INSTITUTE OF AERONAUTICAL ENGINEERING

(Autonomous)

Dundigal, Hyderabad - 500 043

Lecture Notes:

Object Oriented Software Engineering (ACSC19)

Drafted by :

D. Koumudi prasanna(IARE10939

Assistant Professor

Department Of CSIT

Institute of Aeronautical Engineering

August 03, 2022

Contents

List of Figures

Abbreviations

TVC	Thrust Vector Control
LOX	Liquid OXygen
LVDT	Liquid Propellant Rocket Engine
RC	Reinforced Concrete

Symbols

D^{el}	Elasticity tensor
σ	Stress tensor
ε	Strain tensor
V_{eq}	Equivalent velocity
$\dot{m}$	Mass flow rate
I_{sp}	Specific Impulse
c	Effective exhaust velocity
I_t	Total impulse
υ	Exhaust velocity
m_p	Propellant mass
m_e	Empty mass
A_{ex}	Exit Area
p_{ex}	Exhaust pressure
P_{SL-a}	Ambient pressure at sea level
F_{SL-a}	Sea level thrust of the rocket
$\dot{W}_{sp}$	Specific propellant consumption rate
C_w	Weight flow coefficient
C_f	Thrust coefficient

Chapter 1

Introduction to software engineering

Course Outcomes

After successful completion of this module, students should be able to:

CO 1	Outline process models, approaches and techniques for managing software development process.	Understand

What is Software

The product that software professionals build and then support over the long term.

Software encompasses:

Who wishes to buy them

Examples – PC software such as editing, graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists

Customized products

Software that is commissioned by a specific customer to meet their own needs. Examples – embedded control systems, air traffic control software, traffic monitoring systems

Stand-alone systems that are marketed and sold to any customer I. Instructions (computer programs) that when executed provide desired features, function, and performance
II. Data structures that enable the programs to adequately store and manipulate information and
III. Documentation that describes the operation and use of the programs

Software products

Generic products
Stand-alone systems that are marketed and sold to any customer
Who wishes to buy them
Examples – PC software such as editing, graphics programs, project management tools; CAD software; software for specific markets such as appointments systems for dentists.
Customized products
Software that is commissioned by a specific customer to meet their own needs.
Examples – embedded control systems, air traffic control software, traffic monitoring systems.

Why Software is Important

The economies of ALL developed nations are dependent on software.
More and more systems are software controlled (transportation, medical, telecommunications, military, industrial, entertainment, etc…)
Software engineering is concerned with theories, methods and tools for professional software development.
Expenditure on software represents a significant fraction of Gross national product (GNP) in all developed countries

Features of Software

Its characteristics that make it different from other things human being build.

Features of such logical system:

Software is developed or engineered; it is not manufactured in the classical sense which has quality problem.

Software doesn’t ”wear out.‖ but it deteriorates (due to change). Hardware has bathtub curve of failure rate (high failure rate in the beginning, then drop to steady state, then cumulative effects of dust, vibration, abuse occurs).

Although the industry is moving toward component-based construction (e.g. standard screws and off-the-shelf integrated circuits), most software continues to be custom-built. Modern reusable components encapsulate data and processing into software parts to be reused by different programs. E.g. graphical user interface, window, pull-down menus in library etc.

Software Applications

I. System software: such as compilers, editors, file management utilities

II. Application software: stand-alone programs for specific needs.

III. Engineering/scientific software: Characterized by number crunching‖ algorithms. such as automotive stress analysis, molecular biology, orbital dynamics etc

IV. Embedded software resides within a product or system. (key pad control of a microwave oven, digital function of dashboard display in a car)

V. Product-line software focus on a limited marketplace to address mass consumer market. (word processing, graphics, database management)

VI. WebApps (Web applications) network centric software. As web 2.0 emerges, more sophisticated computing environments is supported integrated with remote database and business applications.

VII. AI software uses non-numerical algorithm to solve complex problem. Robotics, expert system, pattern recognition game playing

Software Engineering Definition

The seminal definition

[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines

The IEEE definition:

Software Engineering

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in (1).

Software Engineering A Layered Technology

•Any engineering approach must rest on organizational commitment to quality which fosters a continuous process improvement culture.

•Process layer as the foundation defines a framework with activities for effective delivery of software engineering technology. Establish the context where products (model, data, report, and forms) are produced, milestones are established, quality is ensured and change is managed.

•Method provides technical how-to‘s for building software. It encompasses many tasks including communication, requirement analysis, design modeling, program construction, testing and support.

•Tools provide automated or semi-automated support for the process and methods

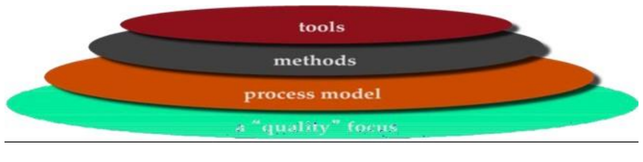


Figure 1.1: Layered Technology

Software Process

•A process is a collection of activities, actions and tasks that are performed when some work product is to be created. It is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work to pick and choose the appropriate set of work actions and tasks.
•Purpose of process is to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it

Five Activities of a Generic Process Framework

•Communication: communicate with customer to understand objectives and gather requirements.
•Planning: creates a map‖ defines the work by describing the tasks, risks and resources, work products and work schedule.
•Modeling: Create a sketch‖, what it looks like architecturally, how the constituent parts fit together and other characteristics.
•Construction: code generation and the testing.
•Deployment: Delivered to the customer who evaluates the products and provides feedback based on the evaluation.
These five framework activities can be used to all software development regardless of the application domain, size of the project, complexity of the efforts etc, though the details will be different in each case.For many software projects, these framework activities are applied iteratively as a project progresses. Each iteration produces a software increment that provides a subset of overall software features and functionality.

Umbrella Activities

Complement the five process framework activities and help team manage and control progress, quality, change, and risk.

•Software project tracking and control: assess progress against the plan and take actions to maintain the schedule.

•Risk management: assesses risks that may affect the outcome and quality.

•Software quality assurance: defines and conduct activities to ensure quality.

•Technical reviews: assesses work products to uncover and remove errors before going to the next activity.

•Measurement: define and collects process, project, and product measures to ensure stake-holder‘s needs are met.

•Software configuration management: manage the effects of change throughout the soft-ware process.

•Reusability management: defines criteria for work product reuse and establishes mech-anism to achieve reusable components.

•Work product preparation and production: create work products such as models, docu-ments, logs, forms and lists.

Adapting a Process Model

The process should be agile and adaptable to problems. Process adopted for one project might be significantly different than a process adopted from another project. (to the prob-lem, the project, the team, organizational culture). Among the differences are:

•the overall flow of activities, actions, and tasks and the interdependencies among them

•the degree to which actions and tasks are defined within each framework activity

•the degree to which work products are identified and required

•the manner which quality assurance activities are applied

•the manner in which project tracking and control activities are applied

•the overall degree of detail and rigor with which the process is described

•the degree to which the customer and other stakeholders are involved with the project
•the level of autonomy given to the software team
•the degree to which team organization and roles are prescribed

Software Process: A Generic Process Model

•A framework for the activities, actions, and tasks that are required to build high-quality software.
•SP defines the approach that is taken as software is engineered.
•Is not equal to software engineering, which also encompasses technologies that populate the process – technical methods and automated tools

A Generic process model

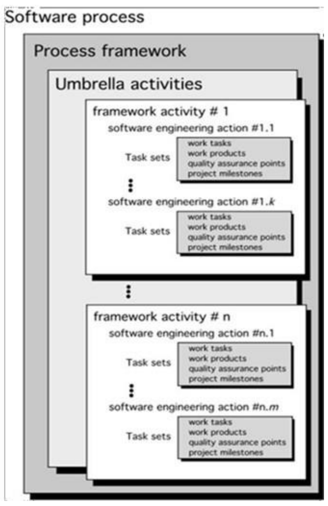


Figure 1.2: A Generic Process Model

A generic process framework for software engineering defines five framework activities-communication, planning, modeling, construction, and deployment. In addition, a set of umbrella activities- project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others are applied throughout the process. Next question is: how the framework activities and the actions and tasks that occur within each activity are organized with respect to sequence and time? See the process flow for answer.

Process Flow

1. Linear process flow executes each of the five activities in sequence
2. An iterative process flow repeats one or more of the activities before proceeding to the next
3. An evolutionary process flow executes the activities in a circular manner. Each circuit leads to a more complete version of the software.
4. A parallel process flow executes one or more activities in parallel with other activities (modeling for one aspect of the software in parallel with construction of another aspect of the software.

Identifying a Task Set

Before you can proceed with the process model, a key question: what actions are appropriate for a framework activity given the nature of the problem, the

characteristics of the people and the stakeholders

A task set defines the actual work to be done to accomplish the objectives of a software engineering action.

A list of the task to be accomplished

A list of the work products to be produced

A list of the quality assurance filters to be applied

Example of a Task Set for Elicitation

The task sets for Requirements gathering action for a simple project may include:

1. Make a list of stakeholders for the project
2. Invite all stakeholders to an informal meeting.
3. Ask each stakeholder to make a list of features and functions required.
4. Discuss requirements and build a final list
5. Prioritize requirements
6. Note areas of uncertainty

Process Patterns

•A process pattern •describes a process-related problem that is encountered during software engineering work
•identifies the environment in which the problem has been encountered, and
•suggests one or more proven solutions to the problem
•Stated in more general terms, a process pattern provides you with a template [Amb98]—a consistent method for describing problem solutions within the context of the software process

1. Problems and solutions associated with a complete process model (e.g. prototyping)
2. Problems and solutions associated with a framework activity (e.g. planning) or
3. An action with a framework activity (e.g. project estimating)

Process Pattern Types
•Stage patterns—defines a problem associated with a framework activity for the process. It includes multiple task patterns as well. For example, Establishing Communication would incorporate the task pattern Requirements Gathering and others
•Task patterns—defines 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 with the process, even when the overall flow of activities is iterative in nature. Example includes

Sprial Model or Prototyping

An Example of Process Pattern

•Describes an approach that may be applicable when stakeholders have a general idea of what must be done but are unsure of specific software requirements

•Pattern name. Requirement Unclear

•Intent. This pattern describes an approach for building a model that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements

•Type. Phase pattern

•Initial context. Conditions must be met (1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding software problem to be solved has been identified by stakeholders ; (4) an initial understanding of project scope, basic business requirements and project constraints has been developed

•Problem. Requirements are hazy or nonexistent. Stakeholders are unsure of what they want

•Solution. A description of the prototyping process would be presented here

•Resulting context. A software prototype that identifies basic requirements. (modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, 1. This prototype may evolve through a series of increments to become the production software or 2. the prototype may be discarded

•Related patterns. Customer Communication, Iterative Design, Iterative Development, Customer Assessment, Requirement Extraction

Process Assessment and Improvement

•The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer‘s needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics

•A number of different approaches to software process assessment and improvement have been proposed over the past few decades

•Standard CMMI Assessment Method for Process Improvement (SCAMPI)—provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment [SEI00]

•CMM-Based Appraisal for Internal Process Improvement (CBA IPI)— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01]

•SPICE (ISO/IEC15504)—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process [ISO08]

•ISO 9001:2000 for Software—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies [Ant06]

software development process models

Prescriptive Process Models

1. Classic Process Models Waterfall Model (Linear Sequential Model)
2. Incremental Process Models - Incremental Model
3. Evolutionary Software Process Models
   •Prototyping
   •Spiral Model
   •Concurrent Development Model

Classic Process Models -Waterfall Model (Linear Sequential Model)

•The waterfall model, sometimes called the classic life cycle
It is the oldest paradigm for Software Engineering. When requirements are well defined and reasonably stable, it leads to a linear fashion
•The waterfall model, sometimes called the classic life cycle, suggests a systematic
•sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software
A variation of waterfall model depicts the relationship of quality assurance actions to the

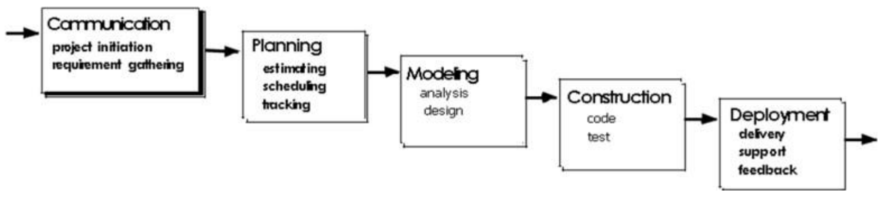


Figure 1.3: software process models

actions associated with communication, modeling and early code construction activates Team first moves down the left side of the V to refine the problem requirements. Once code is generated, the team moves up the right side of the V, performing a series of tests that validate each of the models created as the team moved down the left side
The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work
The V Model

Incremental Process Models- Incremental Model

•When initial requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. A compelling need to expand a limited

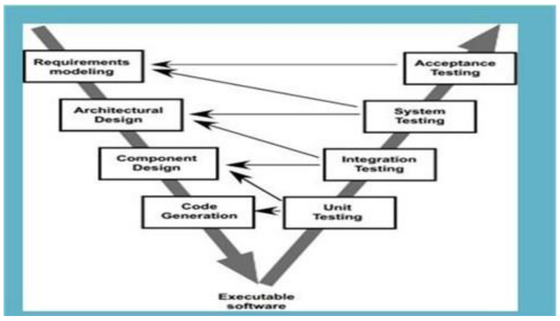


Figure 1.4: v model

The problems that are sometimes encountered when the waterfall model is applied are

•Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds

•It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects

•The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous

set of new functions to a later system release

•It combines elements of linear and parallel process flows. Each linear sequence produces deliverable increments of the software

•The first increment is often a core product with many supplementary features. Users use it and evaluate it with more modifications to better meet the needs

•The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user

•Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project

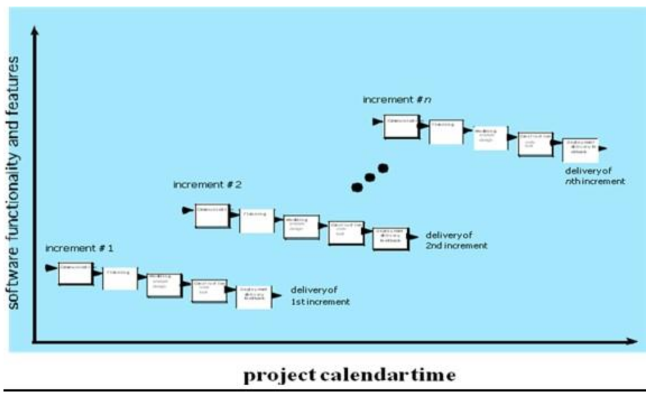

software functionality and features
increment # n
delivery of nth increment
increment # 2
delivery of 2nd increment
increment # 1
delivery of 1st increment
project calendar time

Figure 1.5: project cakender time

Evolutionary Software Process Models

•Prototyping

•Spiral Model

•Concurrent Development Model

•Software system evolves over time as requirements often change as development proceeds. Thus, a straight line to a complete end product is not possible. However, a limited version must be delivered to meet competitive pressure

•Usually a set of core product or system requirements is well understood, but the details and extension have yet to be defined

•You need a process model that has been explicitly designed to accommodate a product that evolved over time

•It is iterative that enables you to develop increasingly more complete version of the software

•Two types are introduced, namely Prototyping and Spiral models

Evolutionary Models: Prototyping

•When to use: Customer defines a set of general objectives but does not identify detailed requirements for functions and features. or Developer may be unsure of the efficiency of an algorithm, the form that human computer interaction should take.

•What step: Begins with communication by meeting with stakeholders to define the objective, identify whatever requirements are known, outline areas where further definition is mandatory. A quick plan for prototyping and modeling (quick design) occur. Quick design focuses on a representation of those aspects the software that will be visible to end users. (interface and output). Design leads to the construction of a prototype which will be deployed and evaluated. Stakeholder‘s comments will be used to refine requirements

•Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. However, engineers may make compromises in order to get a prototype working quickly. The less-than-ideal choice may be adopted forever after you get used to it

Prototyping can be problematic for the following reasons

•Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you hasn‘t considered overall software quality or long-term maintainability

•As a software engineer, you often make implementation compromises in order to get a prototype working quickly

•An inappropriate operating system or programming language may be used simply because it is available and known

•An inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system

Evolutionary Models: The Spiral

•It couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model and is a risk-driven process model generator that is used to guide multi- stakeholder concurrent engineering of software intensive systems

•Two main distinguishing features: one is cyclic approach for incrementally growing a system‘s degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions

•A series of evolutionary releases are delivered. During the early iterations, the release might be a model or prototype. During later iterations, increasingly more complete version of the engineered system are produced

•The first circuit in the clockwise direction might result in the product specification;

subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software

•Each pass results in adjustments to the project plan. Cost and schedule are adjusted based on feedback. Also, the number of iterations will be adjusted by project manager

•Good to develop large-scale system as software evolves as the process progresses and risk should be understood and properly reacted to. Prototyping is used to reduce risk

•However, it may be difficult to convince customers that it is controllable as it demands considerable risk assessment expertise

Concurrent Model

•Allow a software team to represent iterative and concurrent elements of any of the process models. For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following actions: prototyping, analysis and design

•The Figure shows modeling may be in any one of the states at any given time. For example, communication activity has completed its first iteration and in the awaiting changes state. The modeling activity was in inactive state, now makes a transition into the under development state. If customer indicates changes in requirements, the modeling activity moves from the under development state into the awaiting changes state

•Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions and tasks to a sequence of events, it defines a process network. Each activity, action or task on the network exists simultaneously with other activities, actions or tasks. Events generated at one point trigger transitions among the state

Specialized Process Models

Specialized process models take on many of the characteristics of one or more of the traditional models. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen

•Component-Based Development

•The Formal Methods Model

•Aspect-Oriented Software Development

Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable

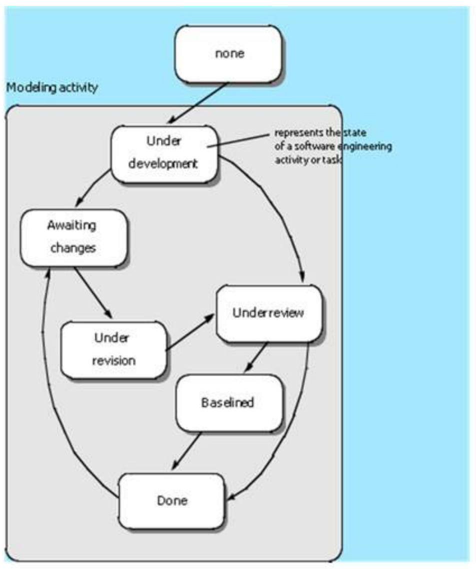


Figure 1.6: concurrent model

the component to be integrated into the software that is to be built

These components can be as either conventional software modules or object-oriented packages or packages of classes

Steps involved in CBS are

•Available component-based products are researched and evaluated for the application domain in question

•Component Integration issues are considered

•A software architecture is designed to accommodate the components

•Components are integrated into the architecture

•Comprehensive testing is conducted to ensure proper functionality

•Component-based development model leads to software reuse and reusability helps software engineers with a number of measurable benefits

•Component-based development leads to a 70 percent reduction in development cycle time, 84 percent reduction in project cost and productivity index of 26.2 compared to an industry norm of 16.9

Formal Methods Model

•Formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software

•They enable software engineers to specify, develop and verify a computer-based system

by applying a rigorous mathematical notation
•Development of formal models is quite time consuming and expensive
•Extensive training is needed in applying formal methods
•Difficult to use the model as a communication mechanism for technically unsophisticated customers
Aspect-oriented Software Development
•The aspect-oriented approach is based on the principle of identifying common program code within certain aspects and placing the common procedures outside the main business logic
•The process of aspect orientation and software development may include modeling, design,programming, reverse-engineering and re-engineering
•The domain of AOSD includes applications, components and databases
•Interaction with and integration into other paradigms is carried out with the help of frameworks, generators, program languages and architecture-description languages (ADL)
The Unified process, personal and team process models
•The Unified Process is an iterative and incremental development process. Unified Process divides the project into four phases

1. Inception
2. Elaboration
3. Construction
4. Transition

•The Inception, Elaboration, Construction and Transition phases are divided into a series of time boxed iterations. (The Inception phase may also be divided into iterations for a large project.)
•Each iteration results in an increment, which is a release of the system that contains added or improved functionality compared with the previous release
•Although most iterations will include work in most of the process disciplines (e.g. Requirements, Design, Implementation, Testing) the relative effort and emphasis will change

over the course of the project

•Risk Focused

The Unified Process requires the project team to focus on addressing the most critical risks early in the project life cycle. The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first. Risk Focused

Inception Phase

Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception Phase is long then it is usually an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process

The following are typical goals for the Inception phase

•Establish a justification or business case for the project

•Establish the project scope and boundary conditions

•Outline the use cases and key requirements that will drive the design tradeoffs

•Outline one or more candidate architectures

•Identify risks

•Prepare a preliminary project schedule and cost estimate

The Lifecycle Objective Milestone marks the end of the Inception phase

Elaboration Phase

During the Elaboration phase the project team is expected to capture a majority of the system requirements. The primary goals of Elaboration are to address known risk factors and to establish and validate the system architecture

Common processes undertaken in this phase include the creation of use case diagrams, conceptual diagrams (class diagrams with only basic notation) and package diagrams (architectural diagrams)

The architecture is validated primarily through the implementation of an Executable Architectural Baseline. This is a partial implementation of the system which includes the core, most architecturally significant, components. It is built in a series of small, time-boxed iterations

By the end of the Elaboration phase the system architecture must have stabilized and the executable architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost

The final Elaboration phase deliverable is a plan (including cost and schedule estimates) for the Construction phase. At this point the plan should be accurate and credible, since it should be based on the Elaboration phase experience and since significant risk factors should have been addressed during the Elaboration phase. The Lifecycle Architecture Milestone marks the end of the Elaboration phase

Construction Phase

Construction is the largest phase in the project. In this phase the remainder of the system is built on the foundation laid in Elaboration. System features are implemented in a series of short, timeboxed iterations. Each iteration results in an executable release of the software. It is customary to write full text use cases during the construction phase and each one becomes the start of a new iteration

Common UML (Unified Modeling Language) diagrams used during this phase include Activity, Sequence, Collaboration, State (Transition) and Interaction Overview diagrams

The Initial Operational Capability Milestone marks the end of the Construction phase

Transition Phase

The final project phase is Transition. In this phase the system is deployed to the target users. Feedback received from an initial release (or initial releases) may result in further refinements to be incorporated over the course of several Transition phase iterations. The Transition phase also includes system conversions and user training

The Product Release Milestone marks the end of the Transition phase

Advantages of UP Software Development

This is a complete methodology in itself with an emphasis on accurate documentation

It is proactively able to resolve the project risks associated with the client's evolving requirements requiring careful change request management Less time is required for integration as the process of integration goes on throughout the software development life cycle

The development time required is less due to reuse of components

Disadvantages of RUP Software Development

The team members need to be expert in their field to develop a software under this methodology

On cutting edge projects which utilise new technology, the reuse of components will not be possible. Hence the time saving one could have made will be impossible to fulfill

Integration throughout the process of software development, in theory sounds a good thing. But on particularly big projects with multiple development streams it will only add to the confusion and cause more issues during the stages of testing

Personal and Team process models

The best software process is one that is close to the people who will be doing the work. The PSP model defines five framework activities

Personal Software Process (PSP)

Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked

High-level design review. Formal verification methods (Chapter 21) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results

Postmortem. Using the measures and metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness

Team Software Process (TSP)

The goal of TSP is to build a self directed project team that organizes itself to produce high-quality software. TSP objectives are

•Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers

agile development

Agile software engineering combines a philosophy and a set of development guidelines. The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity. The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers A manifesto is normally associated with an emerging political movement— one that attacks the old guard and suggests revolutionary change (hopefully for the better). In some ways, that's exactly what agile development is all about. Although the underlying ideas that guide agile development have been with us for many years, it has been less than two decades since these ideas have crystallized into a "movement." In essence, agile 1 methods were developed in an effort to overcome perceived and actual weaknesses in conventional software engineering. Agile development can provide important benefi ts, but it is not applicable to all projects, all products, all people, and all situations. It is also not antithetical to solid software engineering practice and can be applied as an overriding philosophy for all software work. In the modern economy, it is often diffi cult or impossible to predict how a computer-based system (e.g., a mobile application) will evolve as time passes. Market conditions change rapidly, end-user needs evolve, and new competitive threats emerge without warning. In many situations, you won't be able to defi ne requirements fully before the project begins. You must be agile enough to respond to a fl uid business environment.

What is Agility

Just what is agility in the context of software engineering work? Ivar Jacobson [Jac02a] provides a useful discussion: Agility has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to appro-priately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project

Agility and the cost of change

The conventional wisdom in software development (supported by decades of experience) is that the cost of change increases nonlinearly as a project progresses (Figure 5.1 , solid black curve). It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project). A usage scenario might have to be modifi ed, a list Cost of of functions may be extended, or a writt

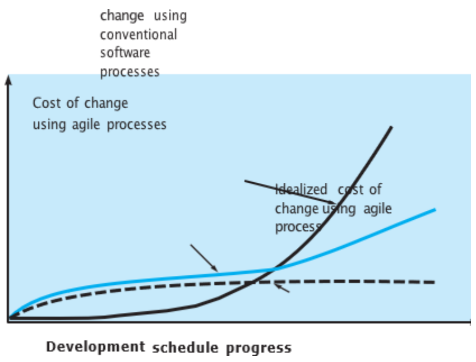


Figure 1.7: Development schedule progress

specification can be edited. The costs of doing this work are minimal, and the time required will not adversely affect the outcome of the project. But what if we fast-forward a number of months? The team is in the middle of validation testing (something that occurs relatively late in the project), and an important stake- holder is requesting a major functional change. The change requires a modifi- cation to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests, and so on. Costs escalate quickly, and the time and cost required to ensure that the change is made without unintended side effects is nontrivial

What is Agile Process

Any agile software process is characterized in a manner that addresses a num- ber of key assumptions [Fow02] about the majority of software projects

1. It is difficult to predict in advance which software requirements will per- sist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds
   For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design

2. Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like. Agility principles

Agility principles

The Agile Alliance (see [Agi03], [Fow01]) defines 12 agility principles for those who want to achieve agility

1. Our highest priority is to satisfy the customer through early and continu- ous delivery of valuable software

2. Welcome changing requirements, even late in development. Agile pro- cesses harness change for the customer's competitive advantage

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. Business people and developers must work together daily throughout the project

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done

6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation

7. Working software is the primary measure of progress

The process

A software process (Chapters 3–5) provides the framework from which a com- prehensive plan for software development can be established. A small num- ber of framework activities are applicable to all software projects, regardless of their size or complexity. A number of different task sets—tasks, milestones, work products, and quality assurance points—enable the framework activities to be adapted to the characteristics of the software project and the requirements of the project team. Finally, umbrella activities—such as software quality assur- ance, software configuration management, and measurement—overlay the pro- cess model. Umbrella activities are independent of any one framework activity and occur throughout the process

The project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones [Jon04] found that “about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays and overruns, or were ter- minated without completion.” Although the success rate for present-day soft- ware projects may have improved somewhat, our project failure rate remains much higher than it should be.1 To avoid project failure, a software project manager and the software engi- neers who build the product must avoid a set of common warning signs, under- stand the critical success factors that lead to good project management, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 31.5 and in the chapters that follow

Project Management

Effective software project management focuses on the four Ps: people, product, process, and project. The order is not arbitrary. The manager who forgets that software engineering work is an intensely human endeavor will never have suc- cess in project management. A manager who fails to encourage comprehensive stakeholder communication early in the evolution of a product risks building an elegant solution for the wrong problem. The manager who pays little attention to the process runs the risk of inserting competent technical methods and tools into a vacuum. The manager who embarks without a solid project plan jeopardizes the success of the project

Software project management: Estimation

Estimation is attempt to determine how much money, effort, resources and time it will take to build a specific software based system or project. Estimation involves answering the following questions

1. How much effort is required to complete each activity

2. How much calendar time is needed to complete each activity

3. What is the total cost of each activity.
   Project cost estimation and project scheduling are normally carried out together
   The costs of development are primarily the costs of the effort involved, so the effort computation is used in both the cost and the schedule estimate. Do some cost estimation before detailed schedules are drawn up
   These initial estimates may be used to establish a budget for the project or to set a price for the software for a customer
   There are three parameters involved in computing the total cost of a software development project
   •Hardware and software costs including maintenance
   •Travel and training costs
   •Effort costs (the costs of paying software engineers)
   The following costs are all part of the total effort cost
   Factors affecting software pricing

4. Costs of providing, heating and lighting office space

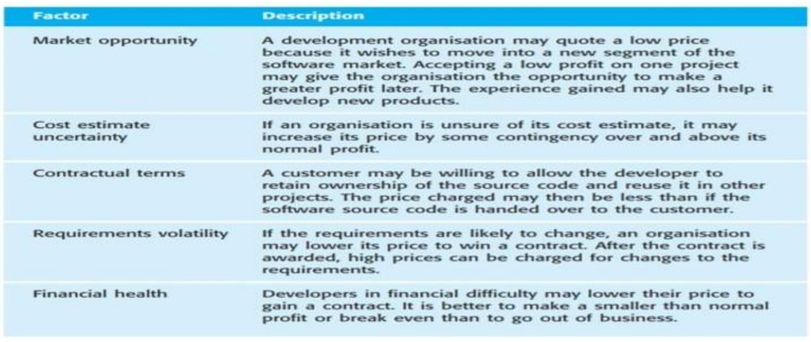
5. Costs of support staff such as accountants, administrators, system managers, cleaners and technicians

6. Costs of networking and communications

7. Costs of central facilities such as a library or recreational facilities

8. Costs of Social Security and employee benefits such as pensions and health insurance

Factors affecting software pricing Introduction about LOC and FP based estimation



Factor	Description
Market opportunity	A development organisation may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organisation the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Cost estimate uncertainty	If an organisation is unsure of its cost estimate, it may increase its price by some contingency over and above its normal profit.
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged may then be less than if the software source code is handed over to the customer.
Requirements volatility	If the requirements are likely to change, an organisation may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.
Financial health	Developers in financial difficulty may lower their price to gain a contract. It is better to make a smaller than normal profit or break even than to go out of business.

Figure 1.8: factor affecting software pricing

Function Points

•STEP 1: measure size in terms of the amount of functionality in a system. Function points are computed by first calculating an unadjusted function point count (UFC). Counts are made for the following categories

– External inputs – those items provided by the user that describe distinct application-oriented data (such as file names and menu selections)

– External outputs – those items provided to the user that generate distinct application-oriented data (such as reports and messages, rather than the individual components of these)

– External inquiries – interactive inputs requiring a response

– External files – machine-readable interfaces to other systems

– Internal files – logical master files in the system

•STEP 2: Multiply each number by a weight factor, according to complexity (simple, average or complex) of the parameter, associated with that number. The value is given by a table • STEP 3: Calculate the total UFP (Unadjusted Function Points)

• STEP 4: Calculate the total TCF (Technical Complexity Factor) by giving a value between 0 and 5 according to the importance of the following points

• STEP 5: Sum the resulting numbers too obtain DI (degree of influence)

Parameter	simple	average	complex
users inputs	3	4	6
users outputs	4	5	7
users requests	3	4	6
files	7	10	15
external interfaces	5	7	10

Figure 1.9: weight factor

- STEP 6: TCF (Technical Complexity Factor) by given by the formula

– TCF=0.65+0.01*DI

- STEP 6: Function Points are by given by the formula

– FP=UFP*TCF

Relation between LOC and FP

– LOC = Language Factor * FP

– where

- LOC (Lines of Code)
- FP (Function Points)

The Basic COCOMO model computes effort as a function of program size. The Basic COCOMO equation is

– E = aKLOC$\hat{b}$

- Effort for three modes of Basic COCOMO
- The intermediate COCOMO model computes effort as a function of program size and

Mode	**a**	**b**
Organic	**2.4**	**1.05**
Semi-detached	**3.0**	**1.12**
Embedded	**3.6**	**1.20**

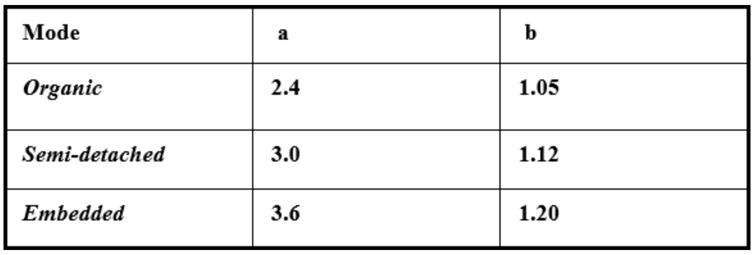
Figure 1.10: cocomo model

a set of cost drivers. The Intermediate COCOMO equation is

– E = aKLOC$\hat{b}$*EAF

- Effort for three modes of intermediate COCOMO

Total EAF = Product of the selected factors



Mode	**a**	**b**
Organic	**2.4**	**1.05**
Semi-detached	**3.0**	**1.12**
Embedded	**3.6**	**1.20**

Figure 1.11: cocomo

Adjusted value of Effort: Adjusted Person Months: APM = (Total EAF) * PM

A development process typically consists of the following stages

- Requirements Analysis
- Design (High Level + Detailed)
- Implementation& Coding • Testing (Unit + Integration)

Error Estimation

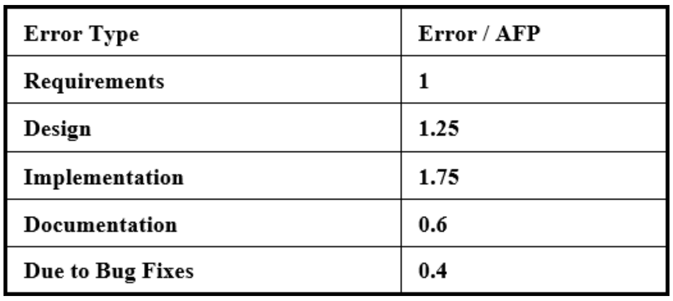
- Calculate the estimated number of errors in your design, i.e.total errors found in requirements, specifications, code, user manuals, and bad fixes

– Adjust the Function Point calculated in step1

AFP = FP ** 1.25

– Use the following table for calculating error estimates

LOC based estimation



Error Type	**Error / AFP**
Requirements	**1**
Design	**1.25**
Implementation	**1.75**
Documentation	**0.6**
Due to Bug Fixes	**0.4**

Figure 1.12: error estimation

- Source lines of code (SLOC), also known as lines of code (LOC), is a software metric used to measure the size of a computer program by counting the number of lines in the text of the program's source code

- SLOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or maintainability once the software is produced
- Lines used for commenting the code and header file are ignored

Two major types of LOC

1. Physical LOC

- Physical LOC is the count of lines in the text of the program's source code including comment lines
- Blank lines are also included unless the lines of code in a section consists of more than 25% blank lines

1. Logical LOC

- Logical LOC attempts to measure the number of executable statements, but their specific definitions are tied to specific computer languages
- Ex: Logical LOC measure for C-like programming languages is the number of statement-terminating semicolons (;)

The problems of lines of code (LOC)

- Different languages lead to different lengths of code
- It is not clear how to count lines of code
- A report, screen, or GUI generator can generate thousands of lines of code in minutes
- Depending on the application, the complexity of code is different

COCOMO model

The software cost estimation provides

- The vital link between the general concepts and techniques of economic analysis and the particular world of software engineering
- Software cost estimation techniques also provides an essential part of the foundation for good software management

Cost of a project

• The cost in a project is due to

– due the requirements for software, hardware and human resources

– the cost of software development is due to the human resources needed

– most cost estimates are measured in person-months (PM)

– the cost of the project depends on the nature and characteristics of the project, at any point, the accuracy of the estimate will depend on the amount of reliable information we have about the final product

The Constructive Cost Model (COCOMO) is the most widely used software estimation

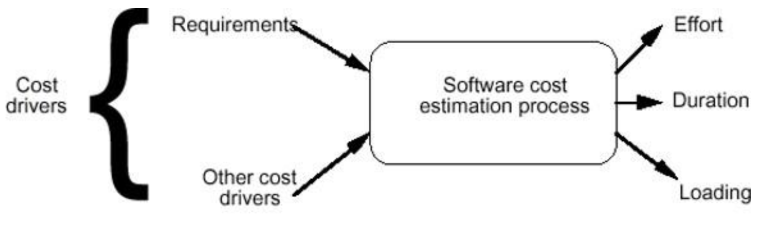


Figure 1. Classical view of software estimation process.

Figure 1.13: classical view of software estimation process

model in the world. The COCOMO model predicts the effort and duration of a project based on inputs relating to the size of the resulting systems and a number of ”cost drives” that affect productivity

• Effort Equation

– PM = C * (KDSI)n (person-months)

• where PM = number of person-month (=152 working hours)

• C = a constant

• KDSI = thousands of ”delivered source instructions” (DSI) and

• n = a constant

• Productivity equation

– (DSI) / (PM)

• where PM = number of person-month (=152 working hours)

• DSI = ”delivered source instructions

• Schedule equation – TDEV = C * (PM)n (months) • where TDEV = number of months estimated for software development

• Average Staffing Equation

– (PM) / (TDEV) (FSP)

• where FSP means Full-time-equivalent Software Personnel

COCOMO is defined in terms of three different models

– Basic model

– Intermediate model, and

– Detailed model

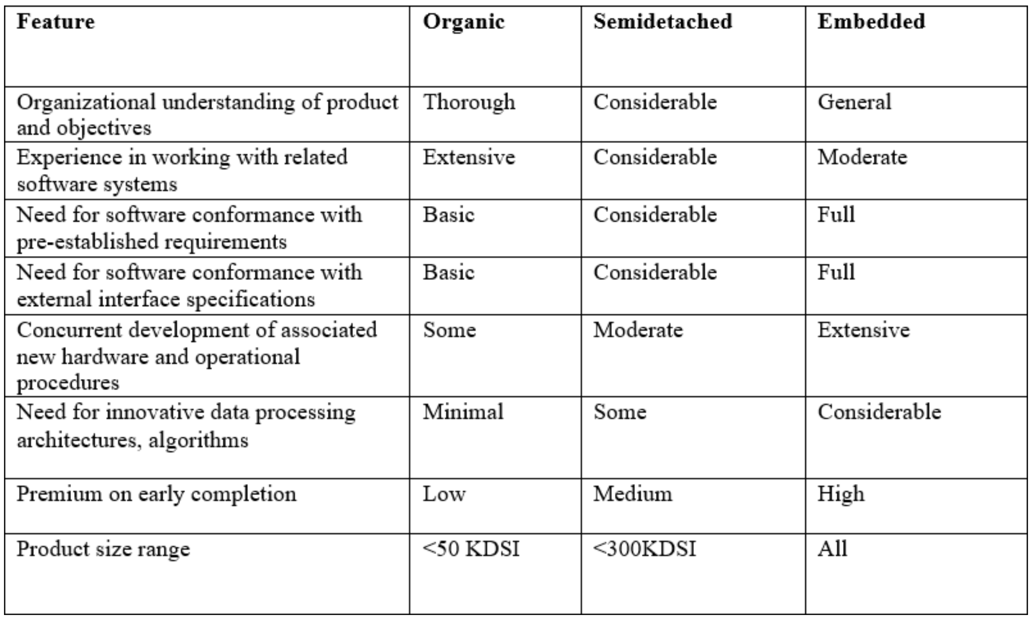
• The more complex models account for more factors that influence software projects, and make more accurate estimates

• The most important factors contributing to a project's duration and cost is the Development Mode

• Organic Mode: The project is developed in a familiar, stable environment, and the product is similar to previously developed products. The product is relatively small, and requires little innovation

• Semidetached Mode: The project's characteristics are intermediate between Organic and Embedded

• Embedded Mode: The project is characterized by tight, inflexible constraints and interface requirements. An embedded mode project will require a great deal of innovation



Feature	**Organic**	**Semidetached**	**Embedded**
Organizational understanding of product and objectives	Thorough	Considerable	General
Experience in working with related software systems	Extensive	Considerable	Moderate
Need for software conformance with pre-established requirements	Basic	Considerable	Full
Need for software conformance with external interface specifications	Basic	Considerable	Full
Concurrent development of associated new hardware and operational procedures	Some	Moderate	Extensive
Need for innovative data processing architectures, algorithms	Minimal	Some	Considerable
Premium on early completion	Low	Medium	High
Product size range	<50 KDSI	<300KDSI	All

Figure 1.14: project

project metrics

These are metrics that relate to Project Quality. They are used to quantify defects, cost, schedule, productivity and estimation of various project resources and deliverables

1. Schedule Variance: Any difference between the scheduled completion of an activity and the actual completion is known as Schedule Variance. Schedule variance = ((Actual calendar days – Planned calendar days) + Start variance)/ Planned calendar days x 100

2. Effort Variance: Difference between the planned outlined effort and the effort required to actually undertake the task is called Effort variance. Effort variance = (Actual Effort – Planned Effort)/ Planned Effort x 100

3. Size Variance: Difference between the estimated size of the project and the actual size of the project (normally in KLOC or FP). Size variance = (Actual size – Estimated size)/ Estimated size x 100

4. Requirement Stability Index: Provides visibility to the magnitude and impact of requirements changes. RSI = 1- ((Number of changed + Number of deleted + Number of added) / Total number of initial requirements) x100
   process and project metrics

Within the context of the software process and the projects that are conducted using the process, a software team is concerned primarily with productivity and quality metrics—measures of software development "output" as a function of effort and time applied and measures of the "fi tness for use" of the work products that are produced. For planning and estimating purposes, our interest is historical. What was software development productivity on past projects? What was the quality of the software that was produced?

How can past productivity and quality data be extrapolated to the present? How can it help us plan and estimate more accurately? In their guidebook on software measurement, Park, Goethert, and Florac [Par96b] note the reasons that we measure: (1) to characterize in an effort to gain an understanding "of processes, products, resources, and environments, and to establish baselines for comparisons with future assessments"; (2) to evaluate "to determine status with respect to plans"; (3) to predict by "gaining understandings of relationships among processes and products and building models of these relationships"; and (4) to improve by "identify[ing] roadblocks, root causes, ineffi ciencies, and other opportunities for improving product quality and process performance." Measurement is a management tool. If conducted properly, it provides a project manager with insight. And as a result, it assists the project manager and the software team in making decisions that will lead to a successful project

METRICS IN THE PROCESS AND PROJECT DOMAINS

Process metrics are collected across all projects and over long periods of time. Their intent is to provide a set of process indicators that lead to long-term software process improvement (Chapter 37). Project metrics enable a software project manager to (1) assess the status of an ongoing project, (2) track potential risks, (3) uncover problem areas before they go "critical," (4) adjust work fl ow or tasks, and (5) evaluate the project team's ability to control quality of software work products. Measures that are collected by a project team and converted into metrics for use during a project can also be transmitted to those with responsibility for software process improvement. For this reason, many of the same metrics are used in both the process and project domains

Process Metrics and Software Process Improvement software process improvement, it is important to note that process is only one of a number of "controllable factors in improving software quality and organizational performance" [Pau94]. Referring to Figure 32.1 , process sits at the center of a triangle connecting three factors that have a profound infl uence on software quality and organizational performance. The skill and motivation of people have been shown [Boe81] to be the most infl uential factors in quality and performance. The complexity of the product can have a substantial impact on quality and team performance. The technology (i.e., the software engineering methods and tools) that populates the process also has an impact. In addition, the process triangle exists within

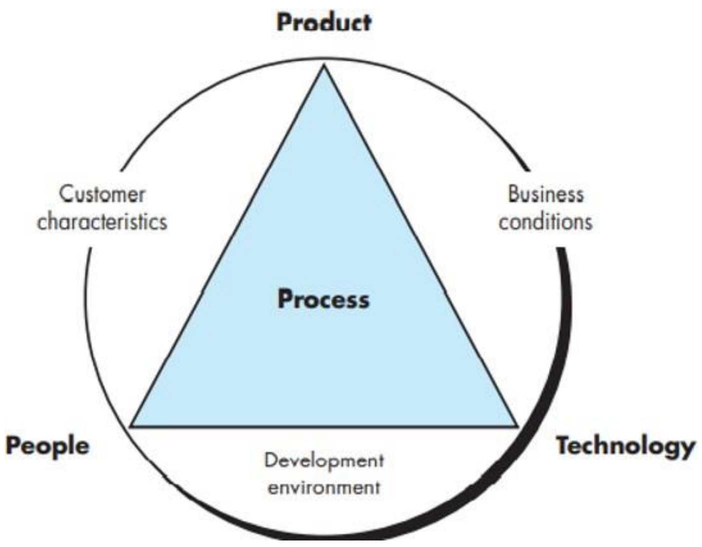


Figure 1.15: process metrics

a circle of environmental conditions that include the development environment (e.g., integrated software tools), business conditions (e.g., deadlines, business rules), and customer characteristics (e.g., ease of communication and collaboration). You can only measure the effi cacy of a software process indirectly. That is, you derive a set of metrics based on the outcomes that can be derived from the process. Outcomes include measures of errors uncovered before release of the software, defects delivered to and reported by end users, work products delivered (productivity), human effort expended, calendar

Project Metrics

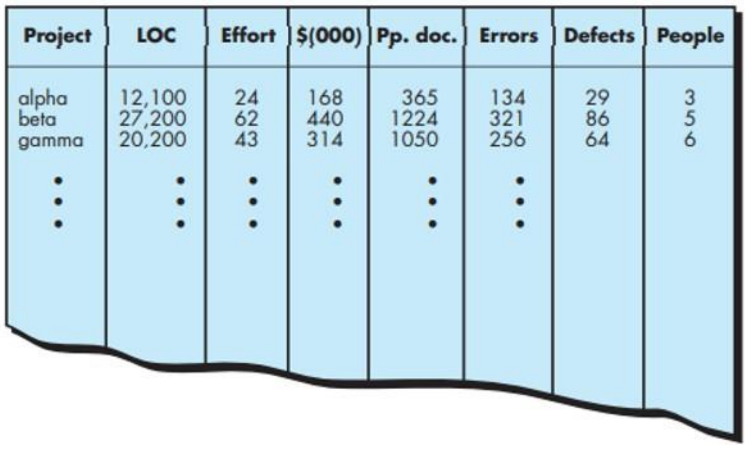
Unlike software process metrics that are used for strategic purposes, software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work fl ow and technical activities. The fi rst application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress. As technical work commences, other project metrics begin to have signifi - cance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from requirements into design, technical metrics (Chapter 30) are collected to assess design quality and to provide indicators that will infl uence the approach taken to code generation and testing. The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality. As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost. Unlike software process metrics that are used for strategic purposes, software project measures are tactical. That is, project metrics and the indicators derived from them are used by a project manager and a software team to adapt project work fl ow and technical activities. The fi rst application of project metrics on most software projects occurs during estimation. Metrics collected from past projects are used as a basis from which effort and time estimates are made for current software work. As a project proceeds, measures of effort and calendar time expended are compared to original estimates (and the project schedule). The project manager uses these data to monitor and control progress. As technical work commences, other project metrics begin to have signifi - cance. Production rates represented in terms of models created, review hours, function points, and delivered source lines are measured. In addition, errors uncovered during each software engineering task are tracked. As the software evolves from requirements into design, technical metrics (Chapter 30) are collected to assess design quality and to provide indicators that will infl uence the approach taken to code generation and testing. The intent of project metrics is twofold. First, these metrics are used to minimize the development schedule by making the adjustments necessary to avoid delays and mitigate potential problems and risks. Second, project metrics are used to assess product quality on an ongoing basis and, when necessary, modify the technical approach to improve quality. As quality improves, defects are minimized, and as the defect count goes down, the amount of rework required during the project is also reduced. This leads to a reduction in overall project cost

SOFTWARE MEASUREMENT

In Chapter 30 we noted that measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the “quality” of bolts produced, measured by counting rejects). Software metrics can be categorized similarly. Direct measures of the software process include cost and effort applied. Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, effi ciency, reliability, maintainability, and many other “–abilities” that are discussed in Chapter 19. The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specifi c conventions for measurement are established in advance. However, the quality and functionality of software or its effi ciency or maintainability are more diffi cult to assess and can be measured only indirectly. We have partitioned the software metrics domain into process, project, and product metrics and noted that product metrics that are private to an individual are often combined to develop project metrics that are public to a software team. Project metrics are then consolidated to create process metrics that are public to the software organization as a whole. But how does an organization combine metrics that come from different individuals or projects? To illustrate, consider a simple example. Individuals on two different project teams record and categorize all errors that they fi nd during the software process. Individual measures are then combined to develop team measures. Team A found 342 errors during the software process prior to release. Team B found 184 errors. All other things being equal, which team is more effective in uncovering errors throughout the process? Because you do not know the size or complexity of the projects, you cannot answer this question. However, if the measures are normalized, it is possible to create software metrics that enable comparison to broader organizational averages

Size Oriented Metrics

Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures The table lists each software development project that has been completed over the past few years and corresponding measures for that project Referring to the table entry (Figure 32.2) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of$168,000. It should be noted that the effort and cost recorded in the table represent all



Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
⋮	⋮	⋮	⋮	⋮	⋮		

Figure 1.16: size oriented metrics

software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects were encountered after release to the customer within the fi rst year of operation. Three people worked on the development of software for project alpha. In order to develop metrics that can be assimilated with similar metrics from other projects, you can choose lines of code as a normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project

- Errors per KLOC (thousand lines of code)
- Defects per KLOC •

$ per KLOC •

Pages of documentation per KLOC In addition, other interesting metrics can be computed

- Errors per person-month
- KLOC per person-month •$ per page of documentation Size-oriented metrics are not universally accepted as the best way to measure the software process. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an “artifact” of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that when productivity is considered, they penalize well-designed but shorter

programs; that they cannot easily accommodate nonprocedural languages; and that their use in estimation requires a level of detail that may be diffi cult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed)

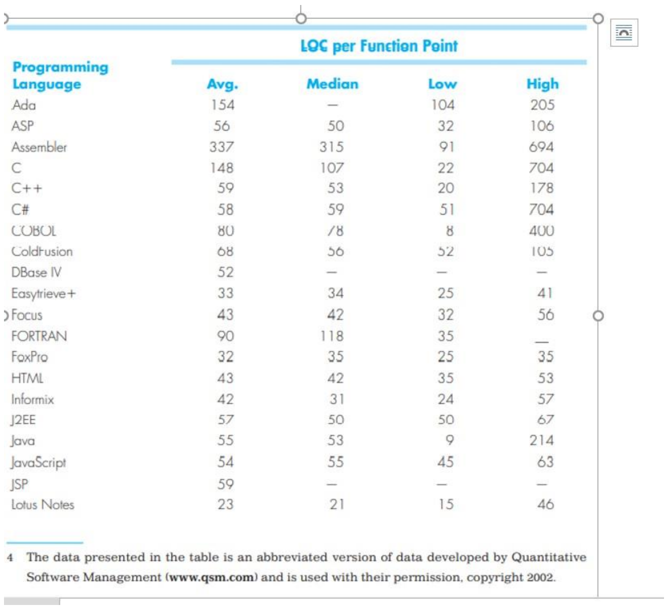
Function-Oriented Metrics

Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. The most widely used functionoriented metric is the function point (FP). Computation of the function point is based on characteristics of the software's information domain and complexity. The mechanics of FP computation have been discussed in Chapter 30

The function point, like the LOC measure, is controversial. Proponents claim that FP is programming language–independent, making it ideal for applications using conventional and nonprocedural languages, and that it is based on data that are more likely to be known early in the evolution of a project, making FP more attractive as an estimation approach. Opponents claim that the method requires some "sleight of hand" in that computation is based on subjective rather than objective data, that counts of the information domain (and other dimensions) can be diffi cult to collect after the fact, and that FP has no direct physical meaning it's just a number

Reconciling LOC and FP Metrics

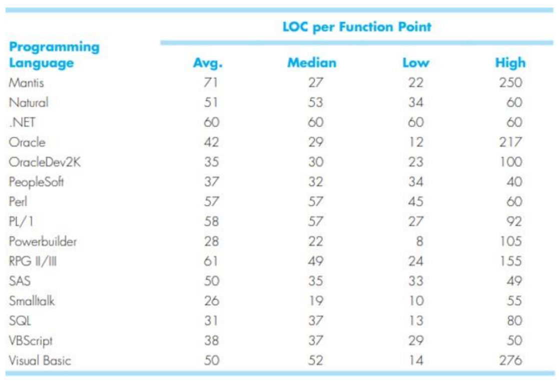
The relationship between lines of code and function points depends upon the programming language that is used to implement the software and the quality of the design. A number of studies have attempted to relate FP and LOC measures. The following table4 [QSM02] provides rough estimates of the average number of lines of code required to build one function point in various programming languages A review of these data indicates that one LOC of C11 provides approximately 2.4 times the "functionality" (on average) as one LOC of C. Furthermore, one LOC of a Smalltalk provides at least four times the functionality of an LOC for a conventional programming language such as Ada, COBOL, or C. Using the information contained in the table, it is possible to "backfi re" [Jon98] existing software to estimate the number of function points, once the total number of programming language statements are known. LOC and FP measures are often used to derive productivity metrics. This invariably leads to a debate about the use of such data. Should the LOC/ person-month (or FP/person-month) of one group be compared to



Programming Language	LOC per Function Point			
	Avg.	Median	Low	High
Ada	154	—	104	205
ASP	56	50	32	106
Assembler	337	315	91	694
C	148	107	22	704
C++	59	53	20	178
C#	58	59	51	704
COBOL	80	78	8	400
ColdFusion	68	56	52	105
DBase IV	52	—	—	—
Easytrieve+	33	34	25	41
Focus	43	42	32	56
FORTRAN	90	118	35	—
FoxPro	32	35	25	35
HTML	43	42	35	53
Informix	42	31	24	57
J2EE	57	50	50	67
Java	55	53	9	214
JavaScript	54	55	45	63
JSP	59	—	—	—
Lotus Notes	23	21	15	46

4 The data presented in the table is an abbreviated version of data developed by Quantitative Software Management (www.qsm.com) and is used with their permission, copyright 2002.

Figure 1.17: LOC per function point



Programming Language	LOC per Function Point			
	Avg.	Median	Low	High
Mantis	71	27	22	250
Natural	51	53	34	60
.NET	60	60	60	60
Oracle	42	29	12	217
OracleDev2K	35	30	23	100
PeopleSoft	37	32	34	40
Perl	57	57	45	60
PL/1	58	57	27	92
Powerbuilder	28	22	8	105
RPG II/III	61	49	24	155
SAS	50	35	33	49
Smalltalk	26	19	10	55
SQL	31	37	13	80
VBScript	38	37	29	50
Visual Basic	50	52	14	276

Figure 1.18: LOC

similar data from another? Should managers appraise the performance of individuals by using these metrics? The answer to these questions is an emphatic no! The reason for this response is that many factors infl uence productivity, making for “ apples and oranges” comparisons that can be easily misinterpreted. Function points and LOC-based metrics have been found to be relatively accurate predictors of software development effort and cost. However, in order to use LOC and FP for estimation (Chapter 33), an historical baseline of information must be established. Within the context of process and project metrics, you should be concerned primarily with productivity and quality—measures of software development “output” as a function of effort and time applied and measures of the “fi tness for use” of the work products that are produced. For process improvement and project planning purposes, your interest is historical. What was software development productivity on past projects? What was the quality of the software that PROCESS AND PROJECT METRICS 713 was produced? How can past productivity and quality data be extrapolated to the present? How can it help us improve the process and plan new projects more accurately

Object-Oriented Metrics

Conventional software project metrics (LOC or FP) can be used to estimate object-oriented software projects. However, these metrics do not provide enough granularity for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process. Lorenz and Kidd [Lor94] suggest the following set of metrics for OO projects: Number of scenario scripts

A scenario script (analogous to a use case) is a detailed sequence of steps that describes the interaction between the user and the application. Each script is organized into triplets of the form initiator, action, participant where initiator is the object that requests some service (that initiates a message), action is the result of the request, and participant is the server object that satisfi es the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed. Number of key classes. Key classes are the “highly independent components” [Lor94] that are defi ned early in object-oriented analysis (Chapter 10). 5 Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development. Number of support classes. Support classes are required to implement the system but are not immediately related to the problem domain. Examples might be user interface (UI) classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defi ned iteratively throughout an evolutionary process. The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development

Average number of support classes

per key class. In general, key classes are known early in the project. Support classes are

defi ned throughout. If the average number of support classes
per key class were known for a given problem domain, estimating (based on total number of classes) would be greatly simplified

object oriented concepts

What is an object-oriented (OO) viewpoint? Why is a method considered to be object oriented? What is an object? As OO concepts gained widespread adherents during the 1980s and 1990s, there were many different opinions about the correct answers to these questions, but today a coherent view of OO concepts has emerged. This appendix is designed to provide you with a brief overview of this important topic and to introduce basic concepts and terminology. To understand the object-oriented point of view, consider an example of a real-world object—the thing you are sitting in right now—a chair. Chair is a subclass of a much larger class that we can call PieceOfFurniture. Individual chairs are members (usually called instances) of the class Chair. A set of generic attributes can be associated with every object in the class PieceOfFurniture. For example, all furniture has a cost, dimensions, weight, location, and color, among many possible attributes. These apply whether we are talking about a table or a chair, a sofa or an armoire. Because Chair is a member of PieceOfFurniture, Chair inherits all attributes defi ned for the class
location = building + fl oor + room
then an operation named move() would modify one or more of the data items (building, fl oor, or room) that form the attribute location. To do this, move must have "knowledge" of these data items. The operation move() could be used for a chair or a table, as long as both are instances of the class PieceOfFurniture. Valid operations for the cl ass PieceOf-Furniture— buy(), sell(), weigh()—are specifi ed as part of the class defi nition and are inherited by all instances of the class
Now that we have introduced a few basic concepts, a more formal defi nition of object oriented will prove more meaningful. Coad and Yourdon [Coa91] defi ne the term this way: Object oriented = objects + classifi cation + inheritance + communication Three of these concepts have already been introduced. Communication is discussed later in this appendix
CLASSES AND OBJECTS

A class is an OO concept that encapsulates the data and procedural abstractions required to describe the content and behavior of some real-world entity. Data abstractions that describe the class are enclosed by a “wall” of procedural abstractions [Tay90] (represented in Figure A2.1) that are capable of manipulating the data in some way. In a well-designed class, the only way to reach the attributes (and operate on them) is to go through one of the methods that form the “wall” illustrated in the fi gure. Therefore, the class encapsulates data (inside the wall) and the processing that manipulates the data (the methods that make up the wall). This achieves information hiding (Chapter 12) and reduces the impact of side effects associated with change. Since the methods tend to manipulate a limited number of attributes, their cohesion is improved, and because communication occurs only through the methods that make up the “wall,” the class tends to be less strongly coupled from other elements of a system

1 It should be noted, however, that coupling can become a serious problem in OO sys-

the class tends to be less strongly coupled from other elements of a system...

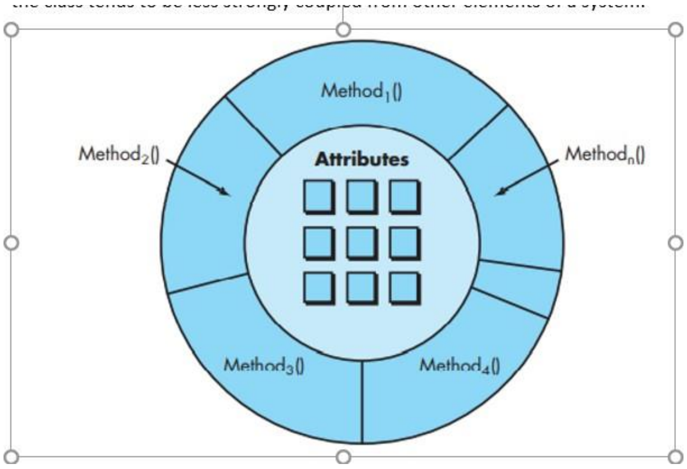


Figure 1.19: classes and objects

tems. It arises when classes from various parts of the system are used as the data types of attributes, and arguments to methods. Even though access to the objects may only be through procedure calls, this does not mean that coupling is necessarily low, just lower than if direct access to the internals of objects were allowed

Stated another way, a class is a generalized description (e.g., a template or blueprint) that describes a collection of similar objects. By defi nition, objects are instances of a specifi c class and inherit its attributes and the operations that are available to manipulate the attributes. A superclass (often called a base class) is a generalization of a set of classes that are related to it. A subclass is a specialization of the superclass. For example, the superclass MotorVehicle is a generalization of the classes Truck, SUV, Automobile, and Van. The subclass Automobile inherits all attributes of MotorVehicle, but in addition, incorporates other attributes that are specifi c only to automobiles. These defi nitions imply the existence of a class hierarchy in which the attributes and operations of the superclass are inherited by subclasses that may each add additional “private” attributes and methods. For example, the operations sitOn() and turn() might be private to the Chair subclass

ATTRIBUTES

You have learned that attributes are attached to classes and that they describe the class in some way. An attribute can take on a value defi ned by an enumerated domain. In most cases, a domain is simply a set of specifi c values. For example, assume that a class Automobile has an attribute color. The domain of values for color is white, black, silver, gray, blue, red, yellow, green. In more complex situations, the domain can be a class. Continuing the example, the class Automobile also has an attribute powerTrain that is itself a class. The class PowerTrain would contain attributes that describe the specifi c engine and transmission for the car. The features (values of the domain) can be augmented by assigning a default value (feature) to an attribute. For example, the color attribute defaults to white. It may also be useful to associate a probability with a particular feature by assigning value, probability pairs. Consider the color attribute for automobile. In some applications (e.g., manufacturing planning) it might be necessary to assign a probability to each of the colors (e.g., white and black are highly probable as automobile colors)

OPERATIONS , METHODS , AND SERVICES

An object encapsulates data (represented as a collection of attributes) and the algorithms that process the data. These algorithms are called operations, methods, or services2 and can be viewed as processing components. Each of the operations that is encapsulated by an object provides a representation of one of the behaviors of the object. For example, the operation GetColor() for the object Automobile will extract the color stored in the color attribute. The implication of the existence of this operation is that the class Automobile has been designed to receive a stimulus (we call the stimulus a message) that requests the color of the particular instance of a class. Whenever an object receives a stimulus, it initiates some behavior. This can be as simple as retrieving the color of automobile or as complex as the initiation of a chain of stimuli that are passed among a variety of different objects. In the latter case, consider an example in which the initial stimulus received by Object 1 results in the generation of two other stimuli that are sent to Object 2 and Object 3. Operations encapsulated by the second and third objects act on the stimuli, returning necessary information to the fi rst object. Object 1 then uses the returned information to satisfy the behavior demanded by the initial stimulus. OBJECT-ORIENTED A NALYSIS AND DESIGN CONCEPTS

Requirements modeling (also called analysis modeling) focuses primarily on classes that are extracted directly from the statement of the problem. These entity classes typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifi cally deleted). Design refi nes and extends the set of entity classes. Boundary and controller classes are developed and/or refi ned during design. Boundary classes create the interface (e.g., interactive screen and printed reports) that the user sees and interacts with as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users. Controller classes are designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, and (4) validation of data communicated between objects or between the user and the application. The concepts discussed in the paragraphs that follow can be useful in analysis and design work

Inheritance

Inheritance is one of the key differentiators between conventional and object-oriented systems. A subclass Y inherits all of the attributes and operations associated with its superclass X. This means that all data structures and algorithms originally designed and implemented for X are immediately available for Y—no further work need be done. Reuse has been accomplished directly. Any change to the attributes or operations contained within a superclass is immediately inherited by all subclasses. Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system. It is important to note that at each level of the class hierarchy new attributes and operations may be added to those that have been inherited from higher levels in the hierarchy. In fact, whenever a new class is to be created, you have a number of options: • The class can be designed and built from scratch. That is, inheritance is not used

- The class hierarchy can be searched to determine if a class higher in the hierarchy contains most of the required attributes and operations. The new class inherits from the higher class and additions may then be added, as required
- The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class
- Characteristics of an existing class can be overridden, and different versions of attributes or operations are implemented for the new class. Like all fundamental design concepts, inheritance can provide signifi cant benefi t for the design, but if it is used inappropriately, 3 it can complicate a design unnecessarily and lead to error-prone software that is diffi cult to maintain

Messages

Classes must interact with one another to achieve design goals. A message stimulates some behavior to occur in the receiving object. The behavior is accomplished when an operation is executed. The interaction between objects is illustrated schematically in Figure A2.2. An operation within SenderObject generates a message of the form message () where the parameters identify ReceiverObject as the object to be stimulated by the message, the operation within ReceiverObject that is to receive the message, and the data items that provide information that is required for the operation to be successful. The collaboration defi ned between classes as part of the analysis model provides useful guidance in the design of messages

For example, designing a subclass that inherits attributes and operations from more than

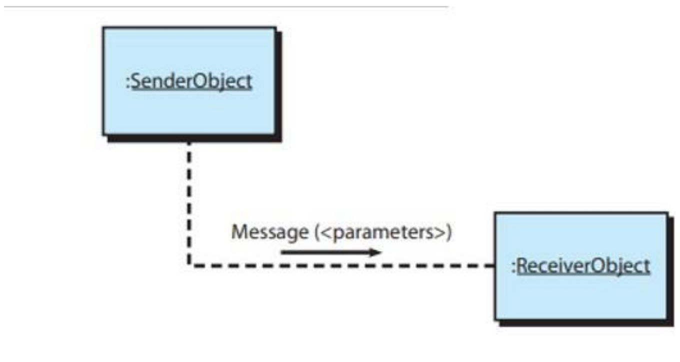


FIGURE A2.2 Message passing between objects

Figure 1.20: message

one superclass (sometimes called “multiple inheritance”) is frowned upon by most design-ers

Cox [Cox86] describes the interchange between classes in the following manner: An object [class] is requested to perform one of its operations by sending it a message telling the object what to do. The receiver [object] responds to the message by fi rst choosing the operation that implements the message name, executing this operation, and then returning control to the caller. Messaging ties an object-oriented system together. Messages provide insight into the behavior of individual objects and the OO system as a whole

Polymorphism. Polymorphism is a characteristic that greatly reduces the effort required to extend the design of an existing object-oriented system. To understand polymorphism, consider a conventional application that must draw four different types of graphs: line graphs, pie charts, histograms, and Kiviat diagrams. Ideally, once data are collected for a particular type of graph, the graph should draw itself. To accomplish this in a conventional application (and maintain module cohesion), it would be necessary to develop drawing modules for each type of graph. Then, within the design, control logic similar to the following would have to be embedded

```
case of graphtype
if graphtype = linegraph then DrawLineGraph (data);
if graphtype = piechart then DrawPieChart (data);
if graphtype = histogram then DrawHisto (data);
if graphtype = kiviat then DrawKiviat (data);
end case;
```

Although this design is reasonably straightforward, adding new graph types could be tricky. A new drawing module would have to be created for each graph type and then the control logic would have to be updated to refl ect the new graph type. To solve this problem, all of the graphs become subclasses of a general class called Graph. Using a concept called overloading [Tay90], each subclass defi nes an operation called draw. An object can send a draw message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own draw operation to create the appropriate graph. Therefore, the design is reduced to

```
draw<graphtype>
```

When a new graph type is to be added to the system, a subclass is created with its own draw operation. But no changes are required within any object that wants a graph drawn because the message draw remains unchanged. To summarize, polymorphism enables a number of different operations to have the same name. This in turn decouples objects from one another, making each more independent

Design classes. The requirements model defi nes a complete set of analysis classes. Each describes some element of the problem domain, focusing on aspects of the problem that are user or customer visible. The level of abstraction of an analysis class is relatively high. As the design model evolves, the software team must defi ne a set of design classes that (1) refi ne the analysis classes by providing design detail that will enable the classes to be implemented and (2) create a new set of design classes that implement a software infrastructure that supports the business solution. Five different types of design classes, each representing a different layer of the design architecture are suggested [Amb01]

- User interface classes defi ne all abstractions that are necessary for human-computer interaction (HCI)
- Business domain classes are often refi nements of the analysis classes defi ned earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain
- Process classes implement lower-level business abstractions required to fully manage the business domain classes
- Persistent classes represent data stores (e.g., a database) that will persist beyond the execution of the software
- System classes implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world

As the architectural design evolves, the software team should develop a complete set of attributes and operations for each design class. The level of abstraction is reduced as each analysis class is transformed into a design representation. That is, analysis classes represent objects (and associated methods that are applied to them) using the jargon of the business domain. Design classes present signifi - cantly more technical detail as a guide for implementation. Arlow and Neustadt [Arl02] suggest that each design class be reviewed to ensure that it is “well formed.” They defi ne four characteristics of a well-formed design class Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class. For example, the class Scene defi ned for video-editing software is complete only if it contains all attributes and methods that can reasonably be associated with the creation of a video scene. Suffi ciency ensures that the design class contains only those methods that are suffi cient to achieve the intent of the class, no more and no less

Primitiveness. Methods associated with a design class should be focused on accomplishing one specifi c function for the class. Once the function has been implemented with a method, the class should not provide another way to accomplish the same thing. For example, the class VideoClip of the video editing software might have attributes start-point and end-point to indicate the start and end points of the clip (note that the raw video loaded into the system may be longer than the clip that is used). The methods, setStartPoint() and setEndPoint() provide the only means for establishing start and end points for the clip. High cohesion. A cohesive design class is single minded. That is, it has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities. For example, the class VideoClip of the video-editing software might contain a set of methods for editing the video clip. As long as each method focuses solely on attributes associated with the video clip, cohesion is maintained

Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled (all design classes collaborate with all other design classes), the system is diffi cult to implement, test, and maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the law of Demeter [Lie03], suggests that a method should only send messages to methods in neighboring classes

Chapter 2

PLANNING AND SCHEDULING

Course Outcomes

After successful completion of this module, students should be able to:

CO 2	Make use of importance of project planning activities for selection and initiation of projects and portfolios.	Apply

Software requirements specification

The production of the requirements stage of the software development process is Software Requirements Specifications (SRS) (also called a requirements document). This report lays a foundation for software engineering activities and is constructing when entire requirements are elicited and analyzed. SRS is a formal report, which acts as a representation of software that enables the customers to review whether it (SRS) is according to their requirements. Also, it comprises user requirements for a system as well as detailed specifications of the system requirements. The SRS is a specification for a specific software product, program, or set of applications that perform particular functions in a specific environment. It serves several goals depending on who is writing it. First, the SRS could be written by the client of a system. Second, the SRS could be written by a developer of the system. The two methods create entirely various situations and establish different purposes for the document altogether. The first case, SRS, is used to define the needs and expectation of the users. The second case, SRS, is written for various purposes and serves as a contract document between customer and developer Characteristics of good SRS

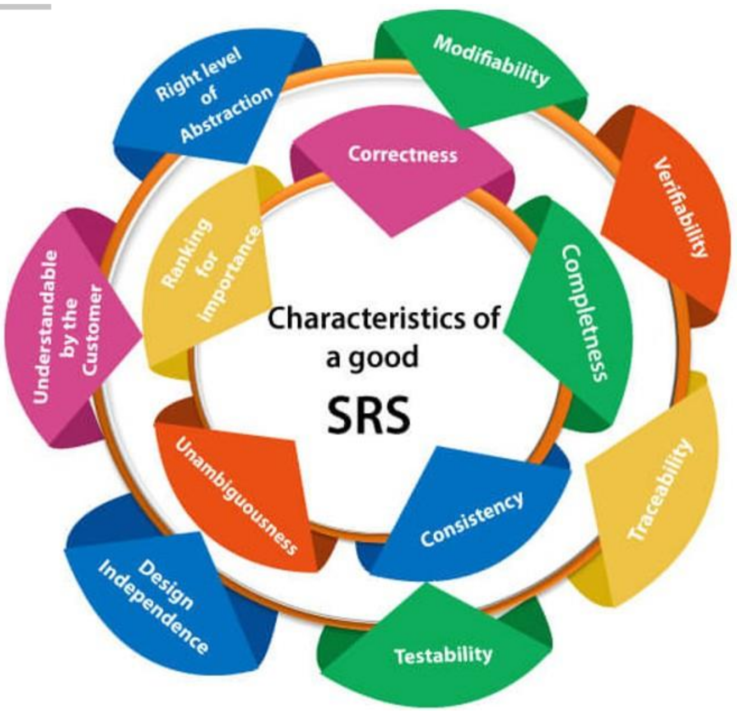


Figure 2.1: software requirement specification

Following are the features of a good SRS document

1. Correctness: User review is used to provide the accuracy of requirements stated in the SRS. SRS is said to be perfect if it covers all the needs that are truly expected from the system

Completeness: The SRS is complete if, and only if, it includes the following elements

1. All essential requirements, whether relating to functionality, performance, design, constraints, attributes, or external interfaces

2. Definition of their responses of the software to all realizable classes of input data in all available categories of situations

prototyping

- When to use: Customer defines a set of general objectives but does not identify detailed requirements for functions and features. or Developer may be unsure of the efficiency of an algorithm, the form that human computer interaction should take
- What step: Begins with communication by meeting with stakeholders to define the objective, identify whatever requirements are known, outline areas where further definition is mandatory. A quick plan for prototyping and modeling (quick design) occur. Quick design focuses on a representation of those aspects the software that will be visible to end users. (interface and output). Design leads to the construction of a prototype which will be deployed and evaluated. Stakeholder‘s comments will be used to refine requirements
- Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. However, engineers may make compromises in order to get a prototype working quickly. The less-than-ideal choice may be adopted forever after you get used to it

Prototyping can be problematic for the following reasons

- Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you hasn‘t considered overall software quality or long-term maintainability
- As a software engineer, you often make implementation compromises in order to get a prototype working quickly
- An inappropriate operating system or programming language may be used simply because it is available and known
- An inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system

1. Classic Process Models Waterfall Model (Linear Sequential Model)
2. Incremental Process Models - Incremental Model
3. Evolutionary Software Process Models
   •Prototyping
   •Spiral Model
   •Concurrent Development Model

Classic Process Models -Waterfall Model (Linear Sequential Model)

•The waterfall model, sometimes called the classic life cycle
It is the oldest paradigm for Software Engineering. When requirements are well defined and reasonably stable, it leads to a linear fashion
•The waterfall model, sometimes called the classic life cycle, suggests a systematic
•sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software
A variation of waterfall model depicts the relationship of quality assurance actions to the

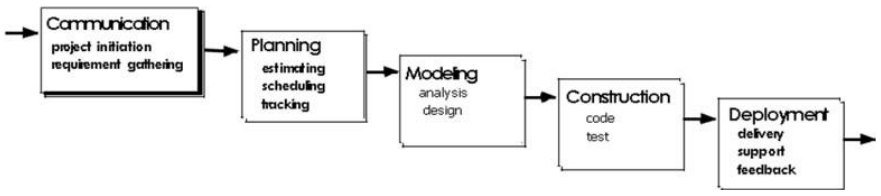


Figure 2.2: software process models

actions associated with communication, modeling and early code construction activates Team first moves down the left side of the V to refine the problem requirements. Once code is generated, the team moves up the right side of the V, performing a series of tests that validate each of the models created as the team moved down the left side
The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work

The V Model

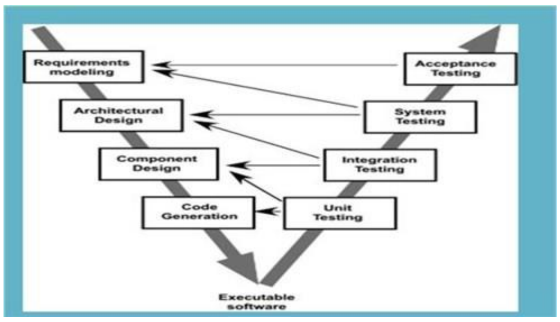


Figure 2.3: v model

The problems that are sometimes encountered when the waterfall model is applied are

•Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds

•It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects

•The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous

Incremental Process Models- Incremental Model

•When initial requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. A compelling need to expand a limited set of new functions to a later system release

•It combines elements of linear and parallel process flows. Each linear sequence produces deliverable increments of the software

•The first increment is often a core product with many supplementary features. Users use it and evaluate it with more modifications to better meet the needs

•The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user

•Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project

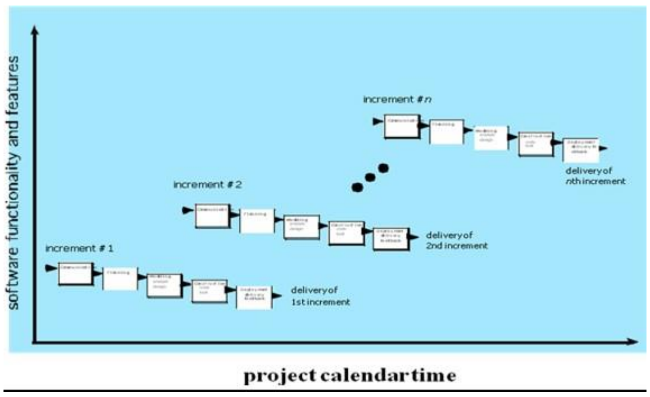

software functionality and features
increment # n
increment # 2
increment # 1
delivery of nth increment
delivery of 2nd increment
delivery of 1st increment
project calendar time

Figure 2.4: project cakender time

Evolutionary Software Process Models

•Prototyping

•Spiral Model

•Concurrent Development Model

•Software system evolves over time as requirements often change as development proceeds. Thus, a straight line to a complete end product is not possible. However, a limited version must be delivered to meet competitive pressure

•Usually a set of core product or system requirements is well understood, but the details and extension have yet to be defined

•You need a process model that has been explicitly designed to accommodate a product that evolved over time

•It is iterative that enables you to develop increasingly more complete version of the software

•Two types are introduced, namely Prototyping and Spiral models

Evolutionary Models: Prototyping

•When to use: Customer defines a set of general objectives but does not identify detailed

requirements for functions and features. or Developer may be unsure of the efficiency of an algorithm, the form that human computer interaction should take.
•What step: Begins with communication by meeting with stakeholders to define the objective, identify whatever requirements are known, outline areas where further definition is mandatory. A quick plan for prototyping and modeling (quick design) occur. Quick design focuses on a representation of those aspects the software that will be visible to end users. (interface and output). Design leads to the construction of a prototype which will be deployed and evaluated. Stakeholder‘s comments will be used to refine requirements
•Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. However, engineers may make compromises in order to get a prototype working quickly. The less-than-ideal choice may be adopted forever after you get used to it
Prototyping can be problematic for the following reasons
•Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you hasn‘t considered overall software quality or long-term maintainability
•As a software engineer, you often make implementation compromises in order to get a prototype working quickly
•An inappropriate operating system or programming language may be used simply because it is available and known
•An inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system
Evolutionary Models: The Spiral
•It couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model and is a risk-driven process model generator that is used to guide multi- stakeholder concurrent engineering of software intensive systems
•Two main distinguishing features: one is cyclic approach for incrementally growing a system‘s degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions

•A series of evolutionary releases are delivered. During the early iterations, the release might be a model or prototype. During later iterations, increasingly more complete version of the engineered system are produced
•The first circuit in the clockwise direction might result in the product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software
•Each pass results in adjustments to the project plan. Cost and schedule are adjusted based on feedback. Also, the number of iterations will be adjusted by project manager
•Good to develop large-scale system as software evolves as the process progresses and risk should be understood and properly reacted to. Prototyping is used to reduce risk
•However, it may be difficult to convince customers that it is controllable as it demands considerable risk assessment expertise
Concurrent Model
•Allow a software team to represent iterative and concurrent elements of any of the process models. For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following actions: prototyping, analysis and design
•The Figure shows modeling may be in any one of the states at any given time. For example, communication activity has completed its first iteration and in the awaiting changes state. The modeling activity was in inactive state, now makes a transition into the under development state. If customer indicates changes in requirements, the modeling activity moves from the under development state into the awaiting changes state
•Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions and tasks to a sequence of events, it defines a process network. Each activity, action or task on the network exists simultaneously with other activities, actions or tasks. Events generated at one point trigger transitions among the state

The project

We conduct planned and controlled software projects for one primary reason—it is the only known way to manage complexity. And yet, software teams still struggle. In a study of 250 large software projects between 1998 and 2004, Capers Jones [Jon04] found that “about 25 were deemed successful in that they achieved their schedule, cost, and quality objectives. About 50 had delays or overruns below 35 percent, while about 175 experienced major delays and overruns, or were ter- minated without completion.” Although the success rate for present-day soft- ware projects may have improved somewhat, our project failure rate remains much higher than it should be.1 To avoid project failure, a software project manager and the software engi- neers who build the product must avoid a set of common warning signs, under- stand the critical success factors that lead to good project man- agement, and develop a commonsense approach for planning, monitoring, and controlling the project. Each of these issues is discussed in Section 31.5 and in the chapters that follow

Planning

The objective of software project planning is to provide a framework that enables the man- ager to make reasonable estimates of resources, cost, and schedule. In addition, estimates should attempt to defi ne best-case and worst-case scenarios so that project outcomes can be bounded. Although there is an inherent degree of uncertainty, the software team em- barks on a plan that has been established as a consequence of these tasks. Therefore, the plan must be adapted and updated
as the project proceeds. In the following sections, each of the activities associated with software project planning is discussed Planning requires you to make an initial commit- ment, even though it’s likely that this “commitment” will be proven wrong. Whenever estimates are made, you look into the future and accept some degree of uncertainty as a matter of course. To quote Frederick Brooks [Bro95]
our techniques of estimating are poorly developed. More seriously, they refl ect an un- voiced assumption that is quite untrue, i.e., that all will go well because we are uncertain of our estimates, software managers often lack the courteous stubbornness to make people wait for a good product
are therefore diffi cult to use during software planning (before a design and code exist). However, other, more subjective assessments of complexity (e.g., function point complex- ity adjustment factors described in Chapter 30) can be established early in the planning

process

Project size is another important factor that can affect the accuracy and effi cacy of es-timates. As size increases, the interdependency among various elements of the software grows rapidly. 2 Problem decomposition, an important approach to estimating, becomes more diffi cult because the refi nement of problem elements may still be formidable. To paraphrase Murphy's law: "What can go wrong will go wrong"—and if there are more things that can fail, more things will fail

Scope

Software scope describes the functions and features that are to be delivered to end users; the data that are input and output; the "content" that is presented to users as a conse-quence of using the software; and the performance, constraints, interfaces, and reliability that bound the system. Scope is defi ned using one of two techniques

1. A narrative description of software scope is developed after communication with all stakeholders

2. A set of use cases 3

is developed by end users

Functions described in the statement of scope (or within the use cases) are evaluated and in some cases refi ned to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented

some degree of decomposition is often useful. Performance considerations encompass pro-cessing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systemsOnce scope has been identifi ed (with the concurrence of the customer), it is

reasonable to ask: "Can we build software to meet this scope? Is the project feasible?" All too often, software engineers rush past these questions (or are pushed past them by impatient managers or other stakeholders), only to become mired in a project that is doomed from the onset Putnam and Myers [Put97a] suggest that scoping is not enough. Once scope is understood, you must work to determine if it can be accomplished within the dimensions of available technology, dollars, time, and resources. This is a crucial, although often overlooked, part of the estimation process

Resource

The second planning task is estimation of the resources required to accomplish the software development effort. Figure 33.1depicts the three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software tools). Each resource is specifi ed with four characteristics: description of the resource, a statement of availability, time when the resource will be required, and duration of time that the resource will be applied. The last two characteristics can be viewed as a time window. Availability of the resource for a specifi ed window must be established at the earliest practical time.

Human Resources

The planner begins by evaluating software scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client-server) are specifi ed. For relatively small projects (a few person-months)

a single individual may perform all software engineering tasks, consulting with specialists

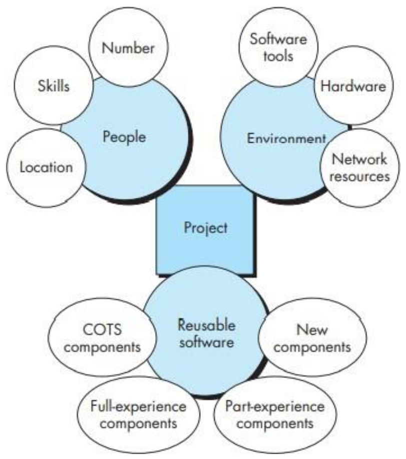


Figure 2.5: resource

as required. For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. Techniques for estimating effort are discussed later in this chapter

Reusable Software Resources

Component-based software engineering (CBSE)4 emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called components, must be cataloged for easy reference, standardized for easy application, and validated for easy integration. Bennatan [Ben00] suggests four software resource categories that should be considered as planning proceeds: off-the-shelf components (existing software that can be acquired from a third party or from a past project), full-experience components (existing specifi cations, designs, code, or test data developed for past projects that are similar to the software to be built for the current project), partial-experience components (existing specifi cations, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modifi cation), and new components (components built by the software team specifi cally for the needs of the current project)

Ironically, reusable software components are often neglected during planning, only to become a paramount concern during the development phase of the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur

Environmental Resources

The environment that supports a software project, often called the software engineering environment (SEE), incorporates hardware and software. Hardware provides a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice. 5 Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available. When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hard

elements being developed by other engineering teams. For example, software for a robotic device used within a manufacturing cell may require a specifi c robot (e.g., a robotic welder) as part of the validation test step; a software project for advanced page layout may need a high-speed digital printing system at some point during development. Each hardware element must be specifi ed as part of planning

Software Estimation

Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can affect the ultimate cost of software and effort applied to develop it. However, software project estimation can be transformed from a black art to a series of systematic steps that provide estimates with acceptable risk. To achieve reliable cost and effort estimates, a number of options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100 percent accurate estimates after the project is complete!).
2. Base estimates on similar projects that have already been completed.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates.
4. Use one or more empirical models for software cost and effort estimation.

Unfortunately, the fi rst option, however attractive, is not practical. Cost estimates must be provided up front. However, you should recognize that the longer you wait, the more you know, and the more you know, the less likely you are to make serious errors in your estimates.
The second option can work reasonably well, if the current project is quite similar to past efforts and other project infl uences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results
he remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a divide-and-conquer approach to software

project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. A model is based on experience (historical data) and takes the form
where d is one of a number of estimated values (e.g., effort, cost, project duration) and vi are selected independent parameters (e.g., estimated LOC or FP). Automated estimation tools implement one or more decomposition techniques or empirical models and provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g., experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data. Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation. In Chapter 32, we examined the characteristics of some of the software metrics that provide the basis for historical estimation data

Empirical Estimation Models

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.9 Values for LOC or FP are estimated using the approach described in Sections 33.6.3 and 33.6.4. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model
The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, you should use the results obtained from such models judiciously
An estimation model should be calibrated to refl ect local conditions. The model should be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results. If agreement is poor, the model must be tuned and retested before it can be used.

The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [Mat94]

E = A + B * (ev) C

where A, B, and C are empirically derived constants, E is effort in person-months, and ev is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (33.3), the majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g., problem complexity, staff experience, development environment). A quick examination of any empirically derived model indicates that it must be calibrated for local needs

The COCOMO II Model

In his classic book on software engineering economics, Barry Boehm [Boe81] introduced a hierarchy of software estimation models bearing the name COCOMO, for COnstructive COst MOdel. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMO II [Boe00]. Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address different “stages” of the software process

Like all estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, 10 function points, and lines of source code

The Software Equation

The software equation [Put92] is a dynamic multivariable model that assumes a specifi c distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4,000 contemporary software projects.

Based on these data, we derive an estimation model of the form

E =effort in person-months or person-years t = project duration in months or years B = "special skills factor" 11 P = "productivity parameter" that refl ects: overall process maturity and management practices, the extent to which good software engineering practices are used, the level of programming languages used, the state of the software environment, the skills and experience of the software team, and the complexity of the application

Typical values might be P 5 2,000 for development of real-time embedded software, P 5 10,000 for telecommunication and systems software, and P 5 28,000 for business systems applications. The productivity parameter can be derived for local conditions using historical data collected from past development efforts

You should note that the software equation has two independent parameters

1 an estimate of size (in LOC) and ,2 an indication of project duration in calendar months or years

To simplify the estimation process and use a more common form for their estimation model, Putnam and Myers [Put92] suggest a set of equations derived from the software equation. Minimum development time is defined as

Specialized Process Models

Specialized process models take on many of the characteristics of one or more of the traditional models. However, these models tend to be applied when a specialized or narrowly defined software engineering approach is chosen

•Component-Based Development

•The Formal Methods Model

•Aspect-Oriented Software Development

Component-Based Development

Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built

These components can be as either conventional software modules or object-oriented packages or packages of classes

Steps involved in CBS are

- Available component-based products are researched and evaluated for the application domain in question

•Component Integration issues are considered

•A software architecture is designed to accommodate the components

•Components are integrated into the architecture

•Comprehensive testing is conducted to ensure proper functionality

•Component-based development model leads to software reuse and reusability helps software engineers with a number of measurable benefits

•Component-based development leads to a 70 percent reduction in development cycle time, 84 percent reduction in project cost and productivity index of 26.2 compared to an industry norm of 16.9

Formal Methods Model

•Formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software

•They enable software engineers to specify, develop and verify a computer-based system by applying a rigorous mathematical notation

•Development of formal models is quite time consuming and expensive

•Extensive training is needed in applying formal methods

•Difficult to use the model as a communication mechanism for technically unsophisticated customers

Aspect-oriented Software Development

•The aspect-oriented approach is based on the principle of identifying common program code within certain aspects and placing the common procedures outside the main business logic

•The process of aspect orientation and software development may include modeling, design,programming, reverse-engineering and re-engineering

•The domain of AOSD includes applications, components and databases

•Interaction with and integration into other paradigms is carried out with the help of frameworks, generators, program languages and architecture-description languages (ADL)

The Unified process, personal and team process models

•The Unified Process is an iterative and incremental development process. Unified Process divides the project into four phases

1. Inception

2. Elaboration
3. Construction
4. Transition

•The Inception, Elaboration, Construction and Transition phases are divided into a series of time boxed iterations. (The Inception phase may also be divided into iterations for a large project.)
•Each iteration results in an increment, which is a release of the system that contains added or improved functionality compared with the previous release
•Although most iterations will include work in most of the process disciplines (e.g. Requirements, Design, Implementation, Testing) the relative effort and emphasis will change over the course of the project
•Risk Focused
The Unified Process requires the project team to focus on addressing the most critical risks early in the project life cycle. The deliverables of each iteration, especially in the Elaboration phase, must be selected in order to ensure that the greatest risks are addressed first. Risk Focused
Inception Phase
Inception is the smallest phase in the project, and ideally it should be quite short. If the Inception Phase is long then it is usually an indication of excessive up-front specification, which is contrary to the spirit of the Unified Process
The following are typical goals for the Inception phase
•Establish a justification or business case for the project
•Establish the project scope and boundary conditions
•Outline the use cases and key requirements that will drive the design tradeoffs
•Outline one or more candidate architectures
•Identify risks
•Prepare a preliminary project schedule and cost estimate
The Lifecycle Objective Milestone marks the end of the Inception phase
Elaboration Phase
During the Elaboration phase the project team is expected to capture a majority of the system requirements. The primary goals of Elaboration are to address known risk factors

and to establish and validate the system architecture

Common processes undertaken in this phase include the creation of use case diagrams, conceptual diagrams (class diagrams with only basic notation) and package diagrams (architectural diagrams)

The architecture is validated primarily through the implementation of an Executable Architectural Baseline. This is a partial implementation of the system which includes the core, most architecturally significant, components. It is built in a series of small, time-boxed iterations

By the end of the Elaboration phase the system architecture must have stabilized and the executable architecture baseline must demonstrate that the architecture will support the key system functionality and exhibit the right behavior in terms of performance, scalability and cost

The final Elaboration phase deliverable is a plan (including cost and schedule estimates) for the Construction phase. At this point the plan should be accurate and credible, since it should be based on the Elaboration phase experience and since significant risk factors should have been addressed during the Elaboration phase. The Lifecycle Architecture Milestone marks the end of the Elaboration phase

Construction Phase

Construction is the largest phase in the project. In this phase the remainder of the system is built on the foundation laid in Elaboration. System features are implemented in a series of short, timeboxed iterations. Each iteration results in an executable release of the software. It is customary to write full text use cases during the construction phase and each one becomes the start of a new iteration

Common UML (Unified Modeling Language) diagrams used during this phase include Activity, Sequence, Collaboration, State (Transition) and Interaction Overview diagrams

The Initial Operational Capability Milestone marks the end of the Construction phase

Transition Phase

The final project phase is Transition. In this phase the system is deployed to the target users. Feedback received from an initial release (or initial releases) may result in further refinements to be incorporated over the course of several Transition phase iterations. The Transition phase also includes system conversions and user training

The Product Release Milestone marks the end of the Transition phase

Advantages of UP Software Development

This is a complete methodology in itself with an emphasis on accurate documentation

It is proactively able to resolve the project risks associated with the client's evolving requirements requiring careful change request management Less time is required for integration as the process of integration goes on throughout the software development life cycle

The development time required is less due to reuse of components

Disadvantages of RUP Software Development

The team members need to be expert in their field to develop a software under this methodology

On cutting edge projects which utilise new technology, the reuse of components will not be possible. Hence the time saving one could have made will be impossible to fulfill

Integration throughout the process of software development, in theory sounds a good thing. But on particularly big projects with multiple development streams it will only add to the confusion and cause more issues during the stages of testing

Personal and Team process models

The best software process is one that is close to the people who will be doing the work. The PSP model defines five framework activities

Personal Software Process (PSP)

Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created

High-level design. External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked

High-level design review. Formal verification methods (Chapter 21) are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results

Development. The component-level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results

Postmortem. Using the measures and metrics collected, the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness

Team Software Process (TSP)

The goal of TSP is to build a self directed project team that organizes itself to produce high-quality software. TSP objectives are

•Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPTs) of 3 to about 20 engineers

•Show managers how to coach and motivate their teams and how to help them sustain peak performance

•Accelerate software process improvement by making CMM23 Level 5 behavior normal and expected

•Provide improvement guidance to high-maturity organizations

•Facilitate university teaching of industrial-grade team skills

Characteristics of good SRS

Following are the features of a good SRS document

. Correctness: User review is used to provide the accuracy of requirements stated in the SRS. SRS is said to be perfect if it covers all the needs that are truly expected from the system

. Completeness: The SRS is complete if, and only if, it includes the following elements

. All essential requirements, whether relating to functionality, performance, design, constraints, attributes, or external interfaces

. Definition of their responses of the software to all realizable classes of input data in all available categories of situations

risk management

When you consider risk in the context of software engineering, Charette’s three conceptual underpinnings are always in evidence. The future is your concern—what risks might cause the software project to go awry? Change is your concern—how will changes in customer requirements, development

REACTIVE VERSUS PROACTIVE RISK STRATEGIES

Reactive risk strategies have been laughingly called the “Indiana Jones school of risk management” [Tho92]. In the movies that carried his name, Indiana Jones, when faced with overwhelming diffi culty, would invariably say, “Don’t worry, I’ll think of something!” Never worrying about problems until they happened, Indy would react in some heroic way Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team fl ies into action in an attempt to correct the problem rapidly. This is often called a fi re-fi ghting mode. When this fails, “crisis management” [Cha92] takes over and the project is in real jeopardy

A considerably more intelligent strategy for risk management is to be proactive. A proactive strategy begins long before technical work is initiated. Potential risks are identifi ed, their probability and impact are assessed, and they

are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner. Throughout the remainder of this chapter, we discuss a proactive strategy for risk management

SOFTWARE RISKS

Although there has been considerable debate about the proper defi nition for software risk, there is general agreement that risk always involves two characteristics: uncertainty—the risk may or may not happen; that is, there are no

100 percent probable risks 1—and loss—if the risk becomes a reality, unwanted consequences or losses will occur [Hig95]. When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffi ng and organization), resource, stakeholder, and requirements problems and their impact on a software project. In Chapter 33, project complexity, size, and the degree of structural uncertainty were also defi ned as project (and estimation) risk factors

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become diffi cult or impossible. Technical risks identify potential design, implementation, interface, verifi cation, and maintenance problems. In addition, specifi cation ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be

RISK IDENTIFICATION

Risk identifi cation is a systematic attempt to specify threats to the project plan(estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a fi rst step toward avoiding them when possible and controlling them when necessary. There are two distinct types of risks for each of the categories that have been presented in Section 35.2. Generic risks are a potential threat to every software
project. Product-specifi c risks can be identifi ed only by those with a clear understanding of the technology, the people, and the environment that is specifi c to the software that is to be built. To identify product-specifi c risks, the project plan and
the software statement of scope are examined, and an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identifi cation and focuses on some subset of known and predictable risks in the following generic subcategories

Product size—Risks associated with the overall size of the software to be built or modifi ed

- Business impact—Risks associated with constraints imposed by management or the marketplace
- Stakeholder characteristics—Risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner

Process defi nition—Risks associated with the degree to which the software process has been defi ned and is followed by the development organization

- Development environment—Risks associated with the availability and quality of the tools to be used to build the product

- Technology to be built—Risks associated with the complexity of the system to be built and the “newness” of the technology that is packaged by the system
- Staff size and experience—Risks associated with the overall technical and project experience of the software engineers who will do the work

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow you to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of “risk components and drivers” [AFC88] are listed along with their probability of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions

A number of comprehensive checklists for software project risk are available on the Web (e.g., [Baa07], [NAS07], [Wor04]). You can use these checklists to gain insight into generic risks for software projects. In addition to the use of checklists, risk patterns [Mil04] have been proposed as a systematic approach to risk identifi cation

Assessing Overall Project Risk

The following questions have been derived from risk data obtained by surveying experienced software project managers in different parts of the world [Kei98]. The questions are ordered by their relative importance to the success of a project

1. Have top software and customer managers formally committed to support the project
2. Are end users enthusiastically committed to the project and the system product to be built
3. requirements fully understood by the software engineering team and its customers
4. Have customers been involved fully in the defi nition of requirements
5. Have customers been involved fully in the defi nition of requirements
6. Do end users have realistic expectations
7. Is the project scope stable

8. Does the software engineering team have the right mix of skills
9. Are project requirements stable
10. Does the project team have experience with the technology to be implemented
11. Is the number of people on the project team adequate to do the job
12. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built

If any one of these questions is answered negatively, mitigation, monitoring, and management steps should be instituted without fail. The degree to which the project is at risk is directly proportional to the number of negative responses to these questions

Risk Components and Drivers

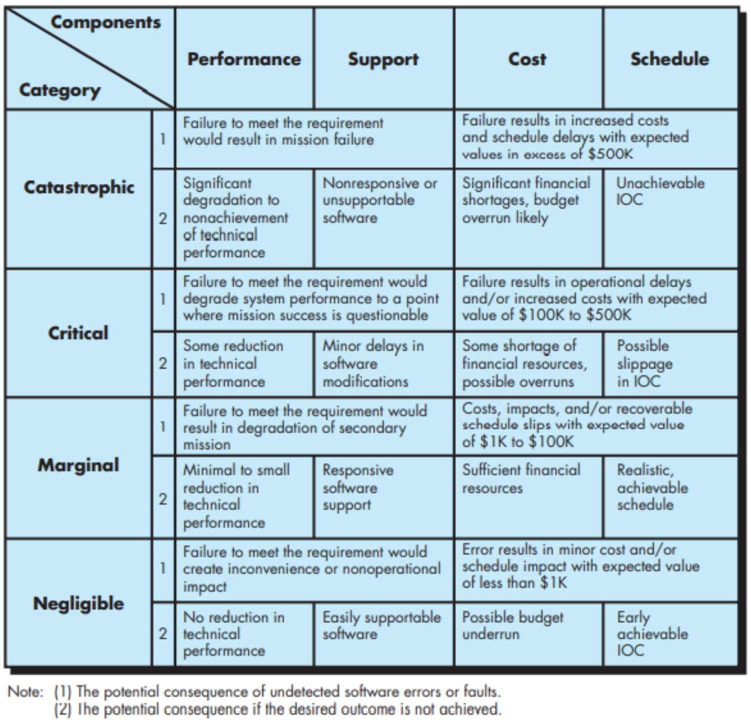
The U.S. Air Force [AFC88] has published a pamphlet that contains excellent guidelines for software risk identifi cation and abatement. The Air Force approach requires that the project manager identify the risk drivers that affect software risk components—performance, cost, support, and schedule. In the context of this discussion, the risk components are defi ned in the following manner
Performance risk—The degree of uncertainty that the product will meet its requirements and be fi t for its intended use. • Cost risk—The degree of uncertainty that the project budget will be maintained
• Support risk—The degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance
• Schedule risk—The degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time
The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic. Referring to Figure 35.1[Boe89], a characterization of the potential consequences of errors (rows labeled 1) or a failure to achieve a desired outcome (rows labeled 2) are described. The impact category is chosen

based on the characterization that best fits the description in the table

RISK PROJECTION

Risk projection, also called risk estimation, attempts to rate each risk in two ways—(1) the likelihood or probability that the risk is real and will occur and (2) the consequences of the problems associated with the risk, should it occur. You work along with other managers and technical staff to perform four risk projection steps

1. Establish a scale that refl ects the perceived likelihood of a risk
2. Delineate the consequences of the risk



Category		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Note: (1) The potential consequence of undetected software errors or faults.
(2) The potential consequence if the desired outcome is not achieved.

Figure 2.6: risk projection

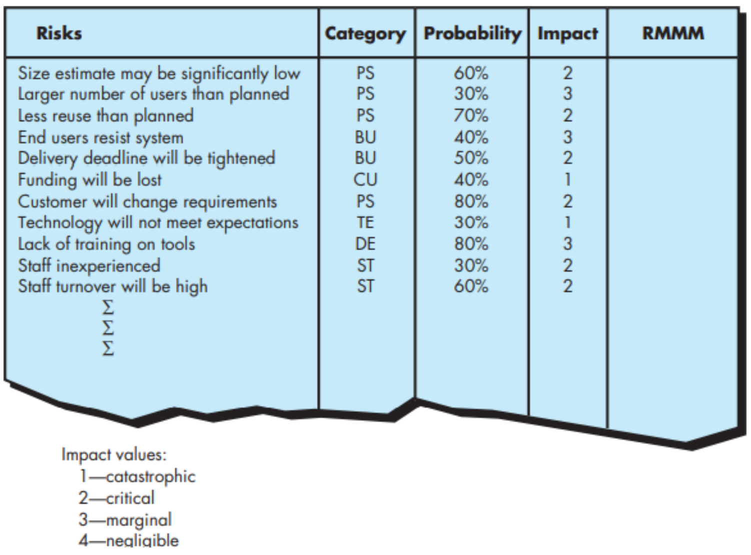
Estimate the impact of the risk on the project and the product

4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings. The intent of these steps is to consider risks in a manner that leads to prioriization. No software team has the resources to address every possible risk with the same degree of

rigor. By prioritizing risks, you can allocate resources where they will have the most impact

Developing a Risk Table

A risk table provides you with a simple technique for risk projection.2 A sample risk table is illustrated in Figure 35.2



Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
Σ				
Σ				
Σ				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

Figure 2.7: risk

You begin by listing all risks (no matter how remote) in the fi rst column of the table. This can be accomplished with the help of the risk item checklists referenced in Section 35.3. Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. One way to accomplish this is to poll individual team members in round-robin fashion until their collective assessment of risk probability begins to converge

reevaluated to accomplish second-order prioritization. Referring to Figure 35.3 , risk impact and probability have a distinct infl uence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a signifi cant amount of management time. However, high-impact risks with moderate to high

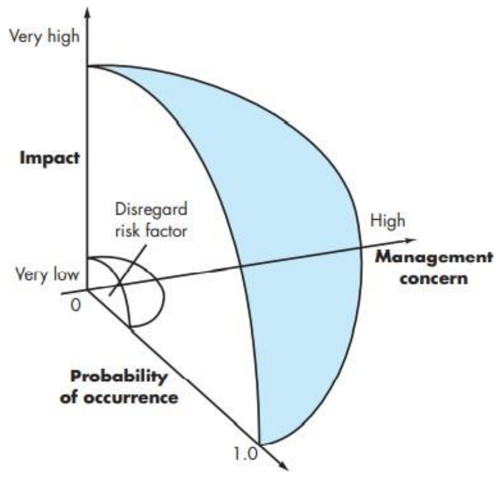


Figure 2.8: risk management

probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow. All risks that lie above the cutoff line should be managed. The column labeled RMMM contains a pointer into a risk mitigation, monitoring, and management
plan or, alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff. The RMMM plan and risk information sheets are discussed in Sections 35.5 and 35.6. Risk probability can be determined by making individual estimates and then developing a single consensus value. Although that approach is workable, more sophisticated techniques for determining risk probability have been developed

Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defi ned external interface to customer hardware (a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many stakeholders are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt. In most cases, you want the “bad news” to occur as soon as possible, but in some cases, the longer the delay, the better

Returning once more to the risk analysis approach proposed by the U.S. Air Force [AFC88], you can apply the following steps to determine the overall consequences of a risk: (1) determine the average probability of occurrence value for each risk component; (2) using Figure 35.1 , determine the impact for each component based on the criteria shown, and (3) complete the risk table and analyze the results as described in the preceding sections. The overall risk exposure, RE, is determined using the following relationship [Hal98] RE=P *C where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur. For example, assume that the software team defi nes a project risk in the following manner: Risk identifi cation. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed. Risk probability. Eighty percent (likely). Risk impact. Sixty reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data

indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be 18 3 100 3 14 5 $25,200. Risk exposure. RE 5 0.80 3 25,200 $̃20,200. Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the fi nal cost estimate fora project

It can also be used to predict the probable increase in staff resources required at various points during the project schedule. The risk projection and analysis techniques described in Sections 35.4.1 and 35.4.2 are applied iteratively as the software project proceeds. 4 The project team should revisit the risk table at regular intervals, reevaluating each risk to

RISK REFINEMENT

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refi ne the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage. One way to do this is to represent the risk in condition-transition-consequence(CTC) format [Glu94]. That is, the risk is stated in the following form: Given that <condition> then there is concern that (possibly) <consequence>

Using the CTC format for the reuse risk noted in Section 35.4.2, you could write: Given that all reusable software components must conform to specifi c design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components

This general condition can be refi ned in the following manner

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards

Subcondition 2. The design standard for component interfaces has not been solidifi ed and may not conform to certain existing reusable components

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment

The consequences associated with these refi ned subconditions remain the same (i.e., 30 percent of software components must be custom engineered), but the refi nement helps to isolate the underlying risks and might lead to easier analysis and response

Software Project Scheduling

- Basic Principles
- The Relationship Between People and Effort
- Effort Distribution
- Software project scheduling is an action that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks
- During early stages of project planning, a macroscopic schedule is developed
- As the project gets under way, each entry on the macroscopic schedule is refined into a detailed schedule. Basic Principles of Project Scheduling

1. Compartmentalization: The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are refined

2. Interdependency: The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence, while others can occur in parallel. Other activities can occur independently

3. Time allocation: Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date. Whether work will be conducted on a full-time or part-time basis

4. Effort validation: Every project has a defined number of people on the software team. The project manager must ensure that no more than the allocated number of people have been scheduled at any given time

5. Defined responsibilities. Every task that is scheduled should be assigned to a specific team member

6. Defined outcomes: Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g., the design of a component) or a part of a work product. Work products are often combined in deliverables

7. Defined milestones: Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved

Each of these principles is applied as the project schedule evolves. The Relationship between People and Effort

- In small software development project a single person can analyze requirements, perform design, generate code, and conduct tests. As the size of a project increases, more people must become involved
- There is a common myth that is still believed by many managers who are responsible for software development projects: If we fall behind schedule, we can always add more programmers and catch up later in the project
- Unfortunately, adding people late in a project often has a disruptive effect on the project, causing schedules to slip even further. The people who are added must learn the system, and the people who teach them are the same people who were doing the work
- While teaching, no work is done, and the project falls further behind. In addition to

the time it takes to learn the system, more people

- Although communication is absolutely essential to successful software development, every new communication path requires additional effort and therefore additional time

Effort distribution

- A recommended distribution of effort across the software process is often referred to as the 40– 20–40 rule
- Forty percent of all effort is allocated to frontend analysis and design. A similar percentage is applied to back-end testing. You can correctly infer that coding (20 percent of effort) is deemphasized
- Work expended on project planning rarely accounts for more than 2 to 3 percent of effort, unless the plan commits an organization to large expenditures with high risk
- Customer communication and requirements analysis may comprise 10 to 25 percent of project effort
- Effort expended on analysis or prototyping should increase in direct proportion with project size and complexity
- A range of 20 to 25 percent of effort is normally applied to software design. Time expended for design review and subsequent iteration must also be considered
- Because of the effort applied to software design, code should follow with relatively little difficulty
- A range of 15 to 20 percent of overall effort can be achieved. Testing and subsequent debugging can account for 30 to 40 percent of software development effort
- The criticality of the software often dictates the amount of testing that is required. If software is human rated (i.e., software failure can result in loss of life), even higher percentages are typical

Object Oriented Estimation and Scheduling

It is worthwhile to supplement conventional software cost estimation methods with a technique that has been designed explicitly for OO software. Lorenz and Kidd [Lor94] suggest the following approach

1. Develop estimates using effort decomposition, FP analysis, and any other method that

is applicable for conventional applications

2. Using the requirements model (Chapter 10), develop use cases and determine a count. Recognize that the number of use cases may change as the project progresses

3. From the requirements model, determine the number of key classes (called analysis classes in Chapter 10)

4. Categorize the type of interface for the application and develop a multiplier for support classes, where the multipliers for no GUI, a text-based user interface, a
conventional GUI, and a complex GUI are: 2.0, 2.25, 2.5, and 3.0, respectively. Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes

5. Multiply the total number of classes (key 1 support) by the average number of work units per class. Lorenz and Kidd suggest 15 to 20 person-days per class

6. Cross-check the class-based estimate by multiplying the average number of work units per use case

SPECIALIZED ESTIMATION TECHNIQUES

The estimation techniques discussed in Sections 33.6 through 33.8 can be used for any software project. However, when a software team encounters an extremely short project duration (weeks rather than months) that is likely to have a continuing stream of changes, project planning in general, and estimation in particular should be abbreviated. 12 In the sections that follow, I examine two specialized estimation technique

Estimation for Agile Development

Because the requirements for an agile project (Chapter 5) are defi ned by a set of user scenarios (e.g., “stories” in Extreme Programming), it is possible to develop an estimation approach that is informal, reasonably disciplined, and meaningful

Each user scenario (the equivalent of a mini use case created at the very start of a project by end users or other stakeholders) is considered separately for estimation purposes

2. The scenario is decomposed into the set of software engineering tasks that will be required to develop it

3a. Each task is estimated separately. Note: Estimation can be based on historical data, an empirical model, or “experience.”

3b. Alternatively, the “volume” of the scenario can be estimated in LOC, FP, or some other volume-oriented measure (e.g., use case count)

4a. Estimates for each task are summed to create an estimate for the scenario

4b. Alternatively, the volume estimate for the scenario is translated into effort using historical data

5. The effort estimates for all scenarios that are to be implemented for a given software increment are summed to develop the effort estimate for the increment

Because the project duration required for the development of a software increment is quite short (typically three to six weeks)

this estimation approach serves two purposes

1 to be certain that the number of scenarios to be included in the increment conforms to the available resources, and

2 to establish a basis for allocating effort as the increment is developed

Estimation for WebApp Projects

WebApp projects often adopt the agile process model. A modifi ed function point measure, coupled with the steps outlined in Section 33.9.1, can be used to develop an estimate for the WebApp. Roetzheim [Roe00] suggests the following approach when adapting function points for WebApp estimation

- Inputs are each input screen or form (for example, CGI or Java), each maintenance screen, and if you use a tab notebook metaphor anywhere, each tab
- Outputs are each static Web page, each dynamic Web page script (for example, ASP, ISAPI, or other DHTML script), and each report (whether Web based or administrative in nature)
- Tables are each logical table in the database plus, if you are using XML to store data in a fi le, each XML object (or collection of XML attributes)
- Interfaces retain their defi nition as logical fi les (for example, unique record formats) into our out-of-the-system boundari

Chapter 3

ANALYSIS

Course Outcomes

After successful completion of this module, students should be able to:

CO3	Analyze functional, behavioral, functional and object-oriented models for software model development.	Analyze
CO4	Outline basic principles, building blocks and views for designing object-oriented architectural view of a system	Understand

Analysis modeling

- Analysis model operates as a link between the ’system description’ and the ’design model’
- In the analysis model, information, functions and the behaviour of the system is defined and these are translated into the architecture, interface and component level design in the ’design modeling’

Elements of the analysis model

1. Scenario based element

- This type of element represents the system user point of view
- Scenario based elements are use case diagram, user stories

2. Class based elements

- The object of this type of element manipulated by the system
- It defines the object,attributes and relationship
- The collaboration is occurring between the classes
- Class based elements are the class diagram, collaboration diagram

3. Behavioral elements

- Behavioral elements represent state of the system and how it is changed by the external events
- The behavioral elements are sequenced diagram, state diagram

4. Flow oriented elements

- An information flows through a computer-based system it gets transformed
- It shows how the data objects are transformed while they flow between the various system functions
- The flow elements are data flow diagram, control flow diagram

Analysis Rules of Thumb

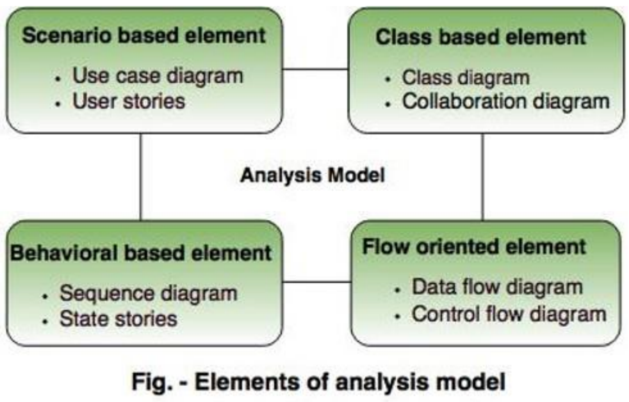


Figure 3.1: elements of analysis model

The rules of thumb that must be followed while creating the analysis model

The rules are as follows

- The model focuses on the requirements in the business domain. The level of abstraction must be high i.e there is no need to give details

- Every element in the model helps in understanding the software requirement and focus on the information, function and behaviour of the system
- The consideration of infrastructure and nonfunctional model delayed in the design

For example, the database is required for a system, but the classes, functions and behavior of the database are not initially required. If these are initially considered then there is a delay in the designing

- Throughout the system minimum coupling is required. The interconnections between the modules is known as ’coupling’
- The analysis model gives value to all the people related to model
- The model should be simple as possible. Because simple model always helps in easy understanding of the requirement

Concepts of data modeling

- Analysis modeling starts with the data modeling
- The software engineer defines all the data object that proceeds within the system and the relationship between data objects are identified. Data objects
- The data object is the representation of composite information
- The composite information means an object has a number of different properties or attribute

For example, Height is a single value so it is not a valid data object, but dimensions contain the height, the width and depth these are defined as an object. Data Attributes

Each of the data object has a set of attributes

Data object has the following characteristics

- Name an instance of the data object
- Describe the instance
- Make reference to another instance in another table ,Relationship

Relationship shows the relationship between data objects and how they are related to each other

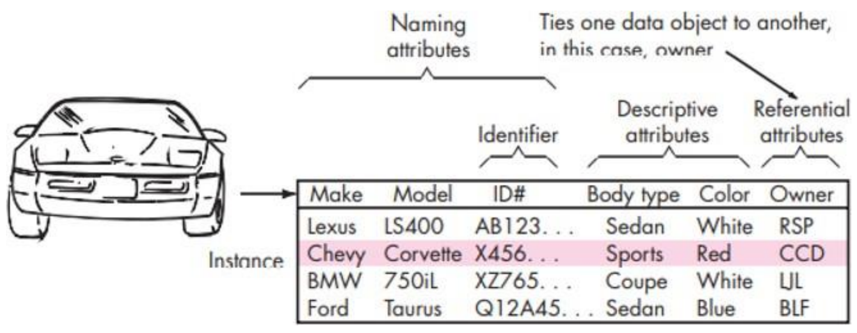
Data modeling

If software requirements include the need to create, extend, or interface with a database or if complex data structures must be constructed and manipulated, the software team may choose to create a data model as part of overall requirements modeling. A software engineer or analyst defines all data objects that are processed within the system, the relationships between the data objects, and other information that is pertinent to the relationships. The entity-relationship diagram (ERD) addresses these issues and represents all data objects that are entered, stored, transformed, and produced within an application Data Objects A data object is a representation of composite information that must be understood by software. By composite information, I mean something that has a number of different properties or attributes

Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object. A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.10 Therefore, the data object can be represented as a table as shown in Figure 6.7. The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color, and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object car

Data Attributes Data attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an identifier—that is, the identifier Relationships Data objects are connected to one another in different ways. Consider the two data objects, person and car. These objects can be represented using the



Make	Model	ID#	Body type	Color	Owner
Lexus	LS400	AB123. . .	Sedan	White	RSP
Chevy	Corvette	X456. . .	Sports	Red	CCD
BMW	750iL	XZ765. . .	Coupe	White	LJL
Ford	Taurus	Q12A45. . .	Sedan	Blue	BLF

Figure 3.2: data object

simple notation illustrated in Figure 6.8a. A connection is established between person and

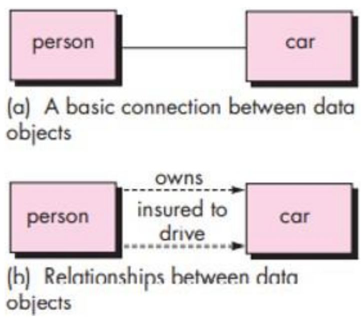

person
car
(a) A basic connection between data objects
owns
insured to drive
person
car
(b) Relationships between data objects

Figure 3.3: Relationship between data and objectt

car because the two objects are related. But what are the relationships? To determine the answer, you should understand the role of people (owners, in this case) and cars within the context of the software to be built. You can establish a set of object/ relationship pairs that define the relevant relationships. For example, • A person owns a car. • A person is insured to drive a car. The relationships owns and insured to drive define the relevant connections between person and car. Figure 6.8b illustrates these object-relationship pairs graphically. The arrows noted in Figure 6.8b provide important information about the directionality of the relationship and often reduce ambiguity or misinterpretations

Functional modeling and Information Flow

unction Modeling & Information Flow Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies hardware, software, and human elements to transform it; and produces output in a variety of forms Structured analysis began as an information flow modeling technique. A rectangle is used to represent an external entity (software, hardware, a person) A circle (sometimes called a bubble) represents a process or transform that is applied to data (or control) and changes it in some way. n arrow represents one or more data items (data objects) and it should be labeled. The double line represents a data store—stored information that is used by the software. First data flow model (sometimes called a level 0 DFD or context diagram) represents the entire system. It provides incremental detail with each s Information Flow model
Creatinga Data Flow Model It enables software engineer to develop models of the information domain and functional domain at the same time. Data flow diagram may be used to represent a system or software at any level of abstraction As DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system. As DFD refinement results in corresponding refinement of data as it moves through the processes that represent the application

Behavioral Modelling

Analysts view the problem as a set of use cases supported by a set of collaborating objects Aids in organizing and defining the software Behavioral models depict this view of the business processes: How the objects interact and form a collaboration to support the use cases An internal view of the business process described by a use case Creating behavioral models is an iterative process which may induce changes in other models
Interaction Diagrams Objects—an instantiation of a class Patient is a class Mary Wilson is an instantiation of the patient class (object) Attributes—characteristics of a class Patient class: name, address, phone, etc. Operations—the behaviors of a class, or an action that an object can perform Messages—information sent to objects to tell them to execute

one of their behaviors A function call from one object to another Types Sequence Diagrams—emphasize message sequence Communication Diagrams—emphasize message flow Sequence Diagrams Illustrate the objects that participate in a single use-case A dynamic model Shows the sequence of messages that pass between objects Aid in understanding real-time specifications and complex use-cases Generic diagram shows all scenarios for a use-case Instance diagrams show a single scenario

Structured Analysis

Structured Analysis is a development method that allows the analyst to understand the system and its activities in a logical way. It is a systematic approach, which uses graphical tools that analyze and refine the objectives of an existing system and develop a new system specification which can be easily understandable by user

It has following attributes

- It is graphic which specifies the presentation of application
- It divides the processes so that it gives a clear picture of system flow. • It is logical rather than physical i.e., the elements of system do not depend on vendor or hardware
- It is an approach that works from high-level overviews to lower-level details

Structured Analysis Tools

During Structured Analysis, various tools and techniques are used for system development. They are

- Data Flow Diagrams
- Data Dictionary
- Decision Trees
- Decision Tables
- Structured English
- Pseudocode

Object Oriented Analysis

the system analysis or object-oriented analysis phase of software development, the system requirements are determined, the classes are identified and the relationships among classes are identified

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modelling, dynamic modelling, and functional modelling

Object Modelling Object modelling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class

The process of object modelling can be visualized in the following steps

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

Dynamic Modelling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling

Dynamic Modelling can be defined as “a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world”

The process of dynamic modelling can be visualized in the following steps

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modelling

Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis

The process of functional modelling can be visualized in the following steps

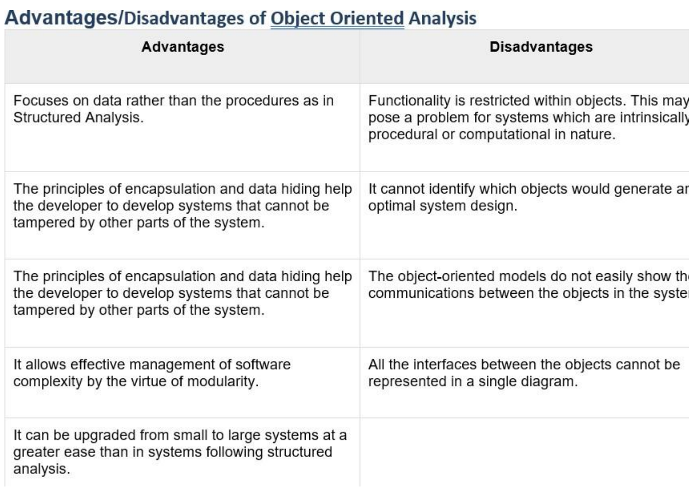
- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

Structured Analysis vs. Object Oriented Analysis

The Structured Analysis/Structured Design (SASD) approach is the traditional approach of software development based upon the waterfall model. The phases of development of a system using SASD are

- Feasibility Study
- Requirement Analysis and Specification
- System Design
- Implementation
- Post-implementation Review

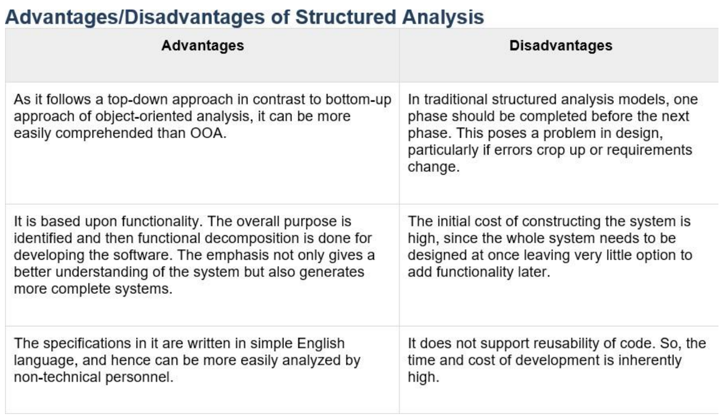
Now, we will look at the relative advantages and disadvantages of structured analysis approach and object-oriented analysis approach



Advantages/Disadvantages of Object Oriented Analysis

Advantages	Disadvantages
Focuses on data rather than the procedures as in Structured Analysis.	Functionality is restricted within objects. This may pose a problem for systems which are intrinsically procedural or computational in nature.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	It cannot identify which objects would generate ar optimal system design.
The principles of encapsulation and data hiding help the developer to develop systems that cannot be tampered by other parts of the system.	The object-oriented models do not easily show th communications between the objects in the syste
It allows effective management of software complexity by the virtue of modularity.	All the interfaces between the objects cannot be represented in a single diagram.
It can be upgraded from small to large systems at a greater ease than in systems following structured analysis.	

Figure 3.4: object oriented analysis


Advantages/Disadvantages of Structured Analysis

Advantages	Disadvantages
As it follows a top-down approach in contrast to bottom-up approach of object-oriented analysis, it can be more easily comprehended than OOA.	In traditional structured analysis models, one phase should be completed before the next phase. This poses a problem in design, particularly if errors crop up or requirements change.
It is based upon functionality. The overall purpose is identified and then functional decomposition is done for developing the software. The emphasis not only gives a better understanding of the system but also generates more complete systems.	The initial cost of constructing the system is high, since the whole system needs to be designed at once leaving very little option to add functionality later.
The specifications in it are written in simple English language, and hence can be more easily analyzed by non-technical personnel.	It does not support reusability of code. So, the time and cost of development is inherently high.

Figure 3.5: structured analysis

Domain Analysis

In the discussion of requirements engineering (Chapter 5), I noted that analysis patterns often reoccur across many applications within a specific business domain. If these patterns are defined and categorized in a manner that allows you to recognize and apply them to solve common problems, the creation of the analysis model is expedited. More important, the likelihood of applying design patterns and executable software components grows dramatically. This improves time-to-market and reduces development costs But

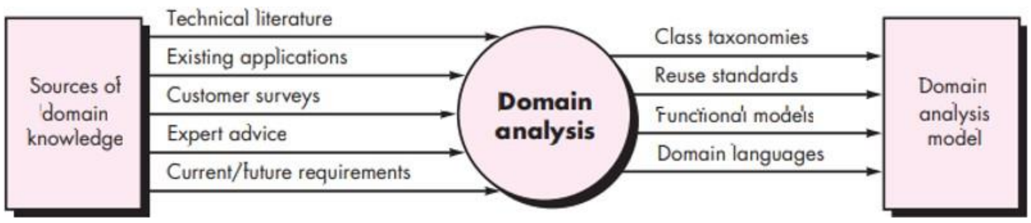


Figure 3.6: domain analysis

how are analysis patterns and classes recognized in the first place? Who defines them, categorizes them, and readies them for use on subsequent projects? The answers to these questions lie in domain analysis. Firesmith [Fir93] describes domain analysis in the following way: Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain. . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks

Object Relationship Model

An Object relational model is a combination of a Object oriented database model and a Relational database model. So, it supports objects, classes, inheritance etc. just like Object Oriented models and has support for data types, tabular structures etc. like Relational data model

One of the major goals of Object relational data model is to close the gap between relational databases and the object oriented practises frequently used in many programming languages such as C++, C#, Java etc

History of Object Relational Data Model

Both Relational data models and Object oriented data models are very useful. But it was felt that they both were lacking in some characteristics and so work was started to build a model that was a combination of them both. Hence, Object relational data model was created as a result of research that was carried out in the 1990's

Advantages of Object Relational model

The advantages of the Object Relational model are

Inheritance

The Object Relational data model allows its users to inherit objects, tables etc. so that they can extend their functionality. Inherited objects contains new attributes as well as the attributes that were inherited

Complex Data Types

Complex data types can be formed using existing data types. This is useful in Object relational data model as complex data types allow better manipulation of the data

Extensibility

The functionality of the system can be extended in Object relational data model. This can be achieved using complex data types as well as advanced concepts of object oriented model such as inheritance

Disadvantages of Object Relational model

The object relational data model can get quite complicated and difficult to handle at times as it is a combination of the Object oriented data model and Relational data model and utilizes the functionalities of both of them

object behaviour model

The modeling notation that I have discussed to this point represents static elements of the requirements model. It is now time to make a transition to the dynamic behavior of the system or product. To accomplish this, you can represent the behavior of the system as a function of specific events and time. The behavioral model indicates how software will respond to external events or stimuli

To create the model, you should perform the following steps

1. Evaluate all use cases to fully understand the sequence of interaction within the system
2. Identify events that drive the interaction sequence and understand how these events relate to specific objects
3. Create a sequence for each use case
4. Build a state diagram for the system
5. Review the behavioral model to verify accuracy and consistency. Each of these steps is discussed in the sections that follow

1 Identifying Events with the Use Case In Chapter 6 you learned that the use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. In Section 7.2.3, I indicated that an event is not the information that has been exchanged, but rather the fact that information has been exchanged. A use case is examined for points of information exchange

To illustrate, we reconsider the use case for a portion of the SafeHome security function. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed. As an example of a typical event, consider the underlined use case phrase “homeowner uses the keypad to key in a four-digit password.” In the context of the requirements model, the object, Homeowner,7 transmits an event to the object ControlPanel. The event might be called password entered. The information State Representations In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.8 The state of a class takes on both passive and active characteristics [Cha93]. A passive state is simply the current status of all of an object’s attributes. For example, the passive state of the class Player (in the video game application discussed in Chapter 6) would include the current position and orientation attributes of Player as well as other features of Player that are relevant to the game (e.g., an attribute that indicates magic wishes remaining)

State diagrams for analysis classes. One component of a behavioral model is a UML state diagram9 that represents active states for each class and the events (triggers) that cause changes between these active states. Figure 7.6 illustrates a state diagram for the ControlPanel object in the SafeHome security function. Each arrow shown in Figure 7.6 represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that triggers the transition. Although the active state

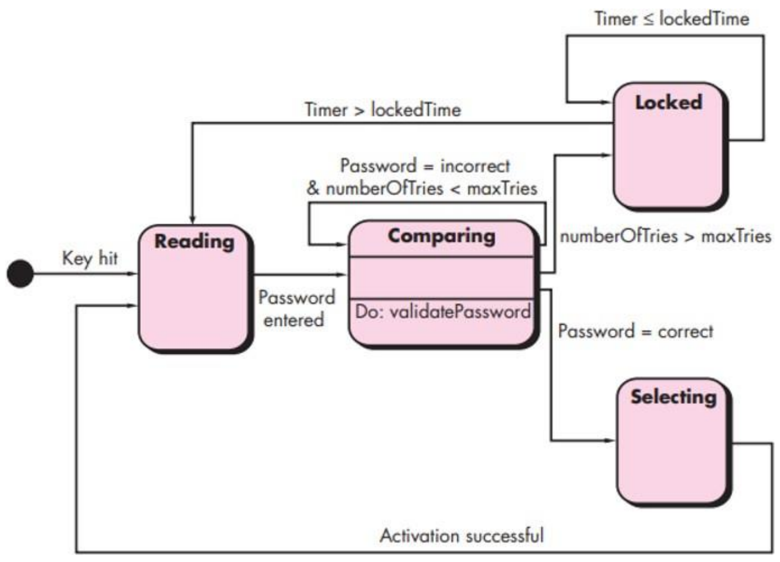


Figure 3.7: object relationship

model provides useful insight into the “life history” of an object, it is possible to specify additional information to provide more depth in understanding the behavior of an object. In addition to specifying the event that causes the transition to occur, you can specify a guard and an action [Cha93]. A guard is a Boolean condition that must be satisfied in order for the transition to occur. For example, the guard for the transition from the “reading” state to the “comparing” state in Figure 7.6 can be determined by examining the use case: if (password input 4 digits) then compare to stored password
Sequence diagrams. The second type of behavioral representation, called a sequence diagram in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modele

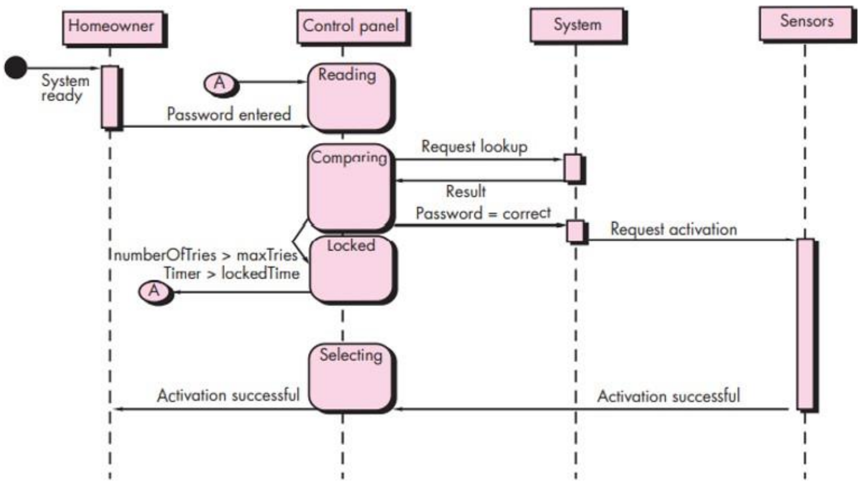


Figure 3.8: class

Design modeling with UML

Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered. In some cases, the domain knowledge is applied to a new problem within the same application domain. In other cases, the domain knowledge captured by a pattern can be applied by analogy to a completely different application domain
Discovering Analysis Patterns The requirements model is comprised of a wide variety of

elements: scenario-based (use cases), data-oriented (the data model), class-based, flow-oriented, and behavioral. Each of these elements examines the problem from a different perspective, and each provides an opportunity to discover patterns that may occur throughout an application domain, or by analogy, across different application domains A Requirements Pattern Example: Actuator-Sensor12 One of the requirements of the SafeHome security function is the ability to monitory security sensors (e.g., break-in sensors, fire, smoke or CO sensors, water sensors)

Constraints

• Each passive sensor must have some method to read sensor input and attributes that represent the sensor value

• Each active sensor must have capabilities to broadcast update messages when its value changes

•Each active sensor should send a life tick, a status message issued within a specified time frame, to detect malfunctions

• Each actuator must have some method to invoke the appropriate response determined by the ComputingComponent

• Each sensor and actuator should have a function implemented to check its own operation state

• Each sensor and actuator should be able to test the validity of the values received or sent and set its operation state if the values are outside of the specifications. Applicability. Useful in any system in which multiple sensors and actuators are present. Structure. A UML class diagram for the Actuator-Sensor pattern is shown in Figure 7.8. Actuator, PassiveSensor, and ActiveSensor are abstract classes and denoted in italics. There are four different types of sensors and ac

Pattern Name. Actuator-Sensor

Intent. Specify various kinds of sensors and actuators in an embedded system. Motivation. Embedded systems usually have various kinds of sensors and actuators. These sensors and actuators are all either directly or indirectly connected to a control unit. Although many of the sensors and actuators look quite different, their behavior is similar enough to structure them into a pattern. The pattern shows how to specify the sensors and actuators for a system, including attributes and operations. The Actuator-Sensor pattern uses a pull mechanism (explicit request for information) for PassiveSensors and a push mechanism (broadcast of information) for the ActiveSensors

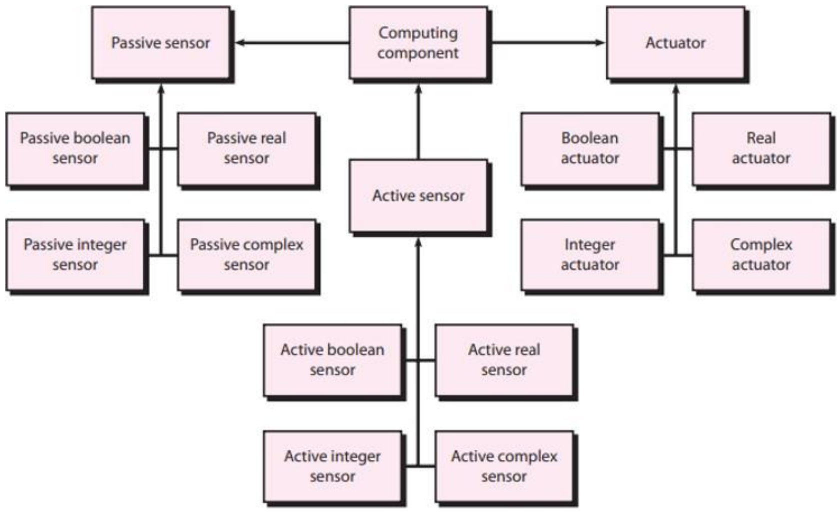


Figure 3.9: uml

Chapter 4

DESIGN

Course Outcomes

After successful completion of this module, students should be able to:

CO 8	Understand the importance and need of Design and explain the various approaches of Designs.	Analyze
CO 9	Demonstrate the need of software Architecture, data design, mapping techniques and explain system design and object design process.	Under-stand

Software design is an iterative process through which requirements are translated into a blueprint‖ for constructing the software. Initially, the blue print depicts a holistic view of software. That is, the design is represented at a high level of abstraction— a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.

Design process

Software design is an iterative process through which requirements are translated into a blueprint‖ for constructing the software. Initially, the blue print depicts a holistic view of software. That is, the design is represented at a high level of abstraction— a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements

Software Quality Guidelines and Attributes

Design Guidelines

- A good design should
- exhibit good architectural structure
- 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 design 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 reputable method that is driven by information obtained during software requirements analysis

Quality Guidelines

A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be
implemented in an evolutionary fashion, thereby facilitating implementation and testing

- A design should be modular; that is, the software should be logically partitioned into elements or subsystems
- A design should contain distinct representations of data, architecture, interfaces, and components
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns
- A design should lead to components that exhibit independent functional characteristics
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis

- A design should be represented using a notation that effectively communicates its meaning

Quality attributes

- Functionality is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are derived and the security of the overall system
- Usability is assessed by considering human factors, overall aesthetics, consistency and documentation
- Reliability is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure, the ability to recover form failure, and the predictability of the program
- Performance is measured by processing speed, response time, resource consumption, throughput, and efficiency
- Supportability combines the ability to extend the program, adaptability, serviceability, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized

The Evolution of Software Design

- The evolution of software design is a continuing process that has now spanned almost six decades
- All these methods have a number of common characteristics

1. A mechanism for the translation of the requirements model into a design representation
2. A notation for representing functional components and their interfaces
3. Heuristics for refinement and partitioning, and
4. Guidelines for quality assessment

Regardless of the design method that is used, you should apply a set of basic concepts to data, architectural, interface, and component-level design. These concepts are considered in the sections that follow

Design Concepts

Abstraction

– Abstraction is the process by which data and programs are defined with a representation

similar in form to its meaning (semantics), while hiding away the implementation details
– Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time
– At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At the lower levels of abstraction, a more detailed description of the solution is provided
Abstraction can be
- Data abstraction is a named collection of data that describes a data object. Data abstraction for door' would encompass a set of attributes that describe the door (e.g. door type, swing direction, opening mechanism, weight, dimensions)
- The procedural abstraction open' would make use of information contained in the attributes of the data abstraction door'

Basic Design Principles
- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Dependency inversion Principle
- Interface segregation Principle
- Procedural abstraction refers to a sequence of instructions that have as specific and

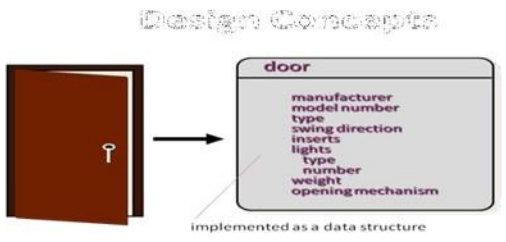


Figure 4.1: procedural abstraction

limited function. The name of the procedural abstraction implies these functions, but specific details are suppressed
e.g. open' for a door. open' implies a long sequence of procedural steps (e.g. walk to the door, reach out and grasp knob, turn knob, turn knob and pull door, step away from moving door, etc) Architecture
– Architecture is the structure or organization of program components (modules), the

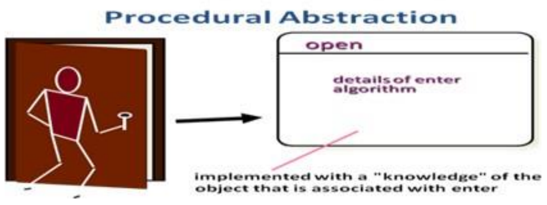

Procedural Abstraction
open
details of enter algorithm
implemented with a "knowledge" of the object that is associated with enter

Figure 4.2: object association

manner in which these components interact, and the structure of data that are used by the components. Architectural design can be represented using

– Structural models represent architecture as an organized collection of program components

– Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks that are encountered in similar types of application

– Dynamic models address the behavioural aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events

– Procedural models focus on the design of the business or technical process that the system must accommodate

– Function models can be used to represent the functional hierarchy of a system

Patterns

– Design pattern describes a design structure that solves a particular design problem within a specific context

– Each design pattern is to provide a description that enables a designer to determine

- Whether the pattern is applicable to the current work
- Whether the pattern can be reused
- Whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern

Separation of Concerns

- Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- A concern is a feature or behavior that is specified as part of the requirements model for the software

- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve
- For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2

modular design

– Software is divided into separately named and addressable components, called modules that are integrated to satisfy problem requirements
– Design has to be modularized so that development can be more easily planned, software increments can be defined and delivered, changes can be more easily accommodated, testing and debugging can be conducted more efficiently and long- term maintenance can be conducted without serious side effects
Information Hiding

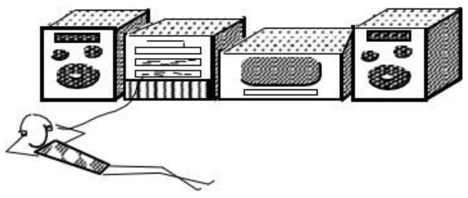


Figure 4.3: modular

–• Reduces the likelihood of side effects
- Limits the global impact of local design decisions
- Emphasizes communication through controlled interfaces
- Discourages the use of global data
- Leads to encapsulation—an attribute of high quality design
- Results in higher quality software

Functional Independence
– Functional independence is achieved by developing modules with single minded‘ function and avoids excessive interaction with other modules
– Independent modules are easier to maintain because secondary effects caused by design

or code modification are limited, error propagation is reduced Independence is assessed using two qualitative criteria

- Cohesion which is an indication of the relative functional strength of a module. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program
- Coupling is an indication of the relative interdependence among modules. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface

Refinement

– Refinement is a process of elaboration. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement occurs

– Refinement helps the designer to reveal low-level details as design progresses

Refactoring

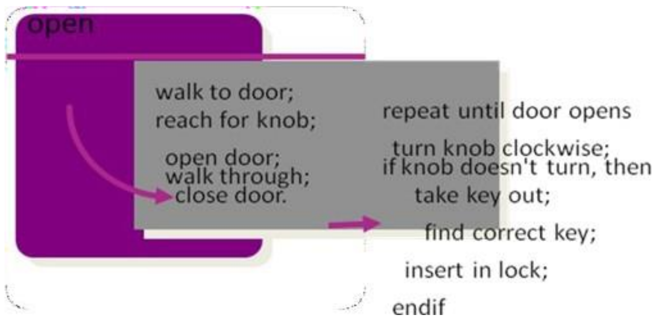


Figure 4.4: refinement

– Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure

– When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design

Aspects

- As requirements analysis occurs, a set of concerns‖ is uncovered. These concerns include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts

- Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently
- In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized. As design begins, requirements are refined into a modular design representation
- Consider two requirements, A and B. Requirement A crosscuts requirement B if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account

Refactoring

- An important design activity suggested for many agile methods, refactoring is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior
- When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design

Object-Oriented Design Concepts

- The object-oriented (OO) paradigm is widely used in modern software engineering
- OO design concepts such as classes and objects, inheritance, messages, and polymorphism, among others

Design classes

– As the design model evolves, a set of design classes are to be defined that

- Refine the analysis classes by providing design detail that will enable the classes to be implemented
- Create a new set of design classes that implement a software infrastructure to support the business solution

The Design Model

Design model can be viewed as

– Process dimension indicating the evolution of the design model as design tasks are executed as part of the software process

– Abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively

Design Model Elements are as follows

- Data design elements
- Architectural design elements
- Interface design elements
- Component-level design elements
- Deployment-level design elements

Data design elements

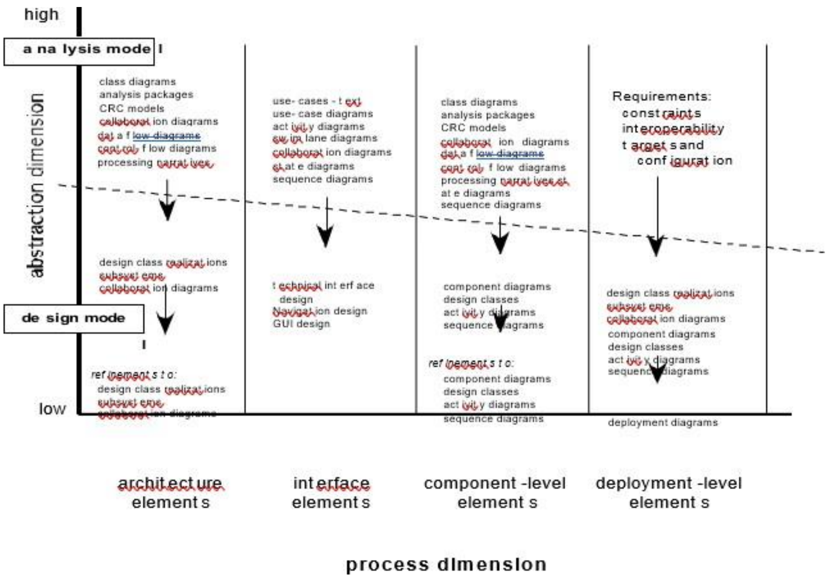


Figure 4.5: process dimension

– Data design creates a model of data and/or information that is represented at a high level of abstraction

– Data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system Architectural level databases and files Component level data structures

Architectural design elements

- The architectural design for software is the equivalent to the floor plan of a house. The floor plan depicts the overall layout of the rooms; their size, shape, and relationship to one another; and the doors and windows that allow movement into and out of the rooms. The floor plan gives us an overall view of the house. Architectural design elements give us an overall view of the software

– The architectural model is derived from

• Information about the application domain for the software to be built

• Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand

• The availability of architectural patterns and styles

Interface design elements

– The interface design elements for software tell how information flows into and out of the system and how it is communicated among the components defined as part of the architecture

– Important elements of interface design • The user interface (UI): Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components). In general, the UI is a unique subsystem within the overall application architecture

• External interfaces to other systems, devices, networks or other producers or consumers of informationThe design of external interfaces requires definitive information about the entity to which information is sent or received

• Internal interfaces between various design componentsThe design of internal interfaces is closely aligned with component-level design

Component-level design elements

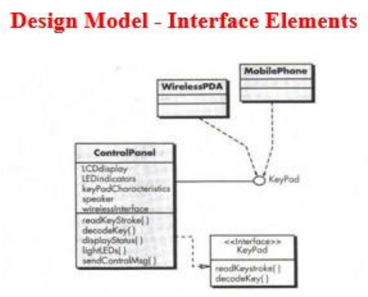


Figure 4.6: design model

• The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. These drawings depict wiring and plumbing within each room, the location of electrical receptacles and wall switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. • Component-level design for software fully describes the internal detail of each software component

• Component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations

• The design details of a component can be modelled at many different levels of abstraction

• An UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode or diagrammatic form (e.g., flowchart or box diagram

Deployment-level design elements

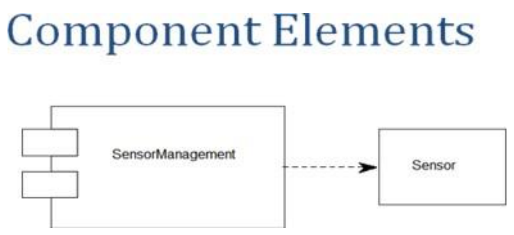


Figure 4.7: component

– Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software

– Deployment diagrams shows the computing environment but does not explicitly indicate configuration details

introduction to software architecture

• When you consider the architecture of a building, many different attributes come to mind. At the most simplistic level, you think about the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole

• The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them

• Software architecture enables to

– Analyze the effectiveness of the design in meeting its stated requirements

– Consider architectural alternatives at a stage when making design changes is still relatively easy

– Reduce the risks associated with the construction of the software

– Architectural design represents the structure of data and program components that are required to build a computer-based system

– It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships
that occur among all architectural components of a system

– Architecture considers two levels of the design – data design and architectural design. Data design enables us to represent the data component of the architecture

– Architectural design focuses on the representation of the structure of software components, their properties, and interactions

Why Is Architecture Important

• Representations of software architecture are an enabler for communication between all parties, interested in the development of a computer-based system

• The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on ultimate success of the system as an operational entity

• Architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together

Architectural Descriptions

• Each of us has a mental image of what the word architecture means. In reality, however, it means different things to different people

• The implication is that different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns

• An architectural description is actually a set of work products that reflect different views of the system

• An architectural description of a software-based system must exhibit characteristics that are analogous to those noted for the office building

• Developers want clear, decisive guidance on how to proceed with design

• Customers want a clear understanding on the environmental changes that must occur and assurances that the architecture will meet their business needs

• Other architects want a clear, salient understanding of the architecture‘s key aspects.‖ Each of these wants‖ is reflected in a different view represented using a different viewpoint

• Each view developed as part of an architectural description addresses a specific stakeholder concern

• To develop each view (and the architectural description as a whole) the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern

• Therefore, architectural decisions themselves can be considered to be one view of the architecture

• The reasons that decisions were made provide insight into the structure of a system and its conformance to stakeholder concerns

Software Architectural Styles

1. A Brief Taxonomy of Architectural Styles
2. Architectural Patterns
3. Organization and Refinement

• The software that is built for computer-based systems exhibit one of many architectural styles

• Each style describes a system category that encompasses

– A set of component types that perform a function required by the system

– A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components

– constraints that define how components can be integrated to form the system

– semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts

A Brief Taxonomy of Architectural Styles

• Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add,

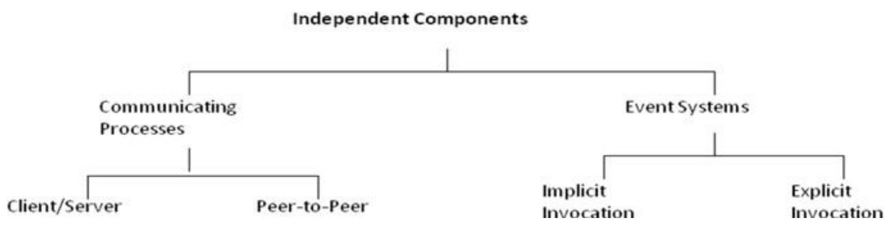


Figure 4.8: independent

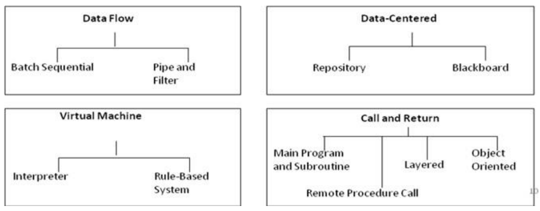


Figure 4.9: architectural

delete, or otherwise modify data within the store

• Illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a blackboard

Data-Centered Style

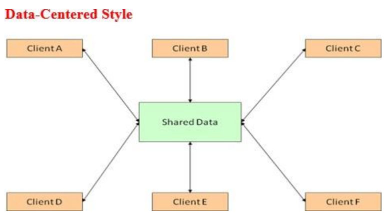


Figure 4.10: centered style

Data-flow architectures

- This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data
- A pipe-and-filter pattern show has a set of components, called filters, connected by pipes that transmit data from one component to the next
- Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form
- However, the filter does not require knowledge of the workings of its neighboring filters

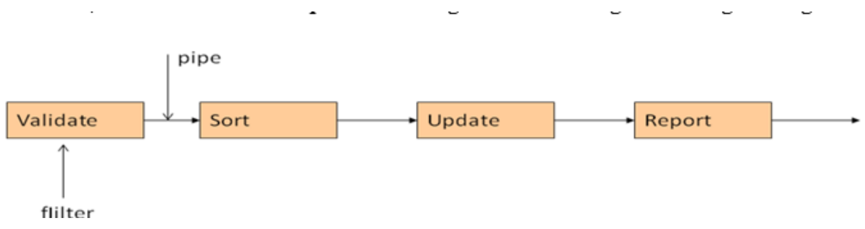


Figure 4.11: data flow

Call and return architectures

- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale
- A number of substyles exist within this category
- Main program/subprogram architectures: This classic program structure decomposes function into a control hierarchy where a main‖ program invokes a number of program components that in turn may invoke still other components. Figure illustrates architecture of this type
- Remote procedure call architectures: The components of a main program/subprogram architecture are distributed across multiple computers on a network

Object-oriented architectures

- The components of a system encapsulate data and the operations that must be applied to manipulate the data
- Communication and coordination between components are accomplished via message passing

Layered architectures

- The basic structure of a layered architecture is illustrated in Figure. • A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set
- At the outer layer, components service user interface operations
- At the inner layer, components perform operating system interfacing
- Intermediate layers provide utility services and application software functions
- As the requirements model is developed, you‘ll notice that the software must address

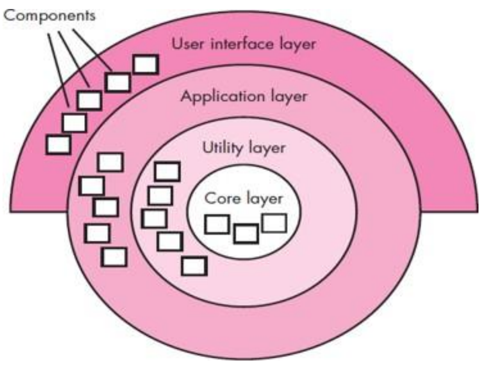


Figure 4.12: layered architecture

a number of broad problems that span the entire application

- For example, the requirements model for virtually every e-commerce application is faced with the following problem: How do we offer a broad array of goods to a broad array of customers and allow those customers to purchase our goods online
- Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design

Organization and Refinement

- Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architec-tural design that is derived

Control • How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized or do components operate asynchronously

Data • How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system

Architectural design

• As architectural design begins, the software to be developed must be put into context—that is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction

1. Represent the system in context
2. Define archetypes
3. Refine the architecture into components
4. Describe instantiations of the system

1. Represent the System in Context

• Use an architectural context diagram (ACD) that shows

– The identification and flow of all information into and out of a system

– The specification of all interfaces

– Any relevant support processing from/by other systems

• An ACD models the manner in which software interacts with entities external to its boundaries

• An ACD identifies systems that interoperate with the target system

– Super-ordinate systems

• Use target system as part of some higher level processing scheme

– Sub-ordinate systems

• those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality

- Peer-level systems

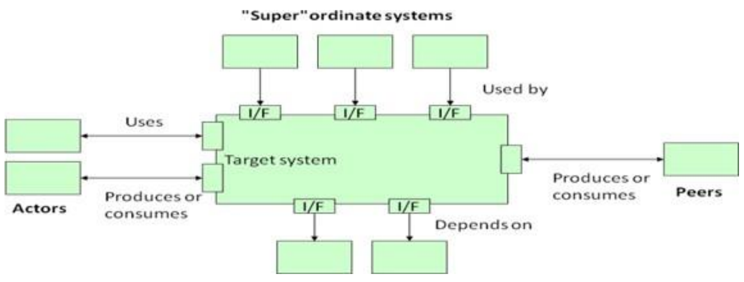


Figure 4.13: super ordinate system

- Interact on a peer-to-peer basis with target system to produced or consumed by peers and target system

– Actors

- People or devices that interact with target system to produce or consume data

2. Define Archetypes

- Archetypes indicate the important abstractions within the problem domain (i.e., they model information)
- An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system
- Only a relatively small set of archetypes is required in order to design even relatively complex systems
- The target system architecture is composed of these archetypes

– They represent stable elements of the architecture

– They may be instantiated in different ways based on the behavior of the system

– They can be derived from the analysis class model

- The archetypes and their relationships can be illustrated in a UML class diagram

Archetypes in Software Architecture

- Node - Represents a cohesive collection of input and output elements of the home security function
- Detector/Sensor - An abstraction that encompasses all sensing equipment that feeds information into the target system
- Indicator - An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring

• Controller - An abstraction that depicts the mechanism that allows the arming or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another

Archetypes – their attributes 3. Refine the Architecture into Components

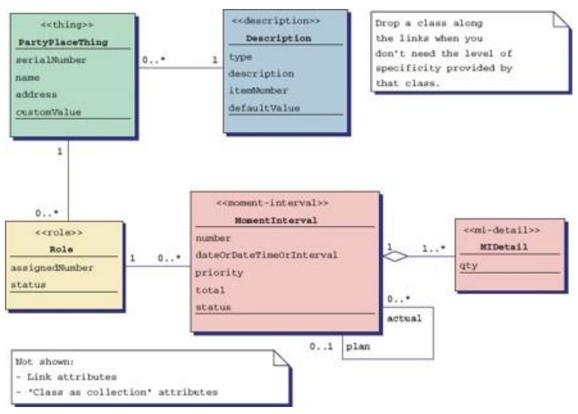


Figure 4.14: attributes

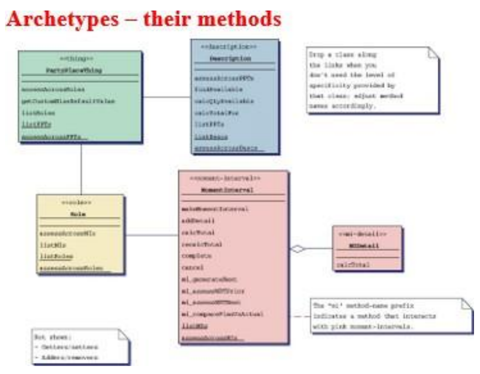


Figure 4.15: methods

• Based on the archetypes, the architectural designer refines the software architecture into components to illustrate the overall structure and architectural style of the system

• These components are derived from various sources

– The application domain provides application components, which are the domain classes in the analysis model that represent entities in the real world

– The infrastructure domain provides design components (i.e., design classes) that enable application components but have no business connection • Examples: memory management, communication, database, and task management

• These components are derived from various sources

– The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface

Refine the Architecture into Components

• Based on the archetypes, the architectural designer refines the software architecture into components to illustrate the overall structure and architectural style of the system

• These components are derived from various sources

– The application domain provides application components, which are the domain classes in the analysis model that represent entities in the real world

– The infrastructure domain provides design components (i.e., design classes) that enable application components but have no business connection

• Examples: memory management, communication, database, and task management

• These components are derived from various sources

– The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface

4. Describe Instantiations of the System

• The architectural design that has been modeled to this point is still relatively high level

• The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified

• However, further refinement (recall that all design is iterative) is still necessary

Assessing alternative architectural design

• Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved

1. An Architecture Trade-Off Analysis Method

The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively

• Collect scenarios. A set of use cases is developed to represent the system from the user‘s point of view

• Elicit requirements, constraints, and environment description. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed

- Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements
- Evaluate quality attributes by considering each attribute in isolation
- Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style
- Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted

2. Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system

3. Architectural Description Languages

- Architectural description language (ADL) provides a semantics and syntax for describing software architecture
- ADL should provide the designer with the ability to decompose architectural components, compose individual components into larger architectural blocks, and represent interfaces (connection mechanisms) between components
- Once descriptive, language based techniques for architectural design has been established; it is more likely that effective assessment methods for architectures will be established as the design evolves

Architectural Mapping using Data Flow

- Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style

– Information must enter and exit software in an external world‖. Such externalized data must be converted into an internal form for processing

Information enters along paths that transform external data into an internal form. These paths are identified are Incoming flow

– Incoming data are transformed through a transform center and move along the paths that now lead out‖ of the software. Data moving along these paths are called Outgoing flow

- Transaction Flow

– Information flow is often characterized by a single data item, called transaction, that triggers other data flow along one of many paths

– Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction

– The transaction is evaluated and, based on its value, flow along one of many action paths is initiated. The hub of information from which many action paths emanate is called a transaction center

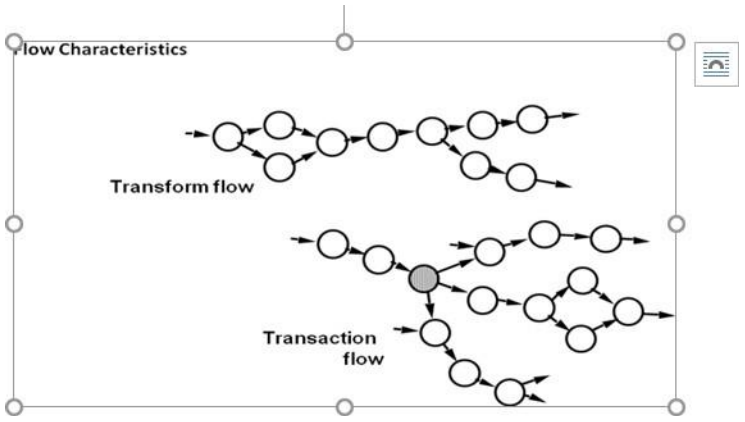


Figure 4.16: flow characteristics

Transform Mapping

1. Review the fundamental system model
2. Review and refine data flow diagrams for the software
3. Determine whether the DFD has transform or transaction flow characteristics
4. Isolate the transform center by specifying incoming and outgoing flow boundaries
5. Perform first-level factoring
6. Perform second-level factoring
7. Refine the first-iteration architecture using design heuristics for improved software quality

Transform Mapping

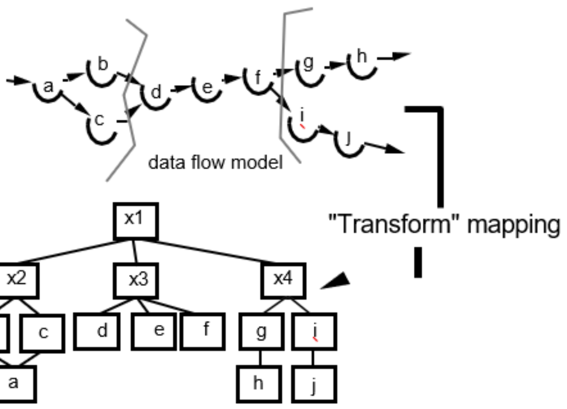


Figure 4.17: transfor mapping

Factoring

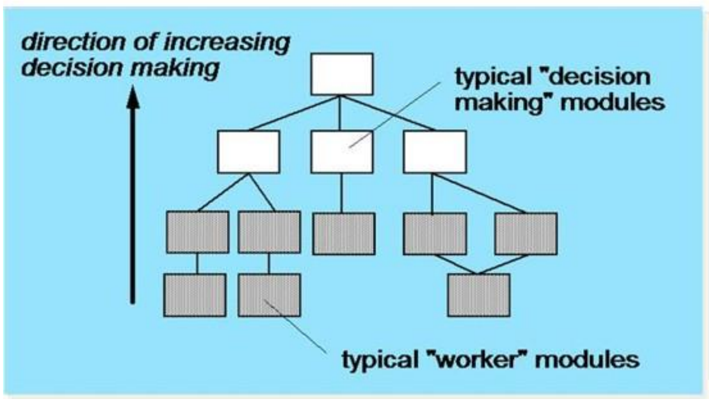
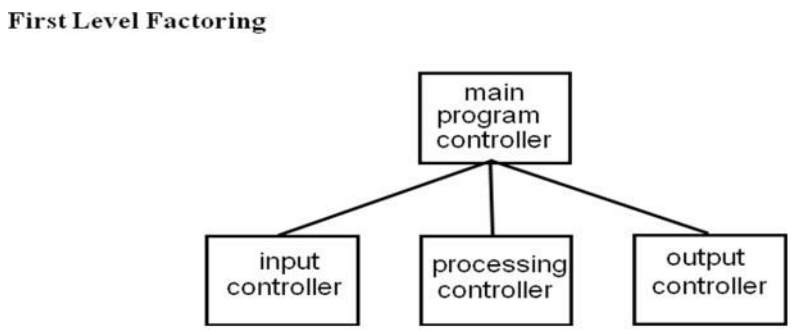

direction of increasing decision making
typical "decision making" modules
typical "worker" modules

Figure 4.18: factoring

Transaction Mapping

1. Review the fundamental system model
2. Review and refine data flow diagrams for the software
3. Determine whether the DFD has transform or transaction flow characteristics
4. Isolate the transaction center and the flow characteristics along each of the action paths
5. Map the DFD in a program structure amenable to transaction processing



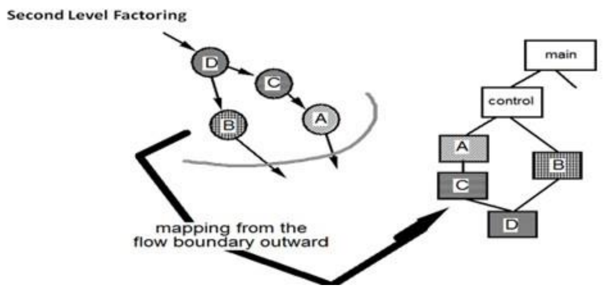


Figure 4.19: first level factoring

6. Factor and refine the transaction structure and the structure of each action path

7. Refine the first-iteration architecture using design heuristics for improved software quality

Design Patterns

- Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged
- A design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem
- A vast array of interface design patterns has been proposed over the past decade

3. Design Issues

As the design of a user interface evolves, four common design issues almost always surface

1. System response time
2. User help facilities

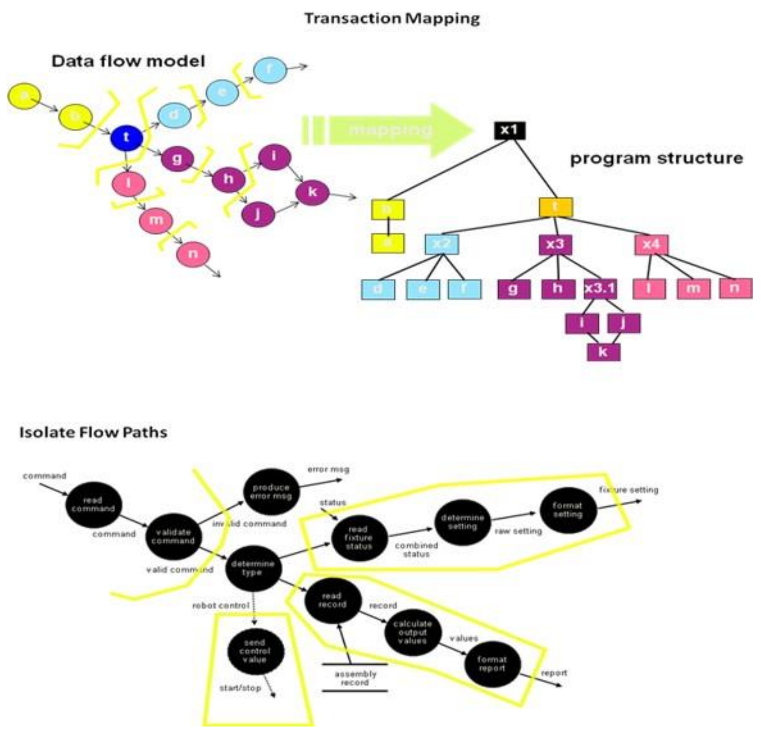


Figure 4.20: transaction mapping

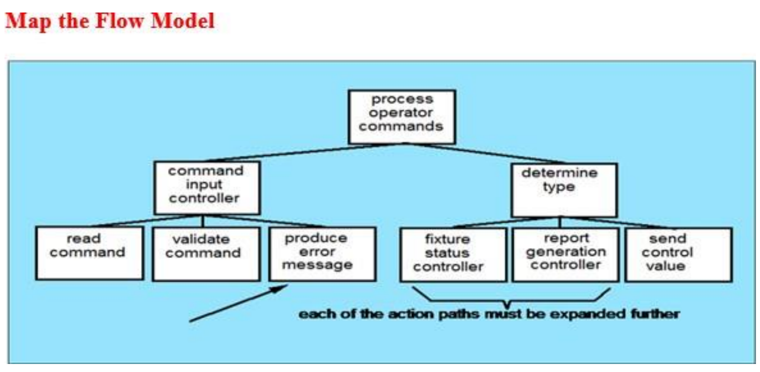

Map the Flow Model
process operator commands
command input controller
determine type
read command
validate command
produce error message
fixture status controller
report generation controller
send control value
each of the action paths must be expanded further

Figure 4.21: map the flow model

3. Error information handling

4. Command labeling

Designing class based components

Designing Class Based Components

1 Basic Design Principles

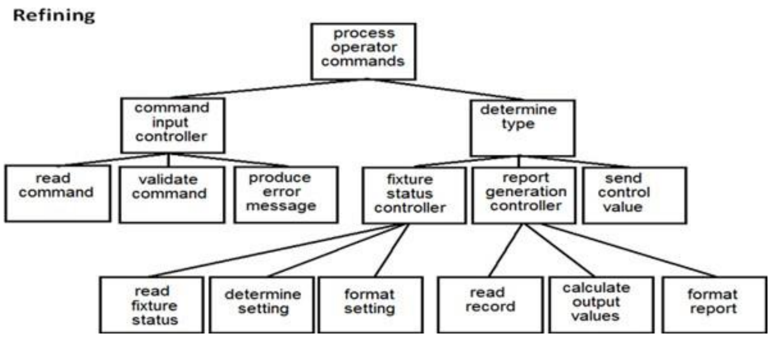


Figure 4.22: refining

2. Component-Level Design Guidelines
3. Cohesion
4. Coupling

Component-level design focuses on the elaboration of analysis classes (problem domain specific classes) and definition and refinement of infrastructure classes Purpose of using design principles is to create designs that are more amenable to change and to reduce propagation of side effects when changes do occur 1. Basic Design Principles

- Single Responsibility Principle
- Open-Closed Principle
- Liskov Substitution Principle
- Dependency inversion Principle
- Interface segregation Principle

2. Component-Level Design Guidelines

In addition to the principles discussed, a set of pragmatic design guidelines can be applied as component-level design proceeds. These guidelines apply to components, their interfaces, and the dependencies and inheritance and inheritance characteristics that have an impact on the resultant design

- Components: Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
- Interfaces: Interfaces provide important information about communication and collaboration

- Dependencies and Inheritance: For improved readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes)

3. Cohesion

- The single-mindedness‖ of a component
- Cohesion implies that a single component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- Types of cohesion

o Functional

o Layer

o Communicational

4. Coupling

- Coupling or Dependency is the degree to which each program module relies on each one of the other modules
- Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa
- Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability

Different categories of coupling

- Content coupling. Occurs when one component surreptitiously modifies data that is internal to another component‖ [Let01]. This violates information hiding—a basic design concept
- Common coupling. Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made
- Control coupling. Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then directs‖ logical flow within B. The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes. If this is overlooked, an error will result. Stamp coupling. Occurs when ClassB is declared as a type for an argument of an operation of

ClassA. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex

- Data coupling. Occurs when operations pass long strings of data arguments. The bandwidth‖ of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult
- Routine call coupling. Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system
- Type use coupling. Occurs when component A uses a data type defined in component B (e.g., this occurs whenever a class declares an instance variable or a local variable as having another class for its type‖ [Let01]). If the type definition changes, every component that uses the definition must also change
- Inclusion or import coupling. Occurs when component A imports or includes a package or the content of component B
- External coupling. Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system

Traditional components

Designing Traditional Components

- Graphical Design Notation
- Tabular Design Notation
- Program Design Language

Conventional design constructs emphasize the maintainability of a functional/procedural program

- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Various notations depict the use of these constructs

– Graphical design notation

- Sequence, if-then-else, selection, repetition

– Tabular design notation

– Program design language

1. Graphical Design Notation

- ‖A picture is worth a thousand words,‖ but it‘s rather important to know which picture and which 1000 words
- There is no question that graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail
- However, if graphical tools are misused, the wrong picture may lead to the wrong software
- The activity diagram allows you to represent sequence, condition, and repetition
- all elements of structured programming—and is a descendent of an earlier pictorial design representation (still used widely) called a flowchart
- A flowchart, like an activity diagram, is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of

control. The sequence is represented as two processing boxes connected by a line (arrow) of control

2. Tabular Design Notation

The following steps are applied to develop a decision table

1. List all actions that can be associated with a specific procedure (or module)
2. List all conditions (or decisions made) during execution of the procedure
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
4. Define rules by indicating what action(s) occurs for a set of conditions

3. Program Design Language

- Program design language (PDL), also called structured English or pseudocode, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English)
- Narrative text (e.g., English) is embedded within a programming language-like syntax. Automated tools (e.g., [Cai03]) can be used to enhance the application of PDL
- A basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and input-output (I/O) constructs

- PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, interprocess synchronization, and many other features

Chapter 5

CO5	Apply design concepts and principles to develop components of a system	Apply
CO6	Distinguish the approaches to verification and validation of system in various stages of development.	Analyze

IMPLEMENTATION, TESTING AND MAINTENANCE

5.1 Top - down,bottom-up

op-down approaches emphasize planning and a complete understanding of the system. It is inherent that no coding can begin until a sufficient level of detail has been reached in the design of at least some part of the system. Top-down approaches are implemented by attaching the stubs in place of the module. This, however, delays testing of the ultimate functional units of a system until significant design is complete

Bottom-up emphasizes coding and early testing, which can begin as soon as the first module has been specified. This approach, however, runs the risk that modules may be coded without having a clear idea of how they link to other parts of the system, and that such linking may not be as easy as first thought. Re-usability of code is one of the main benefits of the bottom-up approach

Top-down design was promoted in the 1970s by IBM researchers Harlan Mills and Niklaus Wirth. Mills developed structured programming concepts for practical use and tested them in a 1969 project to automate the New York Times morgue index. The engineering and management success of this project led to the spread of the top-down approach through IBM and the rest of the computer industry. Among other achievements, Niklaus Wirth,

the developer of Pascal programming language, wrote the influential paper Program Development by Stepwise Refinement. Since Niklaus Wirth went on to develop languages such as Modula and Oberon (where one could define a module before knowing about the entire program specification), one can infer that top-down programming was not strictly what he promoted. Top-down methods were favored in software engineering until the late 1980s,[3] and object-oriented programming assisted in demonstrating the idea that both aspects of top-down and bottom-up programming could be utilized

Modern software design approaches usually combine both top-down and bottom-up approaches. Although an understanding of the complete system is usually considered necessary for good design, leading theoretically to a top-down approach, most software projects attempt to make use of existing code to some degree. Pre-existing modules give designs a bottom-up flavor. Some design approaches also use an approach where a partially functional system is designed and coded to completion, and this system is then expanded to fulfill all the requirements for the project

object oriented product implementation and integration

an object-oriented approach applying an integrated database for production systems. The basis for integration is an object-oriented Product/ Production-Model (PPM) which has been developed at an interdisciplinary research center of the University of Karlsruhe.

Based on this object-oriented PPM, new ways of organizing engineering processes become possible. By using predefined methods of the database objects, the development of modular software tools is enabled. The common database ensures redundancy-free data management, uniform data structures for all relevant views and avoids repeated input of basic data. As an application for the described methods, the paper discusses a new planning

Software Testing methods

Software Testing Fundamentals

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. Therefore, you should design and implement a computer based system or a product with

testability‖ in mind. At the same time, the tests themselves must exhibit a set of characteristics that achieve the goal of finding the most errors with a minimum of effort

Testability. James provides the following definition for testability: Software testability is simply how easily [a computer program] can be tested.‖ The following characteristics lead to testable software

Operability

The better it works, the more efficiently it can be tested.‖ If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts

Observability

What you see is what you test.‖ Inputs provided as part of testing produce distinct outputs. System states and variables are visible or queriable during execution. Incorrect output is easily identified. Internal errors are automatically detected and reported. Source code is accessible

Controllability

The better we can control the software, the more the testing can be automated and optimized.‖ All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured. All code is executable through some combination of input. Software and hardware states and variables can be controlled directly by the test engineer. Tests can be conveniently specified, automated, and reproduced

Decomposability

By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.‖ The software system is built from independent modules that can be tested independently

Simplicity

The less there is to test, the more quickly we can test it.‖ The program should exhibit functional simplicity (e.g., the feature set is the minimum necessary to meet requirements); structural simplicity (e.g., architecture is modularized to limit the propagation of faults), and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance). Stability.

The fewer the changes, the fewer the disruptions to testing.‖ Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests. The software recovers well from failures. Understandability

The more information we have, the smarter we will test.‖ The architectural design and the dependencies between internal, external, and shared components are well understood. Technical documentation is instantly accessible, well organized, specific and detailed, and accurate. Changes to the design are communicated to testers. What is a good Test

Test Characteristics. And what about the tests themselves? Kaner, Falk, and Nguyen [Kan93] suggest the following attributes of a good‖ test: A good test has a high probability of finding an error

To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a graphical user interface is the failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition. A good test is not redundant

Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different). A good test should be best of breed‖ [Kan93]. In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used. A good test should be neither too simple nor too complex

Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately

Internal and External Views of Testing

- Any engineered product (and most other things) can be tested in one of two ways

1 Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function

2 Knowing the internal workings of a product, tests can be conducted to ensure that all gears mesh,‖ that is, internal operations are performed according to specifications and all internal components have been adequately exercised

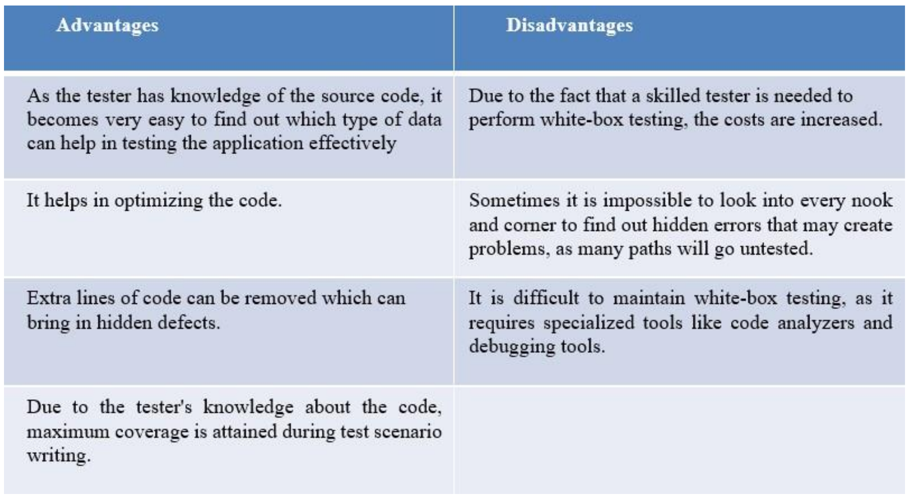
- The first test approach takes an external view and is called black-box testing. The second requires an internal view and is termed white-box testing
- Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software
- White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops
- White-box testing would lead to 100 percent correct programs.‖ need do is define all logical paths, develop test cases to exercise them, and evaluate results, i.e, generate test cases to exercise program logic exhaustively
- A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity

White Box Testing

White Box Testing

- White-box testing is the detailed investigation of internal logic and structure of the code
- White-box testing, sometimes called glass-box testing or open-box testing
- It is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases
- The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately
- Using White-box testing methods, you can derive test cases that

1. Guarantee that all independent paths within a module have been exercised at least once
2. Exercise all logical decisions on their true and false sides
3. Execute all loops at their boundaries and within their operational bounds, and
4. Exercise internal data structures to ensure their validity



Advantages	Disadvantages
As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively	Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased.
It helps in optimizing the code.	Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested.
Extra lines of code can be removed which can bring in hidden defects.	It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.
Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.	

Figure 5.1: white box testing

Basis Path Testing

- Basis path testing is a white-box testing technique
- The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing

Flow Graph Notation

Independent Program Paths

Deriving Test Cases

Graph Matrices

1. Flow Graph Notation

- A simple notation for the representation of control flow, called a flow graph (or program graph)
- The flow graph depicts logical control flow using the notation in the following figure
- Arrows called edges represent flow of control
- Circles called nodes represent one or more actions
- Areas bounded by edges and nodes called regions
- A predicate node is a node containing a condition
- Any procedural design can be translated into a flow graph
- Note that compound Boolean expressions at tests generate at least two predicate node and additional arcs
- To illustrate the use of a flow graph, consider the procedural design representation in

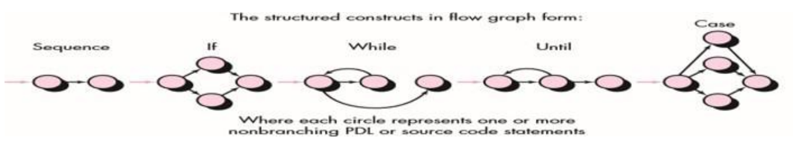

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

Figure 5.2: basis path testing

Figure

- Here, Figure (a) flow chart is used to depict program control structure
- Figure(b) maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart)
- Figure(b), each circle, called a flow graph node, represents one or more procedural statements
- A sequence of process boxes and a decision diamond can map into a single node
- The arrows on the flow graph, called edges or links, represent flow of control and are analogous to flowchart arrows
- An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the flow graph symbol for the if-then-else construct)
- Areas bounded by edges and nodes are called regions. When counting regions

2. Independent Program Paths

- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition
- Cyclomatic complexity is software metric that provides a quantitative measure of the

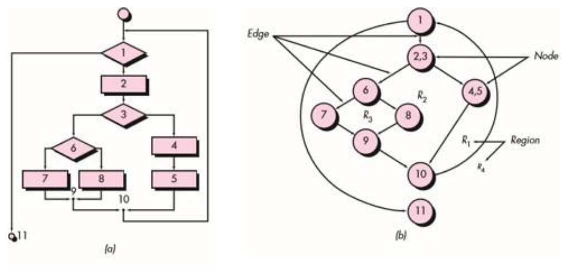


Figure 5.3: flow graph

logical complexity if a program

- When used in the context of the basis path testing method, the value computed for Cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once
- Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. Complexity is computed in one of three ways

1. The number of regions of the flow graph corresponds to the Cyclomatic complexity.
2. Cyclomatic complexity V(G) for a flow graph G is defined as V(G) =E – N + 2 where E is the number of flow graph edges and N is the number of flow graph nodes.
3. Cyclomatic complexity V(G) for a flow graph G is also defined as V(G) = P+ 1 where P is the number of predicate nodes contained in the flow graph G.

3. Deriving Test Cases

The following steps can be applied to derive the basis set: 1. Using the design or code as a foundation, draw a corresponding flow graph

2. Determine the Cyclomatic complexity of the resultant flow graph

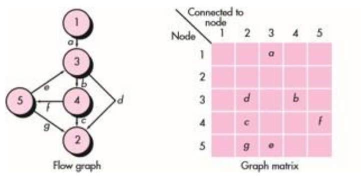
3. Determine a basis set of linearly independent paths

4. Prepare test cases that will force execution of each path in the basis set

4. Graph Matrices

- A data structure, called a graph matrix, can be quite useful for developing a software tool that assists in basis path testing

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes
- A simple example of a flow graph and its corresponding graph matrix is shown in Figure
- Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters
- A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge b
- The graph matrix is nothing more than a tabular representation of a flow graph
- By adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing
- The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist)



Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Figure 5.4: graph matrics

Control Structure Testing

- Although basis path testing is simple and highly effective, it is not sufficient in itself
- Other variations on control structure testing necessary. These broaden testing coverage and improve the quality of white-box testing

1. Condition testing

- Condition testing is a test-case design method that exercises the logical conditions contained in a program module

• A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator

• A relational expression takes the form

E1<relational-operator> E2

• Where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following

• A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses

• The condition testing method focuses on testing each condition in the program to ensure that it does not contain errors

2. Data Flow Testing

• The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program

• To illustrate the data flow testing approach, assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables

• For a statement with S as its statement number

• DEF(S)={X | statement S contains a definition of X} • USE(S) = {X | statement S contains a use of X} • If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S. The definition of variable X at statement S is said to be live at statement S‘ if there exists a path from statement S to statement S‘ that contains no other definition of X

3. Loop Testing

• Loops are the cornerstone for the vast majority of all algorithms implemented in software

• Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs

• Four different classes of loops can be defined

Simple loops

The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop

1. Skip the loop entirely

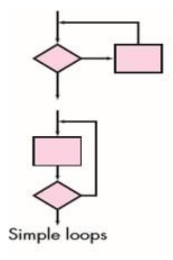


Figure 5.5: loop testing

1. Only one pass through the loop
2. Two passes through the loop
3. m passes through the loop where m < n
4. n - 1, n, n + 1 passes through the loop

2. Nested loops: If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases

- Beizer suggests an approach that will help to reduce the number of tests

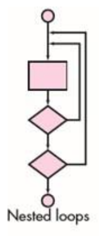


Figure 5.6: nested loops

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to typical‖ values

4. Continue until all loops have been tested.

3. Concatenated loops: In the concatenated loops, if two loops are independent of each other then they are tested using simple loops or else test them as nested loops. However if the loop counter for one loop is used as the initial value for the others, then it will not be considered as an independent loops

4. Unstructured loops: Whenever possible, this class of loops should be redesigned to

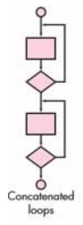


Figure 5.7: concatenated

reflect the use of the structured programming constructs.

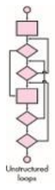


Figure 5.8: unconstructed

Black Box Testing

- Black-box testing, also called behavioral testing
- It focuses on the functional requirements of the software
- Black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program
- Black-box 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. behavior or performance errors

5. initialization and termination errors

Black – Box Testing Techniques

1. Graph-Based Testing Methods
2. Equivalence Partitioning
3. Boundary Value Analysis
4. Orthogonal Array Testing

1. Graph-Based Testing Methods

- The first step in black-box testing is to understand the objects5 that are modeled in software and the relationships that connect these objects
- Next step is to define a series of tests that verify all objects have the expected relationship to one another
- To accomplish these steps, create a graph—a collection of nodes that represent objects, links that represent the relationships between objects, node weights that describe the properties of a node (e.g., a specific data value or state behavior), and link weights that describe some characteristic of a link
- The symbolic representation of a graph is shown in below Figure
- Nodes are represented as circles connected by links that take a number of different forms
- A directed link (represented by an arrow) indicates that a relationship moves in only one direction
- A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions
- Parallel links are used when a number of different relationships are established between graph nodes

2. Equivalence Partitioning

- Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived
- Test-case design for equivalence partitioning is based on an evaluation of equivalence

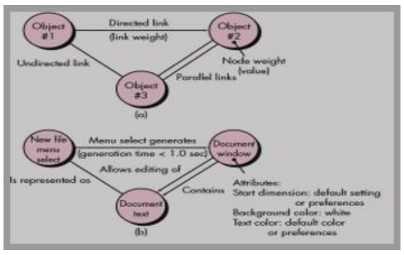


Figure 5.9: graph based testing

classes for an input condition

- Equivalence classes may be defined according to the following guidelines

If an input condition specifies a range, one valid and two invalid equivalence classes are defined

If an input condition requires a specific value, one valid and two invalid equivalence classes are defined

If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined

If an input condition is Boolean, one valid and one invalid class are defined

3. Boundary Value Analysis

- A greater number of errors occurs at the boundaries of the input domain rather than in the center of input domain
- For this reason that boundary value analysis (BVA) has been developed as a testing technique
- Boundary value analysis leads to a selection of test cases that exercise bounding values
- BVA leads to the selection of test cases at the edges‖ of the class. Rather than focusing solely on input conditions
- BVA derives test cases from the output domain also
- Guidelines for BVA are similar in many respects to those provided for equivalence partitioning

1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b

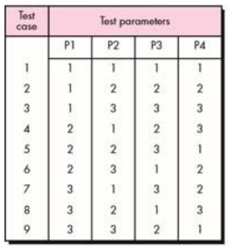
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested

3. Apply guidelines 1 and 2 to out put conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum(and minimum)allowable number of table entries

4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary. Most software engineers intuitively perform BVA to some degree. By applying these guidelines, boundary testing will be more complete, thereby having a higher likelihood for error detection

4. Orthogonal Array Testing

- Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing
- The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software compone
- For example, when a train ticket has to be verified, the factors such as - the number of passengers, ticket number, seat numbers and the train numbers has to be tested, which becomes difficult when a tester verifies input one by one. Hence, it will be more efficient when he combines more inputs together and does testing. Here, use the Orthogonal Array testing method
- When orthogonal array testing occurs, an L9 orthogonal array of test cases is created
- The L9 orthogonal array has a balancing property
- That is, test cases (represented by dark dots in the figure) are dispersed uniformly throughout the test domain,‖ as illustrated in the right-hand cube in Figure
- To illustrate the use of the L9 orthogonal array, consider the send function for a fax



Test case	Test parameters			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

Figure 5.10: orthogonal

application

- Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values
- P1 = 1, send it now : P1 = 2, send it one hour later : P1 = 3, send it after midnight
- P2, P3, and P4 would also take on values of 1, 2, and 3, signifying other send functions
- If a one input item at a time‖ testing strategy were chosen, the following sequence of tests (P1,P2,P3,P4) would be specified: (1,1,1,1),(2,1,1,1),(3,1,1,1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2), and (1, 1, 1, 3)
- The orthogonal array testing approach enables you to provide good test coverage with far fewer test cases than the exhaustive strategy. An L9 orthogonal array for the fax send function is illustrated in Figure

Regression testing

- When any modification or changes are done to the application or even when any small change is done to the code then it can bring unexpected issues. Along with the new changes it becomes very important to test whether the existing functionality is intact or not. This can be achieved by doing the regression testing
- The purpose of the regression testing is to find the bugs which may get introduced accidentally because of the new changes or modification
- During confirmation testing the defect got fixed and that part of the application started working as intended. But there might be a possibility that the fix may have introduced or uncovered a different defect elsewhere in the software. The way to detect these unexpected side-effects‘ of fixes is to do regression testing
- This also ensures that the bugs found earlier are NOT creatable
- Usually the regression testing is done by automation tools because in order to fix the defect the same test is carried out again and again and it will be very tedious and time consuming to do it manually
- During regression testing the test cases are prioritized depending upon the changes done to the feature or module in the application. The feature or module where the changes or modification is done that entire feature is taken into priority for testing

- This testing becomes very important when there are continuous modifications or enhancements done in the application or product. These changes or enhancements should NOT introduce new issues in the existing tested code
- This helps in maintaining the quality of the product along with the new changes in the application
- Example

Let‘s assume that there is an application which maintains the details of all the students in school. This application has four buttons Add, Save, Delete and Refresh. All the buttons functionalities are working as expected. Recently a new button Update‘ is added in the application. This Update‘ button functionality is tested and confirmed that it‘s working as expected. But at the same time it becomes very important to know that the introduction of this new button should not impact the other existing buttons functionality. Along with the

Update‘ button all the other buttons functionality are tested in order to find any new issues in the existing code. This process is known as regression testing. When to use Regression testing it

1. Any new feature is added
2. Any enhancement is done
3. Any bug is fixed
4. Any performance related issue is fixed

Advantages of Regression testing

- It helps us to make sure that any changes like bug fixes or any enhancements to the module or application have not impacted the existing tested code
- It ensures that the bugs found earlier are NOT creatable
- Regression testing can be done by using the automation tools
- It helps in improving the quality of the product

Disadvantages of Regression testing

- If regression testing is done without using automated tools then it can be very tedious and time consuming because here we execute the same set of test cases again and again
- Regression test is required even when a very small change is done in the code because this small modification can bring unexpected issues in the existing functionality

Unit testing

- Unit testing focuses verification effort on the smallest unit of software design—the software component or module
- Targets for Unit Test Cases

– Module interface

- Ensure that information flows properly into and out of the module

– Local data structures

- Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution

– Boundary conditions

- Ensure that the module operates properly at boundary values established to limit or restrict processing

– Independent paths (basis paths)

- Paths are exercised to ensure that all statements in a module have been executed at least once

– Error handling paths

- Ensure that the algorithms respond correctly to specific error conditions

Targets for Unit Test Cases

- Common Computational Errors in Execution Paths

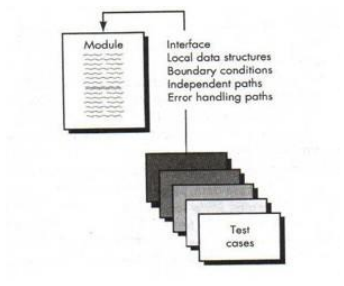


Figure 5.11: unit test cases

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors

- Incorrect symbolic representation of an expression (int vs. float)
- Other Errors to Uncover

– Comparison of different data types

– Incorrect logical operators or precedence

– Expectation of equality when precision error makes equality unlikely (using == with float types)

– Incorrect comparison of variables

– Improper or nonexistent loop termination

– Failure to exit when divergent iteration is encountered

– Improperly modified loop variables

– Boundary value violations

- Problems to uncover in Error Handling

– Error description is unintelligible or ambiguous

– Error noted does not correspond to error encountered

– Error condition causes operating system intervention prior to error handling

– Exception condition processing is incorrect

– Error description does not provide enough information to assist in the location of the cause of the error

- Unit test procedures

– Because a component is not a stand-alone program, driver and / or stub software must be developed for each unit test

- Driver

– A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results

- Stubs

– Serve to replace modules that are subordinate to (called by) the component to be tested

– It uses the module‘s exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing

- Drivers and stubs both represent overhead

– Both must be written but don‘t constitute part of the installed software products

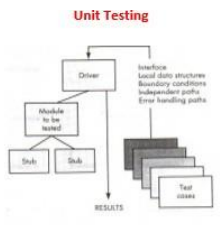


Figure 5.12: unit testing

Integration Testing

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

– Objective is to take unit tested modules and build a program structure based on the prescribed design

– Two Approaches

- Non-incremental Integration Testing
- Incremental Integration Testing
- Non-incremental Integration Testing

– Commonly called the Big Bang‖ approach

– All components are combined in advance

– The entire program is tested as a whole

– Disadvantages

- Chaos results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop
- Incremental Integration Testing

– Three kinds

- Top-down integration
- Bottom-up integration
- Sandwich integration

– The program is constructed and tested in small increments

– Errors are easier to isolate and correct

– Interfaces are more likely to be tested completely

– A systematic test approach is applied

• Top-down Integration

– Modules are integrated by moving downward through the control hierarchy, beginning with the main module

– Subordinate modules are incorporated in either a depth-first or breadth-first fashion

• DF: All modules on a major control path are integrated

• BF: All modules directly subordinate at each level are integrated

– The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module

– Depending on the integration approach selected subordinate stubs are replaced one at a time with actual components

– Tests are conducted as each component is integrated

– On completion of each set of tests, another stub is replaced with real component

• Top-down Integration

– Advantages

• This approach verifies major control or decision points early in the test process

– Disadvantages

• Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded

• Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

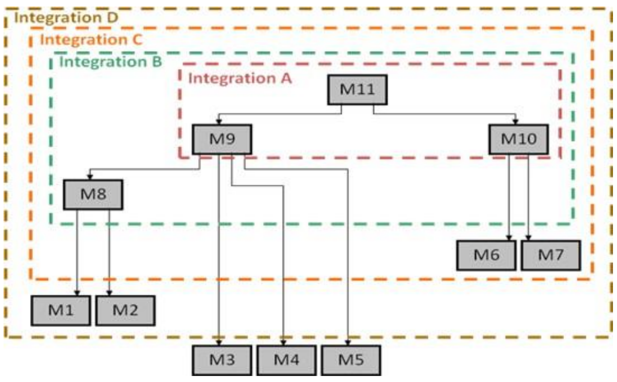


Figure 5.13: top-down integration

Bottom-up Integration

– Integration and testing starts with the most atomic modules (i.e., components at the lowest levels in the program structure) in the control hierarchy

– Begins construction and testing with atomic modules. As components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated

– Low-level components are combined into clusters that perform a specific software sub-function

– A driver is written to coordinate test case input and output

– The cluster is tested

– Drivers are removed and clusters are combined moving upward in the program structure

– As integration moves upward, the need for separate test drivers lessens. If the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified

• Bottom-up Integration

– Advantages

• This approach verifies low-level data processing early in the testing process

• Need for stubs is eliminated

– Disadvantages

• Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version

• Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

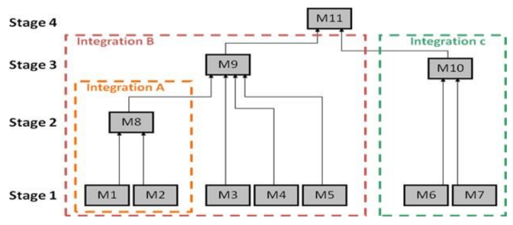


Figure 5.14: integration

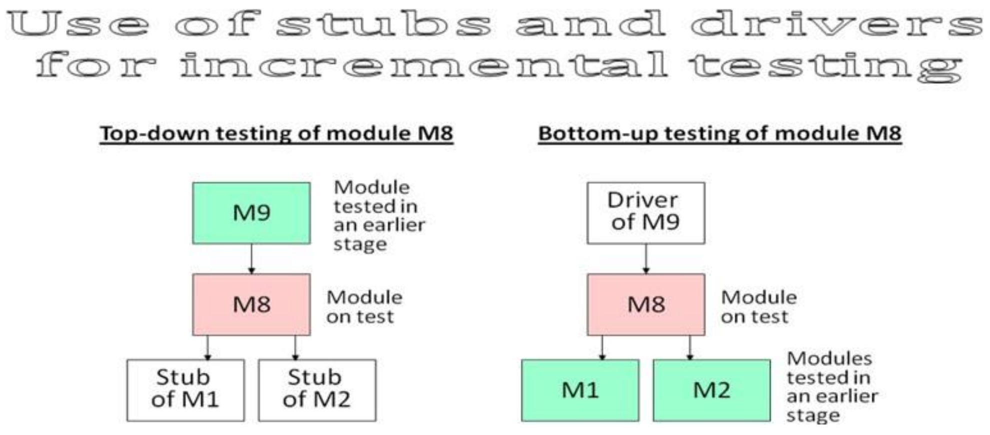

Use of stubs and drivers for incremental testing
Top-down testing of module M8
M9
Module tested in an earlier stage
M8
Module on test
Stub of M1
Stub of M2
Bottom-up testing of module M8
Driver of M9
M8
Module on test
M1
M2
Modules tested in an earlier stage

Figure 5.15: incremental testing

Sandwich Integration

- Consists of a combination of both top-down and bottom-up integration
- Occurs both at the highest level modules and also at the lowest level modules
- Proceeds using functional groups of modules, with each group completed before the next

– High and low-level modules are grouped based on the control and data processing they provide for a specific program feature

– Integration within the group progresses in alternating steps between the high and low level modules of the group

– When integration for a certain functional group is complete, integration and testing moves onto the next group

- Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
- Requires a disciplined approach so that integration doesn‘t tend towards the big bang‖ scenario

Smoke Testing

- Taken from the world of hardware

– Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure

- Designed as a pacing mechanism for time-critical projects

– Allows the software team to assess its project on a frequent basis

• Includes the following activities

– The software is compiled and linked into a build

– A series of breadth tests is designed to expose errors that will keep the build from properly performing its function

• The goal is to uncover show stopper‖ errors that have the highest likelihood of throwing the software project behind schedule

– The build is integrated with other builds and the entire product is smoke tested daily

• Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing

– After a smoke test is completed, detailed test scripts are executed

• Benefits of Smoke Testing

– Integration risk is minimized

• Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact

– The quality of the end-product is improved

• Smoke testing is likely to uncover both functional errors and architectural and component-level design errors

– Error diagnosis and correction are simplified

• Smoke testing will probably uncover errors in the newest components that were integrated

– Progress is easier to assess

• As integration testing progresses, more software has been integrated and more has been demonstrated to work

• Managers get a good indication that progress is being made

Validation Testing

• Validation testing follows integration testing

– The distinction between conventional and object-oriented software disappears

– Focuses on user-visible actions and user-recognizable output from the system

– Demonstrates conformity with requirements

1. Validation-Test Criteria

- Designed to ensure that

– All functional requirements are satisfied

– All behavioral characteristics are achieved

– All performance requirements are attained

– Documentation is correct

– Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)

– After each validation test

– The function or performance characteristic conforms to specification and is accepted

– A deviation from specification is uncovered and a deficiency list is created

– Deviation or error discovered at this stage in a project can rarely be corrected prior to scheduled delivery

2. A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle (activies)

- It is virtually impossible for a software developer to foresee how the customer will really use a program

- Instructions for use may be misinterpreted; strange combinations of data may be regularly used; output that seemed clear to the tester may be unintelligible to a user in the field

- When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal test drive‖ to a planned and systematically executed series of tests

- In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time

- If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one. Most software product builders use a process called alpha and beta testing to uncover errors that only the end user seems able to find

3. Alpha and Beta Testing

– Alpha Testing

• Conducted at the developer‘s site by end users

• Software is used in a natural setting with developers watching intently

• Testing is conducted in a controlled environment

– Beta Testing

• Conducted at end-user sites

• Developer is generally not present

• It serves as a live application of the software in an environment that cannot be controlled by the developer Alpha and Beta Testing

– Beta Testing

• The end-user records all problems that are encountered and reports these to the developers at regular intervals

• After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

System Testing

• System testing is a series of different test whose primary purpose is to fully exercise the computer-based system

• Each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions

1. Recovery testing

– Tests for recovery from system faults

– Forces the software to fail in a variety of ways and verifies that recovery is properly performed

– Tests reinitialization, checkpointing mechanisms, data recovery, and restart for correctness

– If recovery is automatic, reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness

– If recovery requires human intervention, the mean-time-to-repair is evaluated to determine whether it is within acceptable limits

2. Security testing

– Verifies that protection mechanisms built into a system will, in fact, protect it from improper access

– During security testing, the tester plays the role of the individual who desires to penetrate the system

– Anything goes! The tester may attempt to acquire passwords through external clerical means

– may attack the system with custom software designed to break down any defenses that have been constructed

3. Stress testing

– Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

– Stress tests are designed to confront programs with abnormal situations. In essence, the tester who performs stress testing asks: How high can we crank this up before it fails

– A variation of stress testing is a technique called sensitivity testing

• A very small range of data contained within the bounds of valid data for a program may cause extreme and even erroneous processing or profound
performance degradation

• Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing

4. Performance Testing

• Performance testing is designed to test the run-time performance of software within the context of an integrated system

• Performance testing occurs throughout all steps in the testing process. Even at the unit level, the performance of an individual module may be assessed as tests are conducted

• Performance tests are often coupled with stress testing and usually requires both hardware and software instrumentation

• That is, it is often necessary to measure resource utilization (e.g., processor cycles) in an exacting fashion

The art of Debugging

Debugging occurs as a consequence of successful testing. When a test case uncovers an error, debugging is an action that results in the removal of the error

1. The debugging process

The debugging process attempts to match symptom with cause, thereby leading to error correction

The debugging process will usually have one of two outcomes

(1) the cause will be found and corrected or

(2) the cause will not be found. In the latter case, the person performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion

– Debugging process beings with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is observed

– Debugging attempts to match symptom with cause , thereby leading to error correction

- Characteristics of bugs

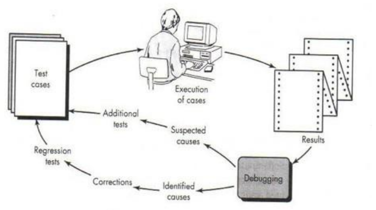


Figure 5.16: debugging

– The symptom and the cause may be geographically remote

– The symptom may disappear (temporarily) when another error is corrected

– The symptom may actually be caused by non-errors

– The symptom may be caused by human error that is not easily traced

– The symptom my be a result of timing problems, rather than processing problems

– It may be difficult to accurately reproduce input conditions

– The symptom may be intermittent

– The symptom may be due to causes that are distributed across a number of tasks running on different processors

2. Psychological Considerations

- Unfortunately, there appears to be some evidence that debugging prowess is an innate human trait. Some people are good at it and others aren‘t
- Although experimental evidence on debugging is open to many interpretations, large

variances in debugging ability have been reported for programmers with the same education and experience

3. Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error or defect
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies

1. Brute force
2. Backtracking
3. Cause elimination

- Brute Force

– Most commonly used and least efficient method for isolating the cause of a software error

– Used when all else fails

– Involves the use of memory dumps, run-time traces, and output statements

– Leads many times to wasted effort and time

- Backtracking

– Can be used successfully in small programs

– The method starts at the location where a symptom has been uncovered

– The source code is then traced backward (manually) until the location of the cause is found

– In large programs, the number of potential backward paths may become unmanageably large

- Cause Elimination

– Involves the use of induction or deduction and introduces the concept of binary partitioning

- Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
- Deduction (general to specific): Show that a specific conclusion follows from a set of general premises

– Data related to the error occurrence are organized to isolate potential causes

– A cause hypothesis is devised, and the aforementioned data are used to prove or disprove the hypothesis

– Alternatively, a list of all possible causes is developed, and tests are conducted to eliminate each cause

– If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

4. Correcting the Error

• Once a bug has been found, it must be corrected

• But the correction of a bug can introduce other errors and therefore do more harm than good

• Van Vleck suggests three simple questions that you should ask before making the correction‖ that removes the cause of a bug

• Three Questions to ask Before Correcting the Error

– Is the cause of the bug reproduced in another part of the program

• Similar errors may be occurring in other parts of the program

– What next bug might be introduced by the fix that I‘m about to make

• The source code (and even the design) should be studied to assess the coupling of logic and data structures related to the fix

– What could we have done to prevent this bug in the first place

• This is the first step toward software quality assurance

• By correcting the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs Coding Practices

• Best coding practices are a set of informal rules that the software development community has learned over time which can help improve the quality of software

• Many computer programs remain in use for far longer than the original authors ever envisaged (sometimes 40 years or more) so any rules need to facilitate both initial development and subsequent maintenance and enhancement by people other than the original authors

• In Ninety-ninety rule, Tim Cargill is credited with this explanation as to why programming projects often run late: ”The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time.” Any guidance which can redress this lack of foresight is worth considering

•The size of a project or program has a significant effect on error rates, programmer productivity, and the amount of management needed

a) Maintainability

b) Dependability

c) Efficiency

d) Usability

Refactoring

- Refactoring is usually motivated by noticing a code smell
- For example the method at hand may be very long, or it may be a near duplicate of another nearby method
- Once recognized, such problems can be addressed by refactoring the source code, or transforming it into a new form that behaves the same as before but that no longer ”smells”

There are two general categories of benefits to the activity of refactoring

software maintenance and reengineering

It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp. This might be achieved by reducing large monolithic routines into a set of
individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments

Extensibility. It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed

- Before applying a refactoring to a section of code, a solid set of automatic unit tests is needed. The tests are used to demonstrate that the behavior of the module is correct before the refactoring
- The tests can never prove that there are no bugs, but the important point is that this process can be cost-effective: good unit tests can catch enough errors to make them worthwhile and to make refactoring safe enough

Bibliography

[1] ABAQUS (2011). Abaqus 6.11 Online Documentation. Dassault Systemes.

[2]Armentrout, D. R. (1981). An analysis of the behavior of steel liner anchorages. PhD thesis, University of Tennessee.

[3] Bower, A. (2011). Applied Mechanics of Solids. Taylor & Francis.

[4]Brown, R. H. and Whitlock, A. R. (1983). Strength of anchor bolts in grouted concrete masonry. Journal of Structural Engineering, 109(6):1362–1374.

[5]Celep, Z. (1988). Rectangular plates resting on tensionless elastic foundation. Journal of Engineering mechanics, 114(12):2083–2092.

[6]Chakraborty, S. (2006). An experimental study on the beehaviour of steel plate-anchor assembly embedded in concrete under biaxial loading. M.tech thesis, Indian Institute of Technology Kanpur.

[7]Cook, R. A. and Klingner, R. E. (1992). Ductile multiple-anchor steel-to-concrete connections. Journal of structural engineering, 118(6):1645–1665.

[8]Damarla, V. N. (1999). An experimental investigation of performance of steel plate-concrete interfaces under combined action of shear and normal forces. Master's thesis, Indian Institute of Technology Kanpur.

[9]Doghri, I. (1993). Fully implicit integration and consistent tangent modulusin elasto-plasticity. International Journal for Numerical Methods in Engineering, 36(22):3915–3932.

[10] FEMA (June, 2007). Interim testing protocols for determining the seismic performance characteristics of structural and nonstructural components. Report 461, Federal Emergency Management Agency.

[11] Furche, J. and Elingehausen, R. (1991). Lateral blow-out failure of headed studs near a free edge. Anchors in Concrete-Design and Behavior, SP-130.

[12] Kallolil, J. J., Chakrabarti, S. K., and Mishra, R. C. (1998). Experimental investigation of embedded steel plates in reinforced concrete structures. Engineering structures, 20(1):105–112.

[13] Krawinkler, H., Zohrei, M., Lashkari-Irvani, B., Cofie, N., and Hadidi-Tamjed, H. (1983). Recommendations for experimental studies on the seismic behavior of steel components and materials. Report, Department of Civil and Environmental Engineering, Stanford Unniversity.

[14] Lemaitre, J. and Chaboche, J. L. (1994). Mechanics of Solid Materials. Cambridge University Press.

[15] Maya, S. (2008). An experimental study on the effect of anchor diameter on the behavior of steel plate-anchor assembly embedded in concrete under biaxial loading. M.tech thesis, Indian Institute of Technology Kanpur.

[16] Sahu, D. K. (2004). Experimental study on the behavior of steel plate-anchor assembly embedded in concrete under cyclic loading. M.tech thesis, Indian Institute of Technology Kanpur.

[17] Sonkar, V. (2007). An experimental study on the behaviour of steel plate-anchor assembly embedded in concrete under constant compressive axial load and cyclic shear. M.tech thesis, Indian Institute of Technology Kanpur.

[18] Thambiratnam, D. P. and Paramasivam, P. (1986). Base plates under axial loads and moments. Journal of Structural Engineering, 112(5):1166–1181.