Software Engineering Notes

Patrick Lindsay

May 6, 2013

Contents

Part I

Notes

Chapter 1

Software Life-Cycle Model

1.1 Overview

<u>Software Engineering</u> - The application of sound engineering practices to software creation and maintenance.

1.1.1 Software development Life Cycle (Traditional Approach)

- Requirements Phase
- Analysis or Specification Phase
- Design Phase
- Implementation/Integration Phase
- Maintenance Phase
- Retirement

Requirements Phase

- Determining the NEEDS and WANTS of the client or customer.
- Determining the constraints of the system.

Analysis or Specification Phase

- After analyzing the requirements, construct a *specification document* which explicitly describes what the product is to do, and the constraints under which it must operate.
- This includes the description of the input, output, actions, and UI.

- The specification document can be used as part of a contract with the client.

Problems with the Spec Document

1. Ambiguity - one sentence may have more than one interpretation.
2. Incompleteness - relevant fact or requirement is left out.
3. Contradiction - two places in the spec document are in conflict.

Design Phase

- Construct an *Architectural Design.*
- Construct a *Detailed Design.*
- Test for *traceability.*

1. Architectural Design - Description of the product in terms of modules.
2. Detailed Design - Description of each module.
3. Traceability - each part of the design can be traced to a statement in the specification document.

Implementation Phase

- Code each module from the detailed design.
- Programmer tests his/her own code separately.
- Modules are combined and tested by developers.
- Product is tested by SQA group. This is called product testing.
- Project is given to the client for acceptance testing.

Maintenance Phase

- Corrective Maintenance - bug squashing
- Enhancement Maintenance - Updates
  - Perfective - client makes new demands
  - Adaptive - changes in the environment of the product requires changes in the software.
- Perform regression testing - insuring that changes have not affected already working functionality.

Retirement Phase

- Determining if desired changes are too costly.
- Determining if a product is obsolete.

1.1.2 Four Components of the Software Engineering Enterprise

The four P's

1. Process
2. Project
3. People
4. Product

Process

- The process is sometimes called the life-cycle model or development sequence.
  - Waterfall
  - Spiral
  - Incremental Build
- Makes use of several process frameworks.
  - Personal Software Process (PSP)
  - Team Software Process (TSP)
  - Capability Maturity Model (CMM)
- Documentation Standards
  - IEEE
  - ANSI

Project

- The set of activities needed to produce the required product.
- Project management is extremely important.
- Many projects are not about developing new products, but maintaining already existing *legacy* systems.

People

- Team Organization
- Team Management
- Relationship with customer or client
- Relationship with end users
- Communication with upper management

Product

Includes

- Requirement Specification Document
- Design Document
- Source Code
- Executable
- User Manuals

Chapter 2

Traditional Software Engineering Process

2.0.3 Historical Influences

- Structured programming (Edsger Dijkstra's letter calling "GOTOs" harmful) uses sequence control, iteration, invoking functions
- Object Oriented paradigm: the use of objects with data and functionality which can represent real-world entities.

Note:
Silver Bullet ca. 1980s
Likened the software crisis to a werewolf. Object Oriented Paradigm was the Silver Bullet.
It did not work as expected.

- Design Patterns: stock of reusable design elements (templates)

2.0.4 Component Reuse

A component as defined by Meyer is "a program element satisfying:"

1. The element may be used by other program elements. (Clients)
2. The clients and their authors do not need to be known to the element's author

2.0.5 Key Expectations of Software Engineering

1. Decide in advance what the specific quality measures are to be for the project and product.
   Predetermine quantitative quality goals.

2. Gather data on all projects to form a basis for estimating future projects.
3. All requirements, designs, code and test materials should be freely and easily available to all members of the team.
   Source code should always be available to all team members in an easily accessible and interpreted way.
   Git, Mercurial, etc.
4. A process should be followed by all team members. *Uniformity*
   (a) Design only against requirements.
   (b) Program only against design.
   (c) Test only against requirements and design.
   Always follow the recipe!
5. Measure and achieve quality goals.

2.1 Methods

Be able to draw and discuss these.

2.1.1 Waterfall Method

See diagram 52.9 on pg 53.

- First described by William*(?)* Royce in 1970.
- No phase is complete until documentation for that phase has been completed and approved by the SQA group.
  Very orderly; heavy on documentation.
- Has been used with great success on a variety of products.
- Feedback loops permits modifications to be made to the previous phase.

Advantages

1. Enforced disciplined approach
2. Requirement that documentation be provided at each phase.
3. All products of the phase must be checked by SQA.
4. Inherent aspect of each phase is testing.

Disadvantages

- The resulting specification document may not be able to be understood by the client.
- It can lead to the construction of product that does not meet the client's needs.

2.1.2 Rapid Prototype Model

SEE DIAGRAM ON PG 55. Construction of a functional subset of the desired product in order to allow the client and the developer to interact.
Keyword is rapid. This is a thrown-together, proof-of-concept type project; a mock-up.

Advantages

- The process is linear and possibly faster than the Waterfall Model
- Increases interaction between client and developer.

Disadvantages

- Client may inaccurately think the product is almost complete when viewing the prototype.
- Developer may attempt to use the prototype as part of the final product.

2.1.3 Waterfall-Rapid Prototype Hybrid

May form a hybrid model using the rapid prototype as the first phase in the Waterfall Model in order to increase interaction but allow for feedback loops within the development of the product.

2.1.4 Incremental Model

Software is implemented, integrated, and tested as a series of incremental builds.
Code pieces providing specific functions.

Advantages

1. Results in builds which can be developed in weeks, not months or years.
2. End user need not learn the entire product at one time.
3. Client need not pay for the entire product at one time.
4. Developer gets paid earlier.
   (At each build delivery)

5. Open-ended design makes maintenance easier.

6. Easier to make changes during development.

Disadvantages

1. Each new build must fit in without destroying existing builds.
   Regression Testing

2. Requires more careful to design to make it open to additions.

3. Can degenerate to a build and fix product if broken into too few builds.

2.1.5 Spiral Model

See Figure 2.12 on pg 63 and Figure 2.13 on pg 65.

- A Waterfall Model with each phase preceded by risk analysis in an attempt to control or resolve risk.

- Each phase is 360.

- The measure of the radius is the cumulative cost to date.

- The measure of the angle is the progress measure.
  Each phase is 360.
  Requires a very experienced engineer.

Advantages

1. The emphasis on alternatives and constraints supports the reuse of existing software.

2. The incorporation of software quality as a specific objective.

3. Answers the question of how much testing should be performed in terms of risks.

4. Maintenance is simply another cycle of the spiral, the same as development.

Disadvantages

1. Intended exclusively for internal development.
   Client and developer are members of the same organization.

2. *Applicable only to large-scale projects.*

3. Must have developers who are skilled at pinpointing the possible risks.

2.2 Agile Methods

2.2.1 General

According to the Agile Manifesto, they value:

- Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation.
- Responsiveness to change over following a plan.
  You don't always have to follow the recipe!

Traits

- Highly iterative
- Pair programming with a focus on teamwork and ego-less programming.
  This is mandatory.
- Early and planned testing.
- Story cards *Similar to storyboards in movies.*
- Refactoring *Turning working code into better code.*
- Feedback

Principles Behind the Agile Manifesto

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive disadvantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.

8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity - the art of maximizing the amount of work not done - is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

2.2.2 Refactoring

- Reduce software complexity
- Improve internal structure while preserving the behavior of the code.
  Prettifying
- Improve software quality
  Readability, execution efficiency, size efficiency, etc.
- Performed in response to "bad smells" in the code, undesirable characteristics
  Martin Fowler
- Automated or manual

Chapter 3

Teams

Team - a group of professionals organized in order to complete the task of creating a large software project.

3.1 Team Structure

3.1.1 Project Factors Related to Structure of the Team

1. Difficulty of the problem
2. Size of the program in LOC or Function Points
   LOC stands for Lines of Code
3. Time the team will stay together
4. Degree of modularity for program
5. Required quality and reliability
6. Rigidity of delivery date
7. Degree of communication required

3.1.2 Jelled Team

- A group of people which are so tightly knit that the attitude is that the whole is greater than the sum of the parts
- Egos are forgotten and the team becomes important
- Exhibits cohesiveness, team spirit, common definition of success
- Generally more productive, more motivated, and happier

3.1.3 Why Don't All Teams Jell?

- A frenzied work atmosphere
- High frustration causing friction among team members
- A "fragmented or poorly coordinated" software process
- An unclear definition on the team
- "Continuous and repeated exposure to failure" - M. Jackman *Homeopathic Remedies for Team Toxicity*

3.1.4 Necessary Team Traits

Personality

- Openness - intellectual curiosity
- Conscientiousness - self-discipline, pushing toward goals
- Extroversion - energy, emotions, seek company of others
  Being a people person
- Agreeableness - compassionate and cooperative
- Neuroticism - how a person responds to stress, emotional stability

3.1.5 How to Make Personality Traits Work

1. Recognize people have different types of personalities
2. Assemble a diverse team covering a range of personalities
3. Create an open, honest, tolerant atmosphere in team meetings

3.2 Roles

- Team Leader - Responsible for overseeing all aspects of the team project; holds the tie-breaking vote.
- Technical Lead - Expert on all technical aspects of the project, in particular the hardware and software used for development.
- Designer - Designs the project and breaks the project into smaller pieces (*Modules*) for the programmers.
- Lead Programmer - Needs to have an understanding of the project as a whole; organizes all of the other programmers.

- Technical Writer - Writes all documentation for the project (*Team meeting minutes, Specification Document, Users' Manual*)
- Configuration Management - Maintains the code base for the project; could include CVS responsibilities.
- Quality Assurance - Writes, maintains, and conducts all testing associated with the project.

3.3 Organizational Structures

- Democratic Team
- Hierarchical or Chief Programmer
- Team Manager/Team Leader
- Synchronize-and-Stabilize Team
- Agile Team

3.3.1 Democratic Team

- Group of up to 10 programmers
- Equal partnership with egoless programming
- Works well if the group is small, highly competent
- Problem with who is in charge
- Positive attitude about finding fault
- Good in research environment with difficult problem

3.3.2 Hierarchical or Chief Programmer

- One overall manager (*Chief Programmer*)
- Everyone understands the lines of authority
  One boss.
- Team members tend to participate less in decisions; decisions are handed down from above
- May have a Programming Secretary and a Backup Programmer for the Chief Programmer
- Difficult to find one person adept at both managing and programming.

3.3.3 Team Manager/Team Leader

See Figure 4.4, pg. 114

- Split the responsibilities of Chief Programmer into Team Manager and Team Leader
- The Team Manager handles the nontechnical management
- The Team Leader deals with the technical issues of the project
- Results in programmers having two bosses
- May be difficult to determine if an issue is technical or nontechnical

Technical Organizational Structure for Large Projects

- One Project Leader oversees several team leaders
- Each team leader has several programmers for which he/she is responsible
- Clear lines of communication
  Two level structure

3.3.4 Synchronize-and-Stabilize Team

Has been used by Microsoft

- Small team led by a manager and having three to eight developers and three to eight testers working one-to-one with the developers
- Developers are given freedom to design and implement their portions as they wish
- Each day, the partial components are tested and debugged.
- Encourages creativity and innovation yet the daily synchronization keeps the project on track.

3.3.5 Agile Team

- Work in pairs (MANDATORY)
- Provide instant review
- Create test cases which are used for daily testing
- Remove the problem if one developer leaves, the knowledge does not disappear about a portion of the project

Chapter 4

Tools

4.1 Tools for Software Engineers

- Analytic (Theoretical) Tools:
    1. Stepwise Refinement
    2. Cost-Benefit Analysis
    3. Divide-and-Conquer
    4. Separation of Concerns
    5. Software Metrics
- Software Tools (CASE: Computer-Aided Software Engineering)

4.1.1 Analytic Tools

Stepwise Refinement

- Process whereby a project is successively decomposed into more detailed instructions.
- In each step, a given task is written as a set of subtasks.
- Term was first coined by Niklaus Wirth in 1971.
- Helps to concentrate on relevant aspects of the current development phase and ignore details that need not be considered.
- A postponement of decisions on details until as late as possible.
- Critical to object-oriented paradigm.

Cost-Benefit Analysis

- Comparing estimated future benefits against future costs for a certain decision.
- Problems occur in that intangible benefits may be hard to quantify.
- May use past experience to project the estimates for benefits or costs.

Divide-and-Conquer

- Most agree this is the oldest analytical tool used in Software Engineering.
- Break a large problem into smaller subprograms that should be easier to solve.
- Idea used in the Unified Process.
- Good concept but no details on how to divide the problem
- *Key difference from Stepwise Refinement is that Divide-and-Conquer does not necessarily procrastinate details.*

Separation of Concerns

- First introduced by Dijkstra in 1974.
- Process of breaking a software project into components which overlap as little as possible in relationship to functionality.
- Regression faults are minimized.
  If every component does one function (High Cohesion), making changes won't affect a lot of things.
- Components are more reusable.

Software Metrics

Measurements used to indicate:

- Size (LOC - Lines of Code)
- Duration (Months, years)
- Effort (Person-Months)
- Quality (Fault Density - Faults/1000LOC)
- Efficiency (Faults/Unit of time)
- Reliability (Mean time between failures)

Product/Process

4.1.2 CASE Tools

- UpperCASE or front-end tools - used in the requirements, analysis and design workflows
- LowerCASE or back-end tools - used in the implementation and maintenance activities

Types of CASE Tools

- Data Dictionary - computerized list of all data defined within the product (type and location defined)
  Could be Upper or LowerCASE
- Consistency Checker - tool which checks that everything in the design is in the specification document and everything in the specification document is in the design.
- Report Generator - tool which generates code needed for producing a report.
- Screen Generator - tool which generates the code necessary for a data capture screen.
- Structured Editor - a text editor which is designed to understand the structure of a program in a programming language, aiding in syntax fault prevention or early detection.
- Pretty Printer or Formatter - code often included with the structured editor which makes use of the language syntax structure to display the code in a standard manner (indenting, highlighting reserved words or comments)
  Try Sublime Text 2!
- On-line Interface Checker - editor know every subprogram declared within the product and their parameter lists.
- Operating System Front End - tool which allows the programmer to give commands to the operating system from within the editor.
- Source Level Debugger *LowerCASE*
- Interactive Source Level Debugger *LowerCASE*
- Version Control Tool - keeps detailed record of each version of the project.
  Check out Git with GitHub or BitBucket!
- Configuration Control Tool - manages multiple variations. *LowerCASE*

Grouped CASE Tools

- CASE Workbench - collection of CASE tools that together support one or two activities.
- CASE Environment - collection of CASE tools which support the entire software process.

4.2 Software Versions

- During maintenance, at least two versions of the product will exist: the old version and the new version.
- Revision - what we call the new version.
- Configuration - the specific version of each artifact from which a given version of the complete product is built.

Chapter 5

Unified Process

5.1 Phases of Unified Process

5.1.1 Inception Phase

- Determine whether it's worthwhile to develop the target product
- Determine economic viability
- Steps
  1. Obtain domain knowledge
  2. Build business model
     Understand how the client operates within the domain.

Questions to consider:

1. Is the proposed software cost effective?
2. Can the proposed software be delivered on time?
3. What risks are involved in developing the software and how can these risks be mitigated?

Identify Risks

1. Technical risks
   - Necessary experience?
   - New hardware needed and will it be delivered on time?
   - Software tools needed?
2. Not getting the requirements right
3. Not getting the architecture (design) right

Documentation in the Inception Phase

- Initial version of the domain model
- Initial version of the business model *How the client operates in their domain.*
- Initial version of the requirements artifacts
- Preliminary version of the analysis artifacts
- Preliminary version of the architecture
- Initial list of risks
- Initial use cases
- Plan for elaboration phase
- Initial version of the business case
  Document that describes the cost-effectiveness of taking on the project

5.1.2 Elaboration Phase

- The aim is to refine the requirements, refine the architecture, monitor the risks, refine the business case, and produce the software project management plan.
- Major activities are refinements of the previous phase.

Elaboration Phase Deliverables

- Completed domain model
- Completed business model
- Completed requirements artifact
- Completed analysis artifacts
- Updated version of the architecture
- Updated list of risks
- Software Project Management Plan (SPMP)
- Completed business case

5.1.3 Construction Phase

- Aim is to produce the first operational-quality version of the product (beta release)
- Emphasis is on implementation and testing
- Components are coded and unit tested
- Components are combined (integrated) and tested again.

Construction Phase Deliverables

- Initial user manual and other manuals as needed
- All the artifacts of the beta release version
- Completed architecture
- Updated risk list
- Software Project Management Plan (SPMP)
- Updated business case if needed

5.1.4 Transition Phase

- Aim is to ensure that the client's requirements have been met
- Driven by feedback from the locations where the beta version is installed and tested
- Manuals are completed
- Faults are corrected
- Discover any unidentified risks

Transition Phase Deliverables

- All the artifacts of the final version
- Completed manuals

Refer to pg. 88 of the book.

Chapter 6

Testing

Testing - continual process carried on during the entire life cycle of software.

V and V

- Verification - determining whether a phase has been correctly carried out (at the end of the phase)
- Validation - testing just before a product is delivered to the client to determine if it satisfies the specifications

Use of the Term Testing We use the term testing instead of V and V because they imply that testing can wait until certain points in the process like the end of a phase, when testing must occur throughout the process.

6.0.5 Fault, Failure, Error, Defect

- Fault - human mistake created in software
- Failure - observed incorrect behavior due to a fault
- Error - amount by which a result is incorrect
- Defect - generic term used to encompass fault, failure, and error

6.1 Software Quality

- Extent to which the product satisfies its specifications
- Adherence to specifications
- Must be ensured at all times, not just added at the end of the product

- Every software developer must be personally responsible for the quality of their work.

6.1.1 Software Quality Assurance

SQA group oversees the quality throughout the process

- Adherence to standards
- Correctness of each workflow
- Independent of the development team

6.1.2 Categories of Testing

1. Non-Execution-Based Testing (Reviews)
    - Walkthroughs
        - 4-6 person team (Experience Senior Staff)
            * Manager of the current workflow
            * Member of the team on the current workflow
            * Member of the team on the subsequent workflow
            * Client Representative
            * SQA group member
        - Material distributed to allow participants to prepare
        - Each reviewer develops two lists: things they don't understand, things they think are incorrect
        - Purpose is to find faults in a workflow or document such as the specification or the design document and create a list of faults for later correction

          Ways to conduct a walkthrough
          (a) Participant Driven - members present their lists to the team creating the document being tested which respond to each item (clarify questions, repair faults)
          (b) Document Driven - chair walks the members through the document with stops at the unclear or possible faults
    - Inspections
        - Proposed by Fagan in 1976 for testing designs and
        - Consists of five formal steps
        - Conducted by team of size 4-5
            * Moderator - both manager and leader of the inspection team
            * Reader - leads team through the document
            * Recorder - produces written report of detected faults

        * Implementer/Tester
    - Steps:
        (a) Overview - presentation by one individual responsible for producing it after which given to participants
        (b) Preparation - participants try to understand the document using lists of fault types and create checklist
        (c) Inspection - one participant walks through using the checklists with the moderator creating a written report
        (d) Rework - using the written report of faults, the individuals which created the document resolve the faults
        (e) Follow-Up - moderator makes sure that everything in the report has been corrected or addressed and that no new faults were created in repairing the original faults

Comparison of Walkthroughs and Inspections

- Walkthroughs are 2 steps and inspections are more formal 5 steps
- Inspections generally take longer but the extra time and effort spent can be cost effective

2. Execution-Based Testing

6.2 Reviews

6.2.1 Strengths of Using Reviews

1. Effective way to find faults in the specification document or the design document.

2. Faults detected earlier in the process are less costly to repair or remove.

6.2.2 Metrics for Inspections

1. Inspection Rate - the number of pages inspected per hour (for specifications and designs) or lines of code inspected per hour (for code inspection)

2. Fault Density - number of faults per page or number of faults per KLOC

3. Fault Detection Rate - number of major and minor faults detected per hour

4. Fault Detection Efficiency - number of major and minor faults detected per person-hour

These methods are used to determine whether the inspection process is working.

Goals and Limits of testing

- Goal: To maximize the number and severity of defects found per dollar spent
- Limit: Testing can only determine the presence of defects, never their absence. Correctness proofs are the only manner of proving the absence of defects.

6.2.3 Levels of Testing

1. Unit Testing - The earliest type of testing, testing the parts of the application (functions, modules, module combinations)
2. Integration Testing - Validates the overall functionality of each stage
3. Acceptance Testing - Validates the final product

6.2.4 Steps for Unit Testing

1. Plan the general approach, resources, and schedule.
2. Determine requirements-based characteristics to be tested.
3. Refine the general plan.
4. Design the set of tests.
5. Implement the refined plan and design.
6. Execute the test procedures.
7. Check for termination.

6.2.5 Test Types

- Black Box - Check correct output for given input as specified by the requirements
- White Box (or Clear Box) - Design test of cover all paths through the application.
- Gray Box - Consider limited details of the application with some of the Black Box techniques.

Black Box Testing

- Equivalence Partitioning - Division of the test input data into subsets
- Boundary Value Analysis - Testing for values which separate acceptable and unacceptable ranges.

White Box Testing

- Statement Coverage - Every statement should be executed by at least one test.
- Decision Coverage - Every branch of decision statements is executed.

6.2.6 Planning Unit Tests

1. Decide what the units are to be and who will test them.
2. Decide how unit tests will be documented.
3. Determine the extent of unit testing.
4. Decide how and where to get the test input.
5. Estimate the resources required.
6. Arrange to track time, number, type, and source of defects.

6.2.7 Unit Testing of Methods

1. Verify operation at normal parameter values.
2. Verify operation at limit parameter values.
3. Verify operation outside parameter values.
4. Ensure that all instructions execute.
5. Check all paths - both sides of branches.
6. Check the use of all called objects.
7. Verify the handling of all data structures.
8. Verify the handling of all files.
9. Check normal termination of all loops.
10. Check abnormal termination of all loops.
11. Check normal termination of all recursion.
12. Check abnormal termination of all recursion.
13. Verify the handling of all error conditions.
14. Check timing and synchronization.
15. Verify all hardware dependencies.

6.2.8 Class Testing

1. Method Combination test
2. Attribute-Oriented Test
3. Testing Class Invariants
4. State-based Tests

6.2.9 What is Tested During Execution Based Testing?

- Utility - How useful it is or how it meets the needs of the user
- Reliability - Measuring the mean time between failures; reliable software rarely fails
- Robustness - valid output for valid input, error conditions handled correctly (satisfying the specifications)
- Performance - How well it meets the constraints
- Correctness - Valid output for valid input

Chapter 7

Modules and Objects

7.1 Modules

Lexically contiguous sequence of statements bounded by boundary symbols, having an associated name.

Cohesion and Coupling

- Cohesion - Measures of software quality relating to modules or classes.
- Coupling - Measures used to quantify characteristics like reusability, reliability, etc.

7.1.1 Cohesion

- The degree of interaction within a module
- A description of the logical relationship between elements of a class
- The type of relationship that exists among the elements of each software entity.

Types of Cohesion

1. Coincidental - No meaningful relationships among the elements of an entity. Difficult to describe the module's function(s)
2. Logical - Performs a series of related actions, one of which is selected by the calling module.
3. Classical or Temporal - Performs a series of actions related in time
4. Procedural - Performs a series of actions related by the sequence of steps to be followed by the product.

5. Communicational - Performs a series of actions related by the sequence of steps to be followed by the product performed on the same data.
6. Informational - Performs multiple functions, each with its own entry point, with independent code for each action, all performed on the same data structure.
7. Functional - Performs a single function or action.

(Note: #3-5 are sometimes called Flowchart Cohesion.)

Notes on Cohesion

- As the cohesion number increases, so do the cohesion level and the desirability.
- Functional cohesion enhances reusability.
- Maintenance is easier to perform on a module with functional cohesion.

7.1.2 Coupling

- The degree of interaction between two modules.
- The type of relationship that exists between two software entities.

Types of Coupling

1. Content - One software entity references the contents of another entity
2. Common - Software entities reference a shared global data structure
3. External - Software entities reference the same externally declared symbol
4. Control - One entity passes control elements as arguments to another entity
5. Stamp - A data structure is passed but not entirely used.
6. Data - One entity calls another and are not coupled as described above (every parameter passed is simple or a data structure entirely used)

7.1.3 Abstraction

1. Data Abstraction
2. Procedural Abstraction

A means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details.

7.1.4 Data Encapsulation

- Data structure along with the actions or operations to be performed on that structure
- An example of abstraction

Procedural Abstraction

Conceptualizing in terms of giving a name to a set of high-level actions which are specified by the body of the procedure.

Information Hiding

- Better named ”detail hiding”
- Concept introduced by Parnas
- Hiding the implementation details of a module from another using it

Abstract Data Type (ADT)

- Specification of a data type along with the operations on that type
- Supports both data and procedural abstraction

Three Concepts Important to the OO Paradigm

1. Inheritance*
2. Polymorphism*
3. Dynamic Binding

** - These are especially important*

Chapter 8

Reusability and Portability

8.1 Reuse

<u>Reuse</u> - Taking components of one product to be used in a different product with different functionality

8.1.1 Types of Reuse

1. **Accidental** or **Opportunistic** Reuse - Developers realize that a component from a previously constructed product can be reused in the new project on which they are currently working.

2. **Deliberate** or **Systematic** Reuse - Components are constructed with the purpose that they will be reused

Advantages of Deliberate Reuse

- Components will be constructed with reuse in mind will be:
  - more likely to be easier and safer to use.
  - better documented.
  - more thoroughly tested.
  - adhere to a uniformity of style.
  - more easily maintained.

Disadvantages of Deliberate Reuse

- Can be expensive to the company
- Can be more time consuming
  - Design

  - Testing
  - Documentation

- May not be reused after time spent making it reusable

8.1.2 Impediments to Reuse

1. Ego
2. Uncertainty of quality
   (Distrust of others' abilities)
3. Economic Reasons
   Afraid you might lose your job/Company won't need you anymore
4. Retrieval
   Can't find it
5. Expense
   To the company
6. Legality
7. For COTS (*Commercial Off-the-Shelf*) components - limited extensibility/modifiability

May occur during design as well as implementation

8.1.3 Types of Design Reuse

1. Company that develops software in one specific domain may set up repository of design components. (Library/Toolkit)
   (a) Tested module designs incorporated
   (b) Reduce design time (and likely implementation)
   (c) Increase design quality
2. Application Frameworks
3. Design Patterns
4. Software Architecture

8.2 Design Patterns

- Christopher Alexander - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."
- Each design pattern is a solution to a general problem in the form of a set of interacting classes that have to be customized to create a specific design.
- Four Essential Elements
  1. Pattern name
  2. Problem it solves
  3. Solution - elements making up the design
  4. Consequences - results and trade offs of applying

8.2.1 List of Design Patterns

1. Abstract Factory
2. Adapter
3. Bridge
4. Builder
5. Chain of Responsibility
6. Command
7. Composite
8. Decorator
9. Faade
10. Factory Method
11. Flyweight
12. Interpreter
13. Iterator
14. Mediator
15. Memento
16. Observer

17. Prototype
18. Proxy
19. Singleton
20. State
21. Strategy
22. Template Method
23. Visitor

8.3 Portability

Software which is significantly easier to modify to run on another platform than it is to code from scratch.

8.3.1 Impediments to Portability

1. Hardware Incompatibilities
2. Operating System Incompatibilities
3. Difference in numeric capabilities (*32-bit v. 64-bit*)
4. Compiler Incompatibilities
5. Data Formats

Chapter 9

Planning and Estimating

9.1 Cost

9.1.1 Cost Estimation

Can occur at:

- Conceptualization Phase
- Requirements Analysis
- Design Phase
- Implementation Phase
- Implementation and Testing

9.1.2 Cost Types

- Internal Cost - cost to developer
  - Salaries
  - Hardware Support
  - Software Support
- External Cost - ???

9.2 Size

Metrics for Size

1. LOC - Lines of Code
2. KDSI - Thousand Delivered Source Instructions

3. Number of Operands and Operators
4. FFP - Files, Flows, and Processes
5. FP - Function Points

9.2.1 LOC and KDSI

Problems With Using LOC or KDSI

1. Creation of source code is only small part of total effort
2. Same product in different languages would have different LOC
3. Some languages have no concept of a line of code (LISP)
4. Question as to whether only to count executable statements (omit declarations, comments, JCL (Job Control Language), reused code?)
5. Not all code that is written is delivered
6. Must wait until the project is finished to get the measurement
7. Code generation tools

9.2.2 FFP

- File - collection of logically or physically related records, permanently resident in the product (transaction and temporary files do not count here)
- Flow - data interface between the product and the environment (screen or report)
- Process functionally defined logical or arithmetic manipulation of the data (sorting, validating, updating, etc.)

9.2.3 FP

Function Points - used to assess the size of a project

1. Identify the functions the application must have.
2. For each function, compute:
   (a) External Inputs - EI
   (b) External Outputs - EO
   (c) External Inquiries - EIN
   (d) Internal Logical Files - ILF
   (e) External Logical Files - ELF

3. Multiply the numbers from step 2 by specified values (see p. 274) This results in an unadjusted function point value (UFP).
4. Compute the Technical Complexity Factor (TCF) using the effect of the 14 general characteristics of the project which have been assigned a value from 0-*not present or no influence* to 5-*strong influence throughout*. The 14 numbers are summed using the Total Degree of Influence (DI). Then the TCF is given by .65+.01*DI (see p.274)
5. FP = UFP * TCF

Converting Function Points to Code

Once the function point value is computed, an estimate for the lines of code by multiplying by a constant associated with the language we plan to use for the application.

Techniques for Estimation of Size

- Use comparisons with past jobs
- Use function point methods
  - Compute unadjusted function points
  - Apply adjustment process
- Use LOC estimates to compute labor and duration using COCOMO formulas.

9.2.4 COCOMO

- Estimation method devised by Boehm in 1981
- COnstructive COst MOdel
- Depends on the LOC of a project
-

9.3 Cost Estimation Techniques

1. Expert Judgment by Analogy - Consult a number of experts who have completed similar type projects.
2. Bottom-up - Estimate each component from the leaf level and add, going upward in design.
3. Algorithmic Cost Estimation - Use FF or FP and comu.

9.3.1 Software Project Management Plan

- Plan which describes the proposed software development in full detail. (*See p. 285*)
- Consists of three main components:
    1. The work to be completed
    2. The resources to be used to do the work
    3. Funding

Work to be Completed

1. Project function - work which continues throughout the project and is not related to any specific workflow.
2. Activity - work which is related to a specific phase or workflow and has a definite beginning and end date

Resources

1. People (varies during the duration of the project)
2. Hardware
3. Support Software (include CASE tools)

Chapter 10

Requirements

10.1 Requirements Analysis

- Missed this. Get it from someone.

10.1.1 Levels of Requirements Analysis

- C-Requirements - Documents what the customer wants and needs in a language clear to the customer (Customer Requirements)
- D-Requirements - Documents the requirements in a specific, structured form (Developer Requirements)

10.1.2 Why Write Requirements Documents?

Without such a document...

1. Team does not really know the goal it is trying to accomplish.
2. Team cannot inspect its work properly
3. Team cannot test its work properly
4. Team cannot track its productivity
5. Team cannot get adequate data on its practices
6. Team cannot predict the size and effort of the next job
7. Team cannot satisfy customer

10.1.3 Each Requirement Must Be...

- Expressed properly
- Made easily accessible
- Numbered
- Accompanied by tests that verify it
- Provided for in the design
- Accounted for by code
- Tested in isolation
- Tested in concert with other requirements
- Validated by testing after the application has been built

10.1.4 Requirements Analysis Process

1. Identify the customer
2. Interview customer representatives
    - Identify wants and needs
    - Exploit tools for expression
    - Sketch GUI's
    - Identify hardware
3. Write C-requirements (Review with customer)
4. Inspect C-Requirements
5. Build D-requirements (After customer approval)

10.2 Sources of Requirements

- Customer Interviews
- Copies of currently used forms or software
- Customer site visit
- Surveillance cameras
- Rapid prototype

10.3 Describing C-Requirements

1. Use Cases
2. Data Flow Diagrams
3. State-Transition Diagrams
4. Drafting User Interfaces (Rapid Prototype)

10.3.1 Use Cases

A concept invented by Jacobson used to express customer requirements in the form of interaction between an actor (type of user) and the application. It is identified by its name and by the type of user of the application (actor). Similar to a user story in Extreme Programming. *See Pg. 318, Fig. 11.1*

10.3.2 Data Flow Diagrams

- Requirements presented as the flow of data among processing elements
- Nodes represent processing units
- Arrows denote the flow of data
- Data stores are denoted by a pair of horizontal lines
- External agencies are represented as rectangles

10.3.3 State-Transition Diagrams

- Application exists as states or situations
- States are represented as nodes
- Transitions between states are represented with arrows
- Can also be used for a design tool

10.3.4 Developing User Interfaces

1. Know your user
   (a) Level of knowledge and experience
      i. Computer literacy
      ii. System experience
      iii. Experience with similar applications
      iv. Education
      v. Reading level

        vi. Typing skill

    (b) Physical characteristics

        i. Age
        ii. Gender
        iii. Handedness
        iv. Physical handicap

    (c) Characteristics of user's tasks and jobs

        i. Type of use (discretionary v. mandatory)
        ii. Frequency of use
        iii. Turnover rate for employees
        iv. Importance of task
        v. Repetitiveness of task
        vi. Training anticipated
        vii. Job Category

    (d) Psychological characteristics of user

        i. Probable attitude toward job
        ii. Motivation
        iii. Cognitive style
            - Verbal v. Spatial
            - Analytic v. Intuitive
            - Concrete v. Abstract

2. Understand the purpose of the user interface in terms of the application's overall purpose.

3. Understand the principles of good screen design.

    (a) Consistency among screens
    (b) Anticipate where the user will usually start
    (c) Apply a hierarchy to emphasize the order of importance
    (d) Apply principles of aesthetics

4. Select the appropriate kind of window

    (a) Property window - display properties of an entity
    (b) Dialog window - obtain information (Input)
    (c) Message window - provide information (Output)
    (d) Palette window - present a set of controls
    (e) Pop-up window - amplify information

5. Develop system menus

   (a) Provide a main menu (between 5 and 9)
   (b) Display all relevant alternatives
   (c) Match the menu structure to the structure of the application's tasks

6. Select the appropriate device-based controls
   This refers to the physical means by which the user communicates their desires with the application (Joysticks, trackballs, graphics tablets, touch screens, mice, microphones, and keyboards.)

7. Select the appropriate screen-based controls
   Icons, buttons, text boxes, radio buttons
   Between 5 and 9, or organized into groups

8. Organize and lay out the window

9. Choose appropriate colors

10.4 CASE Tools for Requirements

- Graphical tool for drawing data flow diagrams
  - + Easier to change a diagram stored in CASE tool than to redraw
  - + Details of the product are stored in the CASE tool so the documentation is always available and updated
  - - Not always user friendly
  - - Almost impossible to program a computer to draw as pleasing a diagram as a human
- Text Editor
- Survey construction tool
- Screen generator (For rapid prototyping)

Chapter 11

Writing the Specification Document

11.1 Specification Document

- Serves as contract between client and developer
- Must be clear and understandable by the client
- Must be complete and detailed for the design team
- Specifies exactly what the product is to do
- Describes the constraints of the system
- Describes acceptance criteria Helps team select solution strategy

11.1.1 Constraints

1. Timing
2. Storage
3. Security
4. Response Time
5. Portability
6. Reliability
7. Parallel Running
8. Deadlines

11.1.2 Acceptance Criteria

- Set of series of tests which can be used to prove the product satisfies the specifications
- Indicates the developer's job is completed
- May be restatements of the constraints

11.1.3 Solution Strategy

- General approach to building the product
- Records are maintained of discarded strategies and why they were discarded. (This will help if the team must justify to SQA why they selected the method they did. It will also aid during maintenance.)

11.2 Specification Document Writing Techniques

1. Informal Methods
    - Natural language, easy to learn
    - Easy for client to understand
    - Easy to incorporate ambiguities, contradictions
2. Semiformal Methods
    - More precise than informal
    - Client can understand
    - Cannot handle timings
3. Formal Methods
    - Most precise
    - Reduce faults
    - Support correctness proofs
    - Difficult to learn
    - Almost impossible for client to understand

11.2.1 Semiformal Methods

1. Data Flow Diagram (DFD) - Gane and Sarsen's graphical technique
2. Problem Statement Language/Problem Statement Analyzer (PSL/PSA) - Computer aided technique

3. Structured Analysis/Design Technique (SADT) - box and arrow diagramming language
4. Software Requirements Engineering Model (SREM) - useful for specifying real-time systems and embedded systems
5. Entity Relationship Modeling (ERM) - data oriented technique useful for specifying databases.

11.2.2 Formal Methods

Easier to validate and code

1. Finite State Machine (FSM)
2. Petri Net - Carl Adam Petri in 1962
3. Z */'zed/* - Formal specification language
4. Anna - formal specification language for Ada
5. Vienna Definition Language (VDL) - based on denotational semantics

Petri Net

See pg. 382

- Useful for concurrent processes and synchronization
- Nondeterministic
- C = P, T, I, O
  - P - Finite set of **places**
  - T - Finite set of **transitions**
  - I - **Input** function mapping T to bags of places
  - O - **Output** function mapping T to bags of places

Z

Four sections:

- Given sets (sets that need not be defined in detail), data types and constants
- State Definition
  Schema - group of variable declarations together with a list of predicates that constrain the possible values of the variables
- Initial State - description of the system when first turned on
- Operations

Analysis of Z

- Easy to find faults in specification written in Z
- Requires writer to be concise
- Allows for proofs
- Not too difficult to learn
- Has decreased the cost of software development
- Easy to translate into natural language for the client

11.3 Testing during Specification Phase

- Walkthroughs
- Inspections
- Correctness proofs

11.4 CASE Tools

- Graphical drawing tool for DFD or Petri Nets
- Data dictionary
- Combined drawing tool and data dictionary (And possibly a consistency checker)
  - Analyst/Designer
  - Software through pictures
  - System Architect

11.5 Metrics

- Size - number of pages in the document
- Quality - measure the number of faults of each type found during inspection
- Effort - counts of files, data items, and operations

Chapter 12

Design

12.1 Implementation

- The process of translating the detailed design into code.
- Unit implementation refers to coding the smallest part of the project which can be separately maintained.

12.1.1 Selection of a Programming Language

1. Client Requests
2. ONly one available on the hardware
3. Most Suitable
    - For the application
    - For the environment
    - For the cost

12.1.2 Goals of Implementation

1. Satisfy the requirements specified in the detailed design.
2. Conform to standards
3. Document work
4. Maintain records of time, defects
5. Promote maintainability

12.1.3 Steps for Implementation

1. Plan the structure of the code (complete detailed design)
2. Self inspect the design or structure
3. Type in the code complying to standards
4. Self inspect your code
5. Compile

12.1.4 Principles of Sound Implementation

1. Try to reuse already existing code if possible.
2. Enforce intentions (Code not to be used anywhere else in the application or to be used in other parts?)
3. Use qualifiers like final, const, and abstract to enforce intentions
   (a) Final classes can't have descendants
   (b) Final methods can't be overridden in inherited classes
   (c) The value of final variables can't be changed
4. Make members inaccessible if they are not specifically intended to be accessed directly
   (a) Make attributes private. Access through public accessors
   (b) Make methods private if they are for use only by methods of the same class.
5. Include examples in documentation
6. List methods alphabetically rather than trying to find a calling order among them. (You may group private, protected, and public methods.)

12.2 Programming Practices

12.2.1 Sound Programming Practices

1. Use consistent and meaningful variable names
2. Use prologue comments for each function giving input, actions, output, author and date, modification and date, files accessed, date tested and approved by
3. Use no more than one statement per line
4. Make good use of white space

5. Restrict levels of nesting to three or four
6. Be consistent with indenting
7. Read in parameters instead of hard-coding constants

12.2.2 Hints for Working with Pointers

- Avoid using pointer parameters in C++, use references instead.
- Never return a pointer to a new heap reference in C++.
- Collect your garbage

12.2.3 Hints for Working with Functions

- Avoid type inquiry. Use virtual functions instead.
- Avoid C++ friend functions except when obviously beneficial.
- Be careful overloading operators.

12.2.4 Hints on Working with Exceptions

- Catch only those exceptions you know how to handle.
- If the present method cannot handle the exception, there has to be a handler in an outer scope that can do so.
- If you can handle part of the exception, then handle and rethrow the exception for handling in the outer scope

12.2.5 Hints on Error Handling

1. Follow agreed-upon development process; inspect
2. Consider introducing classes to encapsulate legal parameter values.
3. Where error handling is specified by requirements, implement
4. For applications that must never crash, anticipate all possible implementation defects (use defaults)
5. Follow a consistent policy for checking parameters

12.3 Programming Standards

12.3.1 Examples of Programming Standards

1. Naming Conventions
    (a) Name with concatenated words
    (b) Begin class names with capitals
    (c) Begin variables with lowercase letters
2. Documentation for Methods
    (a) Preconditions
    (b) Postconditions
    (c) What the method does
    (d) What parameters must be passed
    (e) What exceptions it throws
    (f) Reason for choice of visibility
    (g) Ways in which instance variables are changed
    (h) Known bugs
    (i) Test description
    (j) HIstory of changes
3. Within Methods
    (a) Perform only one operation per line.
    (b) Try to keep the size to one screen.
    (c) Use parentheses within expressions even if they are not needed by the syntax.
4. Documenting Attributes
    (a) Provide all application invariants
    (b) State its purpose

12.4 Tools and Environments for Programming

- IDE (interactive Development Environment) - used for allowing programmers to produce more code in less time
- Drag-and-Drop facilities for forming GUI components
- Graphical representation of directories
- Debuggers
- Wizards

12.5 Defect Severity Classification

- Major - Requirements not satisfied
- Medium - Neither major nor trivial
- Trivial - a defect that will not affect operation or maintenance, but just feels a little ”meh”

12.6 Integration

12.6.1 Terms

- Iteration - a coherent stage of construction
- Build - one of sever carefully planned activities within an iteration

12.6.2 Roadmap for Integration

1. Decode on the extent of all tests.
2. For each iteration:
    - For each build
        (a) Perform regression testing prior to build
        (b) Retest functions if necessary
        (c) Retest modules if necessary
        (d) Test interfaces if required
        (e) Perform build integration tests
    - Perform iteration system and usability test
3. Perform installation tests
4. Perform acceptance tests

12.7 Testing

1. Integration testing
2. Interface testing
3. System testing
4. Usability testing
5. Regression testing
6. Acceptance testing
7. Installation testing

12.7.1 Integration Testing

1. Decide how and where to store, reuse and code the integration test
2. Execute as many unit tests as time allows
3. Exercise regression testing
4. Ensure build requirements are properly specified
5. Exercise use cases that the build should implement (Test against the SRS)
6. Execute the system tests supported by this build

Artifacts Involved in Integration Testing

1. Use case model - set of use cases describing the typical usage of the application and sequence diagrams
2. Test cases
3. Test procedures
4. Test evaluation
5. Test plan
6. Test components - source code for tests
7. Defects - report on defects discovered, classified

12.7.2 Interface Testing

Get this bullet from someone.

12.7.3 System Testing

- Black box which validate the entire application against the requirements
- *Messed this line up. Get it from someone.*
- *Messed this line up. Get it from someone.*

Types of System Testing

1. Reliability/Availability (MTBF)
2. Volume (large amounts of input)
3. Usability
4. Performance

5. Configuration
6. Compatibility
7. Security
8. Resource usage
9. Installability
10. Recoverability
11. Serviceablity

Usability Testing

Attributes for Usability Testing

1. Accessibility - ease of navigation
2. Responsiveness - average time taken to accomplish specified goals
3. Efficiency - how minimal are the require steps to selected functionality
4. Comprehensibility - how easy to understand and use with documentation and help

12.7.4 Regression Testing

Verifying that a project continues to pass the same designated set of system tests that it passed before a change was made.

12.7.5 Acceptance Testing

- Designed to assure the customer that the project agreed upon has, in fact, been built.
- Witnessed officially by a customer representative
- May be the same as some of the system tests

12.7.6 Installation Testing

- Testing the application in the actual target hardware organization where it will be executing
- Perform the same system tests used during integration

12.8 Test Documentation Standard

1. Introduction
2. Test plan - items, scope, approach, resources, schedule, personnel
3. Test design - items, approach and plan in detail
4. Test cases
5. Test procedures - steps for setting up and executing the test cases
6. Test item transmittal report - item under test, physical location of results, person responsible for testing
7. Test log - chronological record, physical location of test, tester name
8. Test incident report - record of an event during testing requiring further investigation
9. Test summary report

12.9 Transition Iterations

- Find defects through customer use
- Test user documentation and help
- Determine realistically whether application meets requirements
- Retire deployment risks
- Satisfy miscellaneous marketing goals

12.10 Alpha Release

- After integration, project is given to in-house users for early prerelease use
- Purpose to provide feeback, defect information and advanced knowledge of the product

12.11 Beta Release

- After integration - project is given to a part of the customer community with the understanding that they report defects found.
- Used to convince customers that there really is a product behind the vendor's promises.

Chapter 13

Post-Delivery Maintenance

Why is Post-Delivery Maintenance Necessary?

1. Corrective Maintenance - correcting a fault
2. Perfective Maintenance - improving the effectiveness of the product

13.1 Facts About Post-Delivery Maintenance

1. More time spent on post-delivery maintenance than any other activity
2. Approximately 67% of the product cost
3. The most challenging of all aspects of software production
4. Incorporates aspects of all of the other workflows of the product.
5. The most thankless development task

13.2 Problems with Post-Delivery Maintenance

1. Product may have not been developed with maintenance in mind.
2. Not considered glamorous
3. Task often given to beginners, not the more experienced programmers
4. Must avoid creating more faults when correcting one

13.3 Requirements for Post-Delivery Maintenance Programmers

1. Ability to locate the cause of a fault

2. Ability to function without good documentation
3. Ability to maintain documentation of changes
4. Ability to perform regression testing
5. Ability to perform requirements analysis, design and implementation for adaptive or perfective maintenance
6. Ability to design tests for new functionality added

13.4 Management of Post-Delivery Maintenance

1. Defect report filed by user
2. Reports may be prioritized
3. Work given to programmer for repair
4. Results tested by SQA and documented
5. Make sure changes do not adversely affect further maintenance
6. Encourage client to avoid the moving target problem

13.5 Special Problems with Object-Oriented Maintenance

1. to understand an object, maintainer must understand the object's hierarchy
2. *Get bullet from someone.*
3. *Get bullet from someone.*

13.6 Reverse Engineering

- The process of recreating the design document and possible the SRS document, given the source code.
- Its need is due to the amount of legacy code in the industry

13.7 CASE Tools for Maintenance

1. Version Control
2. Configuration Control Tool
3. Pretty Printer
4. Graphic tools
5. Display class hierarchy
6. Defect-tracking tool (Bugzilla)

Chapter 14

Detecting Obsolescence

- Obsolescence - the state of being which occurs when an object, service, or practice is no longer wanted even though it may still be in good working order.

14.1 Areas to Consider

A solution may be obsolete if...

- Your system can't handle increased workloads or the way you do business has changed.
- An organization adds headcount instead of reinvesting in technology
- The software engineers from your last project are no longer available
- Your reporting is less than stellar
- The solution doesn't allow you to get back to business quickly
- Your organization is witnessing larger-than-normal turnover or if delivering exceptional customer care is becoming a challenge

14.2 Causes of Software Obsolescence

1. Functional Obsolescence
2. Technological Obsolescence
3. Logistical Obsolescence

14.2.1 Functional Obsolescence

- The software does not perform in the manner for which it was created
- May be cause by hardware changes, requirement changes or other software changes in a larger system

14.2.2 Technological Obsolescence

- Inability to renew licensing agreements
- Software maintenance (support) terminates
- The original supplier no longer markets and sells the software
- More advanced software available
- Software written in obsolete, outdated, or inefficient language

14.2.3 Logistical Obsolescence

- Digital media obsolescence
- Formatting changes

14.3 How to Measure Obsolescence

1. Physical Inspection of the code for inefficient layout, structural deficiencies, excessive or deficient capacities
2. Comparative Analysis of existing software with a newer version of itself (if available)

14.4 Cost of Management of Software Obsolescence

- Mitigation
- Re-development
- Re-qualifying
- Rehosting
- Media Management

14.4.1 Mitigation

- Software License Downgrade - negotiated with the software vendor and enables the users to expand or extend the authorized use of an older product by purchasing additional licenses of the latest version and applying those licenses to the older product
- Source Code Purchase - In some cases, the original vendor may allow the customer to purchase the source code for the product
  This may be negotiated in the original contract for the software.
- Third Party Support - In some cases a third-party can be contracted to continue support of an obsolete software application

14.4.2 Re-Development

- Software is modified to operate correctly with new application hardware (or with other new software). This can range from retesting the software to re-architecting and may also include: re-integration, data migration, re-training and document revision

14.4.3 Re-Qualifying

- Either modified or unmodified legacy software must be tested in a new environment

14.4.4 Rehosting

- Modifying existing software to operate correctly in a new development environment (also called technology porting). Rehosting is applicable to legacy software systems originally created with languages and systems that have, themselves become obsolete.

14.4.5 Media Management

- Storage and maintenance of the medium the software is archived on is a critical element of software obsolescence. There are several problems (and costs) involved that depend on the type of media, the method of storing the media, and version control.

Part II

Glossary

- **Software Engineering** - Application of sound engineering practices to software creation and maintenance.
- **Specification Document** - Explicitly describes what the product is to do and the constraints under which it must operate
- **Ambiguity** - One sentence may have more than one interpretation
- **Incompleteness** - When a relevant fact or requirement is left out
- **Contradiction** - When two places in the specification document are in conflict
- **Architectural Design** - Description of the product in terms of modules
- **Detailed Design** - Description of each module
- **Tracebility** - Each part of the design can be tracked back to a statement in the specification document
- **Product Testing** - When the SQA group tests the product in Implementation Phase of development
- **Maintenance**
  - **Corrective Maintenance** - Bug squashing
  - **Enhancement Maintenance** - Updates
    - **Perfective** - Clients make new demands
    - **Adaptive** - Changes in the environment of the product require changes in the software
- **Regression Testing** - Ensuring that changes have not affected already working functionality
- **The Four P's**
  - **Process** - Sometimes called the life-cycle model or development sequence
  - **Project** - Set of activities needed to produce the required product
  - **People** - Has to do with team organization and management and the team's relationship with clients, users, and management
  - **Product** - Includes the Spec Doc, Design Doc, Source Code, Executable, and User Manuals
- **Structured Programming** - Uses sequence control, iteration, and functions
- **Object Oriented Paradigm** - Use of objects with data and functionality which can represent real-world entities

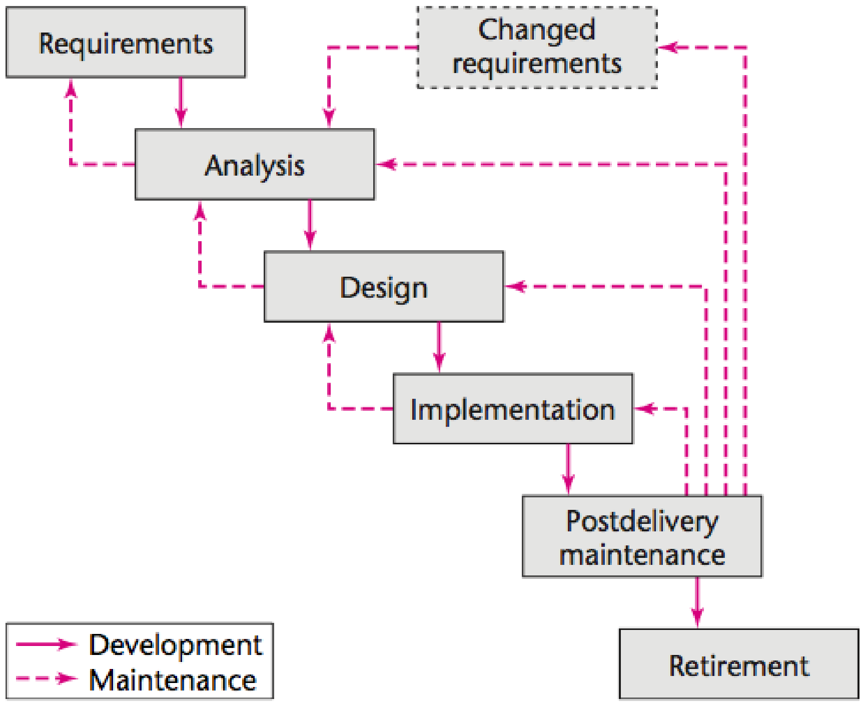

Figure 14.1: The Waterfall Method

- **Design Patterns -** Stock of reusable design elements
- **Development Methods**
  - **Waterfall Method -** No phase is complete until documentation for that phase has been completed and approved by SQA group
  - **Rapid Prototype Model -** Construction of a functional subset of the desired product in order to allow the client and developer to interact
  - **Incremental Build Model -** Software is implemented, integrated, and tested as a series of incremental builds
    - **Incremental Build -** Code piece providing a specific function
  - **Spiral Model -** A variation of the Waterfall Method in which each phase is preceded by risk analysis in an attempt to control risk
- **Teams**
  - **Team -** Group of professionals organized in order to complete the task of creating a large software project

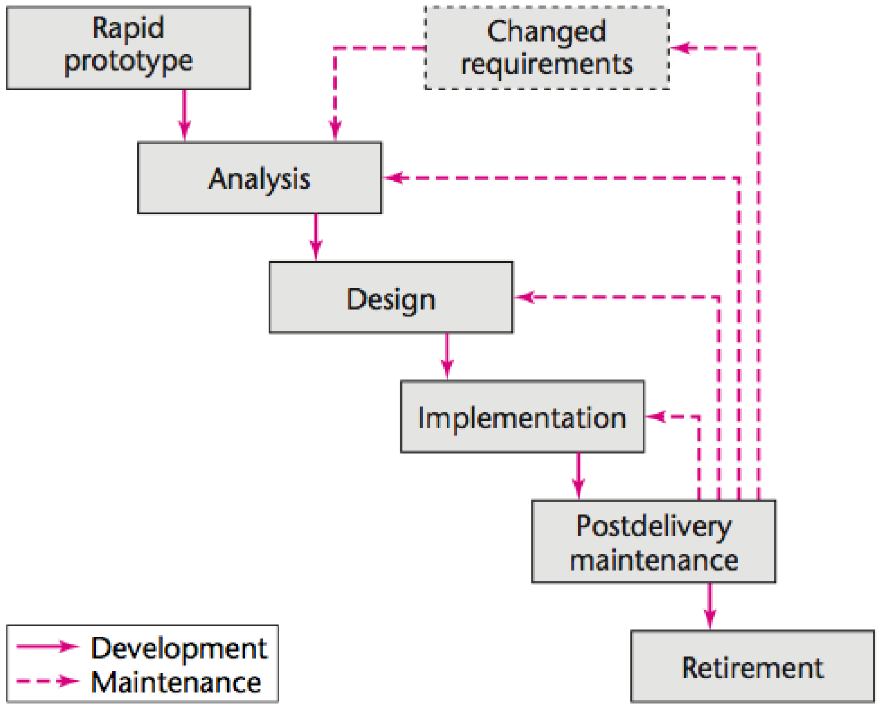

Figure 14.2: The Rapid Prototype Method

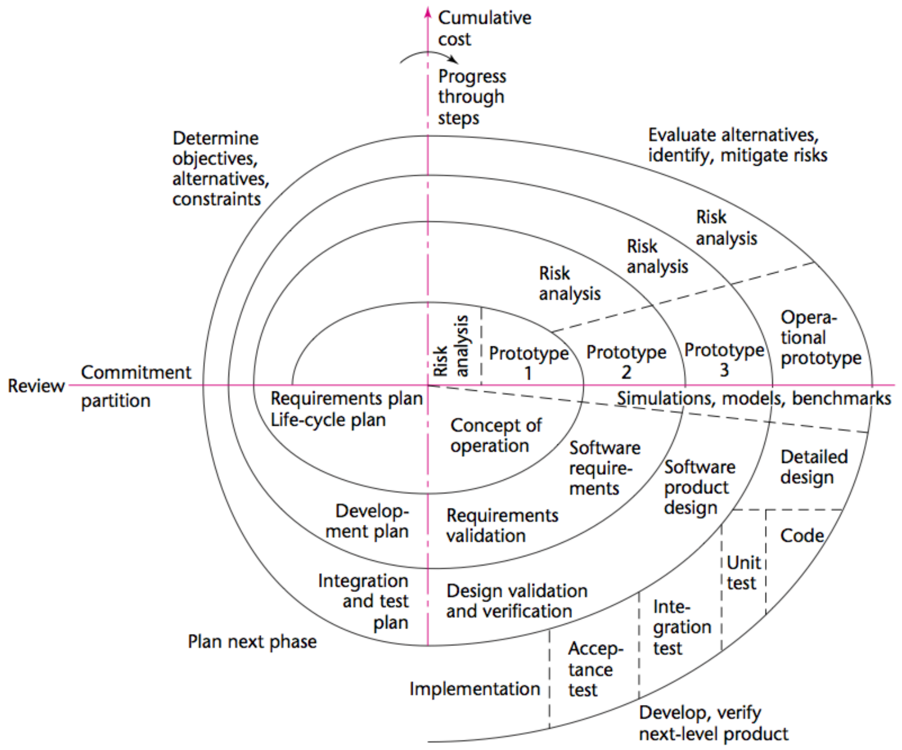

Figure 14.3: The Spiral Model

  - **Jelled Team** - Group of people who are so tightly knit that the attitude is that the whole is greater than the sum of the parts
  - **Team Roles**
    * **Team Leader** - Responsible for overseeing all aspects of the team project, holds the tie-breaking vote
    * **Technical Lead** - Expert on all technical aspects of the project, in particular the hardware and software used for development
    * **Designer** - Designs the project and breaks the project into smaller pieces (modules) for the programmers
    * **Lead Programmer** - needs to have an understanding of the project as a whole; organizes all of the other programmers
    * **Technical Writer** - Writes all documentation for the project
    * **Configuration Management** - Maintains the code base for the project, could include CVS responsibilities
    * **Quality Assurance (SQA)** - Writes, maintains, and conducts all testing associated with the project

- **Stepwise Refinement** - Process whereby a project is successively decomposed into more detailed instructions
- **Divide-and-Conquer** - Break a large problem into smaller subproblems that should be easier to solve
- **Separation of Concerns** - Process of breaking a software project into components which overlap as little as possible in relationship to functionality
- **CASE Tools** - Computer-Aided Software Engineering
  - **UpperCASE** - Front end; used in requirements, analysis, and design workflows
  - **LowerCASE** - Back end; used in implementation and maintenance activities
  - **Data Dictionary** - Computerized list of all data defined within the product (Type and location defined)
  - **Consistency Checker** - Tool which checks that everything in the design is in the spec doc and everything in the spec doc is in the design
  - **Report Generator** - Tool which generates the code necessary for producing a report (Memory Dumps, etc. ?)
  - **Screen Generator** - Tool which generates the code necessary for a data capture screen
  - **Structured Editor** - Text editor designed to understand the structure of a program in a programming language, aiding in syntax fault prevention or early detection

  - **"Pretty Printer"/Formatter** - Code often included with the structured editor which makes use of the language syntax structure to display the code in a standard manner (Indenting, syntax highlighting, etc.)
  - **On-Line Interface Checker** - Editor knows every subprogram declared within the product and its parameter lists
  - **OS Front End** - Tool which allows the programmer to give commands to the OS from within the editor
  - **Source Level Debugger**
  - **Interactive Source Level Debugger**
  - **Version Control Tool** - Keeps detailed record of each version of the project
  - **Configuration Control Tool** - Manages multiple variations

- **Configuration** - Specific version of each artifact from which a given version of the complete product is built
- **Verification** - Determining whether a phase has been correctly carried out (Takes place at the end of the phase)
- **Validation** - Testing just before a product is delivered to a client to determine if it satisfies the specifications
- **Inception Phase** - Determine whether it is worthwhile to develop the target product
- **Elaboration Phase** - Refine the requirements and architecture, monitor the risks, refine the business case, and produce the Software Project Management Plan (SPMP)
- **Construction Phase** - Produce the first operation-quality version of the product (Beta Release)
- **Transition Phase** - Aim is to ensure that the client's requirements have been met
- **Defects**
  - **Fault** - Human mistake created in software
  - **Failure** - Observed incorrect behavior due to a fault
  - **Error** - Amount by which a result is incorrect
  - **Defect** - Generic term which encompasses Fault, Failure, and Error
- **Software Quality** - Extent to which the product satisfies its expectations

- **Participant Driven Walkthrough -** Members present lists of things they do not understand and things they think are incorrect to the team that is creating the document that is being tested. The team responds to each item.
- **Document Driven Walkthrough -** Chair walks the members through the document with stops at the unclear points or possible faults
- **Inspections**
  - **Overview -** Presentation by one individual responsible for producing it, after which it is given to the participants
  - **Preparation -** Participants try to understand the documents using lists of fault types and create checklists
  - **Inspection -** One participant walks through, using the checklists, while the moderator creates the written report
  - **Rework -** Using the written report of faults, the team resolves the faults
  - **Follow-up -** The moderator makes sure that everything in the report has been corrected or addressed and that no new faults were created in repairing the original faults.
- **Inspection Rate -** Number of pages inspected per hour (for specifications and designs) or LOC/hr (for code inspection)
- **Fault Density -** Number of faults per page or per KLOC
- **Fault Detection Rate -** Number of major and minor faults detected per hour
- **Fault Detection Efficiency -** Number of major and minor faults detected per person-hour
- **Unit Testing -** Earliest types of testing, testing the parts of the application (functions, modules, module combinations)
- **Integration Testing -** Validates the overall functionality of each stage
- **Acceptance Testing -** Validates the final product
- **Black Box Test -** Checking correct output for given input as specified by the requirements
- **White Box Test -** Design test to cover all paths through the application
- **Gray Box Test -** Consider limited details of the application with some of the black box techniques
- **Equivalence Partitioning -** Division of the test input data into subsets

- **Boundary Value Analysis** - Testing for values which separate acceptable and unacceptable ranges
- **Statement Coverage** - Every statement should be executed by at least one test
- **Decision Coverage** - Every brand of decision statements is executed by at least one test
- **Utility** - How useful it is or how it meets the needs of the user
- **Reliability** - Measure the mean time between failures; reliable software rarely fails
- **Robustness** - Valid output for valid input, error conditions handled correctly (satisfying the specifications)
- **Performance** - How well it meets constraints
- **Correctness** - Valid output for valid input
- **Module** - Lexically contiguous sequence of statements bounded by boundary symbols, having an associated name
- **Cohesion** - Degree of interaction within a module
  - **Coincidental** - No meaningful relationships among the elements of an entity
  - **Logical** - Performs a series of related actions, one of which is selected by the calling module
  - **Classical or Temporal** - Performs a series of related actions related over time
  - **Procedural** - Performs a series of actions related by the sequence of steps to be followed by the product
  - **Communicational** - Performs a series of actions related by the sequence of steps to be followed by the product performed on the same data (Passing the same value to each function in Procedural Cohesion)
  - **Information** - Performs multiple functions, each with its own entry point and independent code for each action, all performed on the same data structure
  - **Functional** - Performs a single function or action
- **Coupling** - Degree of interaction between two modules
  - **Content** - Software entity references the contents of another entity
  - **Common** - Software entities reference a shared global data structure

  - **External** - Software entities reference the same externally declared symbol
  - **Control** - One entity passes control elements as arguments to another entity
  - **Stamp** - A data structure is passed but not entirely used
  - **Data** - One entity calls another and are not coupled as described above

- **Abstraction** - A means of achieving stepwise refinement by suppressing unnecessary details and accentuating relevant details
- **Data Abstraction** - Data structure along with the actions or operations to be performed on that structure
- **Procedural Abstraction** - Conceptualizing in terms of giving a name to a set of high-level actions which are specified by the body of the procedure
- **Information Hiding** - Hiding the implementation details of a module from another using it
- **Abstract Data Type** - Specification of a data type along with the operations on that type
- **Reuse** - Taking components of one product to be used in a different product with different functionality
  - **Accidental/Opportunistic Reuse** - Developers realize that a component from a previous project can be used in a current project
  - **Deliberate/Systematic Reuse** - Components are constructed with the intent that they will be reusable
- **Portability** - Software which is significantly easier to modify to run on another platform than it is to recode from scratch
- **Internal Cost** - Cost to the developer to develop the product
- **External Cost** - Cost to the client
- **File** - Collection of logically or physically related records permanently resident in the product (Transaction and Temporary files don't count here)
- **Flow** - Data interface between the product and the environment (screen of report)
- **Process** - Functionally defined logical or arithmetic manipulation of the data (Sorting, validating, updating, etc.)
- **Function Point** - Used to assess the size of the project

- **Product Metrics -** Measure some aspect of the product itself, such as its size or reliability
- **Process Metrics -** Used by the developers to deduce information about the software process
- **Component Reuse -** Element may be used by another program's element

Part III

Useful Information

Chapter 15

Test 1

15.1 Draw and Discuss

- Waterfall Method
- Life-Cycle Model

15.2 Definitions

- Traceability
- Cohesion
- Coupling

15.3 Things to Know

- Advantages and Disadvantages of the Waterfall Method (Section 2.1.1)
- Given a project situation, recommend a life-cycle model to use
- Project factors related to programming team structure (Section 3.1)
- Comparison of walkthroughs and inspections (Section 6.1.2)
- Cohesion (Section 7.1.1)
- Coupling (Section 7.1.2)
- Advantages and Disadvantages of Deliberate Reuse (Section 2.0.4 8.1)
- Project Function and Activity (Section 9.3.1)

Chapter 16

Agile Presentations

16.1 Kanban

Ankur Patel

- Japanese for 'Visual Card'
- David Anderson, 2004-2007
- Kanban method was originally created by Taiichi Onho in the late 1940s.
- Implement business concept of JIT at TPS
  Anderson adapted it to software
- Lots of whiteboards, sticky notes, and sections of each software dev cycle to increase visibility of workflow.
- Columns and subcolumns
  - Columns for phases
  - Subcolumns for 'To-Do', 'Doing', 'Done'
- Bottlenecks the team's efforts.
- Flexible

16.2 Agile Scaling Model

Cody Herring

- Way of beginning the software development process.
- Scott Ambler
- Three Categories

  - Core Agile Development
    - Beginning role in the development cycle
  - Agile Delivery
    - Repeatable results with just the right amount of ceremony for the situations they face.
  - Agile at Scale

- First step to take
  - Adopt disciplined agile delivery life cycle
- Complementary strategy to other methodologies

16.3 Agile Software Development

Jared Cox

- James Highsmith and Sam Bayer
- Came from RSD
  - Like RSD, but more iterative
- Six Characteristics
  - Mission-driven
  - Component-based
  - Iterative
  - Time-boxing
  - Risk-driven
  - Change-tolerant
- Three Step Life Cycle
  - Speculate
  - Collaborate
  - Learn

16.4 Crystal Methods

James Reatherford

- Alistair Cockburn
- Series of related methods for projects of various size and complexity

- Incremental
- Real crystals have two main properties: color and hardness
  - Color - size of team
  - Hardness - complexity of the project in terms of cost of failure
- Very much like agile manifesto

16.5 Crystal Clear

Ashutosh Karki

- Lowest hardness of Crystal Family
- Osmotic communication
- Optimized

16.6 Agile Modeling

CJ Stokes

- Scott Ambler
- Values
  - Communication
  - Simplicity
  - Feedback
  - Courage
  - Humility
- Principles
  - Software is primary
  - Enabling the next effort is secondary
  - Travel light
  - Assume simplicity
  - Embrace change
  - Incremental change
  - Model with a purpose
  - Multiple models
  - Quality work

  - Rapid feedback
- Practices
  - Iterative and Incremental Modeling
  - Teamwork

16.7 Extreme Programming

16.7.1 Phillip Clark

- Kent Beck
- Heavily based on test-driven development and iterative and incremental development
- Takes common sense principles to an extreme level
- Coding is the key activity.
- System architecture is not documented (The code is the documentation)
- Four values
  1. Communication
  2. Simplicity
     - "Live for today"
  3. Feedback
     - "Learning to drive"
  4. Courage
- Twelve Core Practices
  1. The Planning Game
  2. Small releases
  3. Metaphor
  4. Simple design
  5. Testing
  6. Refactoring
  7. Pair programming
  8. Collective ownership
  9. Continuous integration
  10. 40-hour week
      - Not allowed to work overtime two weeks in a row.

11. On-site customer
12. Coding standards

- Cool features
    - Food
    - Parties after releases
    - Interesting office arrangement
    - No documentation

16.7.2 Michael Debs

- Working software over comprehensive documentation
- Responding to change over following a plan
- Simplicity
- Refactoring
- Pair programming
- Collective ownership
    - Trust
- Small Development teams

16.8 DSDM

Patrick Lindsay

16.9 Scrum

Atticus Wright

- Four main tenets
    - Individuals and interactions over precesses and tools
        - Scrum dev teams are self-organizing and rely heavily on communication
        - Daily scrums
    - Working software over documentation
        - The goal of each sprint is to have developed functionality for a project that could be released to the public
    - Customer collaboration over contract negotiation

        * Product owner works with stakeholders

- Works well in chaotic, changing environments
- Produces working software early and often
- Features with high priorities are first
- Catch and fix problem quickly
- Increases morale and productivity
- Supports knowledge sharing
- May be a wrapper for other methods

- Weaknesses
  - Can result in inadequate documentation with too much information inside team members' minds
  - Teams may try to modify Scrum to prevent failure
  - Not great for novices
  - May prevent formation of long-term goals
  - Not suitable for teams of more than 10 people
- Sped up, less structured spiral model
- Incremental
- Lots of scrum in lots of places

16.10 Enterprise Agile

Alla Salah

- Created by Mike Beedle in 1999
- Hybrid of Scrum and XP
- Mainly used in organizations with really big projects
- Can operate in concurrent multi-project environment
- Emphasizes project reusability
- Features from scrum form its management structure
- Some of XP's features for the actual development process
- Daily scrum meetings, scrum master, sprint backlogs

- Pair programming, refactoring, product owner
- Four main phases
  - Planning - sprint backlog and overall project organization
  - Designing - constructing the class and object diagram
  - Coding - pair programming and refactoring
  - Testing - validation
- Advantages
  - Eventually develop large repository of reusable models
  - Scalable for project size and complexity
- Disadvantages
  - Reusable code is expensive
  - Requires competent team members

16.11 Feature Driven Development

Hong Bin Yu

- First proposed by Peter Coad.
- Agile, highly adaptive software development process
  - Has lots of short term iteration
  - Quality is concerned at all steps
  - Delivers frequent client-valued results at all steps
  - Reports accurate significant changes
- Five Phases
  1. Develop an overall model
  2. Build a feature list
  3. Plan by feature
  4. Design by feature
  5. Build by feature
- Eight Practices
  1. Domain object modeling
  2. Developing by feature
  3. Class (code) ownership

    - "Everybody gets one." - Every developer is an expert on one part of the project
  4. Feature teams
  5. Inspection
  6. Regular build schedule
  7. Configuration management
  8. Reporting/Visibility of results

- Has a mathematical formula for project progress

16.12 Pragmatic Programming

Ricky Moore

- Not so well defined
- Main point is to prevent problems
- Invented by Andrew Hunt and David Thomas - *The Pragmatic Programmer*
- Techniques are complimentary to other agile methods

16.13 Internet-Speed Development Method

Elisabeth McClellan

- Quick and dirty
- Main idea is to beat your competitors to the punch
- Similar to code-and-fix
- Doesn't concern itself with maintenance; procrastinates debugging
- Love code reuse
- Stable architecture is suggested, but not mandatory

16.14 Lean Software Development

Jordan Shook

- Mary and Tom Poppendieck - 2003
- Seven Principles

    1. Eliminate Waste
    2. Amplify Learning
    3. Decide as late as possible
    4. Deliver as fast as possible
    5. Empower the team
    6. Build integrity into the code
    7. See the whole

- No unnecessary documentation
- Can allow creativity, but that can be a bad thing

16.15 Agile Unified Process

Jason Smith

- Scott Ambler - 2005
- Simple easy to understand approach to business software using Agile principles
- Nature
    - Inception
    - Elaboration
    - Construction
    - Transition
- Philosophies
    1. Staff needs to know what they are doing
    2. Simplicity
    3. Agility
    4. Focus on high-level activities
    5. Tool independence
    6. You'll want to tailor this product to meet your own needs
- Best Practices
    - Sign off artifacts
    - Need-based artifacts
    - Discuss, visualize and write
    - Single repository

  - Requirement prioritization
  - Requirement envisioning
  - Executable specifications
  - Model storming
  - Model in advance
  - Multiple models
  - Effective communication
  - Accept change
- Delivery over time

16.16 Graphical System Design

Pengcheng Zhao

- The process of defining and developing the hardware and software architecture, components, modules, and system
- Goal is to satisfy requirements with a graphical programming language
- Mainly used to develop embedded systems
- Problem - software engineers are usually short on knowledge about specific hardwares, but hardware experts can't completely understand algorithmic software development
- Both parts can use the same tools (LabVIEW)
- The team size can be two or more
- Save time and money