Software Engineering Notes

Software and Software Engineering

Overview

- Software is designed and built by software engineers.
- Software is used by virtually everyone in society.
- Software is pervasive in our commerce, our culture, and our everyday lives.
- Software engineers have a moral obligation to build reliable software that does no harm to other people.
- Software engineers view computer software, as being made up of the programs, documents, and data required to design and build the system.
- Software users are only concerned with whether or not software products meet their expectations and make their tasks easier to complete.

Important Questions for Software Engineers

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed?

Software

- Software is both a product and a vehicle for delivering a product (information).
- Software is engineered not manufactured.
- Software does not wear out, but it does deteriorate.
- Industry is moving toward component-based software construction, but most software is still custom-built.

Software Application Domains

- System software
- Application software
- Engineering or Scientific Software
- Embedded software
- Product-line software (includes entertainment software)
- Web-Applications
- Artificial intelligence software

New Software Challenges

- Open-world computing

    - Creating software to allow machines of all sizes to communicate with each other across vast networks
- Netsourcing
    - Architecting simple and sophisticated applications that benefit targeted end-user markets worldwide
- Open Source
    - Distributing source code for computing applications so customers can make local modifications easily and reliably

Reasons for Legacy System Evolution

- Software must be adapted to meet needs of new computing environments or technology
- Software must be enhanced to implement new business requirements
- Software must be extended to make it interoperable with more modern system components
- Software must be re-architected to make it viable within a network environment

Unique Nature of Web Apps

- Network intensive
- Concurrency
- Unpredictable load
- Availability (24/7/365)
- Data driven
- Content sensitive
- Continuous evolution
- Immediacy (short time to market)
- Security
- Aesthetics

Software Engineering Realities

- Problem should be understood before software solution is developed
- Design is a pivotal activity
- Software should exhibit high quality
- Software should be maintainable

Software Engineering

- Software engineering is the establishment of sound engineering principles in order to obtain reliable and efficient software in an economical manner.
- Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
- Software engineering encompasses a process, management techniques, technical methods, and the use of tools.

Generic Software Process Framework

- Communication (customer collaboration and requirement gathering)
- Planning (establishes engineering work plan, describes technical risks, lists resources required, work products produced, and defines work schedule)
- Modeling (creation of models to help developers and customers understand the requires and software design)
- Construction (code generation and testing)
- Deployment (software delivered for customer evaluation and feedback)

Software Engineering Umbrella Activities

- Software project tracking and control (allows team to assess progress and take corrective action to maintain schedule)
- Risk management (assess risks that may affect project outcomes or quality)
- Software quality assurance (activities required to maintain software quality)
- Technical reviews (assess engineering work products to uncover and remove errors before they propagate to next activity)
- Measurement (define and collect process, project, and product measures to assist team in delivering software meeting customer needs)
- Software configuration management (manage effects of change)
- Reusability management (defines criteria for work product reuse and establish mechanisms to achieve component reuse)
- Work product preparation and production (activities to create models, documents, logs, forms, lists, etc.)

Attributes for Comparing Process Models

- Overall flow and level of interdependencies among tasks
- Degree to which work tasks are defined within each framework activity
- Degree to which work products are identified and required
- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor of process description
- Degree to which stakeholders are involved in the project
- Level of autonomy given to project team
- Degree to which team organization and roles are prescribed

Essence of Practice

- Understand the problem (communication and analysis)
- Plan a solution (software design)
- Carry out the plan (code generation)
- Examine the result for accuracy (testing and quality assurance)

Understand the Problem

- Who are the stakeholders?
- What functions and features are required to solve the problem?
- Is it possible to create smaller problems that are easier to understand?
- Can a graphic analysis model be created?

Plan the Solution

- Have you seen similar problems before?
- Has a similar problem been solved?
- Can readily solvable subproblems be defined?
- Can a design model be created?

Carry Out the Plan

- Does solution conform to the plan?
- Is each solution component provably correct?

Examine the Result

- Is it possible to test each component part of the solution?
- Does the solution produce results that conform to the data, functions, and features required?

Software Practice Core Principles

1. Software exists to provide value to its users
2. Keep it simple stupid (KISS)
3. Clear vision is essential to the success of any software project
4. Always specify, design, and implement knowing that someone else will have to understand what you have done to carry out his or her tasks
5. Be open to future changes, don't code yourself into a corner
6. Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems that require them
7. Placing clear complete thought before any action almost always produces better results

Software Creation

- Almost every software project is precipitated by a business need (e.g. correct a system defect, adapt system to changing environment, extend existing system, create new system)
- Many times an engineering effort will only succeed if the software created for the project succeeds
- The market will only accept a product is the software embedded within it meets the customer's stated or unstated needs

Process Models

Overview

- The roadmap to building high quality software products is software process.
- Software processes are adapted to meet the needs of software engineers and managers as they undertake the development of a software product.
- A software process provides a framework for managing activities that can very easily get out of control.
- Modern software processes must be agile, demanding only those activities, controls, and work products appropriate for team or product.
- Different types of projects require different software processes.
- The software engineer's work products (programs, documentation, data) are produced as consequences of the activities defined by the software process.
- The best indicators of how well a software process has worked are the quality, timeliness, and long-term viability of the resulting software product.

Software Process

- Framework for the activities, actions, and tasks required to build high quality software
- Defines approach taken as software is engineered
- Adapted by creative, knowledgeable software engineers so that it is appropriate for the products they build and the demands of the marketplace

Generic Process Framework

- Communication
- Planning
- Modeling
- Construction
- Deployment

Umbrella Activities (applied throughout process)

- Software project tracking and control
- Risk management
- Software quality assurance
- Formal technical reviews
- Measurement
- Software configuration management
- Reusability management
- Work product preparation and production

Process Flow

- Describes how each of the five framework activities, actions, and tasks are organized with respect to sequence and time

- *Linear process flow* executes each of the framework activities in order beginning with communication and ending with deployment
- *Iterative process flow* executes the activities in a circular manner creating a more complete version of the software with each circuit or iteration
- *Parallel process flow* executes one on more activities in parallel with other activities

Task Sets

- Each software engineering action associated with a framework activity can be represented by different task sets
- Small one person projects do not require task sets that are as large and detailed as complex projects team oriented project task sets
- Task sets are adapted to meet the specific needs of a software project and the project team characteristics

Process Patterns

- Templates or methods for describing project solutions within the context of software processes
- Software teams can combine patterns to construct processes that best meet the needs of specific projects

Process Pattern Template

- Meaningful pattern name
- Forces (environment in which the pattern is encountered and indicators that make problems visible and affect their solution)
- Type
    - Stage patterns (define problems with a framework activity for the process)
    - Task patterns (define problems associated with engineering action or work task relevant to successful software engineering practice)
    - Phase patterns (define the sequence or flow of framework activities that occur within a process)
- Initial context (describes conditions that must be present prior to using pattern)
    - What organizational or team activities have taken place?
    - What is the entry state for the process?
    - What software engineering or project information already exists?
- Solution (describes how to implement pattern correctly)
- Resulting context (describes conditions that result when pattern has been implemented successfully)
    - What organization or team activities must have occurred?
    - What is the exit state for the process?
    - What software engineering information of project information has been developed?
- Related patterns (links to patterns directly related to this one)
- Known uses/examples (instances in which pattern is applicable)

Process Assessment and Improvement

- Standard CMMI Assessment Method for Process Improvement (SCAMPI) provides a five step process assessment model that incorporates five phases (initiating, diagnosing, establishing, acting, learning)
- CMM-Based Appraisal for Internal Process Improvement (CBAIPI) provides diagnostic technique for assessing the relative maturity of a software organization
- SPICE (ISO/IE15504) standard defines a set of requirements for process assessment
- ISO 9001:2000 for Software defines requirements for a quality management system that will produce higher quality products and improve customer satisfaction

Prescriptive Process Models

- Originally proposed to bring order to the chaos of software development
- They brought to software engineering work and provide reasonable guidance to software teams
- They have not provided a definitive answer to the problems of software development in an ever changing computing environment

Software Process Models

- Waterfall Model (classic life cycle - old fashioned but reasonable approach when requirements are well understood)
- Incremental Models (deliver software in small but usable pieces, each piece builds on pieces already delivered)
- Evolutionary Models
  - Prototyping Model (good first step when customer has a legitimate need, but is clueless about the details, developer needs to resist pressure to extend a rough prototype into a production product)
  - Spiral Model (couples iterative nature of prototyping with the controlled and systematic aspects of the Waterfall Model)
- Concurrent Development Model (concurrent engineering - allows software teams to represent the iterative and concurrent element of any process model)

Specialized Process Models

- Component-Based Development (spiral model variation in which applications are built from prepackaged software components called classes)
- Formal Methods Model (rigorous mathematical notation used to specify, design, and verify computer-based systems)
- Aspect-Oriented Software Development (aspect-oriented programming - provides a process for defining, specifying, designing, and constructing software aspects like user interfaces, security, and memory management that impact many parts of the system being developed)

Unified Process

- Use-case driven, architecture centric, iterative, and incremental software process

- Attempts to draw on best features of traditional software process models and implements many features of agile software development
- Phases
    - Inception phase (customer communication and planning)
    - Elaboration phase (communication and modeling)
    - Construction phase
    - Transition phase (customer delivery and feedback)
    - Production phase (software monitoring and support)

Personal Software Process (PSP)

- Emphasizes personal measurement of both work products and the quality of the work products
- Stresses importance of indentifying errors early and to understand the types of errors likely to be made
- Framework activities
    - Planning (size and resource estimates based on requirements)
    - High-level design (external specifications developed for components and component level design is created)
    - High-level design review (formal verification methods used to uncover design errors, metrics maintained for important tasks)
    - Development (component level design refined, code is generated, reviewed, compiled, and tested, metric maintained for important tasks and work results)
    - Postmortem (effectiveness of processes is determined using measures and metrics collected, results of analysis should provide guidance for modifying the process to improve its effectiveness)

Team Software Process

- Objectives
    - Build self-directed teams that plan and track their work, establish goals, and own their processes and plans
    - Show managers how to coach and motivate their teams and maintain peak performance
    - Accelerate software process improvement by making CCM Level 5 behavior normal and expected
    - Provide improvement guidance to high-maturity organizations
    - Facilitate university teaching of industrial team skills
- Scripts for Project Activities
    - Project launch
    - High Level Design
    - Implementation
    - Integration and system testing
    - Postmortem

Process Technology Tools

- Used to adapt process models to be used by software project team

- Allow organizations to build automated models of common process framework, task sets, and umbrella activities
- These automated models can be used to determine workflow and examine alternative process structures
- Tools can be used to allocate, monitor, and even control all software engineering tasks defined as part of the process model

Agile Development

Overview

- Agile software engineering represents a reasonable compromise between to conventional software engineering for certain classes of software and certain types of software projects
- Agile development processes can deliver successful systems quickly
- Agile development stresses continuous communication and collaboration among developers and customers
- Agile software engineering embraces a philosophy that encourages customer satisfaction, incremental software delivery, small project teams (composed of software engineers and stakeholders), informal methods, and minimal software engineering work products
- Agile software engineering development guidelines stress on-time delivery of an operational software increment over analysis and design (the only really important work product is an operational software increment)

Manifesto for Agile Software Development

- Proposes that it may be better to value:
    - Individuals and interactions over processes and tools
    - Working software over comprehensive documentation
    - Customer collaboration over contract negotiation
    - Responding to change over following a plan
- While the items on the right are still important the items on the left are more valuable under this philosophy

Agility

- An agile team is able to respond to changes during project development
- Agile development recognizes that project plans must be flexible
- Agility encourages team structures and attitudes that make communication among developers and customers more facile
- Eliminates the separation between customers and developers
- Agility emphasizes the importance of rapid delivery of operational software and de-emphasizes importance of intermediate work products
- Agility can be applied to any software process as long as the project team is allowed to streamline tasks and conduct planning in way that eliminate non-essential work products
- The costs of change increase rapidly as a project proceeds to completion, the earlier a change is made the less costly it will be
- Agile processes may flatten the cost of change curve by allowing a project team to make changes late in the project at much lower costs

Agile Processes

- Are based on three key assumptions
  1. It is difficult to predict in advance which requirements or customer priorities will change and which will not
  2. For many types of software design and construction activities are interleaved (construction is used to prove the design)
  3. Analysis, design, and testing are not as predictable from a planning perspective as one might like them to be
- Agile processes must be adapt incrementally to manage unpredictability
- Incremental adaptation requires customer feedback based on evaluation of delivered software increments (executable prototypes) over short time periods

Agility Principles

1. Highest priority is to satisfy customer through early and continuous delivery of valuable software
2. Welcome changing requirements even late in development, accommodating change is viewed as increasing the customer's competitive advantage
3. Delivering working software frequently with a preference for shorter delivery schedules (e.g. every 2 or 3 weeks)
4. Business people and developers must work together daily during the project
5. Build projects around motivated individuals, given them the environment and support they need, trust them to get the job done
6. Face-to-face communication is the most effective method of conveying information within the development team
7. Working software is the primary measure of progress
8. Agile processes support sustainable development, developers and customers should be able to continue development indefinitely
9. Continuous attention to technical excellence and good design enhances agility
10. Simplicity (defined as maximizing the work not done) is essential
11. The best architectures, requirements, and design emerge from self-organizing teams
12. At regular intervals teams reflects how to become more effective and adjusts its behavior accordingly

Human Factors

- Traits that need to exist in members of agile development teams:
  - Competence
  - Common focus
  - Collaboration
  - Decision-making ability
  - Fuzzy-problem solving ability
  - Mutual trust and respect
  - Self-organization

Agile Process Models

- Extreme Programming (XP)
- Adaptive Software Development (ASD)
- Scrum
- Dynamic Systems Development Method (DSDM)
- Crystal
- Feature Driven Development (FDD)
- Lean Software Development (LSD)
- Agile Modeling (AM)
- Agile Unified Process (AUP)

Extreme Programming

- Relies on object-oriented approach
- Values
    - Communication (close, informal between developers and stakeholders)
    - Simplicity (developers design for current needs, not future needs)
    - Feedback (implemented software – unit tests, customer – user stories guide acceptance tests, software team – iterative planning)
    - Courage (design for today not tomorrow)
    - Respect (stakeholders and team members for the software product)
- Key activities
    - Planning (user stories created and ordered by customer values)
    - Design (simple designs preferred, CRC cards and design prototypes are only work products, encourages use of refactoring)
    - Coding (focuses on unit tests to exercise stories, emphasizes use of pairs programming to create story code, continuous integration and smoke testing is utilized)
    - Testing (unit tests created before coding are implemented using an automated testing framework to encourage use of regression testing, integration and validation testing done on daily basis, acceptance tests focus on system features and functions viewable by the customer)

Industrial XP

- Readiness acceptance
    - Does an appropriate development environment exists to support IXP?
    - Will the team be populated by stakeholders?
    - Does the organization have a distinct quality program that support continuous process improvement?
    - Will the organizational culture support new values of the agile team?
    - Will the broader project community be populated appropriately?
- Project community (finding the right people for the project team)
- Project chartering (determining whether or not an appropriate business justification exists to justify the project)
- Test-driven management (used to establish measurable destinations and criteria for determining when each is reached)
- Retrospectives (specialized technical review focusing on issues, events, and lessons-learned across a software increment or entire software release)
- Continuous learning (vital part of continuous process improvement)

XP Issues

- Requirement volatility (can easily lead for scope creep that causes changes to earlier work design for the then current needs)
- Conflicting customer needs (many project with many customers make it hard to assimilate all customer needs)
- Requirements expressed informally (with only user stories and acceptance tests, its hard to avoid omissions and inconsistencies)
- Lack of formal design (complex systems may need a formal architectural design to ensure a product that exhibits quality and maintainability)

Adaptive Software Development

- Self-organization arises when independent agents cooperate to create a solution to a problem that is beyond the capability of any individual agent
- Emphasizes self-organizing teams, interpersonal collaboration, and both individual and team learning
- Adaptive cycle characteristics
    - Mission-driven
    - Component-based
    - Iterative
    - Time-boxed
    - Risk driven and change-tolerant
- Phases
    - Speculation (project initiated and adaptive cycle planning takes place)
    - Collaboration (requires teamwork from a jelled team, joint application development is preferred requirements gathering approach)
    - Learning (components implemented and testes, focus groups provide feedback, formal technical reviews, postmortems)

Scrum

- Scrum principles
    - Small working team used to maximize communication, minimize overhead, and maximize sharing of informal knowledge
    - Process must be adaptable to both technical and business challenges to ensure best product produced
    - Process yields frequent increments that can be inspected, adjusted, tested, documented and built on
    - Development work and people performing it are partitioned into clean, low coupling partitions
    - Testing and documentation is performed as the product is built
    - Provides the ability to declare the product done whenever required
- Process patterns defining development activities
    - Backlog (prioritized list of requirements or features the provide business value to customer, items can be added at any time)
    - Sprints (work units required to achieve one of the backlog items, must fir into a predefined time-box, affected backlog items frozen)

    - Scrum meetings (15 minute daily meetings)
        - What was done since last meeting?
        - What obstacles were encountered?
        - What will be done by the next meeting?
    - Demos (deliver software increment to customer for evaluation)

Dynamic Systems Development Method

- Provides a framework for building and maintaining systems which meet tight time constraints using incremental prototyping in a controlled environment
- Uses Pareto principle (80% of project can be delivered in 20% required to deliver the entire project)
- Each increment only delivers enough functionality to move to the next increment
- Uses time boxes to fix time and resources to determine how much functionality will be delivered in each increment
- Guiding principles
    - Active user involvement
    - Teams empowered to make decisions
    - Fitness foe business purpose is criterion for deliverable acceptance
    - Iterative and incremental develop needed to converge on accurate business solution
    - All changes made during development are reversible
    - Requirements are baselined at a high level
    - Testing integrates throughout life-cycle
    - Collaborative and cooperative approach between stakeholders
- Life cycle activities
    - Feasibility study (establishes requirements and constraints)
    - Business study (establishes functional and information requirements needed to provide business value)
    - Functional model iteration (produces set of incremental prototypes to demonstrate functionality to customer)
    - Design and build iteration (revisits prototypes to ensure they provide business value for end users, may occur concurrently with functional model iteration)
    - Implementation (latest iteration placed in operational environment)

Crystal

- Development approach that puts a premium on maneuverability during a resource-limited game of invention and communication with the primary goal of delivering useful software and a secondary goal of setting up for the next game
- Incremental development strategy used with 1 to 3 month time lines
- Reflection workshops conducted before project begins, during increment development activity, and after increment is delivered

Feature Driven Development

- Practical process model for object-oriented software engineering
- Feature is a client-valued function, can be implemented in two weeks or less
- FDD Philosophy

    - Emphasizes collaboration among team members
    - Manages problem and project complexity using feature-based decomposition followed integration of software increments
    - Technical communication using verbal, graphical, and textual means
    - Software quality encouraged by using incremental development, design and code inspections, SQA audits, metric collection, and use of patterns (analysis, design, construction)
- Framework activities
    - Develop overall model (contains set of classes depicting business model of application to be built)
    - Build features list (features extracted from domain model, features are categorized and prioritized, work is broken up into two week chunks)
    - Plan by feature (features assessed based on priority, effort, technical issues, schedule dependencies)
    - Design by feature (classes relevant to feature are chosen, class and method prologs are written, preliminary design detail developed, owner assigned to each class, owner responsible for maintaining design document for his or her own work packages)
    - Build by feature (class owner translates design into source code and performs unit testing, integration performed by chief programmer)

Lean Software Development Principles

- Eliminate waste
- Build quality in
- Create knowledge
- Defer commitment
- Deliver fast
- Respect people
- Optimize the whole

Agile Modeling

- Practice-based methodology for effective modeling and documentation of software systems in a light-weight manner
- Modeling principles
    - Model with a purpose
    - Use multiple models
    - Travel light (only keep models with long-term value)
    - Content is more important than representation
    - Know the models and tools you use to create them
    - Adapt locally
- Requirements gathering and analysis modeling
    - Work collaboratively to find out what customer wants to do
    - Once requirements model is built collaborative analysis modeling continues with the customer
- Architectural modeling
    - Derives preliminary architecture from analysis model
    - Architectural model mist be realistic for the environment and must be understandable by developers

Agile Unified Process

- Adopts classic UP phased activities (inception, elaboration, construction, transition) to enable team to visualize overall software project flow
- Within each activity team iterates to achieve agility and delivers meaningful software increments to end-users as rapidly as possible
- Each AUP iteration addresses
    - Modeling (UML representations of business and problem domains)
    - Implementation (models translated into source code)
    - Testing (uncover errors and ensure source code meets requirements)
    - Deployment (software increment delivery and acquiring feedback)
    - Configuration and project management (change management, risk management, persistent work product control)
    - Environment management (standards, tools, technology)

Principles that Guide Practice

Overview

This chapter describes professional practice as the concepts, principles, methods, and tools used by software engineers and managers to plan and develop software. Software engineers must be concerned both with the technical details of doing things and the things that are needed to build high-quality computer software. Software process provides the project stakeholders with a roadmap to build quality products. Professional practice provides software engineers with the detail needed to travel the road. Software practice encompasses the technical activities needed to produce the work products defined by the software process model chosen for a project.

Software Practice Core Principles

8. Software exists to provide value to its users
9. Keep it simple stupid (KISS)
10. Clear vision is essential to the success of any software project
11. Always specify, design, and implement knowing that someone else will have to understand what you have done to carryout his or her tasks
12. Be open to future changes, don't code yourself into a corner
13. Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems that require them
14. Placing clear complete thought before any action almost always produces better results

Principles that Guide Process

1. Be agile
2. Focus on quality at every step
3. Be ready to adapt
4. Build an effective team
5. Establish mechanisms for communications and control
6. Manage change
7. Assess risk
8. Create work products that provide value for others

Principles that Guide Practice

1. Divide and conquer
2. Understand the use of abstraction
3. Strive for consistency
4. Focus of the transfer of information
5. Build software that exhibits effective modularity
6. Look for patterns
7. When possible, represent the problem and its solution from a number of different perspectives
8. Remember that someone will maintain the software

Principles of Effective Communication

1. Listen
2. Prepare before you communicate
3. Have a facilitator for any communication meeting
4. Face-to-face communication is best
5. Take notes and document decisions
6. Strive for collaboration
7. Stay focused and modularize your discussion
8. Draw a picture if something is unclear
9. Move on once you agree, move on when you can't agree, move on if something unclear can't be clarified at the moment
10. Negotiation is not a contest or game

Planning Principles

1. Understand scope of project
2. Involve customer in planning activities
3. Recognize that planning Is iterative
4. Make estimates based on what you know
5. Consider risk as you define the plan
6. Be realistic
7. Adjust the granularity as you define the plan
8. Define how you intend to measure quality
9. Describe how you intend to accommodate change
10. Track the plan frequently and make adjustments as required

Modeling Classes

- Requirements (analysis) models – represent customer requirements by depicting the software in three domains (information, function, behavior)
- Design models – represent characteristics of software that help practitioners to construct s it effectively (architecture, user interface, component-level detail)

Requirements Modeling Principles

1. Problem information domain must be represented and understood
2. Functions performed by the software must be defined
3. Software behavior must be represented as a consequence of external events
4. Models depicting the information, function, and behavior must be partitioned in manner that uncovers detail in a hierarchical fashion
5. The analysis task should move from essential information toward implementation detail

Design Modeling Principles

1. Design should be traceable to the requirements model
2. Always consider the architecture of the system to be built
3. Data design is as important as algorithm design
4. Internal and external interfaces must be design with care
5. User interface design should be tuned to the needs of the end-user and must focus on use of use
6. Component-level design should be functionally independent
7. Components should be loosely coupled to one another and to the external environment
8. Design representations should be easy to understand
9. Design should be developed iteratively and designer should strive to simplify design with each iteration

Agile Modeling Principles

1. Primary goal of the software team is to build software not create models
2. Don't create any more models than you have to
3. Strive to produce the simplest model that will describe the problem or software
4. Build models in a way that makes them amenable to change
5. Be able to state the explicit purpose for each model created
6. Adapt models to the system at hand
7. Try to build useful models, forget about trying to build perfect models
8. Don't be dogmatic about model syntax as long as the model communicates content successfully
9. If your instincts tell you there is something wrong with the model then you probably have a reason to be concerned
10. Get feedback as soon as you can

Construction Activities

- Coding includes
    - Direct creation of programming language source code
    - Automatic generation of source code using a design-like representation of component to be built
    - Automatic generation of executable code using a "fourth generation programming language
- Testing levels
    - Unit testing
    - Integration testing

    - Validation testing
    - Acceptance testing

Coding Principles

- Preparation - before writing any code be sure you:
    - Understand problem to solve
    - Understand basic design principles
    - Pick a programming language that meets the needs of the software to be built and the environment
    - Select a programming environment that contains the right tools
    - Create a set of unit tests to be applied once your code is completed
- Coding - as you begin writing code be sure you:
    - Use structured programming practices
    - Consider using pairs programming
    - Select data structures that meet the needs of the design
    - Understand software architecture and create interfaces consistent with the architecture
    - Keep conditional logic as simple as possible
    - Create nested loops in a way that allows them to be testable
    - Select meaningful variable names consistent with local standards
    - Write code that is self-documenting
    - Use a visual layout for you code that aids understanding
- Validation - after your complete your first coding pass be sure you:
    - Conduct a code walkthrough when appropriate
    - Perform unit tests and correct uncovered errors
    - Refactor the code

Testing Objectives

- Testing is the process of executing a program with the intent of finding an error
- A good test is one that has a high probability of finding an undiscovered error
- A successful test is one the uncovers and undiscovered error

Testing Principles

1. All tests should be traceable to customer requirements
2. Tests should be planed long before testing begins
3. Pareto Principle applies to testing (80% of errors are found in 20% of code)
4. Testing should begin “in the small” and progress toward testing “in the large”
5. Exhaustive testing is not possible

Deployment Actions

- Delivery
- Support
- Feedback

Deployment Principles

1. Customer software expectations must be managed
2. Complete delivery package should be assembled and tested
3. Support regime must be established before software is delivered
4. Appropriate instructional materials must be supplied to end-users
5. Buggy software should be fixed before it is delivered

Understanding Requirements

Overview

- Requirements engineering helps software engineers better understand the problems they are trying to solve.
- Building an elegant computer solution that ignores the customer's needs helps no one.
- It is very important to understand the customer's wants and needs before you begin designing or building a computer-based solution.
- The requirements engineering process begins with inception, moves on to elicitation, negotiation, problem specification, and ends with review or validation of the specification.
- The intent of requirements engineering is to produce a written understanding of the customer's problem.
- Several different work products might be used to communicate this understanding (user scenarios, function and feature lists, analysis models, or specifications).

Requirements Engineering

- Must be adapted to the needs of a specific process, project, product, or people doing the work.
- Begins during the software engineering communication activity and continues into the modeling activity.
- In some cases requirements engineering may be abbreviated, but it is never abandoned.
- It is essential that the software engineering team understand the requirements of a problem before the team tries to solve the problem.

Requirements Engineering Tasks

- Inception (software engineers use context-free questions to establish a basic understanding of the problem, the people who want a solution, the nature of the solution, and the effectiveness of the collaboration between customers and developers)
- Elicitation (find out from customers what the product objectives are, what is to be done, how the product fits into business needs, and how the product is used on a day to day basis)
- Elaboration (focuses on developing a refined technical model of software function, behavior, and information)
- Negotiation (requirements are categorized and organized into subsets, relations among requirements identified, requirements reviewed for correctness, requirements prioritized based on customer needs)

- Specification (written work products produced describing the function, performance, and development constraints for a computer-based system)
- Requirements validation (formal technical reviews used to examine the specification work products to ensure requirement quality and that all work products conform to agreed upon standards for the process, project, and products)
- Requirements management (activities that help project team to identify, control, and track requirements and changes as project proceeds, similar to software configuration management (SCM) techniques

Initiating Requirements Engineering Process

- Identify stakeholders
- Recognize the existence of multiple stakeholder viewpoints
- Work toward collaboration among stakeholders
- These context-free questions focus on customer, stakeholders, overall goals, and benefits of the system
    - Who is behind the request for work?
    - Who will use the solution?
    - What will be the economic benefit of a successful solution?
    - Is there another source for the solution needed?
- The next set of questions enable developer to better understand the problem and the customer's perceptions of the solution
    - How would you characterize good output form a successful solution?
    - What problem(s) will this solution address?
    - Can you describe the business environment in which the solution will be used?
    - Will special performance constraints affect the way the solution os approached?
- The final set of questions focuses on communication effectiveness
    - Are you the best person to give "official" answers to these questions?
    - Are my questions relevant to your problem?
    - Am I asking too many questions?
    - Can anyone else provide additional information?
    - Should I be asking you anything else?

Eliciting Requirements

- Goal is to identify the problem, propose solution elements, negotiate approaches, and specify preliminary set of solutions requirements
- Collaborative requirements gathering guidelines
    - Meetings attended by both developers and customers
    - Rules for preparation and participation are established
    - Flexible agenda is used
    - Facilitator controls the meeting
    - Definition mechanism (e.g. stickers, flip sheets, electronic bulletin board) used to gauge group consensus

Quality function deployment (QFD)

- Quality management technique that translates customer needs into technical software requirements expressed as a **customer voice table**

- Identifies three types of requirements (normal, expected, exciting)
- In customer meetings **function deployment** is used to determine value of each function that is required for the system
- **Information deployment** identifies both data objects and events that the system must consume or produce (these are linked to functions)
- **Task deployment** examines the system behavior in the context of its environment
- **Value analysis** is conducted to determine relative priority of each requirement generated by the deployment activities

Elicitation Work Products

- Statement of need and feasibility
- Bounded statement of scope for system or product
- List of stakeholders involved in requirements elicitation
- Description of system's technical environment
- List of requirements organized by function and applicable domain constraints
- Set of usage scenarios (use-cases) that provide use insight into operation of deployed system
- Prototypes developed to better understand requirements

Elicitation Problems

- Scope – system boundaries ill-defined
- Understanding – customers not sure what's needed or can't communicate it
- Volatility – requirements changes over time

Developing Use-Cases

- Each use-case tells stylized story about how end-users interact with the system under a specific set of circumstances
- First step is to identify **actors** (people or devices) that use the system in the context of the function and behavior of the system to be described
    - Who are the primary (interact with each other) or secondary (support system) actors?
    - What are the actor's goals?
    - What preconditions must exist before story begins?
    - What are the main tasks or functions performed by each actor?
    - What exceptions might be considered as the story is described?
    - What variations in actor interactions are possible?
    - What system information will the actor acquire, produce, or change?
    - Will the actor need to inform the system about external environment changes?
    - What information does the actor desire from the system?
    - Does the actor need to be informed about unexpected changes?
- Next step is to elaborate the basic use case to provide a more detailed description needed to populate a use-case template

Use-case template

- Use Case Name
- Primary actor
- Goal in context
- Preconditions
- Trigger
- Scenario details
- Exceptions
- Priority
- When available
- Frequency of use
- Channel to actor
- Secondary actors
- Channels to secondary actors
- Open issues

Analysis Model

- Intent is to provide descriptions of required information, functional, and behavioral domains for computer-based systems
- Analysis Model Elements
    - Scenario-based elements (use cases describe system from user perspective)
    - Class-based elements (relationships among objects manipulated by actors and their attributes are depicted as classes)
    - Behavioral elements (depict system and class behavior as states and transitions between states)
    - Flow-oriented elements (shows how information flows through the system and is transformed by the system functions)

Analysis Patterns

- Suggest solutions (a class, a function, or a behavior) that can be reused when modeling future applications
- Can speed up the development of abstract analysis models by providing reusable analysis models with their advantages and disadvantages
- Facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions to common patterns

Negotiating Requirements

- Intent is to develop a project plan that meets stakeholder needs and real-world constraints (time, people, budget) placed on the software team
- Negotiation activities
    - Identification of system key stakeholders
    - Determination of stakeholders' "win conditions"
    - Negotiate to reconcile stakeholders' win conditions into "win-win" result for all stakeholders (including developers)

- Goal is to produce a win-win result before proceeding to subsequent software engineering activities

Requirement Review (Validation)

- Is each requirement consistent with overall project or system objective?
- Are all requirements specified at the appropriate level off abstraction?
- Is each requirement essential to system objective or is it an add-on feature?
- Is each requirement bounded and unambiguous?
- Do you know the source for each requirement?
- Do requirements conflict with one another?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable once implemented?
- Does the requirements model reflect the information, function, and behavior of the system to be built?
- Has the requirements model been partitioned in a way that exposes more detailed system information progressively?
- Have all the requirements patterns been properly validated and are they consistent with customer requirements?

Requirements Modeling

The requirements model is the first technical representation of a system. Requirements modeling process uses a combination of text and diagrams to represent software requirements (data, function, and behavior) in an understandable way. Software engineers build requirements models using requirements elicited from customers. Building analysis models helps to make it easier to uncover requirement inconsistencies and omissions. This chapter covers three perspectives of requirements modeling: scenario-based, data (information), and class-based. Requirements modeling work products must be reviewed for completeness, correctness, and consistency.

Requirements Models

- Scenario-based (system from the user's point of view)
- Data (shows how data are transformed inside the system)
- Class-oriented (defines objects, attributes, and relationships)
- Flow-oriented (shows how data are transformed inside the system)
- Behavioral (show the impact of events on the system states)

Requirements Model Objectives

- Describe what the customer requires.
- Establish a basis for the creation of a software design.
- Devise a set of requirements that can be validated once the software is built.

Analysis Model Rules of Thumb

- The model should focus on requirements that are visible within the problem or business domain and be written as a relatively high level of abstraction.
- Each element of the analysis model should add to the understanding of the requirements and provide insight into the information domain, function, and behavior of the system.
- Delay consideration of infrastructure and other non-functional models until design.
- Minimize coupling throughout the system.
- Be certain the analysis model provides value to all stakeholders.
- Keep the model as simple as possible.

Domain Analysis

- Umbrella activity involving the Identification, analysis, and specification of common requirements from a specific application domain, typically for reuse in multiple projects
- Object-oriented domain analysis involves the identification, analysis, and specification of reusable capabilities within a specific application domain in terms of common objects, classes, subassemblies, and frameworks

Requirements Modeling Approaches

- **Structured analysis** considers data and processes that transform data as separate entities
    - Data objects are modeled to define their attributes and relationships
    - Process are modeled to show how they transform data as it flows thought the system
- **Object-oriented analysis** focuses on the definition of classes and the manner in which they collaborate to effect the customer requirements

Scenario-Based Modeling

- Makes use of use cases to capture the ways in which end-users will interact with the system
- UML requirements modeling begins with the creation of scenarios in the from of use cases, activity diagrams, and swim lane diagrams

Developing Use Cases

- Use cases capture the interactions between actors (i.e. entities that consume or produce information)
- Begin by listing the activities performed by a single actor to accomplish a single function
- Continue this process for each actor and each system function
- Use-cases are written first in narrative form and then mapped to a template if more formality is required
- Each primary scenarios should be reviewed and refined to see if alternative interactions are possible
    - Can the actor take some other action at this point?
    - Is it possible that the actor will encounter an error condition at some point? If so, what?
    - Is it possible that the actor will encounter some other behavior at some point? If so, what?

Exceptions

- Describe situations (failures or user choices) that cause the system to exhibit unusual behavior
- Brainstorming should be used to derive a reasonably complete set of exceptions for each use case
    - Are there cases where a validation function occurs for the use case?
    - Are there cases where a supporting function (actor) fails to respond appropriately?
    - Can poor system performance result in unexpected or improper use actions?
- Handling exceptions may require the creation of additional use cases

UML Activity Diagrams

- Supplements use-case by providing graphical representation of the interaction flow within a specific scenario
- Similar to flow chart
    - Rounded rectangles used to represent functions
    - Diamonds used to represent decision points
    - Labeled arrows represent system flow
    - Solid horizontal lines indicate parallel activities

UML Swimlane Diagrams

- Variation of activity diagrams used show flow of activities in use case as well as indicating which actor has responsibility for activity rectangle actions
- Responsibilities are represented by parallel line segments that divide the diagram vertically headed by the responsible actor

Data Objects

- Data object - any person, organization, device, or software product that produces or consumes information
- Attributes - name a data object instance, describe its characteristics, or make reference to another data object
- Relationships - indicate the manner in which data objects are connected to one another

Class-based Modeling

- Represents objects system manipulates, operations applied to objects, and collaborations occurring between classes
- Elements of class model include: classes, objects, attributes, operations, CRC models, collaboration diagrams, and packages

Identifying Analysis Classes

- Examine the problem statement and try to find nouns that fit the following categories and produce or consume information (i.e. grammatical parse)
    - External entities (systems, devices, people)
    - Things (e.g. reports, displays, letters, signals)
    - Events occurring during system operation
    - Roles (e.g. manager, engineer, salesperson)
    - Organizational units (e.g. division, group, team)
    - Places
    - Structures (e.g. sensors, vehicles, computers)

Class Selection Criteria

- Consider whether each potential class satisfies one of these criteria as well
    - Contains information that should be retained
    - Provides needed services
    - Contains multiple attributes
    - Has common set of attributes that apply to all class instances
    - Has common set of operations that apply to all object instances
    - Represents external entity that produces or consumes information

Specifying Class Attributes

- Examine the processing narrative or use-case and select the things that reasonably can belong to each class
- Ask what data items (either composite or elementary) fully define this class in the context of the problem at hand?

Defining Operations

- Look at the verbs in the processing narrative and identify operations reasonably belonging to each class that (i.e. grammatical parse)
    - manipulate data
    - perform computation
    - inquire about the state of an object
    - monitor object for occurrence of controlling event
- Divide operations into sub-operations as needed
- Also consider communications that need to occur between objects and define operations as needed

Class-Responsibility-Collaborator (CRC) Modeling

- Develop a set of index cards that represent the system classes
- One class per card

- Cards are divide into three sections (class name, class responsibilities, class collaborators)
- Once a complete CRC card set is developed it is reviewed examining the usage scenarios

Classes

- Entity classes extracted directly from problem statement (things stored in a database and persist throughout application)
- Boundary classes used to create the interface that user sees and interacts with as software is used
- Controller classes manage unit of work from start to finish
    - Create or update entity objects
    - Instantiate boundary objects
    - Complex communication between sets of objects
    - Validation of data communicated between actors

Allocating Responsibilities to Classes

- Distribute system intelligence evenly
- State each responsibility as generally as possible
- Information and its related behaviors should reside within the same class
- Localize all information about one thing in a single class
- Share responsibilities among related classes when appropriate

Collaborations

- Any time a class cannot fulfill a responsibility on its own it needs to interact with another class
- A server object interacts with a client object to fulfill some responsibility

Reviewing CRC Models

- Each review participant is given a subset of the CRC cards (collaborating cards must be separated)
- All use-case scenarios and use-case diagrams should be organized into categories
- Review leader chooses a use-case scenario and begins reading it out loud
- Each time a named object is read a token is passed to the reviewer holding the object's card
- When the reviewer receives the token, he or she is asked to describe the responsibilities listed on the card
- The group determines whether one of the responsibilities on the card satisfy the use-case requirement or not
- If the responsibilities and collaborations on the index card cannot accommodate the use-case requirements then modifications need to be made to the card set

Associations and Dependencies

- Association - present any time two classes are related to one another in some fashion

    - association **multiplicity** or cardinality can be indicated in a UML class diagram (e.g. 0..1, 1..1, 0.., 1..)
- Dependency – client/server relationship between two classes
    - dependency relationships are indicated in class diagrams using stereotype names surrounded by angle brackets (e.g. <<stereotype>>)

Analysis Packages

- Categorization is an important part of analysis modeling
- Analysis packages are made up of classes having the same categorization
- In class diagrams visibility of class elements can be indicated using a + (public), - (private), # (package)

Requirements Modeling

Requirements modeling has many different dimensions. The discussion in this chapter focuses on flow-oriented models, behavioral models, and patterns. This chapter also discusses WebApp requirements models. Flow-oriented modeling shows how data objects are transformed by processing functions. Behavioral modeling depicts the systems states and the impact of events on system states. Pattern-based modeling makes use of existing domain knowledge to facilitate requirements modeling. Software engineers build models using requirements elicited from stakeholders. Developer insights into software requirements grows in direct proportion to the number of different representations used in modeling. It is not always possible to develop every model for every project given the available project resources. Requirements modeling work products must be reviewed for correctness, completeness, consistency, and relevancy to stakeholder needs.

Flow-oriented Modeling

- Data flow diagrams (DFD) show the relationships of external entities, process or transforms, data items, and data stores
- DFD's take an input-process-output view of the system
- DFD's cannot show procedural detail (e.g. conditionals or loops) only the flow of data through the software
- In DFD's data objects are represented by labeled arrows and data transformations are represented by circles
- First DFD (known as the level 0 or context diagram) represents system as a whole
- Subsequent data flow diagrams refine the context diagram providing increasing levels of detail
- Refinement from one DFD level to the next should follow approximately a 1:5 ratio (this ratio will reduce as the refinement proceeds)

Creating Data Flow Diagram

- Level 0 data flow diagram should depict the system as a single bubble
- Primary input and output should be carefully noted
- Refinement should begin by consolidating candidate processes, data objects, and data stores to be represented at the next level

- Label all arrows with meaningful names
- Information flow continuity must be maintained from one level to level
- Refine one bubble at a time
- Write a process specification (PSPEC) for each bubble in the final DFD
- PSPEC is a "mini-spec" describing the process algorithm written using text narrative, a program design language (PDL), equations, tables, or UML activity diagrams

Creating Control Flow Model

- Begin by stripping all the data flow arrows form the DFD
- Events (solid arrows) and control items (dashed arrows) are added to the control flow diagram (CFD)
- Create a control specification (CSPEC) for each bubble in the final CFD
- CSPEC contains a state diagram that is a sequential specification of the behavior and may also contain a program activation table that is a combinatorial specification of system behavior

Behavioral Modeling

- A state transition diagrams (STD) represents the system states and events that trigger state transitions
- STD's indicate actions (e.g. process activation) taken as a consequence of a particular event
- A state is any observable mode of behavior

Creating Behavior Models

- Evaluate all use-cases to understand the sequence of interaction within the system
- Identify events that drive the interaction sequence and how these events relate to specific objects
- Create a sequence or event-trace for each use-case
- Build a state transition diagram for the system
- Review the behavior model to verify accuracy and consistency

UML State Diagrams

- Round corned rectangles are used for each state
    - Passive states show the current status of object attributes
    - Active states indicate current status of object as it undergoes transformation or processing
- Arrows connecting states are labeled with the name of the event that triggers the transition from one state to the other
- Guards limiting the transition from one state to the next may be specified as Boolean conditions involving object attributes in the use-case narratives

UML Sequence Diagrams

- Built from use-case descriptions by determining how events cause transitions from one object to another
- Key classes and actors are shown across the top
- Object and actor activations are shown as vertical rectangles arranged along vertical dashed lines called lifelines
- Arrows connecting activations are labeled with the name of the event that triggers the transition from one class or actor to another
- Object flow among objects and actors may be represented by labeling a dashed horizontal line with the name of the object being passed
- States may be shown along the lifelines

Analysis Patterns

- Discovered (not created) during requirements engineering work
- Should be documented by describing the general problem pattern is applicable to, the prescribed solution, assumptions, constraints, advantages, disadvantages, and references to known examples
- Documented analysis patterns are stored in an indexed repository facilitate its reuse by other team members

Conditions Favoring WebApp Requirements Modeling

- Large or complex WebApp to be built
- Large number of stakeholders
- Large number developers on WebApp team
- Development team members have not worked together before
- WebApp success will have strong bearing on success of company

WebApp Requirements Modeling

- Inputs – any information collected during communication activity
- Outptus – models for WebApp content, function, user interaction, environment, infrastructure

WebApp Requirements Models

- Content – content (text, graphics, images, audio, video) provided by WebApp
- Interaction – describes user interaction with WebApp
- Functional – defines operations applied to the WebApp content and other content independent processing functions
- Navigation – defines overall navigation strategy for the WebApp
- Configuration – describes WebApp environmental infrastructure in detail

Content Model

- Structural elements that represent WebApp content requirements
- WebApp content objects – text, graphics, photographs, video images, audio

- Includes all analysis classes – user visible entities created or manipulated as end-users interact with WebApp
- Analysis classes defined by class diagrams showing attributes, operations, and class collaborations
- Content model is derived from careful examination of WebApp use-cases

Interaction Model

- Use-cases – dominant element of WebApp interaction models
- Sequence diagrams – provide representation of manner in which user actions collaborate with analysis classes
- State diagrams – indicates information required to move users between states and represents behavioral information, can also depict potential navigation pathways
- User interface prototypes – layout of content presentation, interaction mechanisms, and overall aesthetic of user interface

Functional Model

- User observable behavior delivered to WebApp end-users
- Operations contained in analysis classes to implement class behaviors
- UML activity diagrams used to model both

Configuration Model

- May be a list of server-side and client-side attributes for the WebApp
- UML deployment diagrams can be used for complex configuration architectures

Navigation Model

- Web engineers consider requirements that dictate how each type of user will navigate from one content object to another
- Navigation mechanics are defined as part of design
- Web engineers and stakeholders must determine navigation requirements

Quality Concepts

Overview

This chapter provides an introduction to software quality. Software quality is the concern of every software process stakeholder. If a software team stresses quality in all software engineering activities, it reduces the amount of rework that must be done. This results in lower costs and improved time-to-market. To achieve high quality software, four elements must be present: proven software engineering process and practice, solid project management, comprehensive quality control, and the presence of a quality assurance infrastructure. Software that meets its customer's needs, performs accurately and reliably, and provides value to all who use it. Developers track quality by examining the results of all

quality control activities. Developers measure quality by examining errors before delivery and defects released to the field.

What is Quality?

- The *transcendental view* - quality is something that you immediately recognize, but cannot explicitly define.
- The *user view* - quality in terms of meeting an end-user's specific goals.
- The *manufacturer's view* - quality in terms of the conformance to the original specification of the product.
- The *product view* - quality can be tied to inherent characteristics (e.g., functions and features) of a product.
- The *value-based view* - quality based on how much a customer is willing to pay for a product.
- Quality of design - refers to characteristics designers specify for the end product to be constructed
- Quality of conformance - degree to which design specifications are followed in manufacturing the product

Software Quality

- Software quality defined as an effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.
- An *effective software process* establishes the infrastructure that supports any effort at building a high quality software product.
- A *useful product* delivers the content, functions, and features that the end-user desires, but as important, it delivers these assets in a reliable, error free way.
- By *adding value for both the producer and user* of a software product, high quality software provides benefits for the software organization and the end-user community.

Garvin's Quality Dimensions

- Performance quality
- Feature quality
- Reliability
- Conformance
- Durability
- Serviceability
- Aesthetics
- Perception

McCall's Quality Factors

- *Correctness* - extent to which a program satisfies its specification and fulfills the customer's mission objectives

- *Reliability* - extent to which a program can be expected to perform its intended function with required precision
- *Efficiency* - amount of computing resources and code required by a program to perform its function
- *Integrity* - extent to which access to software or data by unauthorized persons can be controlled
- *Usability* - effort required to learn, operate, prepare input for, and interpret output of a program
- *Maintainability* - effort required to locate and fix an error in a program
- *Flexibility* - effort required to modify an operational program
- *Testability* - effort required to test a program to ensure that it performs its intended function
- *Portability* - effort required to transfer the program from one hardware and/or software system environment to another
- *Reusability* - extent to which a program [or parts of a program] can be reused in other applications
- *Interoperability.* Effort required to couple one system to another

ISO 9126 Quality Factors

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Measuring Quality

- General quality dimensions and factors are not adequate for assessing the quality of an application in concrete terms
- Project teams need to develop a set of targeted questions to assess the degree to which each quality factor has been satisfied in the application
- Subjective measures of software quality may be viewed as little more than personal opinion
- Software metrics represent indirect measures of some manifestation of quality and are an attempt to quantify the assessment of software quality

Software Quality Dilemma

- If you produce software with terrible quality you lose because no one will but it
- If you spend infinite time and money to create software you lose because you will go out of business without bringing the software to market
- The trick is to balance the construction costs and the product quality
- Producing software using a “good enough” attitude may leave your production team exposed to serious liability issues resulting from product failures after release

- Developers need to realize the taking time to do things right mean that they don't need to find the resources to do it over again

Cost of Quality

- *Prevention costs* - quality planning, formal technical reviews, test equipment, training
- *Appraisal co*sts - in-process and inter-process inspection, equipment calibration and maintenance, testing
- *Internal failure costs* - rework, repair, failure mode analysis
- *External failure cos*ts - complaint resolution, product return and replacement, help line support, warranty work

Low Quality Software

- Low quality software increases risks for both developers and end-users
- Risks are areas of uncertainty in the development process such that if they occur may result in unwanted consequences or losses
- When systems are delivered late, fail to deliver functionality, and does not meet customer expectations litigation ensues
- Low quality software is easier to hack and can increase the security risks for the application once deployed
- A secure system cannot be built without focusing on quality (security, reliability, dependability) during the design phase
- Low quality software is liable to contain architectural flaws as well as implementation problems (bugs)

Impact of Management Actions

- Estimation decisions – irrational delivery date estimates cause teams to take short-cuts that can lead to reduced product quality
- Scheduling decisions – failing to pay attention to task dependencies when creating the project schedule may force the project team to test modules without their subcomponents and quality may suffer
- Risk-oriented decisions – reacting to each crisis as it arises rather than building in mechanisms to monitor risks and having established contingency plans may result in products having reduced quality

Achieving Software Quality

- Software quality is the result of good project management and solid engineering practice
- To build high quality software you must understand the problem to be solved and be capable of creating a quality design the conforms to the problem requirements
- Eliminating architectural flaws during design can improve quality
- Project management – project plan includes explicit techniques for quality and change management

- Quality control - series of inspections, reviews, and tests used to ensure conformance of a work product to its specifications
- Quality assurance - consists of the auditing and reporting procedures used to provide management with data needed to make proactive decisions

Review Techniques

Overview

People discover mistakes as they develop software engineering work products. Technical reviews are the most effective technique for finding mistakes early in the software process. If you find an error early in the process, it is less expensive to correct. In addition, errors have a way of amplifying as the process proceeds. Reviews save time by reducing the amount of rework that will be required late in the project. In general, six steps are employed: planning, preparation, structuring the meeting, noting errors, making corrections, and verifying that corrections have been performed properly. The output of a review is a list of issues and/or errors that have been uncovered, as well as the technical status of the work product reviewed.

Review Goals

- Point out needed improvements in the product of a single person or team
- Confirm those parts of a product in which improvement is either not desired or not needed
- Achieve technical work of more uniform, or at least more predictable, quality than can be achieved without reviews, in order to make technical work more manageable

Software Defect Cost Impact

- Defects or faults are quality problems discovered after software has been released to end-user or another software process framework activity
- Industry studies suggest that design activities introduce 50-65% of all defects or errors during the software process
- Review techniques have been shown to be up to 75% effective in uncovering design flaws which ultimately reduces the cost of subsequent activities in the software process
- Defect amplification models can be used to show that the benefits of detecting and removing defects from activities that occur early in the software process

Review Metrics

- *Preparation effort,* E_p - the effort (in person-hours) required to review a work product prior to the actual review meeting
- *Assessment effort,* E_a - the effort (in person-hours) that is expending during the actual review
- *Rework effort,* E_r - the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- *Work product size, WPS* - a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)

- *Minor errors found,* Err_{minor} - the number of errors found that can be categorized as minor (requiring less than some pre-specified effort to correct)
- *Major errors found,* Err_{major} - the number of errors found that can be categorized as major (requiring more than some pre-specified effort to correct)
- *Total review effort,* $E_{review} = E_p + E_a + E_r$
- *Total number of errors discovered,* $Err_{tot} = Err_{minir} + Err_{major}$
- *Defect density* $= Err_{tot} / WPS$

Review Cost Effectiveness

- Software review organizations can only assess the effectiveness and cost benefits after reviews are completed, review metrics collected, average data computed, and downstream software quality is measured by testing
- Some people have found a 10 to 1 return on inspection costs, accelerated product delivery times, and productivity increases
- Review costs benefits are most pronounced during the latter phases of software process leading up to product deployment

Review Formality

- Review formality increases as:
  - degree to which distinct roles are defined for the reviewers
  - amount of planning and preparation for the review increases
  - distinct structure for review (including tasks and internal work products) is defined
  - follow-up the reviewers occurs for any corrections that are made

Informal Reviews

- Simple desk check of a work product or casual meeting
- Efficacy of informal reviews is improved by developing and using checklists for each major work product to be reviewed
- Pair programming might be viewed as continuous as relying on continuous desk checks as code is being created

Formal Technical Review (FTR) Objectives

- Uncover errors in function, logic, or implementation for any representation of the software
- Verify that the software under review meets its requirements
- Ensure that the software has been represented according to predefined standards
- Achieve software that is developed in a uniform manner
- Make projects more manageable
- Serve as a training ground, enabling junior engineers to observe different approaches to software analysis, design, and implementation
- Serves to promote backup and continuity because a number of people become familiar with parts of the software that they may not have otherwise seen

Formal Technical Reviews

- Involves 3 to 5 people (including reviewers)
- Advance preparation (no more than 2 hours per person) required
- Duration of review meeting should be less than 2 hours
- Focus of review (walkthrough or inspection) is on a discrete work product
- Review leader organizes the review meeting at the producer's request
- Reviewers ask questions that enable the producer to discover his or her own error (the product is under review not the producer)
- Producer of the work product walks the reviewers through the product
- Recorder writes down any significant issues raised during the review
- Reviewers decide to accept or reject the work product and whether to require additional reviews of product or not

Review Summary Report

- What was reviewed?
- Who reviewed it?
- What were the findings and conclusions?

Review Issues List

- Identifies problem areas within product
- Serves as action list to guide the work product creator as corrections are made

Formal Technical Review Guidelines

1. Review the product not the producer.
2. Set an agenda and maintain it.
3. Limit rebuttal and debate.
4. Enunciate problem area, but don’t attempt to solve every problem noted.
5. Take written notes.
6. Limit number of participants and insist on advance preparation.
7. Develop a checklist for each product that is likely to be reviewed.
8. Allocate resources and schedule time for all reviewers.
9. Conduct meaningful training for all reviewers.
10. Review your early reviews,

Sample Driven Reviews

- Samples of all software engineering work products are reviewed to determine the most error-prone
- Full FTR resources are focused on the likely error-prone work products based on sampling results
- To be effective the sample driven review process must be driven by quantitative measures of the work products

1. Inspect a representative fraction of the content of each software work product (i) and record the number of faults (f_i) found within (a_i)
2. Develop a gross estimate of the number of faults within work product i by multiplying f_i by $1/a_i$
3. Sort work products in descending order according to the gross estimate of the number of faults in each
4. Focus on available review resources on those work products with the highest estimated number of faults

Software Quality Assurance

Overview

This chapter provides an introduction to software quality assurance. Software quality assurance (SQA) is the concern of every software engineer to reduce costs and improve product time-to-market. A Software Quality Assurance Plan is not merely another name for a test plan, though test plans are included in an SQA plan. SQA activities are performed on every software project. Use of metrics is an important part of developing a strategy to improve the quality of both software processes and work products.

Software Quality Assurance

- Umbrella activity applied throughout the software process
- Planned and systematic pattern of actions required to ensure high quality in software
- Responsibility of many stakeholders (software engineers, project managers, customers, salespeople, SQA group)

SQA Questions

- Does the software adequately meet its quality factors?
- Has software development been conducted according to pre-established standards?
- Have technical disciplines performed their SQA roles properly?

Quality Assurance Elements

- Standards – ensure that standards are adopted and follwed
- Reviews and audits – audits are reviews performed by SQA personnel to ensure hat quality guidelines are followed for all software engineering work
- Testing – ensure that testing id properly planned and conducted
- Error/defect collection and analysis – collects and analyses error and defect data to better understand how errors are introduced and can be eliminated
- Changes management – ensures that adequate change management practices have been instituted
- Education – takes lead in software process improvement and educational program
- Vendor management – suggests specific quality practices vendor should follow and incorporates quality mandates in vendor contracts
- Security management – ensures use of appropriate process and technology to achieve desired security level

- Safety – responsible for assessing impact of software failure and initiating steps to reduce risk
- Risk management – ensures risk management activities are properly conducted and that contingency plans have been established

SQA Tasks

- Prepare SQA plan for the project.
- Participate in the development of the project's software process description.
- Review software engineering activities to verify compliance with the defined software process.
- Audit designated software work products to verify compliance with those defined as part of the software process.
- Ensure that any deviations in software or work products are documented and handled according to a documented procedure.
- Record any evidence of noncompliance and reports them to management.

SQA Goals

- Requirements quality
  - Ambiguity
  - Completeness
  - Volatility
  - Traceability
  - Model clarity
- Design quality
  - Architectural integrity
  - Component completeness
  - Interface complexity
  - Patterns
- Code quality
  - Complexity
  - Maintainability
  - Understandability
  - Reusability
  - Documentation
- Quality control effectiveness
  - Resource allocation
  - Completion rate
  - Review effectiveness
  - Testing effectiveness

Formal SQA

- Assumes that a rigorous syntax and semantics can be defined for every programming language

- Allows the use of a rigorous approach to the specification of software requirements
- Applies mathematical proof of correctness techniques to demonstrate that a program conforms to its specification

Statistical Quality Assurance

1. Information about software defects is collected and categorized
2. Each defect is traced back to its cause
3. Using the Pareto principle (80% of the defects can be traced to 20% of the causes) isolate the "vital few" defect causes
4. Move to correct the problems that caused the defects in the “vital few”

Six Sigma Software Engineering

- *Define* customer requirements, deliverables, and project goals via well-defined methods of customer communication.
- *Measure* each existing process and its output to determine current quality performance (e.g. compute defect metrics)
- *Analyze* defect metrics and determine viral few causes.
- For an existing process that needs improvement
    - I*mprove* process by eliminating the root causes for defects
    - *Control* future work to ensure that future work does not reintroduce causes of defects
- If new processes are being developed
    - *Design* each new process to avoid root causes of defects and to meet customer requirements
    - *Verify* that the process model will avoid defects and meet customer requirements

Software Reliability

- Defined as the probability of failure free operation of a computer program in a specified environment for a specified time period
- Can be measured directly and estimated using historical and developmental data (unlike many other software quality factors)
- Software reliability problems can usually be traced back to errors in design or implementation.
- Measures of Reliability
    - Mean time between failure (MTBF) = MTTF + MTTR
    - MTTF = mean time to failure
    - MTTR = mean time to repair
    - Availability = [MTTF / (MTTF + MTTR)] x 100%

Software Safety

- Defined as a software quality assurance activity that focuses on identifying potential hazards that may cause a software system to fail.
- Early identification of software hazards allows developers to specify design features to can eliminate or at least control the impact of potential hazards.

- Software reliability involves determining the likelihood that a failure will occur, while software safety examines the ways in which failures may result in conditions that can lead to a mishap.

ISO 9000 Quality Standards

- Quality assurance systems are defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.
- ISO 9000 describes the quality elements that must be present for a quality assurance system to be compliant with the standard, but it does not describe how an organization should implement these elements.
- ISO 9001:2000 is the quality standard that contains 20 requirements that must be present in an effective software quality assurance system.

SQA Plan

- Management section - describes the place of SQA in the structure of the organization
- Documentation section - describes each work product produced as part of the software process
- Standards, practices, and conventions section - lists all applicable standards/practices applied during the software process and any metrics to be collected as part of the software engineering work
- Reviews and audits section - provides an overview of the approach used in the reviews and audits to be conducted during the project
- Test section - references the test plan and procedure document and defines test record keeping requirements
- Problem reporting and corrective action section - defines procedures for reporting, tracking, and resolving errors or defects, identifies organizational responsibilities for these activities
- Other - tools, SQA methods, change control, record keeping, training, and risk management

Software Testing Strategies

Overview

This chapter describes several approaches to testing software. Software testing must be planned carefully to avoid wasting development time and resources. Testing begins “in the small” and progresses “to the large”. Initially individual components are tested and debugged. After the individual components have been tested and added to the system, integration testing takes place. Once the full software product is completed, system testing is performed. The Test Specification document should be reviewed like all other software engineering work products.

Strategic Approach to Software Testing

- Many software errors are eliminated before testing begins by conducting effective technical reviews
- Testing begins at the component level and works outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.

- The developer of the software conducts testing and may be assisted by independent test groups for large projects.
- Testing and debugging are different activities.
- Debugging must be accommodated in any testing strategy.

Verification and Validation

- Make a distinction between *verification* (are we building the product right?) and *validation* (are we building the right product?)
- Software testing is only one element of Software Quality Assurance (SQA)
- Quality must be built in to the development process, you can't use testing to add quality after the fact

Organizing for Software Testing

- The role of the Independent Test Group (ITG) is to remove the conflict of interest inherent when the builder is testing his or her own product.
- Misconceptions regarding the use of independent testing teams
  - The developer should do no testing at all
  - Software is tossed "over the wall" to people to test it mercilessly
  - Testers are not involved with the project until it is time for it to be tested
- The developer and ITGC must work together throughout the software project to ensure that thorough tests will be conducted

Software Testing Strategy

- Unit Testing – makes heavy use of testing techniques that exercise specific control paths to detect errors in each software component individually
- Integration Testing – focuses on issues associated with verification and program construction as components begin interacting with one another
- Validation Testing – provides assurance that the software validation criteria (established during requirements analysis) meets all functional, behavioral, and performance requirements
- System Testing – verifies that all system elements mesh properly and that overall system function and performance has been achieved

Strategic Testing Issues

- Specify product requirements in a quantifiable manner before testing starts.
- Specify testing objectives explicitly.
- Identify categories of users for the software and develop a profile for each.
- Develop a test plan that emphasizes rapid cycle testing.
- Build robust software that is designed to test itself.
- Use effective formal reviews as a filter prior to testing.
- Conduct formal technical reviews to assess the test strategy and test cases.
- Develop a continuous improvement approach for the testing process.

Unit Testing

- Module interfaces are tested for proper information flow.
- Local data are examined to ensure that integrity is maintained.
- Boundary conditions are tested.
- Basis (independent) path are tested.
- All error handling paths should be tested.
- Drivers and/or stubs need to be developed to test incomplete software.

Integration Testing

- Sandwich testing uses top-down tests for upper levels of program structure coupled with bottom-up tests for subordinate levels
- Testers should strive to indentify critical modules having the following requirements
- Overall plan for integration of software and the specific tests are documented in a test specification

Integration Testing Strategies

- Top-down integration testing
  1. Main control module used as a test driver and stubs are substitutes for components directly subordinate to it.
  2. Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
  3. Tests are conducted as each component is integrated.
  4. On completion of each set of tests and other stub is replaced with a real component.
  5. Regression testing may be used to ensure that new errors not introduced.

- Bottom-up integration testing
  1. Low level components are combined into clusters that perform a specific software function.
  2. A driver (control program) is written to coordinate test case input and output.
  3. The cluster is tested.
  4. Drivers are removed and clusters are combined moving upward in the program structure.

- Regression testing – used to check for defects propagated to other modules by changes made to existing program
  1. Representative sample of existing test cases is used to exercise all software functions.
  2. Additional test cases focusing software functions likely to be affected by the change.
  3. Tests cases that focus on the changed software components.

- Smoke testing
  1. Software components already translated into code are integrated into a build.
  2. A series of tests designed to expose errors that will keep the build from performing its functions are created.
  3. The build is integrated with the other builds and the entire product is smoke tested daily (either top-down or bottom integration may be used).

General Software Test Criteria

- Interface integrity – internal and external module interfaces are tested as each module or cluster is added to the software
- Functional validity – test to uncover functional defects in the software
- Information content – test for errors in local or global data structures
- Performance – verify specified performance bounds are tested

Object-Oriented Test Strategies

- Unit Testing – components being tested are classes not modules
- Integration Testing – as classes are integrated into the architecture regression tests are run to uncover communication and collaboration errors between objects
- Systems Testing – the system as a whole is tested to uncover requirement errors

Object-Oriented Unit Testing

- smallest testable unit is the encapsulated class or object
- similar to system testing of conventional software
- do not test operations in isolation from one another
- driven by class operations and state behavior, not algorithmic detail and data flow across module interface

Object-Oriented Integration Testing

- focuses on groups of classes that collaborate or communicate in some manner
- integration of operations one at a time into classes is often meaningless
- **thread-based testing** – testing all classes required to respond to one system input or event
- **use-based testing** – begins by testing independent classes (classes that use very few server classes) first and the dependent classes that make use of them
- **cluster testing** – groups of collaborating classes are tested for interaction errors
- regression testing is important as each thread, cluster, or subsystem is added to the system

WebApp Testing Strategies

1. WebApp content model is reviewed to uncover errors.
2. Interface model is reviewed to ensure all use-cases are accommodated.
3. Design model for WebApp is reviewed to uncover navigation errors.
4. User interface is tested to uncover presentation errors and/or navigation mechanics problems.
5. Selected functional components are unit tested.
6. Navigation throughout the architecture is tested.
7. WebApp is implemented in a variety of different environmental configurations and the compatibility of WebApp with each is assessed.
8. Security tests are conducted.
9. Performance tests are conducted.
10. WebApp is tested by a controlled and monitored group of end-users (looking for content errors, navigation errors, usability concerns, compatibility issues, reliability, and performance).

Validation Testing

- Focuses on visible user actions and user recognizable outputs from the system
- Validation tests are based on the use-case scenarios, the behavior model, and the event flow diagram created in the analysis model
  - Must ensure that each function or performance characteristic conforms to its specification.
  - Deviations (deficiencies) must be negotiated with the customer to establish a means for resolving the errors.
- Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.

Acceptance Testing

- Making sure the software works correctly for intended user in his or her normal work environment.
- Alpha test – version of the complete software is tested by customer under the supervision of the developer at the developer's site
- Beta test – version of the complete software is tested by customer at his or her own site without the developer being present

System Testing

- Series of tests whose purpose is to exercise a computer-based system
- The focus of these system tests cases identify interfacing errors
- Recovery testing – checks the system's ability to recover from failures
- Security testing – verifies that system protection mechanism prevent improper penetration or data alteration
- Stress testing – program is checked to see how well it deals with abnormal resource demands (i.e. quantity, frequency, or volume)
- Performance testing – designed to test the run-time performance of software, especially real-time software
- Deployment (or configuration) testing – exercises the software in each of the environment in which it is to operate

Bug Causes

- The symptom and the cause may be geographically remote (symptom may appear in one part of a program).
- The symptom may disappear (temporarily) when another error is corrected.
- The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
- The symptom may be caused by human error that is not easily traced.
- The symptom may be a result of timing problems, rather than processing problems.
- It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

- The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
- The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Debugging Strategies

- Debugging (removal of a defect) occurs as a consequence of successful testing.
- Some people are better at debugging than others.
- Common approaches (may be partially automated with debugging tools):
  - **Brute force** – memory dumps and run-time traces are examined for clues to error causes
  - **Backtracking** – source code is examined by looking backwards from symptom to potential causes of errors
  - **Cause elimination** – uses binary partitioning to reduce the number of locations potential where errors can exist)

Bug Removal Considerations

- Is the cause of the bug reproduced in another part of the program?
- What "next bug" might be introduced by the fix that is being proposed?
- What could have been done to prevent this bug in the first place?

Testing Conventional Applications

Overview

The importance of software testing to software quality can not be overemphasized. Once source code has been generated, software must be tested to allow errors to be identified and removed before delivery to the customer. While it is not possible to remove every error in a large software package, the software engineer's goal is to remove as many as possible early in the software development cycle. It is important to remember that testing can only find errors it cannot prove that a program is free of bugs. Two basic test techniques exist for testing coventional software, testing module input/output (black-box) and exercising the internal logic of software components (white-box). Formal technical reviews by themselves cannot find all software defects, test data must also be used. For large software projects, separate test teams may be used to develop and execute the set of test cases used in testing. Testing must be planned and designed.

Software Testing Objectives

- Testing is the process of executing a program with the intent of finding errors.
- A good test case is one with a high probability of finding an as-yet undiscovered error.
- A successful test is one that discovers an as-yet-undiscovered error.

Software Testability Checklist

- Operability – the better it works the more efficiently it can be tested
- Observabilty – what you see is what you test

- Controllability – the better software can be controlled the more testing can be automated and optimized
- Decomposability – by controlling the scope of testing, the more quickly problems can be isolated and retested intelligently
- Simplicity – the less there is to test, the more quickly we can test
- Stability – the fewer the changes, the fewer the disruptions to testing
- Understandability – the more information known, the smarter the testing

Good Test Attributes

- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be best of breed.
- A good test should not be too simple or too complex.

Test Case Design Strategies

- Black-box or behavioral testing – knowing the specified function a product is to perform and demonstrating correct operation based solely on its specification without regard for its internal logic
- White-box or glass-box testing – knowing the internal workings of a product, tests are performed to check the workings of all possible logic paths

White-Box Testing Questions

- Can you guarantee that all independent paths within a module will be executed at least once?
- Can you exercise all logical decisions on their true and false branches?
- Will all loops execute at their boundaries and within their operational bounds?
- Can you exercise internal data structures to ensure their validity?

Basis Path Testing

- White-box technique usually based on the program flow graph
- The cyclomatic complexity of the program computed from its flow graph using the formula $V(G) = E – N + 2$ or by counting the conditional statements in the program design language (PDL) representation and adding 1
- Determine the basis set of linearly independent paths (the cardinality of this set id the program cyclomatic complexity)
- Prepare test cases that will force the execution of each path in the basis set.

Control Structure Testing

- White-box technique focusing on control structures present in the software
- Condition testing (e.g. branch testing) – focuses on testing each decision statement in a software module, it is important to ensure coverage of all logical combinations of data that may be processed by the module (a truth table may be helpful)
- Data flow testing – selects test paths based according to the locations of variable definitions and uses in the program (e.g. definition use chains)

- Loop testing – focuses on the validity of the program loop constructs (i.e. simple loops, concatenated loops, nested loops, unstructured loops), involves checking to ensure loops start and stop when they are supposed to (unstructured loops should be redesigned whenever possible)

Black-Box Testing Questions

- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

Graph-based Testing Methods

- Black-box methods based on the nature of the relationships (links) among the program objects (nodes), test cases are designed to traverse the entire graph
- Transaction flow testing – nodes represent steps in some transaction and links represent logical connections between steps that need to be validated
- Finite state modeling – nodes represent user observable states of the software and links represent transitions between states
- Data flow modeling – nodes are data objects and links are transformations from one data object to another
- Timing modeling – nodes are program objects and links are sequential connections between these objects, link weights are required execution times

Equivalence Partitioning

- Black-box technique that divides the input domain into classes of data from which test cases can be derived
- An ideal test case uncovers a class of errors that might require many arbitrary test cases to be executed before a general error is observed
- Equivalence class guidelines:
  1. If input condition specifies a range, one valid and two invalid equivalence classes are defined
  2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
  3. If an input condition specifies a member of a set, one valid and one invalid equivalence class is defined
  4. If an input condition is Boolean, one valid and one invalid equivalence class is defined

Boundary Value Analysis

- Black-box technique that focuses on the boundaries of the input domain rather than its center

- BVA guidelines:
  1. If input condition specifies a range bounded by values a and b, test cases should include a and b, values just above and just below a and b
  2. If an input condition specifies and number of values, test cases should be exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values
  3. Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maxim output reports
  4. If internal program data structures have boundaries (e.g. size limitations), be certain to test the boundaries

Orthogonal Array Testing

- Black-box technique that enables the design of a reasonably small set of test cases that provide maximum test coverage
- Focus is on categories of faulty logic likely to be present in the software component (without examining the code)
- Priorities for assessing tests using an orthogonal array
  1. Detect and isolate all single mode faults
  2. Detect all double mode faults
  3. Mutimode faults

Model-Based Testing

- Black-bix testing technique using information contained in the requirements model as a basis for test case generation
- Steps for MBT
  1. Analyze an existing behavior model for the software or create one.
  2. Traverse behavioral model and specify inputs that force software to make transition from state to state.
  3. Review behavioral model and note expected outputs as software makes transition from state to state.
  4. Execute test cases.
  5. Compare actual and expected results (take corrective action as required).

Specialized Testing

- Graphical User Interface (GUI) – test cases can be developed from behavioral model of user interface, use of automated testing tools is strongly recommended.
- Client/Sever Architectures – operational profiles derived from usage scenarios are tested at three levels (client application "disconnected mode", client and server software without network, complete application)
  - Applications function tests
  - Server tests
  - Database tests
  - Transaction tests
  - Network communications tests
- Documentation and Help

    - Review and inspection to check for editorial errors
    - Black-Box for live tests
        - Graph-based testing to describe program use
        - Equivalence partitioning and boundary value analysis to describe classes of input and interactions
- Real-Time Systems
    1. Task testing – test each task independently
    2. Behavioral testing – using technique similar to equivalence partitioning of external event models created by automated tools
    3. Intertask testing – testing for timing errors (e.g. synchronization and communication errors)
    4. System testing – testing full system, especially the handling of Boolean events (interrupts), test cased based on state model and control specification

Testing Patterns

- Provide software engineers with useful advice as testing activities begin
- Provide a vocabulary for problem-solvers
- Can focus attention on forces behind a problem (when and why a solution applies)
- Encourage iterative thinking (each solution creates a new context in which problems can be solved

Testing Object-Oriented Applications

Overview

It is important to test object-oriented at several different levels to uncover errors that may occur as classes collaborate with one another and with other subsystems. The process of testing object-oriented systems begins with a review of the object-oriented analysis and design models. Once the code is written object-oriented testing begins by testing "in the small" with class testing (class operations and collaborations). As classes are integrated to become subsystems class collaboration problems are investigated using thread-based testing, use-based testing, cluster testing, and fault-based approaches. Use-cases from the analysis model are used to uncover software validation errors. The primary work product is a set of documented test cases with defined expected results and the actual results recorded.

OO Testing

- Requires definition of testing to be broadened to include error discovery techniques applied to object-oriented analysis (OOA) and deign (OOD) models
- Significant strategy changes need to be made to unit and integration testing
- Test case design must take into account the unique characteristic of object-oriented (OO) software
- All object-oriented models should be reviewed for correctness, completeness, and consistency as part of the testing process

OOA and OOD Model Review

- The analysis and design models cannot be tested because they are not executable

- The syntax correctness of the analysis and design models can check for proper use of notation and modeling conventions
- The semantic correctness of the analysis and design models are assessed based on their conformance to the real world problem domain (as determined by domain experts)

OO Model Consistency

- Judged by considering relationships among object-oriented model entities
- The analysis model can be used to facilitate the steps below for each iteration of the requirements model
  1. Revisit the CRC model and object-relationship model
  2. Inspect the description of each CRC card to determine if a delegated responsibility is part of the collaborator's definition
  3. Invert the connection to be sure that each collaborator that is asked for service is receiving requests from a reasonable source.
  4. Using the inverted connections from step 3, determine whether additional classes might be required or whether responsibilities are properly grouped among the classes.
  5. Determine whether widely requested responsibilities might be combined into a single responsibility.
- Once the deign model is created you should conduct reviews of the system design and object design
- The system design is reviewed by examining the object-behavior model and mapping the required system behavior against subsystems designed to accomplish the behavior

Software Testing Strategy for Object-Oriented Architectures

- Unit Testing – called *class testing* in OO circles, components being tested are classes and their behaviors not modules
- Integration Testing – as classes are integrated into the architecture regression tests are run to uncover communication and collaboration errors between objects
  - *Thread-based testing* – tests one thread at a time (set of classes required to respond to one input or event)
  - *Use-based testing* – tests independent classes (those that use very through server classes) first then tests dependent classes (those that use independent classes) until entire system is tested
  - *Cluster testing* – set of collaborating classes (identified from CRC card model) is exercised using test cases designed to uncover collaboration errors
- Validation Testing – testing strategy where the system as a whole is tested to uncover requirement errors, uses conventional black box testing methods

Comparison Testing

- Black-box testing for safety critical systems in which independently developed implementations of redundant systems are tested for conformance to specifications
- Often equivalence class partitioning is used to develop a common set of test cases for each implementation

OO Test Case Design

1. Each test case should be uniquely identified and be explicitly associated with a class to be tested
2. State the purpose of each test

3. List the testing steps for each test including:
   a. list of states to test for each object involved in the test
   b. list of messages and operations to exercised as a consequence of the test
   c. list of exceptions that may occur as the object is tested
   d. list of external conditions needed to be changed for the test
   e. supplementary information required to understand or implement the test

OO Test Case Design

- White-box testing methods can be applied to testing the code used to implement class operations, but not much else
- Black-box testing methods are appropriate for testing OO systems just as they are for testing conventional systems

OO Fault-Based Testing
- Best reserved for operations and the class level
- Uses the inheritance structure
- Tester examines the OOA model and hypothesizes a set of plausible defects that may be encountered in operation calls and message connections and builds appropriate test cases
- Misses incorrect specification and errors in subsystem interactions
- Finds client errors not server errors

Class Hierarchy and Test Cases

- Subclasses may contain operations that are inherited from super classes
- Subclasses may contain operations that were redefined rather than inherited
- All classes derived from a previously tested base class need to be tested thoroughly

OO Scenario-Based Testing

- Using the user tasks described in the use-cases and building the test cases from the tasks and their variants
- Uncovers errors that occur when any actor interacts with the OO software
- Concentrates on what the user does, not what the product does
- You can get a higher return on your effort by spending more time on reviewing the use-cases as they are created, than spending more time on use-case testing

OO Testing – Surface Structure and Deep Structure
- Testing surface structure – exercising the structure observable by end-user, this often involves observing and interviewing users as they manipulate system objects
- Testing deep structure – exercising internal program structure - the dependencies, behaviors, and communications mechanisms established as part of the system and object design

Class Level Testing Methods
- Random testing – requires large numbers data permutations and combinations and can be inefficient
- Partition testing – reduces the number of test cases required to test a class
  - **State-based** partitioning – tests designed in way so that operations that cause state changes are tested separately from those that do not

- **Attribute-based** partitioning – for each class attribute, operations are classified according to those that use the attribute, those that modify the attribute, and those that do not use or modify the attribute
- **Category-based** partitioning – operations are categorized according to the function they perform: initialization, computation, query, termination

Inter-Class Test Case Design

- Multiple class testing
  1. For each client class use the list of class operators to generate random test sequences that send messages to other server classes
  2. For each message generated determine the collaborator class and the corresponding server object operator
  3. For each server class operator (invoked by a client object message) determine the message it transmits
  4. For each message, determine the next level of operators that are invoked and incorporate them into the test sequence
- Tests derived from behavior models
  - Test cases must cover all states in the state transition diagram
  - Breadth first traversal of the state model can be used (test one transition at a time and only make use of previously tested transitions when testing a new transition)
  - Test cases can also be derived to ensure that all behaviors for the class have been adequately exercised

Testing Web Applications

Overview

This chapter describes Web testing as a collection of activities whose purpose is to uncover errors in WebApp content, function, usability, navigability, performance, capacity, and security. A testing startegy that involves both reviews and executable testing is applied throughout the WebE process. The WebApp testing process involves all project stakeholders. Web testing begins with user-visible aspects of WebApps and proceeds to exercise technology and infrastructure. Seven testing steps are performed: content testing, interface testing, navigation testing, component testing, configuration testing, performance testing, and security testing. In sometimes a test plan is written. A suite of test cases is always developed for every testing step and an archive of testing results is maintained for future use.

Dimensions of Quality

- *Content* evaluated at both syntactic and semantic levels
- *Function* tested to uncover lack of conformance to requirements
- *Structure* is assessed to ensure proper content and function are delivered
- *Usability* is tested to ensure that each category of user can be supported as new content or functionality is added
- *Navigability* is tested to ensure that all navigation syntax and semantics are exercised
- *Performance* is tested under a variety of operating conditions, configurations, and loading to ensure a reasonable level of user response
- *Compatibility* tested by executing WebApp using a variety of client and server configurations
- *Interoperability* tested to ensure proper interfaces to other applications and databases
- *Security* is tested by assessing potential vulnerabilities and trying to exploit each of them

Characteristics of WebApp Errors

- Many types of WebApp tests uncover problems evidenced on the client side using an specific interface (e.g. may be an error symptom, not the error itself)
- It may be difficult to reproduce errors outside of the environment in which the error was originally encountered
- Many errors can be traced to the WebApp configuration, incorrect design, or improper HTML
- It is hard to determine whether errors are caused by problems with the server, the client, or the network itself
- Some errors are attributable to problems in the static operating environment and some are attributable to the dynamic operating environment

Testing WebApps for Errors

11. WebApp content model is reviewed to uncover errors.
12. Interface model is reviewed to ensure all use-cases are accommodated.
13. Design model for WebApp is reviewed to uncover navigation errors.
14. User interface is tested to uncover presentation errors and/or navigation mechanics problems.
15. Selected functional components are unit tested.
16. Navigation throughout the architecture is tested.
17. WebApp is implemented in a variety of different environmental configurations and the compatibility of WebApp with each is assessed.
18. Security tests are conducted.
19. Performance tests are conducted.
20. WebApp is tested by a controlled and monitored group of end-users (looking for content errors, navigation errors, usability concerns, compatibility issues, reliability, and performance).

Web Engineering Test Plan Elements

1. Task set to be applied during testing
2. Work products to be produced as each testing task is executed
3. Evaluation and recording methods for testing results

Web Testing Process

- Content testing – tries to uncover content errors
- Interface testing – exercises interaction mechanisms and validates aesthetic aspects of user interface
- Navigation testing – makes use of use-cases in the design of test cases that exercise each usage scenario against the navigation design (used as part of WebApp integration testing)
- Component testing – exercises the WebApp content and functional units (used as part of WebApp integration testing)
- Configuration testing – attempts to uncover errors traceable to a specific client or server environment (cross-reference table is useful)

- Security testing – tests designed to exploit WebApp or environment vulnerabilities
- Performance testing – series of tests designed to assess:
  - WebApp response time and reliability under varying system loads
  - Which WebApp components are responsible for system degradation
  - How performance degradation impacts overall WebApp requirements

Content Testing Objectives

- Uncover syntactic errors in all media (e.g. typos)
- Uncover semantic errors (e.g. errors in completeness or accuracy)
- Find errors in organization or structure of content presented to end-user
- Questions to be answered
  - Is the information factually accurate?
  - Is the information concise and to the point?
  - Is the layout of the content object easy for the user to understand?
  - Can information embedded within a content object be found easily?
  - Have proper references been provided for all information derived from other sources?
  - Is the information presented consistent internally and consistent with information presented in other content objects?
  - Is the content offensive, misleading, or does it open the door to litigation?
  - Does the content infringe on existing copyrights or trademarks?
  - Does the content contain internal links that supplement existing content? Are the links correct?
  - Does the aesthetic style of the content conflict with the aesthetic style of the interface?

Database Testing Problems

- The original query must be checked to uncover errors in translating the user's request to SQL
- Problems in communicating between the WebApp server and Database server need to be tested.
- Need to demonstrate the validity of the raw data from the database to the WebApp and the validity of the transformations applied to the raw data.
- Need to test validity of dynamic content object formats transmitted to the user and the validity of the transformations to make the data visible to the user.

User Interface Testing

- During requirements testing the interface model reviewed to ensure it corresponds to stakeholder requirements and the requirements model
- During design interface model is reviewed to ensure generic user interface quality criteria have been achieved and that application-specific issues have been properly addressed
- During testing focus shifts to application-specific aspects of user intyeraction as manifested the user interface syntax and semantics

User Interface Testing Strategy

- Interface features are tested to ensure that design rules, aesthetics, and related visual content

is available for user without error.
- Individual interface mechanisms are tested using unit testing strategies.
- Each interface mechanism is tested in the context of a use-case of navigation semantic unit (e.g. thread) for a specific user category
- Complete interface is tested against selected use-cases and navigation semantic unit to uncover interface semantic errors
- Interface is tested in a variety of environments to ensure compatibility

Testable WebApp Interface Mechanisms

- Links (each link is listed and tested)
- Forms (check labels, field navigation, data entry, error checking, data transmission, meaningful error messages)
- Client-side scripting (black box testing and compatibility tests)
- Dynamic HTML (correctness of generated HTML and compatibility tests)
- Client-side pop-up windows (proper size and placement of pop-up, working controls, consistent with aesthetic appearance of Web page)
- CGI scripts (black box, data integrity, and performance testing)
- Streaming content (demonstrate existence, accuracy, and control over content display)
- Cookies (check that server constructs cookie correctly, cookie transmitted correctly, ensure proper level of persistence, check to see WebApp attaches the correct cookies to server requests)
- Application specific interface mechanisms

Usability Testing

- Define set of usability testing categories and identify goals for each
  - *Interactivity* – interaction mechanisms are easy to understand and use
  - *Layout* – navigation, content, and functions allows user to find them quickly
  - *Readability* – content understandable
  - *Aesthetics* – graphic design supports easy of use
  - *Display characteristics* – WebApp makes good use of screen size and resolution
  - *Time sensitivity* – content and features can be acquired in timely manner
  - *Personalization* – adaptive interfaces
  - *Accessibility* – special needs users
- Design tests the will enable each goal to be evaluated
- Select participants to conduct the tests
- Instrument participants' interactions with the WebApp during testing
- Develop method for assessing usability of the WebApp

Compatibility Testing

- Define a set of commonly encountered client-side computing configurations and their variants
- Organize this information (computing platform, typical display devices, operating system, available browsers, connection speeds) in a tree structure
- Derive compatibility validation test suite from existing interface tests, navigation tests, performance tests, and security tests

- Goal is to uncover execution problems that can be traced to configuration differences

Component-Level (Function) Testing

- Black box and white box testing of each WebApp function
- Useful test case design methods
  - Equivalence partitioning
  - Boundary value analysis (esp. form field values)
  - Path testing
  - Forced error testing

Navigation Testing

- Need to ensure that all mechanisms that allow the WebApp to user to travel through the WebApp are functional
- Need to validate that each navigation semantic unit (NSU) can be achieved by the appropriate user category

Testing Navigation Syntax

- Navigational Links
- Redirects
- Bookmarks
- Frames and framesets
- Site maps
- Internal search engines

Testing Navigation Semantics

- Navigation semantic units are defined by a set of pathways that connect navigation nodes
- Each NSU must allows a user from a defined user category achieve specific requirements defined by a use-case
- Testing needs to ensure that each path is executed in its entity without error
- Every relevant path must be tested
- User must be given guidance to follow or discontinue each path based on current location in site map

Configuration Testing

- Server-side Issues
  - Compatibility of WebApp with server OS
  - Correct file and directory creation by WebApp
  - System security measures do not degrade user service by WebApp
  - Testing WebApp with distributed server configuration
  - WebApp properly integrated with database software
  - Correct execution of WebApp scripts
  - Examination system administration errors for impact on WebApp
  - On-site testing of proxy servers
- Client-side issues

- Hardware
- Operating systems
- Browser software
- User interface components
- Plug-ins
- Connectivity

Testable Security Elements

- Firewalls
- Authentication
- Encryption
- Authorization

Performance Testing

- Used to performance problems that can result from lack of server-side resources, inappropriate network bandwidth, inadequate database capabilities, faulty operating system capabilities, poorly designed WebApp functionality, and hardware/software issues
- Intent is to discover how system responds to loading and collect metrics that will lead to improve performance
  - Does the server response time degrade to a point where it is noticeable and unacceptable?
  - At what point (in terms of users, transactions or data loading) does performance become unacceptable?
  - What system components are responsible for performance degradation?
  - What is the average response time for users under a variety of loading conditions?
  - Does performance degradation have an impact on system security?
  - Is WebApp reliability or accuracy affected as the load on the system grows?
  - What happens when loads that are greater than maximum server capacity are applied?
  - Does performance degradation have an impact on company revenues?

Performance – Load Testing

- Examines real-world conditions at variety of load level and in a variety of combinations
- Determine combinations of N, T, and D that cause performance to degrade or fail completely
  N = number of concurrent users
  T = number of on-line transactions per unit of time
  D = data load processed by server per transaction
- Overall through put is computed using the equation
  $P = N * T * D$

Performance – Stress Testing

- Forces loading to be increases to breaking point to determine how much capacity the WebApp can handle
  - Does system degrade gracefully?
  - Are users made aware that they cannot reach the server?

- Does server queue resource requests during heavy demand and then process the queue when demand lessens?
- Are transactions lost as capacity is exceeded?
- Is data integrity affected when capacity is exceeded?
- How long till system comes back on-line after a failure?
- Are certain WebApp functions discontinued as capacity is reached?

Product Metrics for Software

Overview

This chapter describes the use of product metrics in the software quality assurance process. Software engineers use product metrics to help them assess the quality of the design and construction the software product being built. Product metrics provide software engineers with a basis to conduct analysis, design, coding, and testing more objectively. Qualitative criteria for assessing software quality are not always sufficient by themselves. The process of using product metrics begins by deriving the software measures and metrics that are appropriate for the software representation under consideration. Then data are collected and metrics are computed. The metrics are computed and compared to pre-established guidelines and historical data. The results of these comparisons are used to guide modifications made to work products arising from analysis, design, coding, or testing.

Definitions

- Measure – provides a quantitative indication of the extent, amount, capacity, or size of some attribute of a product or process
- Measurement – act of determining a measure
- Metric – statistic that relates individual measures to one another
- Indicator – metric or combination of metrics that provide insight into the software process, software project, or the product itself to make things better

Benefits of Product Metrics

1. Assist in the evaluation of the analysis and evaluation model
2. Provide indication of procedural design complexity and source code complexity
3. Facilitate design of more effective testing

Measurement Process Activities

- *Formulation* – derivation of software measures and metrics appropriate for software representation being considered
- *Collection* – mechanism used to accumulate the date used to derive the software metrics
- *Analysis* – computation of metrics
- *Interpretation* – evaluation of metrics that results in gaining insight into quality of the work product
- *Feedback* – recommendations derived from interpretation of the metrics is transmitted to the software development team

Metrics Characterization and Validation Principles

- A metric should have desirable mathematical properties
- The value of a metric should increase when positive software traits occur or decrease when undesirable software traits are encountered
- Each metric should be validated empirically in several contexts before it is used to make decisions

Measurement Collection and Analysis Principles

1. Automate data collection and analysis whenever possible
2. Use valid statistical techniques to establish relationships between internal product attributes and external quality characteristics
3. Establish interpretive guidelines and recommendations for each metric

Goal-Oriented Software Measurement (GQM)

- A goal definition template can be used to define each measurement goal
- GQM emphasizes the need
    1. establish explicit measurement goal specific to the process activity or product characteristic being assessed
    2. define a set of questions that must be answered in order to achieve the goal
    3. identify well-formulated metrics that help to answer these questions

Attributes of Effective Software Metrics

- Simple and computable
- Empirically and intuitively persuasive
- Consistent and objective
- Consistent in use of units and measures
- Programming language independent
- Provide an effective mechanism for quality feedback

Requirements Model Metrics

- Function-based metrics
    - Function points
- Specification quality metrics (Davis)\
    - Specificity
    - Completeness

Architectural Design Metrics

- Structural complexity (based on module fanout)
- Data complexity (based on module interface inputs and outputs)
- System complexity (sum of structural and data complexity)

- Morphology (number of nodes and arcs in program graph)
- Design structure quality index (DSQI)

OO Design Metrics

- **Size**(population, volume, length, functionality)
- **Complexity** (how classes interrelate to one another)
- **Coupling** (physical connections between design elements)
- **Sufficiency** (how well design components reflect all properties of the problem domain)
- **Completeness** (coverage of all parts of problem domain)
- **Cohesion** (manner in which all operations work together)
- **Primitiveness** (degree to which attributes and operations are atomic)
- **Similarity** (degree to which two or more classes are alike)
- **Volatility** (likelihood a design component will change)

Class-Oriented Metrics

- Chidamber and Kemerer (CK) Metrics Suite
  - weighted metrics per class (WMC)
  - depth of inheritance tree (DIT)
  - number of children (NOC)
  - coupling between object classes (CBO)
  - response for a class(RFC)
  - lack of cohesion in methods (LCOM)
- Harrison, Counsel, and Nithi (MOOD) Metrics Suite
  - method inheritance factor (MIF)
  - coupling factor (CF)
  - polymorphism factor (PF)
- Lorenz and Kidd
  - class size (CS)
  - number of operations overridden by a subclass (NOO)
  - number of operations added by a subclass (NOA)
  - specialization index (SI)

Component-Level Design Metrics

- Cohesion metrics (data slice, data tokens, glue tokens, superglue tokens, stickiness)
- Coupling metrics (data and control flow, global, environmental)
- Complexity metrics (e.g. cyclomatic complexity)

Operation-Oriented Metrics

- Average operation size (OSavg)
- Operation complexity (OC)
- Average number of parameters per operation (NPavg)

Using WebApp Design Metrics

- Is the WebApp interface usable?
- Are the aesthetics of the WebApp pleasing to the user and appropriate for the information domain?
- Is the content designed to impart the most information for the least amount of effort?
- Is navigation efficient and straightforward?
- Has the WebApp architecture been designed to accommodate special goals and objectives of users, content structure, functionality, and effective navigation flow?
- Are the WebApp components designed to reduce procedural complexity and enhance correctness, reliability, and performance?

WebApp Interface Metrics

- Layout appropriateness
- Layout complexity
- Layout region complexity
- Recognition complexity
- Recognition time
- Typing effort
- Mouse pick effort
- Selection complexity
- Content acquisition time

Aesthetic (graphic layout) metrics

- Word count
- Body text percentage
- Emphasized body text %
- Text positioning count
- Text cluster count
- Link count
- Page size
- Graphic percentage
- Graphics count
- Color count
- Font count

Content Metrics

- Page wait
- Page complexity
- Graphic complexity
- Audio complexity
- Video complexity
- Animation complexity
- Scanned image complexity

Navigation Metrics

- Page link complexity
- Connectivity
- Connectivity density

Halstead's Software Science (Source Code Metrics)

- Overall program length
- Potential minimum algorithm volume
- Actual algorithm volume (number of bits used to specify program)
- Program level (software complexity)
- Language level (constant for given language)

Testing Metrics

- Metrics that predict the likely number of tests required during various testing phases
  - Architectural design metrics
  - Cyclomatic complexity can target modules that are candidates for extensive unit testing
  - Halstead effort
- Metrics that focus on test coverage for a given component
  - Cyclomatic complexity lies at the core of basis path testing

Object-Oriented Testing Metrics

- Encapsulation
  - Lack of cohesion in methods (LCOM)
  - Percent public and protected (PAP)
  - Public access to data members (PAD)
- Inheritance
  - Number of root classes (NOR)
  - Fan in (FIN)
  - Number of children (NOC)
  - Depth of inheritance tree (DIT)

Maintenance Metrics

- Software Maturity Index (IEEE Standard 982.1-1988)
- SMI approaches 1.0 as product begins to stabilize

Project Management Concepts

Overview

- Project management involves the planning, monitoring, and control of people, process, and events that occur during software development.

- Everyone manages, but the scope of each person's management activities varies according his or her role in the project.
- Software needs to be managed because it is a complex undertaking with a long duration time.
- Managers must focus on the fours P's to be successful (people, product, process, and project).
- A project plan is a document that defines the four P's in such a way as to ensure a cost effective, high quality software product.
- The only way to be sure that a project plan worked correctly is by observing that a high quality product was delivered on time and under budget.

Management Spectrum

- People (recruiting, selection, performance management, training, compensation, career development, organization, work design, team/culture development)
- Product (product objectives, scope, alternative solutions, constraint tradeoffs)
- Process (framework activities populated with tasks, milestones, work products, and QA points)
- Project (planning, monitoring, controlling)

People

- Stakeholders (senior managers, project managers, practitioners, customers, end-users)
- Team leadership model (motivation, organization, innovation)
- Characteristics of effective project managers (problem solving, managerial identity, achievement, influence and team building)

Factors Affecting Team Organization

- Difficulty of problem to be solved
- Size of resulting program
- Team lifetime
- Degree to which problem can be modularized
- Required quality and reliability of the system to be built
- Rigidity of the delivery date
- Degree of communication required for the project

Team Organizational Paradigms

- *Closed paradigm* (top level problem solving and internal coordination managed by team leader, good for projects that repeat past efforts)
- *Random paradigm* (team loosely structured success depends on initiative of individual team members, paradigm excels when innovation and technical breakthroughs required)
- *Open paradigm* (rotating task coordinators and group consensus, good for solving complex problems – not always efficient as other paradigms)
- *Synchronous paradigm* (relies on natural problem compartmentalization and team organized to require little active communication with each other)

Toxic Team Environment Characteristics

1. Frenzied work atmosphere where team members waste energy and lose focus on work objectives
2. High frustration and group friction caused by personal, business, or technological problems
3. Fragmented or poorly coordinated procedures or improperly chosen process model blocks accomplishments
4. Unclear role definition that results in lack of accountability or finger pointing
5. Repeated exposure to failure that leads to loss of confidence and lower morale

Agile Teams

- Teams have significant autonomy to make their own project management and technical decisions
- Planning kept to minimum and is constrained only by business requirements and organizational standards
- Team self-organizes as project proceeds to maximum contributes of each individual's talents
- May conduct daily (10 – 20 minute) meeting to synchronized and coordinate each day's work
    - What has been accomplished since the last meeting?
    - What needs to be accomplished by the next meeting?
    - How will each team member contribute to accomplishing what needs to be done?
    - What roadblocks exist that have to be overcome?

Coordination and Communication Issues

- Formal, impersonal approaches (e.g. documents, milestones, memos)
- Formal interpersonal approaches (e.g. review meetings, inspections)
- Informal interpersonal approaches (e.g. information meetings, problem solving)
- Electronic communication (e.g. e-mail, bulletin boards, video conferencing)
- Interpersonal networking (e.g. informal discussion with people other than project team members)

The Product

- Software scope (context, information objectives, function, and performance)
- Problem decomposition (partitioning or problem elaboration - focus is on functionality to be delivered and the process used to deliver it)

The Process

- Process model chosen must be appropriate for the:
    - customers and developers
    - characteristics of the product
    - project development environment
- Project planning begins with melding the product and the process
- Each function to be engineered must pass though the set of framework activities defined for a software organization

- Work tasks may vary but the common process framework (CPF) is invariant (project size does not change the CPF)
- The detail of the actual work tasks used to complete each framework activity and dependent on the size and complexity of the project
- The job of the software engineer is to estimate the resources required to move each function though the framework activities to produce each work product
- Project decomposition begins when the project manager tries to determine how to accomplish each CPF activity

Signs of Potential Project Failure

1. Developers do not understand customer's needs
2. Product scope poorly defined
3. Changes poorly managed
4. Chosen technology changes
5. Business needs change or ill-defined
6. Deadlines unrealistic
7. Users are resistant
8. Sponsorship lost or never obtained
9. Project team members lack appropriate skills
10. Managers and practitioners avoid best practices and lessons learned

Avoiding Project Failure

1. Start on the right foot
2. Maintain momentum
3. Track progress
4. Make smart decisions
5. Conduct a postmortem analysis

W5HH Principle

- Why is the system being developed?
- What will be done by When?
- Who is responsible for a function?
- Where are they organizationally located?
- How will the job be done technically and managerially?
- How much of each resource is needed?

Critical Practices

- Formal risk management
- Empirical cost and schedule estimation
- Metric-based project management
- Earned value tracking
- Defect tracking against quality targets
- People-aware program management

Process and Project Metrics

Overview

Software process and project metrics are quantitative measures that enable software engineers to gain insight into the efficiency of the software process and the projects conducted using the process framework. In software project management, we are primarily concerned with productivity and quality metrics. There are four reasons for measuring software processes, products, and resources (to characterize, to evaluate, to predict, and to improve).

Process and Project Metrics

- Metrics should be collected so that process and product indicators can be ascertained
- *Process metrics* used to provide indictors that lead to long term process improvement
- *Project metrics* enable project manager to
  - Assess status of ongoing project
  - Track potential risks
  - Uncover problem are before they go critical
  - Adjust work flow or tasks
  - Evaluate the project team's ability to control quality of software wrok products

Process Metrics

- Private process metrics (e.g. defect rates by individual or module) are only known to by the individual or team concerned.
- Public process metrics enable organizations to make strategic changes to improve the software process.
- Metrics should not be used to evaluate the performance of individuals.
- Statistical software process improvement helps and organization to discover where they are strong and where are week.

Statistical Process Control

1. Errors are categorized by their origin
2. Record cost to correct each error and defect
3. Count number of errors and defects in each category
4. Overall cost of errors and defects computed for each category
5. Identify category with greatest cost to organization
6. Develop plans to eliminate the most costly class of errors and defects or at least reduce their frequency

Project Metrics

- A software team can use software project metrics to adapt project workflow and technical activities.
- Project metrics are used to avoid development schedule delays, to mitigate potential risks, and to assess product quality on an on-going basis.
- Every project should measure its inputs (resources), outputs (deliverables), and results (effectiveness of deliverables).

Software Measurement

- *Direct process measures* include cost and effort.
- *Direct process measures* include lines of code (LOC), execution speed, memory size, defects rep orted over some time period.
- *Indirect product measures* examine the quality of the software product itself (e.g. functionality, complexity, efficiency, reliability, maintainability).

Size-Oriented Metrics

- Derived by normalizing (dividing) any direct measure (e.g. defects or human effort) associated with the product or project by LOC.
- Size oriented metrics are widely used but their validity and applicability is widely debated.

Function-Oriented Metrics

- Function points are computed from direct measures of the information domain of a business software application and assessment of its complexity.
- Once computed function points are used like LOC to normalize measures for software productivity, quality, and other attributes.
- The relationship of LOC and function points depends on the language used to implement the software.

Reconciling LOC and FP Metrics

- The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design
- Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost
- Using LOC and FP for estimation a historical baseline of information must be established.

Object-Oriented Metrics

- Number of scenario scripts (NSS)
- Number of key classes (NKC)
- Number of support classes (e.g. UI classes, database access classes, computations classes, etc.)
- Average number of support classes per key class
- Number of subsystems (NSUB)

Use Case-Oriented Metrics

- Describe (indirectly) user-visible functions and features in language independent manner
- Number of use case is directly proportional to LOC size of application and number of test cases needed
- However use cases do not come in standard sizes and use as a normalization measure is suspect
- Use case points have been suggested as a mechanism for estimating effort

WebApp Project Metrics

- Number of static Web pages (N_{sp})
- Number of dynamic Web pages (N_{dp})
- Customization index: $C = N_{sp} / (N_{dp} + N_{sp})$
- Number of internal page links
- Number of persistent data objects
- Number of external systems interfaced
- Number of static content objects
- Number of dynamic content objects
- Number of executable functions

Software Quality Metrics

- Factors assessing software quality come from three distinct points of view (product operation, product revision, product modification).
- Software quality factors requiring measures include
  - correctness (defects per KLOC)
  - maintainability (mean time to change)
  - integrity (threat and security)
  - usability (easy to learn, easy to use, productivity increase, user attitude)
- Defect removal efficiency (DRE) is a measure of the filtering ability of the quality assurance and control activities as they are applied through out the process framework

  DRE = E / (E + D)
  E = number of errors found before delivery of work product
  D = number of defects found after work product delivery

Integrating Metrics with Software Process

- Many software developers do not collect measures.
- Without measurement it is impossible to determine whether a process is improving or not.
- Baseline metrics data should be collected from a large, representative sampling of past software projects.
- Getting this historic project data is very difficult, if the previous developers did not collect data in an on-going manner.

Arguments for Software Metrics

- If you don't measure you have no way of determining any improvement
- By requesting and evaluating productivity and quality measures software teams can establish meaningful goals for process improvement
- Software project managers are concerned with developing project estimates, producing high quality systems, and delivering product on time
- Using measurement to establish a project baseline helps to make project managers tasks possible

Baselines

- Establishing a metrics baseline can benefit portions of the process, project, and product levels
- Baseline data must often be collected by historical investigation of past project (better to collect while projects are on-going)
- To be effective the baseline data needs to have the following attributes:
  - data must be reasonably accurate, not guesstimates
  - data should be collected for as many projects as possible
  - measures must be consistent
  - applications should be similar to work that is to be estimated

Metrics for Small Organizations

- Most software organizations have fewer than 20 software engineers.
- Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect.
- Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

Establishing a Software Metrics Program

1. Identify business goal
2. Identify what you want to know
3. Identify subgoals
4. Identify subgoal entities and attributes
5. Formalize measurement goals
6. Identify quantifiable questions and indicators related to subgoals
7. Identify data elements needed to be collected to construct the indicators
8. Define measures to be used and create operational definitions for them
9. Identify actions needed to implement the measures
10. Prepare a plan to implement the measures

Estimation for Software Projects

Overview

Software planning involves estimating how much time, effort, money, and resources will be required to build a specific software system. After the project scope is determined and the problem is decomposed into smaller problems, software managers use historical project data (as well as personal experience and intuition) to determine estimates for each. The final estimates are typically adjusted by taking project complexity and risk into account. The resulting work product is called a project management plan. Managers will not know that they have done a good job estimating until the project post mortem. It is essential to track resources and revise estimates as a project progresses.

Project Planning Objectives

- To provide a framework that enables software manager to make a reasonable estimate of resources, cost, and schedule.
- 'Best case' and 'worst case' scenarios should be used to bound project outcomes.
- Estimates should be updated as the project progresses.

Estimation Reliability Factors

- Project complexity
- Project size
- Degree of structural uncertainty (degree to which requirements have solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information processed)
- Availability of historical information

Project Planning Process

1. Establish project scope
2. Determine feasibility
3. Analyze risks
4. Determine requires resources
    a. Determine required human resources
    b. Define reusable software resources
    c. Identify environmental resources
5. Estimate cost and effort
    a. Decompose the problem
    b. Develop two or more estimates
    c. Reconcile the estimates
6. Develop project schedule
    a. Establish a meaningful task set
    b. Define task network
    c. Use scheduling tools to develop timeline chart

d. Define schedule tracking mechanisms

Software Scope

- Describes the data to be processed and produced, control parameters, function, performance, constraints, external interfaces, and reliability.
- Often functions described in the software scope statement are refined to allow for better estimates of cost and schedule.

Customer Communication and Scope

- Determine the customer's overall goals for the proposed system and any expected benefits.
- Determine the customer's perceptions concerning the nature if a good solution to the problem.
- Evaluate the effectiveness of the customer meeting.

Feasibility

- Technical feasibility is not a good enough reason to build a product.
- The product must meet the customer's needs and not be available as an off-the-shelf purchase.

Estimation of Resources

- Human Resources (number of people required and skills needed to complete the development project)
- Reusable Software Resources (off-the-shelf components, full-experience components, partial-experience components, new components)
- Environment Resources (hardware and software required to be accessible by software team during the development process)

Software Project Estimation Options

- Delay estimation until late in the project.
- Base estimates on similar projects already completed.
- Use simple decomposition techniques to estimate project cost and effort.
- Use empirical models for software cost and effort estimation.
- Automated tools may assist with project decomposition and estimation.

Decomposition Techniques

- Software sizing (fuzzy logic, function point, standard component, change)
- Problem-based estimation (using LOC decomposition focuses on software functions, using FP decomposition focuses on information domain characteristics)

- Process-based estimation (decomposition based on tasks required to complete the software process framework)
- Use-case estimation (promising, but controversial due to lack of standardization of use cases)

Causes of Estimation Reconciliation Problems

- Project scope is not adequately understood or misinterpreted by planner
- Productivity data used for problem-based estimation techniques is inappropriate or obsolete for the application

Empirical Estimation Models

- Typically derived from regression analysis of historical software project data with estimated person-months as the dependent variable and KLOC, FP, or object points as independent variables.
- Constructive Cost Model (COCOMO) is an example of a static estimation model.
- COCOMO II is a hierarchy of estimation models that take the process phase into account making it more of dynamic estimation model.
- The Software Equation is an example of a dynamic estimation model.

Estimation for Object-Oriented Projects

1. Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
2. Using the requirements model (Chapter 6), develop use cases and determine a count. Recognize that the number of use cases may change as the project progresses.
3. From the requirements model, determine the number of key classes (called analysis classes in Chapter 6).
4. Categorize the type of interface for the application and develop a multiplier for support classes
5. Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.
6. Multiply the total number of classes (key + support) by the average number of work-units per class.
7. Cross check the class-based estimate by multiplying the average number of work-units per use case.

Estimation for Agile Development

1. Each user scenario is considered separately
2. The scenario is decomposed into a set of engineering tasks
3. Each task is estimated separately
    a. May use historical data, empirical model, or experience
    b. Scenario volume can be estimated (LOC, FP, use-case count, etc.)
4. Total scenario estimate computed
    a. Sum estimates for each task
    b. Translate volume estimate to effort using historical dsata

5. The effort estimates for all scenarios in the increment are summed to get an increment estimate

Estimation for WebApp Projects

- Modify the function point estimation procedure as follows
  - *Inputs* are each input screen or form, each maintenance screen, and if you use a tab notebook metaphor each tab.
  - *Outputs* are each static Web page, each dynamic web page script and each report
  - *Tables* are each logical table in the database plus, if you are using XML to store data in a file, each XML object (or collection of XML attributes).
  - *Interfaces* retain their definition as logical files into our out-of-the-system boundaries.
  - *Queries* are each externally published or use a message-oriented interface.

Make-Buy Decision

- It may be more cost effective to acquire a piece of software rather than develop it.
- Decision tree analysis provides a systematic way to sort through the make-buy decision.
- As a rule outsourcing software development requires more skillful management than in-house development of the same product.

Risk Management

Overview

Risks are potential problems that might affect the successful completion of a software project. Risks involve uncertainty and potential losses. Risk analysis and management is intended to help a software team understand and manage uncertainty during the development process. The important thing is to remember that things can go wrong and to make plans to minimize their impact when they do. The work product is called a Risk Mitigation, Monitoring, and Management Plan (RMMM) or a set of Risk Information Sheets (RIS).

Risk Strategies

- Reactive strategies - very common, also known as fire fighting, project team sets resources aside to deal with problems and does nothing until a risk becomes a problem
- Proactive strategies - risk management begins long before technical work starts, risks are identified and prioritized by importance, then team builds a plan to avoid risks if they can or minimize them if the risks turn into problems

Software Risks

- *Project risks* - threaten the project plan
- *Technical risks* - threaten product quality and the timeliness of the schedule

- *Business risks* - threaten the viability of the software to be built (market risks, strategic risks, sales risks, management risks, budget risks)
- *Known risks* - predictable from careful evaluation of current project plan and those extrapolated from past project experience
- *Unknown risks* - some problems simply occur without warning

Risk Identification

- *Product-specific risks* - the project plan and software statement of scope are examined to identify any special characteristics of the product that may threaten the project plan
- *Generic risks* - are potential threats to every software product (product size, business impact, customer characteristics, process definition, development environment, technology to be built, staff size and experience)

Risk Checklist Items

- Product size
- Business impact
- Stakeholder characteristics
- Process definition
- Development environment
- Technology to be built
- Staff size and experience

Risk Assessment Questions

1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project?
3. Are requirement fully understood by developers and customers?
4. Were customers fully involved in requirements definition?
5. Do end-users have realistic expectations?
6. Is project scope stable?
7. Does software team have the right skill set?
8. Are project requirements (scope) stable?
9. Does the project team have experience with technology to be implemented?
10. Is the number of people on project team adequate to do the job?
11. Do all stakeholders agree on the importance of the project the requirements for the systems being built?

Risk Impact

- Risk components - performance, cost, support, schedule
- Risk impact - negligible, marginal, critical, catastrophic
- The risk drivers affecting each risk component are classified according to their impact category and the potential consequences of each undetected software fault or unachieved project outcome are described

Risk Projection (Estimation)

1. Establish a scale that reflects the perceived likelihood of each risk
2. Delineate the consequences of the risk
3. Estimate the impact of the risk on the project and product
4. Note the overall accuracy of the risk projection to avoid misunderstandings

Risk Table Construction

- List all risks in the first column of the table
- Classify each risk and enter the category label in column two
- Determine a probability for each risk and enter it into column three
- Enter the severity of each risk (negligible, marginal, critical, catastrophic) in column four
- Sort the table by probability and impact value
- Determine the criteria for deciding where the sorted table will be divided into the first priority concerns and the second priority concerns
- First priority concerns must be managed (a fifth column can be added to contain a pointer into the RMMM)

Assessing Risk Impact

- Factors affecting risk consequences - nature (types of problems arising), scope (combines severity with extent of project affected), timing (when and how long impact is felt)
- If costs are associated with each risk table entry Halstead's risk exposure metric can be computed (RE = Probability * Cost) and added to the risk table.

Risk Assessment

1. Define referent levels for each project risk that can cause project termination (performance degradation, cost overrun, support difficulty, schedule slippage).
2. Attempt to develop a relationship between each risk triple (risk, probability, impact) and each of the reference levels.
3. Predict the set of referent points that define a region of termination, bounded by a curve or areas of uncertainty.
4. Try to predict how combinations of risks will affect a referent level.

Risk Refinement

- Process of restating the risks as a set of more detailed risks that will be easier to mitigate, monitor, and manage.
- CTC (condition-transition-consequence) format may be a good representation for the detailed risks (e.g. given that <condition> then there is a concern that (possibly) <consequence>).

Risk Mitigation, Monitoring, and Management

- *Risk mitigation* (proactive planing for risk avoidance)
- *Risk monitoring*
  - Assessing whether predicted risks actually occur
  - Ensuring risk aversion steps are being properly applied
  - Collecting information for future risk analysis, attempt to determine which risks caused which problems
  - Determining what risks cause which project problems
- *Risk management and contingency planing* (actions to be taken in the event that mitigation steps have failed and the risk has become a live problem)

Safety Risks and Hazards

- Risks are also associated with software failures that occur in the field after the development project has ended.
- Computers control many mission critical applications in modern times (weapons systems, flight control, industrial processes, etc.).
- Software safety and hazard analysis are quality assurance activities that are of particular concern for these types of applications and are discussed later in the text.

Risk Information Sheets

- Alternative to RMMM in which each risk is documented individually.
- Often risk information sheets (RIS) are maintained using a database system.
- RIS components - risk id, date, probability, impact, description, refinement, mitigation/monitoring, management/contingency/trigger, status, originator, assigned staff member.