to use for contracted software
    - Don't know the outcome at beginning of project
    - Only as good at mitigating risk as engineers are at identifying risks

Evolutionary

- Description
  - Develop system concept as you move through the project
  - Develop prototypes including real requirements analysis, real design, and real maintainable code
- Advantages
  - Manage changing requirements
  - Unsure of optimal architecture or algorithms
  - Produces steady, visible signs of progress
- Disadvantages
  - Don't know time required to complete project
  - Can become an excuse to do code-and-fix

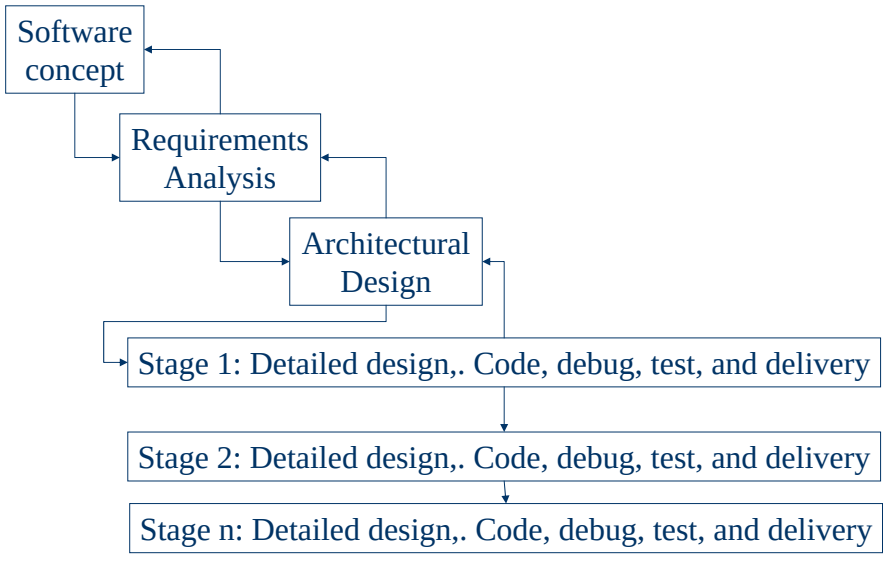
Evolutionary



Incremental

- Description
  - Also known as Staged Delivery
  - Deliver software in successive stages throughout the project
- Advantages
  - Put useful functionality into hands of customer early
  - Provides tangible signs of progress
- Disadvantages
  - Won't work without careful planning
  - Determining stage dependencies

Incremental



Capability Maturity Model® - Integration (CMMI)

- Software Engineering Institute at Carnegie Mellon Institute in Pittsburgh established standards and guidance for developing software engineering disciplines and management
  → the Capability Maturity Model (CMM)
  - has become widespread among mature software development organizations, especially for those developing large scale software in a competitive procurement environment;
  - government and corporate software customers have increasingly required that proposals include information about a software development organization's certified level of maturity.

Capability Maturity Model® - Integration (CMMI)

- CMM recognized five steps:
  1. Level 1 (Initial) - Processes are ad hoc and occasionally chaotic.
  2. Level 2 (Repeatable) - Basic project management processes are established to track cost, schedule and functionality.
  3. Level 3 (Defined) - Management and engineering processes are documented and integrated into a standard software process.
  4. Level 4 (Managed) - Detailed measures of the software process and product quality are collected.
  5. Level 5 (Optimizing) - Continuous process improvement is aided by quantitative feedback from the process and from piloting innovative ideas and technologies.
- Around year 2000, the SEI introduced CMMI.

Rational Unified Process

- The Rational Unified Process (RUP) is
  - an iterative software development process created by the Rational Software Corporation, now a division of IBM;
  - an extensive refinement of the (generic) Unified Process.
  - an adaptable process framework, intended to be tailored by the development organizations and teams that will select the elements of the process that are appropriate for their needs.
  - also a software process product.

Estimation

- Size Estimation
- Effort Estimation
- Schedule Estimation

Most projects overshoot their estimated schedules by anywhere from 25% to 100%

Without an accurate schedule estimate, there is no foundation for effective planning

Estimation

- Constructing software is like constructing a house: you can't tell exactly how much it is going to cost until you know exactly what "it" is.
- As with building a house, you can either build your dream house – expense be hanged – or you can build to a budget, you have to be very flexible about the product characteristics.
- Whether you build to a budget or not, software development is a process of gradual refinement, so some imprecision is unavoidable. Unlike building a home, in software the only way to refine the product concept and thereby the estimate is to actually build the software.
- Estimates can be refined over the course of a project. Promise your customer that you will provide more refined estimates at each stage.

Estimation Process

- Estimate the size of the product (number of lines of code or function points)
  - First need to estimate the size of the program to be build
- Estimate the effort (man-months)
  - Need accurate size estimates and historical data on past projects
- Estimate the schedule (calendar months)
  - With size and effort estimates, schedule is easy
  - Selling the schedule is HARD
- Provide estimates in ranges and periodically refine the ranges to provide increasing precision as the project progresses

Size Estimation

- Use an algorithmic approach, such as function points, that estimates program size from program features.
- Use size-estimation software that estimates program size from your description of program features (screens, dialogs, files, database tables, etc.)
- Estimate each major piece of the new system as a percentage of the size of a similar piece of an old system. Add the pieces to get the total size.

Size estimation should be in terms of lines-of-code

Estimation Tips

- Avoid off-the-cuff estimates
- Allow time for the estimate, and plan it
- Use data form previous projects
- Use developer-based estimates
- Estimate by walk-through
- Estimate by categories
- Estimate at a low level of detail
- Don’t omit common tasks
- Use software estimation tools
- Use several different estimation techniques, and compare the results
- Change estimation practices as the project progresses

Effort Estimation

- Use estimation software to create an effort estimate directly from the size estimate
- Use organization's historical data to determine how much effort previous projects of the estimated size have taken
- Use an algorithmic approach such as Barry Boehm's COCOMO model or Putnam and Myer's lifecycle model to convert a lines-of-code estimate into an effort estimate

Effort estimates should be in terms of man-months

Schedule Estimation

- Can get schedule estimate from effort estimate with the following equation

  Schedule in months = 3.0 * man-months$^{1/3}$
- Example
  - 65 man-months to build project
  - 12 months = 3*65$^{1/3}$
- 65 man-months / 12 months = 5 or 6 team members

One of the common problems with schedule estimates is that they are usually done so crudely that people pad them to give themselves a margin of error.

Best Practices

- Change Board
  - Group that controls changes to software
  - Efficacy
    - Potential reduction from nominal schedule: Fair
    - Improvement in progress visibility: Fair
    - Effect on schedule risk: Decreased Risk
    - Chance of first-time success: Very Good
  - Major Risk
    - Approving to few or too many changes

Best Practices

- Daily Build and Smoke Test
  - Product is built every day (compiled, linked, and combined into an executable program) – the product is then tested to see if it “smokes”
  - Efficacy
    - Potential reduction from nominal schedule: Good
    - Improvement in progress visibility: Good
    - Effect on schedule risk: Decreased Risk
    - Chance of first-time success: Very Good
  - Major Risk
    - Pressure to release interim versions of a product too frequently

Best Practices

- Designing for Change
  - Identifying likely changes, developing a change plan, and hiding design decisions so that changes do not ripple through a program.
  - Efficacy
    - Potential reduction from nominal schedule: Fair
    - Improvement in progress visibility: None
    - Effect on schedule risk: Decreased Risk
    - Chance of first-time success: Good
    - Chance of long-term success: Excellent
  - Major Risk
    - Overeliance on the use of programming languages to solve design problems rather than on change-oriented design practices

Best Practices

- Evolutionary Delivery
  - Deliver selected portions of the software earlier than would otherwise be possible.
  - Efficacy
    - Potential reduction from nominal schedule: Good
    - Improvement in progress visibility: Excellent
    - Effect on schedule risk: Decreased Risk
    - Chance of first-time success: Very Good
    - Chance of long-term success: Excellent
  - Major Risk
    - Feature creek, diminished project control, unrealistic schedule and budget expectations, inefficient use of development time by developers.

Best Practices

- Evolutionary Prototyping
  - System developed in increments so that it can readily be modified in response to end-user and customer feedback.
  - Efficacy
    - Potential reduction from nominal schedule: Excellent
    - Improvement in progress visibility: Excellent
    - Effect on schedule risk: Increased Risk
    - Chance of first-time success: Very Good
    - Chance of long-term success: Excellent
  - Major Risk
    - Unrealistic schedule and budget expectations, inefficient use of prototyping time, unrealistic performance expectations, poor design, poor maintainability

Best Practices

- Goal Setting
  - Use goals to motivate software developers (Shorten schedule, decrease risk, maximum visibility)
  - Efficacy
    - Potential reduction from nominal schedule: Very Good, None, None
    - Improvement in progress visibility: None, Good, Excellent
    - Effect on schedule risk: Increased Risk, Decreased Risk, Decreased Risk
    - Chance of first-time success: Good, Good, Good
    - Chance of long-term success: Very Good, Very Good, Very Good
  - Major Risk
    - Significant loss of motivation if goals are changed

Best Practices

- Inspections
  - Formal technical review
  - Efficacy
    - Potential reduction from nominal schedule: Very Good
    - Improvement in progress visibility: Fair
    - Effect on schedule risk: Decreased Risk
    - Chance of first-time success: Good
    - Chance of long-term success: Excellent
  - Major Risk
    - None

Best Practices

- Joint Application Development (JAD)
  - Requirements-definition and user-interfaced design methodology in which end-users, executives, and developers attend intense off-site meetings to work out a system's details.
  - Efficacy
    - Potential reduction from nominal schedule: Good
    - Improvement in progress visibility: Fair
    - Effect on schedule risk: Decreased Risk
    - Chance of first-time success: Good
    - Chance of long-term success: Excellent
  - Major Risk
    - Unrealistic productivity expectations following the JAD sessions, premature, inaccurate estimates of remaining work following JAD sessions