

INF3705 - Advanced Systems Development

Summary of Textbook Chapters

Software Engineering: A Practitioners' Approach 7th ed. - Roger S Pressman

<u>Chapter</u>	<u>Page</u>
Chapter 2 - Process Models	2
Chapter 3 - Agile Development	9
Chapter 4 - Principles that Guide Practice	20
Chapter 5 - Understanding Requirements	27
Chapter 8 - Design Concepts	34
Chapter 10 - Component-Level Design	42
Chapter 17 - Software Testing Strategies	52
Chapter 18 - Testing Conventional Applications	61
Chapter 19 - Testing Object-Oriented Applications	68

These summaries are from the publishers website. The layout has (hopefully) been improved to enhance readability, but no changes have been made to content.

Ron Barnard

Ver 3.0 Oct 2012
Corrected spelling & grammatical errors, and minor improvements.
Added chapter summaries.

Chapter 2 – 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
- Modelling
- 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 or 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:

- **Pattern Name** - Give the pattern a meaningful name.
- **Forces** - environment in which the pattern is encountered and issues that make problems visible and affect their solution.
- **Type** -
 - *Stage patterns* (define a problem associated with a framework activity for the process).
 - *Task patterns* (define a problem associated with a software engineering action or work task, and relevant to successful software engineering practice).
 - *Phase patterns* (define the sequence of framework activities that occur within a process, even when overall flow is iterative. eg: Spiral, Prototyping).
- **Initial context** - describes conditions under which the pattern applies.
 - What organizational or team activities have already taken place ?
 - What is the entry state for the process ?
 - What software engineering or project information already exists ?
- **Problem** - Specific problem to be solved by the pattern.
- **Solution** - describes how to implement the pattern successfully.
- **Resulting context** - describes conditions that result when the 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 or project information has been developed ?
- **Related patterns** - list of all process patterns directly related to this one.
- **Known uses & examples** - specific instances in which the 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 (CBA IPI)** 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 have brought a certain amount of useful structure 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.
- **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.

Activities can be in one of seven states: awaiting change, inactive, under development, under revision, under review, baselined, done.

Specialized Process Models

- **Component-Based Development** - spiral model variation in which applications are built from prepackaged software components, either conventional modules, or object-oriented classes, or packages of classes.
- **Formal Methods Model** - rigorous mathematical notation used to specify, design, and verify computer-based systems. Time consuming & expensive.
- **Aspect-Oriented Software Development (AOSD)** - (aspect-oriented programming (AOP) - 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 of the best principles of agile software development.
- Phases
 - Inception phase (customer communication and planning)
 - Elaboration phase (communication and modelling)
 - Construction phase
 - Transition phase (customer delivery and feedback)
 - Production phase (deployment, 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 identifying 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. Metrics 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 (TSP)

- 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 CMM Level 5 behaviour normal and expected. (CMM – Capability Maturity Model).
 - Provide improvement guidance to high-maturity organizations.
 - Facilitate university teaching of industrial-grade team skills.
- Scripts for Project Activities (Framework)
 - 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.

Chapter 2 - Summary

A generic process model for software engineering encompasses a set of framework and umbrella activities, actions, and work tasks. Each of a variety of process models can be described by a different process flow – a description of how the framework activities, actions, and tasks are organized sequentially and chronologically. Process patterns can be used to solve common problems that are encountered as part of the software process.

Prescriptive process models have been applied for many years in an effort to bring order and structure to software development. Each of these models suggests a somewhat different process flow, but all perform the same set of generic framework activities: communication, planning, modelling, construction and deployment.

Sequential process models, such as the Waterfall (and V) models, are the oldest software engineering paradigms. They suggest a linear process flow that is often inconsistent with modern realities (eg: continuous change, evolving systems, tight time lines) in the software world. They do, however, have applicability in situations where requirements are well defined and stable.

Incremental process models are iterative in nature and produce working versions of software quite rapidly. Evolutionary process models recognize the iterative, incremental nature of most software engineering projects and are designed to accommodate change. Evolutionary models, such as prototyping and the spiral model, produce incremental work products (or working versions of the software) quickly. These models can be adopted to apply across all software engineering activities - from concept development to long-term system maintenance.

The concurrent process model allows a software team to represent iterative and concurrent elements of any process model. Specialized models include the component-based model that emphasizes component reuse and assembly; the formal methods model that encourages a mathematically approach to software development and verification; and the aspect-oriented model that accommodates cross-cutting concerns spanning the entire system architecture. The Unified Process is a “use case driven, architecture-centric, iterative and incremental” software process designed as a framework for UML methods and tools.

Personal and Team models for the software process have been proposed. Both emphasize measurement, planning, and self-direction as key ingredients for a successful software process.

Chapter 3 – Agile Development

Overview

- Agile software engineering represents a reasonable alternative 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 a way that eliminates 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 able to 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, give 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 maintain a constant pace 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 designs emerge from self-organizing teams.
12. At regular intervals teams reflect on how to become more effective, and adjust their behaviour 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) (Most widely used Agile approach)
- Adaptive Software Development (ASD)
- Scrum
- Dynamic Systems Development Method (DSDM)
- Crystal
- Feature Driven Development (FDD)
- Lean Software Development (LSD)
- Agile Modelling (AM)
- Agile Unified Process (AUP)

Extreme Programming (XP) (Most widely used agile approach)

- Relies on object-oriented approach.
- Values
 - Communication - close, informal (verbal) collaboration 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 (discipline) - design for today not tomorrow.
 - Respect - stakeholders and team members for the software product.
- Key activities
 - **Planning** - *Listening* - “user stories” created and ordered by customer values.
 - **Design** - simple designs preferred, CRC cards and design prototypes are only work products. Difficult problem – *spike solution*. Encourages use of refactoring.
 - **Coding** - focuses on unit tests to exercise stories, emphasizes use of *pair 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

Differs most from XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices. Incorporates six new practices -

- **Readiness assessment:**
 - Does an appropriate development environment exist to support IXP ?
 - Will the team be populated by the proper set of stakeholders ?
 - Does the organization have a distinct quality program and support continuous process improvement ?

- Will the organizational culture support new values of an 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

- **Requirements volatility** - can easily lead to scope creep that causes changes to earlier work design to accommodate current needs.
- **Conflicting customer needs** - projects with many customers may make it hard to assimilate all customer needs.
- **Requirements expressed informally** - with only user stories and acceptance tests, it is 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 (ASD)

- 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 release plan
 - 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 tested, 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 within framework activities of: requirements, analysis, design, evolution, delivery.
 - **Backlog** - prioritized list of requirements or features that provide business value to customer, items can be added at any time.
 - **Sprints** - work units required to achieve one of the backlog items, must fit into a predefined time-box, affected backlog items frozen.
 - **Scrum meetings** - Scrum Master - 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 (DSDM)

- 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% of the time 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 for business purpose is criterion for deliverable acceptance
 - Iterative and incremental development 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 & 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 “manoeuvrability” 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”.
- Set of example Agile processes that have proven effective for different types of projects. Each have core elements, common to all, and roles, process patterns, work products, and practice that are unique to each.

- Reflection workshops conducted before project begins, during increment development activity, and after increment is delivered.

Feature Driven Development (FDD)

- Practical process model for object-oriented software engineering.
- “Feature” is a client-valued function, that 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 by integration of software increments.
 - Technical communication using verbal, graphical, and text-based 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 (LSD)

- Eliminate waste
- Build quality in
- Create knowledge
- Defer commitment

- Deliver fast
- Respect people
- Optimize the whole

Agile Modelling (AM)

- Practice-based methodology for effective modelling and documentation of software systems in a light-weight manner.
- Modelling 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 (Modelling approach should be adapted to needs of agile team)
- Requirements gathering and analysis modelling
 - Work collaboratively to find out what customer wants to do.
 - Once requirements model is built collaborative analysis modelling continues with the customer.
- Architectural modelling
 - Derives preliminary architecture from analysis model.
 - Architectural model must be realistic for the environment and must be understandable by developers.

Agile Unified Process (AUP)

- Adopts classic UP phased activities (inception, elaboration, construction, transition) to enable team to visualize overall software project flow (linear sequence).
- Within each activity team iterates to achieve agility and delivers meaningful software increments to end-users as rapidly as possible.
- “Serial-in-the-large” and “iterative-in-the-small” approach.

- Each AUP iteration addresses:
 - Modelling - 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.
-

Chapter 3 - Summary

In a modern economy, market conditions change rapidly, customer and end-user needs evolve, and new competitive threats emerge without warning. Practitioners must approach software engineering in a manner that allows them to remain agile – to define manoeuvrable, lean process that can accommodate the needs of modern business.

An agile philosophy for software engineering stresses four key issues: the importance of self-organizing teams that have control over the work they perform, communication and collaboration between team members and between practitioners and their customers, a recognition that change represents an opportunity, and an emphasis on rapid delivery of software that satisfies the customer. Agile process models have been designed to address each of these issues.

Extreme programming (XP) is the most widely used agile process. Organized as four framework activities – planning, design, coding, and testing – XP suggests a number of innovative and powerful techniques that allow an agile team to create frequent software releases that deliver features and functionality that have been described and prioritized by stakeholders.

Other agile process models also stress human collaboration and team self-organization, but define their own framework activities and select different points of emphasis. For example, ASD uses an iterative process that incorporates adaptive cycle planning, relatively rigorous requirement gathering methods, and an iterative development cycle that incorporates customer focus groups

and formal technical reviews as real-time feedback mechanisms. Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight time lines, changing requirements, and business criticality. Each process pattern defines a set of development tasks and allows the Scrum team to construct a process that is adapted to the needs of the project. The Dynamic Systems Development Method (DSDM) advocates the use of time-box scheduling and suggests that only enough work is required for each software increment to facilitate movement to the next increment. Crystal is a family of agile process models that can be adopted to the specific characteristics of a project.

Feature Driven Development (FDD) is somewhat more “formal” than other agile methods, but still maintains agility by focusing the project team on the development of features – a client-valued function that can be implemented in two weeks or less. Lean Software Development (LSD) has adopted the principles of lean manufacturing to the world of software engineering. Agile modelling (AM) suggests that modelling is essential for all systems, but the the complexity, type, and size of the model must be tuned to the software to be built. The Agile Unified Process (AUP) adopts a “serial in the large” and “iterative in the small” philosophy for building software.

Chapter 4 – 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

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 carryout 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

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 co-ordination
6. Manage change
7. Assess risk
8. Create work products that provide value for others

Principles that Guide Practice

1. Divide and conquer (Separation of Concerns)
2. Understand the use of abstraction (Simplification)
3. Strive for consistency
4. Focus on 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 (leader) 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 the scope of the project
2. Involve stakeholders 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 (level of detail) as you define the plan
8. Define how you intend to ensure quality
9. Describe how you intend to accommodate change
10. Track the plan frequently and make adjustments as required

Classes of Models

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

Requirements Modelling Principles

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

Design Modelling 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 design of processing functions
4. Internal and external interfaces must be designed with care
5. User interface design should be tuned to the needs of the end-user and must focus on ease 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 (models) should be easy to understand
9. Design should be developed iteratively and designer should strive to simplify design with each iteration

Agile Modelling 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 the component to be built
 - Automatic generation of executable code using a "fourth-generation programming language".
- Testing levels
 - Unit testing – initial focus of testing is at the component level.
 - Integration testing
 - Validation testing
 - Acceptance testing

Coding Principles

- Preparation - before writing any code be sure you:

Chapter 4 - Principles that Guide Practice

- Understand the problem to be solved
 - Understand basic design principles
 - Pick a programming language that meets the needs of the software to be built and the environment in which it will operate
 - 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 the use of pair programming
 - Select data structures that meet the needs of the design
 - Understand the software architecture and create interfaces consistent with it
 - Keep conditional logic as simple as possible
 - Create nested loops in a way that allows them to be easily testable
 - Select meaningful variable names consistent with local standards
 - Write code that is self-documenting
 - Use a visual layout for your code that aids understanding.
- Validation - after you complete your first coding pass be sure you:
 - Conduct a code walkthrough when appropriate
 - Perform unit tests and correct errors
 - Refactor the code

Testing Objectives

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

Testing Principles

- All tests should be traceable to customer requirements
- Tests should be planned long before testing begins
- Pareto Principle applies to testing (80% of errors are found in 20% of code)
- Testing should begin “in the small” and progress toward testing “in the large”
- 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

Chapter 4 - Summary

Software engineering practice encompasses principles, concepts, methods, and tools that software engineers apply throughout the software process. Every software engineering project is different. Yet, a set of generic principles apply to the process as a whole and to the practice of each framework activity regardless of the project or the product.

A set of core principles help in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it navigates through the software process. At the level of of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately

deploy the software in the user community.

Communication principles focus on the need to reduce noise and improve bandwidth as the conversation between developer and customer progresses. Both parties must collaborate for the best communication to occur.

Planning principles provide guidelines for constructing the best map for the journey to a completed system or product. The plan may be designed solely for a single software increment, or it may be defined for the entire project. Regardless, it must address what will be done, who will do it, and when the work will be completed.

Modelling encompasses both analysis and design, describing representations of the software that progressively becomes more detailed. The intent of the models is to solidify understanding of the work to be done and to provide technical guidance to those who will implement the software. Modelling principles serve as a foundation for the methods and notation that are used to create representations of the software.

Construction incorporates a coding and testing cycle in which source code for a component is generated and tested. Coding principles define generic actions that should occur before code is written, while it is being created, and after it has been completed. Although there are many testing principles, only one is dominant: testing is a process of executing a program with the intent of finding an error.

Deployment occurs as each software increment is presented to the customer and encompasses delivery, support, and feedback. Key principles for delivery consider managing customer expectations and providing the customer with appropriate support information for the software. Support demands advance preparation. Feedback allows the customer to suggest changes that have business value and provide the developer with input for the next iterative software engineering cycle.

Chapter 5 – 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 (usage 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 modelling 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, behaviour, 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 you need ?
- The next set of questions enable a developer to better understand the problem and the customer's perceptions of the solution -
 - How would you characterize "good" output from 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 is 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 (Requirements gathering)

- Goal is to identify the problem, propose solution elements, negotiate approaches, and specify preliminary set of solution 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 behaviour 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 the 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 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 behaviour 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 (Requirements model)

- Intent is to provide descriptions of required information, functional, and behavioural domains for computer-based systems. Model changes dynamically as more is learnt about the system to be built. Analysis model is a snapshot of requirements at any given time.
- 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.
 - Behavioural elements - depict system and class behaviour 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 behaviour) that can be reused when modelling 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.

Requirements Review (Validation)

- Is each requirement consistent with overall project or system objective ?
- Are all requirements specified at the appropriate level of 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 behaviour 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 ?

Chapter 5 - Summary

Requirements engineering tasks are conducted to establish a solid foundation for design and construction. Requirements engineering occurs during the communication and modelling activities that have been defined for the generic software process. Seven distinct requirements engineering functions – inception, elicitation, elaboration, negotiation, specification, validation, and management – are conducted by members of the software team.

At project inception, stakeholders establish basic problem requirements, define overriding project constraints, and address major features and functions that must be present for the system to meet its objectives. This information is refined and expanded during elicitation – a requirements gathering activity that makes use of facilitated meetings, QFD, and the development of usage scenarios.

Elaboration further expands requirements in a model – a collection of scenario-based, class-based, behavioural, and flow-oriented elements. The model may reference analysis patterns, solutions for analysis problems that have been seen to reoccur across different applications.

As requirements are identified and the requirements model is being created, the software team and other project stakeholders negotiate the priority, availability, and relative cost of each requirement. The intent of this negotiation is to develop a realistic project plan. In addition, each requirement and the requirements model as a whole are validated against customer need to ensure that the right system is built.

Chapter 8 - Design Concepts

Overview

A software design creates a meaningful engineering representation (or model) of some software product that is to be built. Designers must strive to acquire a repertoire of alternative design information and learn to choose the elements that best match the analysis model. A design model can be traced to the customer's requirements and can be assessed for quality against predefined criteria. During the design process the software requirements model (data, function, behaviour) is transformed into design models that describe the details of the data structures, system architecture, interfaces, and components necessary to implement the system. Each design product is reviewed for quality (i.e. identify and correct errors, inconsistencies, or omissions, whether better alternatives exist, and whether the design model can be implemented within the project constraints) before moving to the next phase of software development.

Software Design

- Encompasses the set of principles, concepts, and practices that lead to the development of a high quality system or product.
- Design principles establish an overriding philosophy that guides the designer as the work is performed.
- Design concepts must be understood before the mechanics of design practice are applied.
- Goal of design engineering is to produce a model or representation that is bug free (firmness), suitable for its intended uses (commodity), and pleasurable to use (delight).
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve.

Software Engineering Design

- Data/Class design - created by transforming the analysis model class-based elements (class diagrams, analysis packages, CRC models, collaboration diagrams) into classes and data structures required to implement the software.
- Architectural design - defines the relationships among the major structural elements of

the software, it is derived from the class-based elements and flow-oriented elements (data flow diagrams, control flow diagrams, processing narratives) of the analysis model.

- Interface design - describes how the software elements, hardware elements, and end-users communicate with one another, it is derived from the analysis model scenario-based elements (use-case text, use-case diagrams, activity diagrams, swim lane diagrams), flow-oriented elements, and behavioural elements (state diagrams, sequence diagrams).
- Component-level design - created by transforming the structural elements defined by the software architecture into a procedural description of the software components using information obtained from the analysis model class-based elements, flow-oriented elements, and behavioural elements.

Software Quality Attributes

A good design must:

- implement all explicit requirements from the analysis model and accommodate all implicit requirements desired by stakeholders.
- be a readable and understandable guide for those who generate code, test components, or support the system.
- provide a complete picture (data, function, behaviour) of the software from an implementation perspective

Design Quality Guidelines

A design should:

- exhibit an architecture that:
 - has been created using recognizable architectural styles or patterns;
 - is composed of components that exhibit good design characteristics;
 - can be implemented in an evolutionary manner.
- be modular.
- contain distinct representations of data, architecture, interfaces, and components (modules).
- lead to data structures that are appropriate for the objects to be implemented and be drawn from recognizable data patterns.
- lead to components that exhibit independent functional characteristics.

- lead to interfaces that reduce the complexity of connections between modules and with the external environment.
- be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- be represented using a notation that effectively communicates its meaning.

FURPS Quality Attributes

- **Functionality** – feature set and program capabilities.
- **Usability** – human factors (aesthetics, consistency, documentation).
- **Reliability** – frequency and severity of failure.
- **Performance** – processing speed, response time, throughput, efficiency.
- **Supportability** – maintainability (extensibility, adaptability, serviceability), testability, compatibility, configurability.

Generic Design Task Set

1. Examine information domain model and design appropriate data structures for data objects and their attributes.
2. Select an architectural style appropriate to the software based on the analysis model.
3. Partition the analysis model into design subsystems and allocate these subsystems within the architecture:
 - Be certain each subsystem is functionally cohesive
 - Design subsystem interfaces
 - Allocate analysis classes or functions to subsystems
4. Create a set of design classes:
 - Translate analysis class into design class
 - Check each class against design criteria and consider inheritance issues
 - Define methods and messages for each design class
 - Evaluate and select design patterns for each design class or subsystem after considering alternatives
 - Review design classes and revise as needed

5. Design any interface required with external systems or devices.
6. Design user interface:
 - Review task analyses
 - Specify action sequences based on user scenarios
 - Define interface objects and control mechanisms
 - Review interface design and revise as needed.
7. Conduct component level design:
 - Specify algorithms at low level of detail
 - Refine interface of each component
 - Define component level data structures
 - Review components and correct all errors uncovered
8. Develop deployment model

Important Design Questions

- What criteria can be used to partition software into individual components ?
- How is function or data structure detail separated from conceptual representation of the software ?
- What uniform criteria define the technical quality of a software design ?

Design Concepts

- **Abstraction** – allows designers to focus on solving a problem without being concerned about irrelevant lower level details (*procedural abstraction* - named sequence of events and *data abstraction* – named collection of data objects).
- **Software Architecture** – overall structure of the software components and the ways in which that structure provides conceptual integrity for a system:
 - Structural models – architecture as organized collection of components
 - Framework models – attempt to identify repeatable architectural patterns
 - Dynamic models – indicate how program structure changes as a function of external events
 - Process models – focus on the design of the business or technical process that system must accommodate
 - Functional models – used to represent system functional hierarchy

- **Design Patterns** – description of a design structure that solves a particular design problem within a specific context and its impact when applied.
- **Separation of concerns** – any complex problem is solvable by subdividing it into pieces that can be solved independently.
- **Modularity** - the degree to which software can be understood by examining its components independently of one another.
- **Information Hiding** – information (data and procedures) contained within a module is inaccessible to modules that have no need for such information.
- **Functional Independence** – achieved by developing modules with single-minded purpose and an aversion to excessive interaction with other models:
 - Cohesion - qualitative indication of the degree to which a module focuses on just one thing
 - Coupling - qualitative indication of the degree to which a module is connected to other modules and to the outside world.
- **Refinement** – process of elaboration where the designer provides successively more detail for each design component.
- **Aspects** – a representation of cross-cutting “concerns” (requirements) that must be accommodated as refinement and modularization occur.
- **Refactoring** – process of changing a software system in such a way that internal structure is improved without altering the external behaviour or code design.

Design Classes

- Refine analysis classes by providing detail needed to implement the classes and implement a software infrastructure that supports the business solution.
- Five types of design classes can be developed to support the design architecture:
 - **User interface classes** – abstractions needed for human-computer interaction (HCI)
 - **Business domain classes** – refinements of earlier analysis classes
 - **Process classes** – implement lower level business abstractions
 - **Persistent classes** – data stores that persist beyond software execution
 - **System classes** – implement software management and control functions

Design Class Characteristics

- **Complete** (includes all necessary attributes and methods) **and sufficient** (contains only those methods needed to achieve class intent).
- **Primitiveness** – each class method focuses on providing one service.
- **High cohesion** – small, focused, single-minded classes.
- **Low coupling** – class collaboration kept to minimum.

Design Model

- **Process dimension** – indicates design model evolution as design tasks are executed during software process:
 - Data elements
 - Architecture elements
 - Interface elements
 - Component-level elements
 - Deployment-level elements
- **Abstraction dimension** – represents level of detail as each analysis model element is transformed into a design equivalent and refined:
 - High level (analysis model elements)
 - Low level (design model elements)
- Many UML diagrams used in the design model are refinements of diagrams created in the analysis model (more implementation specific detail is provided)
- Design patterns may be applied at any point in the design process

Data Design

- High level model depicting user's view of the data or information.
- Design of data structures and operators is essential to creation of high-quality applications.
- Translation of data model into database is critical to achieving system business objectives.
- Reorganizing databases into a data warehouse enables data mining or knowledge discovery that can impact on the success of the business itself.

Architectural Design

- Provides an overall view of the software product.
- Derived from
 - Information about the application domain relevant to software
 - Relationships and collaborations among specific analysis model elements
 - Availability of architectural patterns and styles.
- Usually depicted as a set of interconnected systems that are often derived from the analysis packages within the requirements model.

Interface Design

- Interface is a set of operations that describes the externally observable behaviour of a class and provides access to its public operations.
- Important elements
 - User interface (UI)
 - External interfaces to other systems
 - Internal interfaces between various design components.
- Modelled using UML communication diagrams (called collaboration diagrams in UML 1.x)

Component-Level Design

- Describes the internal detail of each software component.
- Defines
 - Data structures for all local data objects
 - Algorithmic detail for all component processing functions
 - Interface that allows access to all component operations.
- Modelled using UML component diagrams, UML activity diagrams, pseudocode (PDL), and sometimes flowcharts.

Deployment-Level Design

- Indicates how software functionality and subsystems will be allocated within the physical computing environment.
- Modelled using UML deployment diagrams.
- *Descriptor form* deployment diagrams show the computing environment but do not indicate configuration details.

- *Instance form* deployment diagrams identifying specific named hardware configurations are developed during the latter stages of design.

Chapter 8 - Summary

Software architecture provides a holistic view of the system to be built. It depicts the structure and organization of software components, their properties, and the connections between them. Software components include program modules and the various data representations that are manipulated by the program. Therefore, data design is an integral part of the derivation of the software architecture. Architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative system structures.

A number of different architectural styles and patterns are available to the software engineer and may be applied within a given architectural genre. Each style describes a system category that encompasses a set of components that perform a function required by a system, a set of connections that enable communication, co-ordination, and co-operation among components; constraints that define how components can be integrated to form the system; and semantic models that enable a designer to understand the overall properties of a system.

In a general sense, architectural design is accomplished using four distinct steps. First, the system must be represented in context. That is, the designer should define the external entities that the software interacts with and the nature of the interaction. Once context has been specified, the designer should identify a set of top-level abstractions, called archetypes, that represent pivotal elements of the systems' behaviour or function. After abstractions have been defined, the design begins to move closer to the implementation domain. Components are identified and represented within the context of an architecture that supports them. Finally, specific instantiations of the architecture are developed to "prove" the design in a real-world context.

As a simple example of architectural design, the mapping method presented in this chapter uses data flow characteristics to derive a commonly used architectural style. A data flow diagram is mapped into program structure using a transform mapping approach. Transform mapping is applied to an information flow that exhibits distinct boundaries between incoming and outgoing data. The DFD is mapped into a structure that allocates control to input, processing, and output along three separately factored module hierarchies. Once an architecture has been derived, it is elaborated and then analyzed using quality criteria.

Chapter 10 – Component-Level Design

Overview

The purpose of component-level design is to define data structures, algorithms, interface characteristics, and communication mechanisms for each software component identified in the architectural design. Component-level design occurs after the data and architectural designs are established. The component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built. The work product produced is a design for each software component, represented using graphical, tabular, or text-based notation. Design walkthroughs are conducted to determine correctness of the data transformation or control transformation allocated to each component during earlier design steps.

Component Definitions

- Component is a modular, deployable, replaceable part of a system that encapsulates implementation and exposes a set of interfaces.
- **Object-oriented view** is that a component contains a set of collaborating classes
 - Each elaborated class includes all attributes and operations relevant to its implementation
 - All interfaces that enable communication and collaboration with other design classes must also be defined
 - Analysis classes and infrastructure classes serve as the basis for object-oriented elaboration.
- **Traditional view** is that a component (or module) resides in the software and serves one of three roles:
 1. *Control components* co-ordinate invocation of all other problem domain components
 2. *Problem domain components* implement a function required by the customer
 3. *Infrastructure components* are responsible for functions needed to support the processing required in a domain application
 - The analysis model data flow diagram is mapped into a module hierarchy as the starting point for the component derivation.
- **Process-Related view** emphasizes building systems out of existing components chosen from a catalogue of reusable components as a means of populating the architecture.

Class-based Component Design

- Focuses on the elaboration of domain specific analysis classes and the definition of infrastructure classes.
- Detailed description of class attributes, operations, and interfaces is required prior to beginning construction activities.

Class-based Component Design Principles

- **Open-Closed Principle (OCP)** – class should be open for extension but closed for modification.
- **Liskov Substitution Principle (LSP)** – subclasses should be substitutable for their base classes.
- **Dependency Inversion Principle (DIP)** – depend on abstractions, do not depend on concretions.
- **Interface Segregation Principle (ISP)** – many client specific interfaces are better than one general purpose interface.
- **Release Reuse Equivalency Principle (REP)** – the granule of reuse is the granule of release.
- **Common Closure Principle (CCP)** – classes that change together belong together.
- **Common Reuse Principle (CRP)** – Classes that can't be reused together should not be grouped together.

Component-Level Design Guidelines

- Components
 - Establish naming conventions during architectural modelling
 - Architectural component names should have meaning to stakeholders
 - Infrastructure component names should reflect implementation specific meanings
 - Use of stereotypes may help identify the nature of components.
- Interfaces
 - Use lollipop representation rather than formal UML box and arrow notation
 - For consistency interfaces should flow from the left-hand side of the component box
 - Show only the interfaces relevant to the component under consideration.

- **Dependencies and Inheritance**
 - For improved readability model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes)
 - Component interdependencies should be represented by interfaces rather than component to component dependencies

Cohesion (lowest to highest)

A qualitative measure of the “single-mindedness” of a module – should perform a single task. High cohesion is desirable.

- **Functional cohesion** – module performs one and only one function.
- **Layer cohesion** – exhibited by packages, components, and classes, when a higher level layer accesses the services of a lower layer, but lower level layers do not access higher level layer services.
- **Communicational cohesion** – operations that access same data are grouped in same class.

Classes and components that exhibit cohesion are relatively easy to implement, test and maintain.

Coupling

A qualitative measure of the degree to which classes are connected to one another. Low coupling is desirable.

- **Content coupling** – occurs when one component surreptitiously modifies internal data in another component. (Violates information hiding).
- **Common coupling** – occurs when several components make use of a global variable.
- **Control coupling** – occurs when one component passes control flags as arguments to another.
- **Stamp coupling** – occurs when parts of larger data structures are passed between components.
- **Data coupling** – occurs when long strings of arguments are passed between components.
- **Routine call coupling** – occurs when one operation invokes another.

- **Type use coupling** – occurs when one component uses a data type defined in another
- **Inclusion or import coupling** – occurs when one component imports a package or uses the content of another.
- **External coupling** – occurs when a component communicates or collaborates with infrastructure components (e.g. database).

Software must communicate internally & externally. Therefore, coupling is a fact of life. However, coupling should be reduced whenever possible.

Conducting Component-Level Design

1. Identify all design classes that correspond to the problem domain.
2. Identify all design classes that correspond to the infrastructure domain.
3. Elaborate all design classes that are not acquired as reusable components.
 - a) Specify message details when classes or components collaborate.
 - b) Identify appropriate interfaces for each component.
 - c) Elaborate attributes and define data types and data structures required to implement them.
 - d) Describe processing flow within each operation in detail.
4. Identify persistent data sources (databases and files) and identify the classes required to manage them.
5. Develop and elaborate behavioural representations for each class or component.
6. Elaborate deployment diagrams to provide additional implementation detail.
7. Refactor every component-level design representation and always consider alternatives.

WebApp Component-Level Design

- Boundary between content and function often blurred.
- WebApp component is defined as either a:
 - well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or
 - cohesive package of content and functionality that provides the end-user with some required capability.

WebApp Component-Level Content Design

- Focuses on content objects and the manner in which they may be packaged for presentation to the end-user.
- As the WebApp size increases so does the need for formal representations and easy content reference and manipulation.
- For highly dynamic content a clear structural model incorporating content components should be established.

WebApp Component-Level Functional Design

- WebApps provide sophisticated processing functions:
 - perform dynamic processing to create content and navigational capability
 - provide business domain appropriate computation or data processing
 - provide database query and access
 - establish interfaces with external corporate systems
- WebApp functionality is delivered as a series of components developed in parallel with the information architecture.
- During architectural design WebApp content and functionality are combined to create a functional architecture
- The functional architecture is a representation of the functional domain of the WebApp and describes how the components interact with each other

Traditional Component-Level Design

- Each block of code has a single entry at the top.
- Each block of code has a single exit at the bottom.
- Only three control structures are required: sequence, condition (if-then-else), and repetition (looping).
- Reduces program complexity by enhancing readability, testability, and maintainability. Use of a limited number of constructs contributes to “chunking”.

Design Notation

- Graphical
 - UML activity diagrams
 - Flowcharts – arrows for flow of control, diamonds for decisions, rectangles for processes

- Tabular
 - Decision table – subsets of system conditions and actions are associated with each other to define the rules for processing inputs and events.
- Program Design Language (PDL)
 - Structured English or pseudocode used to describe processing details
 - Fixed syntax with keywords providing for representation of all structured constructs, data declarations, and module definitions
 - Free syntax of natural language for describing processing features
 - Data declaration facilities for simple and complex data structures
 - Subprogram definition and invocation facilities

Component-Based Development

- CBSE (Computer-based software engineering) is a process that emphasizes the design and construction of computer-based systems from a catalogue of reusable software components.
- CBSE is time and cost effective.
- Requires software engineers to reuse rather than reinvent.
- Management can be convinced to incur the additional expense required to create reusable components by amortizing the cost over multiple projects.
- Libraries can be created to make reusable components easy to locate and easy to incorporate them in new systems

Domain Engineering

- Intent is to identify, construct, catalogue, and disseminate a set of software components applicable to existing and future products in a particular application domain.
- Overall goal is to establish mechanisms that allow for component sharing and reuse during work on both new and existing systems.
- Includes three major activities: analysis, construction, dissemination.

Domain analysis steps

1. Define application domain to be investigated
2. Categorize items extracted from domain
3. Collect representative sample of applications in the domain
4. Analyze each application from sample and define analysis classes
5. Develop an analysis (requirements) model for classes

Component Qualification

- Ensures that a candidate component will: perform function required, fit into the architectural style specified for system, and exhibit required quality characteristics.
- Factors to consider:
 - Application programming interface (API)
 - Development and integration tools required by the component
 - Run-time requirements (resource usage or performance)
 - Service requirements (component dependencies)
 - Security features
 - Embedded design assumptions
 - Exception handling

Component Adaptation

- Seeks to avoid conflicts between the qualified component and the application architectures to facilitate easy integration.
- Common adaptation techniques:
 - *White-box wrapping* – integration conflicts removed by making code-level modifications to the code
 - *Grey-box wrapping* – used when component library provides a component extension language or API that allows conflicts to be removed or masked
 - *Black-box wrapping* – requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts.

Component Composition Architectural Ingredients

- Assembles qualified, adapted, and engineered components to populate established application architecture.
- Ingredients for creating an effective infrastructure to enable components to coordinate with one another to perform common tasks:
 - **Data exchange model** – similar to drag and drop type mechanisms should be defined for all reusable components, allow human-to-software and component-to-component transfer.
 - **Automation** – tools, macros, scripts should be implemented to facilitate interaction between reusable components.

- **Structured storage** – heterogeneous data should be organized and contained in a single data structure rather than several separate files.
- **Underlying object model** – ensures that components developed in different languages are interoperable across computing platforms.

Representative Component Standards

- Object Management Group (OMG)/CORBA (common object request broker architecture)
- Microsoft COM (component object model) and .net framework
- Sun JavaBeans Components (Bean Development Kit)

Analysis and Design for Reuse

- The requirements model needs to be analyzed to determine the elements that point to existing reusable components.
- Specification matching can help find components to extract from the reuse library.
- If no reusable components can be found, developers should design a new component keeping reuse in mind.
- When designing for reuse developers should consider several key issues:
 - **Standard data** – standard global data structures identified in application domain and components developed to use these data structures.
 - **Standard interface protocols** – three interface levels should be established (intramodular, external technical, and human/machine).
 - **Program templates** – used in architectural design of new programs.

Classifying and Retrieving Components

- Describing reusable components
 - *concept* - what the component does
 - *content* - how the concept is realized
 - *context* - specifies conceptual, operational, and implementation features of the software component within its domain of application.

- Reuse environment elements:
 - **component database** capable of storing software components and classification information to allow their retrieval.
 - **library management system** to allow access to database.
 - **software component retrieval system** that enables client software to retrieve components and services from library server.
 - **CBSE tools** that support integration of reused components into a new design or implementation.
-

Chapter 10 - Summary

The component-level design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

Three different views of component-level design may be taken, depending on the nature of the software to be developed. The object-oriented view focuses on the elaboration of design classes that come from both the problem and infrastructure domain. The traditional view refines three different types of components or modules: control modules, problem domain modules, and infrastructure modules. In both cases, basic design principles and concepts that lead to high-quality software are applied. When considered from a process viewpoint, component-level design draws on reusable software components and design patterns that are pivotal elements of component-based software engineering.

A number of important principles and concepts guide the designer as classes are elaborated. Ideas encompassed in the Open-Closed Principle and the Dependency Inversion Principle and concepts such as coupling and cohesion guide the software engineer in building testable, implementable, and maintainable software components. To conduct component-level design in this context, classes are elaborated specifying messaging details, identifying appropriate interfaces, elaborating attributes and defining data structures to implement them, describing processing flow within each operation, and representing behaviour at a class or component level. In every case, design iteration (refactoring) is an essential activity.

Traditional component-level design requires the representation of data structures, interfaces, and algorithms for a program module in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of a number of design notations that represent component-level detail in either graphical, tabular, or text-based formats.

Component-level design for WebApps considers both content and functionality as it is delivered by a Web-based system. Content design at the component level focuses on content objectives and the manner in which they may be packaged for presentation to a WebApp end user. Functional design for WebApps focuses on processing functions that manipulate content, perform computations, query and access a database, and establish interfaces with other systems. All component-level design principles and guidelines apply.

Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.

Component-based software engineering identifies, constructs, catalogs, and disseminates, a set of software components in a particular application domain. These components are then qualified, adapted, and integrated for use in a new system. Reusable components should be designed within an environment that establishes standard data structures, interface protocols, and program architectures for each application domain.

Chapter 17 - 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 into 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 ITG 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, behavioural, 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 (technical) 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.

Test Strategies for Conventional Software

Unit Testing

Focuses on the internal processing logic and data structures within the boundaries of a component.

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

Integration Testing

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

- *Sandwich testing* uses top-down tests for upper levels of program structure coupled with bottom-up tests for subordinate levels.
- Testers should strive to identify *critical modules* having one or more of the following characteristics:
 - Addresses several software requirements;
 - has a high level of control (resides relatively high in the program structure);
 - is complex, or error prone;
 - has definite performance requirements.
- Critical modules should be tested as early as possible. Regression tests should focus on critical module function.
- 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, another stub is replaced with a real component.
 5. Regression testing may be used to ensure that new errors are 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:

Regression testing is the re-execution of a subset of tests that have already been conducted, to ensure that changes have not propagated unintended side effects.

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 on 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-up 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** – test to verify specified performance bounds.

Test Strategies for Object-Oriented Software

- 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

- An encapsulated class is the focus of unit testing. Operations (methods) within the class are the smallest testable units.
- equivalent to unit testing of conventional software.
- do not test operations in isolation from one another.
- driven by class operations and state behaviour, not algorithmic detail and data flow across module interface.

Object-Oriented Integration Testing

Object-oriented software does not have an obvious hierarchical control structure, so traditional top-down and bottom-up strategies have little meaning.

- 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 then 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

Adopts the same basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems.

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. Each functional component is 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

Begins at the culmination of integration testing. At the validation or system level, the distinction between conventional software, object-oriented software, and WebApps disappears.

- Focuses on visible user actions and user recognizable outputs from the system.
- Validation tests are based on the use-case scenarios, the behaviour 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, catalogued, 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 fully exercise a computer-based system.
- The focus of these system test cases is to identify interfacing errors.
- **Recovery testing** – checks the system's ability to recover from failures.
- **Security testing** – verifies that system protection mechanisms 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 environments in which it is to operate

Debugging Process

- The symptom and the cause may be geographically remote (symptom may appear in one part of a program, while the cause may be elsewhere).
- 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 isolate potential causes of errors.

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 ?

Chapter 17 - Summary

Software testing accounts for the largest percentage of technical effort in the software process. Regardless of the type of software you build, a strategy for systematic test planning, execution, and control begins by considering small elements of the software and moves outward toward the program as a whole.

The objective of software testing is to uncover errors. For conventional software, this objective is achieved through a series of test steps. Unit and integration tests concentrate on functional verification of a component and incorporation of components into the software architecture. Validation testing demonstrates traceability to software requirements, and system testing validates software once it has been incorporated into a larger system. Each test step is accomplished through a series of systematic test techniques that assist in the design of test cases. With each testing step, the level of abstraction with which software is considered is broadened.

The strategy for testing object-oriented software begins with tests that exercise the operations within a class, and then moves to thread-based testing for integration. Threads are sets of classes that respond to an input or event. Use-based tests focus on classes that do not collaborate heavily with other classes.

WebApps are tested in much the same way as OO systems. However, tests are designed to exercise content, functionality, the interface, navigation, and aspects of WebApp performance and security.

Unlike testing (a systematic, planned activity), debugging can be viewed as an art. Beginning with a symptomatic indication of a problem, the debugging activity must track down the cause of an error. Of the many resources available during debugging, the most valuable is the counsel of other members of the software engineering staff.

Chapter 18 – Testing Conventional Applications

Overview

The importance of software testing to software quality can not be over-emphasized. 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 conventional 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 undiscovered error.
- A successful test is one that discovers an undiscovered error.

Software Testability Checklist (How easily a software program can be tested)

- **Operability** – the better it works the more efficiently it can be tested.
- **Observability** – 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. (Every test should have a different purpose).
- A good test should be “best of breed”. (Highest likelihood of finding an error).
- A good test should not be too simple or too complex.

Test Case Design Strategies

- Black-box or behavioural 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 and internal components.

White-Box Testing

Using White-Box testing methods, one can derive test cases that -

- guarantee that all independent paths within a module will be executed at least once;
- exercise all logical decisions on their true and false branches;
- execute all loops at their boundaries and within their operational bounds;
- 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 is 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 definitions and uses of variables 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 (Behavioural Testing)

Attempts to find errors in the following categories:

1. incorrect or missing functions;
2. interface errors;
3. errors in data structures, or external database access;
4. behaviour or performance errors;
5. initialization and termination errors.

Tests are designed to answer the following questions:

- How is functional validity tested ?
- How are system behaviour 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 modelling** – nodes represent steps in some transaction and links represent logical connections between steps that need to be validated.
- **Finite state modelling** – nodes represent user observable states of the software and links represent transitions between states.
- **Data flow modelling** – nodes are data objects and links are transformations from one data object to another.
- **Timing modelling** – 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 otherwise 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 centre.
- BVA guidelines:

- If input condition specifies a range bounded by values a and b, test cases should include a and b, and values just above and just below a and b.
- If an input condition specifies a number of values, test cases should exercise the minimum and maximum numbers, as well as values just above and just below the minimum and maximum values.
- Apply guidelines 1 and 2 to output conditions, test cases should be designed to produce the minimum and maximum output reports.
- 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).
- Assessing test results using an orthogonal array:
 1. Detect and isolate all single mode faults
 2. Detect all double mode faults
 3. Multimode faults.

Model-Based Testing

- Black-box testing technique using information contained in the requirements model as a basis for test case generation.
- Steps for MBT:
 1. Analyze an existing behaviour model for the software or create one.
 2. Traverse behavioural model and specify inputs that force software to make transition from state to state.
 3. Review behavioural 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 behavioural model of user interface, use of automated testing tools is strongly recommended.
- Client/Server 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 methods 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. Behavioural testing – Using system models, it is possible to simulate the behaviour of real-time systems, as a consequence of external events. Use technique similar to equivalence partitioning to test behaviour.
 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 cases based on state model and control specification.

Patterns for Software Testing

- 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).

Chapter 18 - Summary

The primary objective for test-case design is to derive a set of tests that have the highest likelihood for uncovering errors in software. To accomplish this objective, two different categories of test-case design techniques are used: white-box testing and black-box testing.

White-box tests focus on the program control structure. Test cases are derived to ensure that all statements in the program have been executed at least once during testing and that all logical conditions have been exercised. Basis path testing, a white-box technique, makes use of program graphs (or graph matrices) to derive the set of linearly independent tests that will ensure statement coverage. Condition and data flow testing further exercise program logic, and loop testing complements other white-box techniques by providing a procedure for exercising loops of varying degrees of complexity.

Hetzel [Het84] describes white-box testing as “testing in the small”. His implication is that the white-box tests that have been considered in this chapter are typically applied to small program components (eg: modules or small groups of modules). Black-box testing, on the other hand, broadens your focus and might be called “testing in the large”.

Black-box tests are designed to validate functional requirements without regard to the internal workings of a program. Black-box testing techniques focus on the information domain of the software, deriving test cases by partitioning the input and output domain of a program in a manner that provides thorough test coverage. Equivalence partitioning divides the input domain into classes of data that are likely to exercise a specific software function. Boundary value analysis probes the program's ability to handle data at the limits of acceptability. Orthogonal array testing provides an efficient, systematic method for testing systems with small numbers of input parameters. Model-based testing uses elements of the requirements model to test the behaviour of an application.

Specialized testing methods encompass a broad array of software capabilities and application areas. Testing for graphical user interfaces, client-server architectures, documentation and help facilities, and real-time systems each require specialized guidelines and techniques.

Experienced software developers often say, “Testing never ends, it just gets transferred from you [the software engineer] to your customer. Every time your customer uses the program, a test is being conducted.” By applying test-case design, you can achieve more complete testing and thereby uncover and correct the highest number of errors before the “customer's tests” begin.

Chapter 19 – Testing Object-Oriented Applications

Overview

It is important to test object-oriented applications 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 design (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 be checked for proper use of notation and modeling conventions by means of technical reviews.
- 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 ensure 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 design model is created you should conduct reviews of the system design and object design.
- The system design is reviewed by examining the object-behaviour model and mapping the required system behaviour against subsystems designed to accomplish the behaviour.

Software Testing Strategies for Object-Oriented Architectures

- **Unit Testing** – called *class testing* in OO circles, components being tested are classes and their behaviours, 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 few 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.

OO Testing Methods

Overall approach to OO test-case design:

1. Each test case should be uniquely identified and be explicitly associated with the 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 class involved in the test.
 - b) list of messages and operations to be exercised as a consequence of the test.
 - c) list of exceptions that may occur as the class is tested.
 - d) list of external conditions that must exist to properly conduct the test.
 - e) supplementary information required to understand or implement the test

Applicability of Conventional Test Case Design Methods

- 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 at 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

- Capture the user tasks described in the use-cases, and build 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, behaviours, and communications mechanisms established as part of the system design.

Class Level Testing Methods

- Random testing – requires large numbers of 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 a 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 operations to generate random test sequences that send messages to other server classes.
 2. For each message generated, determine the collaborator class and the corresponding operation in the server object.
 3. For each operation in the server object (invoked by a client object message) determine the message that it transmits.
 4. For each message, determine the next level of operations that are invoked and incorporate them into the test sequence.

 - Tests derived from behaviour 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 behaviours for the class have been adequately exercised
-

Chapter 19 - Summary

The overall objective of object-oriented testing – to find the maximum number of errors with a minimum amount of effort is identical to the objective of conventional software testing. But the strategy and tactics for OO testing differ significantly. The view of testing broadens to include the review of both the requirements and design model. In addition, the focus of testing moves away from the procedural component (the module) and toward the class.

Because the OO requirements and design models and the resulting source code are semantically coupled, testing (in the form of technical reviews) begins during the modelling activity. For this reason, the review of CRC, object-relationship, and object-behaviour models can be viewed as first-stage testing.

Once code is available, unit testing is applied for each class. The design of tests for a class uses a variety of methods: fault-based testing, random testing, and partition testing. Each of these methods exercise the operations encapsulated by the class. Test sequences are designed to ensure that relevant operations are exercised. The state of the class, represented by the values of its attributes, is examined to determine if errors exist.

Integration testing can be accomplished using a thread-based or use-based strategy. Thread-based testing integrates the set of classes that collaborate to respond to one input or event. Use-based testing constructs the system in layers, beginning with those classes that do not make use of server classes. Integration test-case design methods can also make use of random and partition tests. In addition, scenario-based testing and tests derived from behavioural models can be used to test a class and its collaborators. A test sequence tracks the flow of operations across class collaborations.

OO system validation testing is black-box oriented and can be accomplished by applying the same black-box methods discussed for conventional software. However, scenario-based testing dominates the validation of OO systems, making the use case a primary driver for validation testing.
