

UNIT - 1

21CSC303J- Software Engineering Project Management (M.Tech Integrated)

These slides are designed to accompany *Software Engineering: A Practitioner's Approach*, 7th McGrawHill, 2009) Slides copyright 2009 by Roger Pressman.

SOFTWARE ENGINEERING

- Some realities:
 - *to understand the problem before a software solution is developed*
 - *design becomes a essential activity*
 - *software should exhibit high quality*
 - *software should be maintainable*

SOFTWARE ENGINEERING

- The IEEE definition:
 - *Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software*

THE EVOLVING ROLE OF SOFTWARE

- Software takes on a dual role. It is a product and, at the same time, the vehicle for delivering a product
- As a product, it delivers the computing potential embodied by computer hardware or, more broadly, a network of computers that are accessible by local hardware
- As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating systems), the communication of information (networks), and the creation and control of other programs (software tools and environments)

Changing Nature of Software

1. System software: Infrastructure software come under this category like compilers, operating systems, editors, drivers, etc. Basically system software is a collection of programs to provide service to other programs.
2. Real time software: These software are used to monitor, control and analyze real world events as they occur. An example may be software required for weather forecasting. Such software will gather and process the status of temperature, humidity and other environmental parameters to forecast the weather.

Changing Nature of Software

3. Embedded software: This type of software is placed in “Read-Only-Memory (ROM)” of the product and control the various functions of the product. The product could be an aircraft, automobile, security system, signalling system, control unit of power plants, etc. The embedded software handles hardware components and is also termed as intelligent software .

4. Business software : This is the largest application area. The software designed to process business applications is called business software. Business software could be payroll, file monitoring system, employee management, account management. It may also be a data warehousing tool which helps us to take decisions based on available data.

Management information system, enterprise resource planning (ERP) and such other software are popular examples of business software

Changing Nature of Software

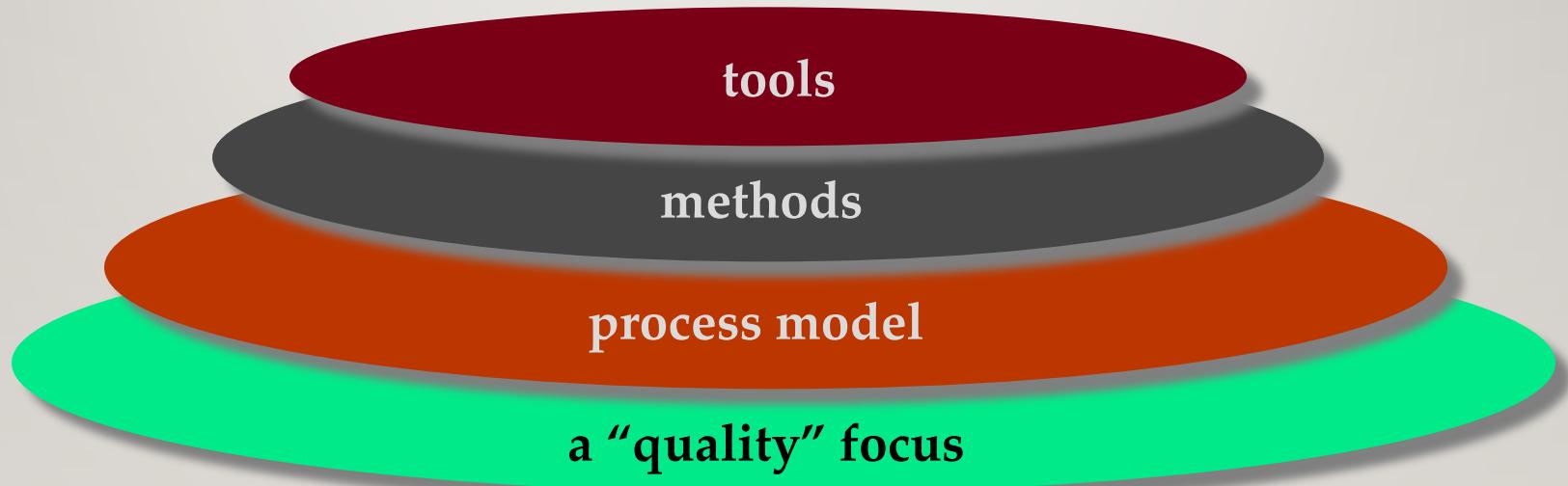
5. Personal computer software :The software used in personal computers are covered in this category. Examples are word processors, computer graphics, multimedia and animating tools, database management, computer games etc. This is a very upcoming area and many big organisations are concentrating their effort here due to large customer base.

6. Artificial intelligence software: Artificial Intelligence software makes use of non numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis. Examples are expert systems, artificial neural network,signal processing software etc.

Changing Nature of Software

8. Web based software: The software related to web applications come under this category. Examples are CGI, HTML, Java, Perl, DHTML etc.

A LAYERED TECHNOLOGY



*Software
Engineering*

-
- Software engineering is a **layered technology**
 - The foundation for software engineering is the *process layer*
 - **Process** defines a **framework** that must be established for effective delivery of software
 - Software engineering *methods provide the technical how-to's for building software*

-
- Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.
 - Software engineering *tools provide automated or semi-automated support for the* process and the methods.

THE ESSENCE OF PRACTICE

- Polya suggests:
 1. *Understand the problem* (communication and analysis).
 2. *Plan a solution* (modeling and software design).
 3. *Carry out the plan* (code generation).
 4. *Examine the result for accuracy* (testing and quality assurance).

UNDERSTAND THE PROBLEM

- who are the stakeholders?
- *What are the unknowns?* What data, functions, and features are required to properly solve the problem?
- *Can the problem be compartmentalized?* Is it possible to represent smaller problems that may be easier to understand?
- *Can the problem be represented graphically?* Can an analysis model be created?

PLAN THE SOLUTION

- *Have you seen similar problems before?* Are there patterns that are familiar in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- *Has a similar problem been solved?* If so, are elements of the solution reusable?
- *Can subproblems be defined?* If so, are solutions readily apparent for the subproblems?
- *Can you represent a solution in a manner that leads to effective implementation?* Can a design model be created?

CARRY OUT THE PLAN

- *Does the solution conform to the plan?* Is source code traceable to the design model?
- *Is each component part of the solution provably correct?* Has the design and code been reviewed, or better?

EXAMINE THE RESULT

- *Is it possible to test each component part of the solution?*
Has a reasonable testing strategy been implemented?
- *Does the solution produce results that conform to the data, functions, and features that are required?* Has the software been validated against all stakeholder requirements?

HOW IT ALL STARTS

- *SafeHome:*
 - Every software project is precipitated by some business need—
 - the need to **correct a defect** in an existing application;
 - the need to **adapt** a ‘legacy system’ to a changing business environment;
 - the need **to extend** the functions and features of an existing application, or
 - the need **to create** a new product, service, or system.

Process Models

WHAT / WHO / WHY IS PROCESS MODELS?

- **What** : a series of predictable steps--- a road map that helps you create a timely, high-quality results.
- **Who** : Software engineers and their managers, clients also
- **Why** : Provides stability, control, and organization to an activity that can if left uncontrolled, become quite chaotic.
- **What Work products** : Programs, documents, and data

WHAT / WHO / WHY IS PROCESS MODELS?

- **What are the steps** : The process you adopt depends on the software that you are building. One process might be good for aircraft avionic system, while an entirely different process would be used for website creation.
- **How to ensure right:** A number of software process assessment mechanisms can be used to determine the maturity of the software process. However, the quality, timeliness and long-term feasibility of the software are the best indicators of the efficacy of the process you use.

DEFINITION OF SOFTWARE PROCESS

- A *process* is *a collection of activities, actions, and tasks* that are performed when some work product is to be created
- An *activity* strives to achieve a broad *objective*
- An *action* encompasses a set of tasks that produce a major work product
- A *task* focuses on a small, but well-defined *objective* that produces a tangible outcome

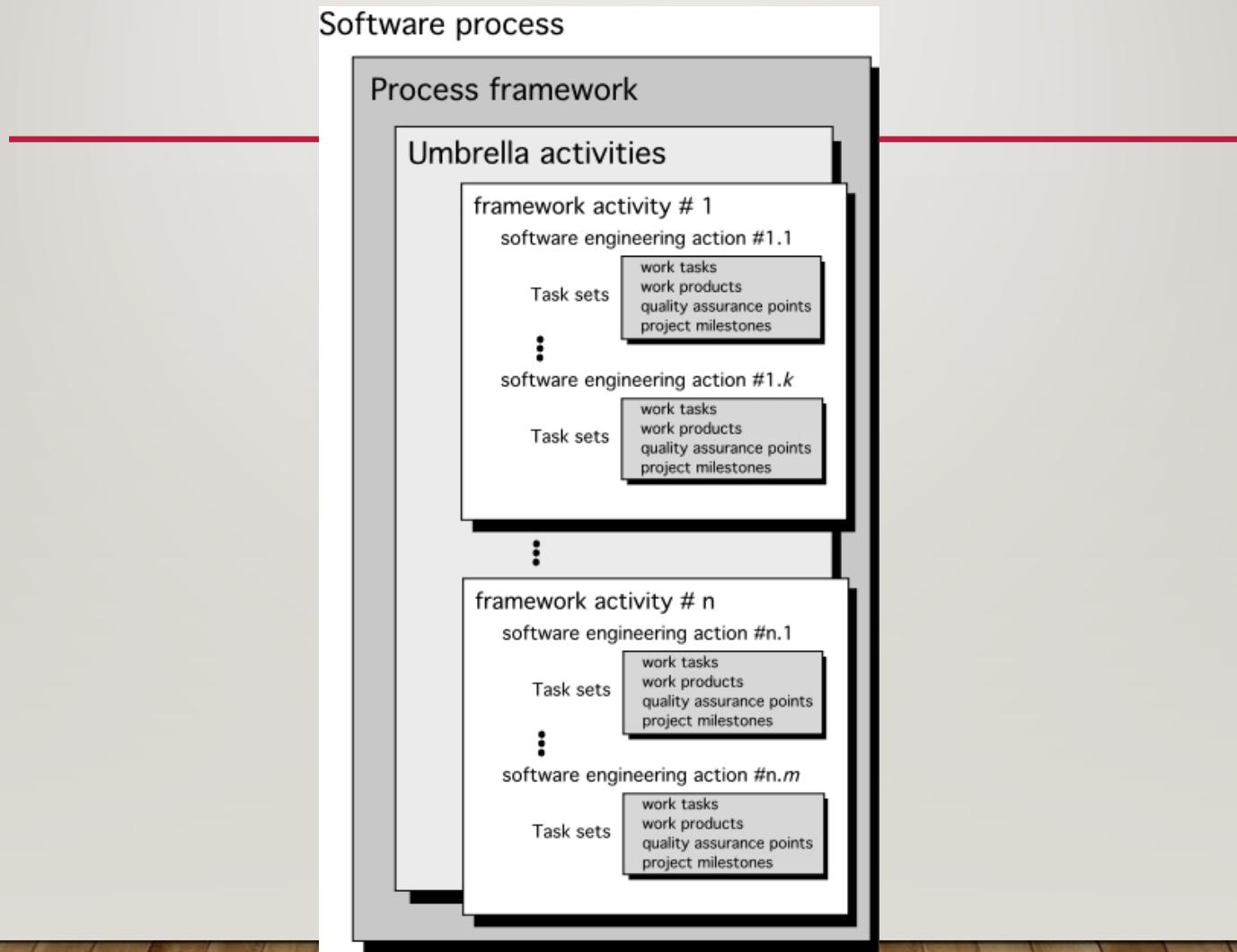
PROCESS FRAMEWORK(Phases)

- Communication
- Planning
- Modeling
- Construction
- Deployment

Umbrella activities

e.g. Risk management, Measurement, Reviews, SCM

A GENERIC PROCESS MODEL



A GENERIC PROCESS MODEL

- a generic process framework for software engineering defines five framework activities-communication, planning, modeling, construction, and deployment.
- **Communication:** to gather requirements that help define software features and functions.
- **Planning:** describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

A GENERIC PROCESS MODEL

- **Modeling** : create a “sketch” of the thing so that you’ll understand the big picture. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you’re going to solve it.
- **Construction**. This activity combines code generation and the testing that is required to uncover errors in the code.

A GENERIC PROCESS MODEL

- **Deployment:** The software is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.
- In addition, a set of umbrella activities- project tracking and control, risk management, quality assurance, configuration management, technical reviews, Measurement , Reusability management and Work product preparation and production are applied throughout the process.

A GENERIC PROCESS MODEL

- **Software project tracking and control** —allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.
- **Risk management** —assesses risks that may affect the outcome of the project or the quality of the product.
- **Software quality assurance** —defines and conducts the activities to monitoring the **software** engineering processes and methods used to ensure **quality** .

A GENERIC PROCESS MODEL

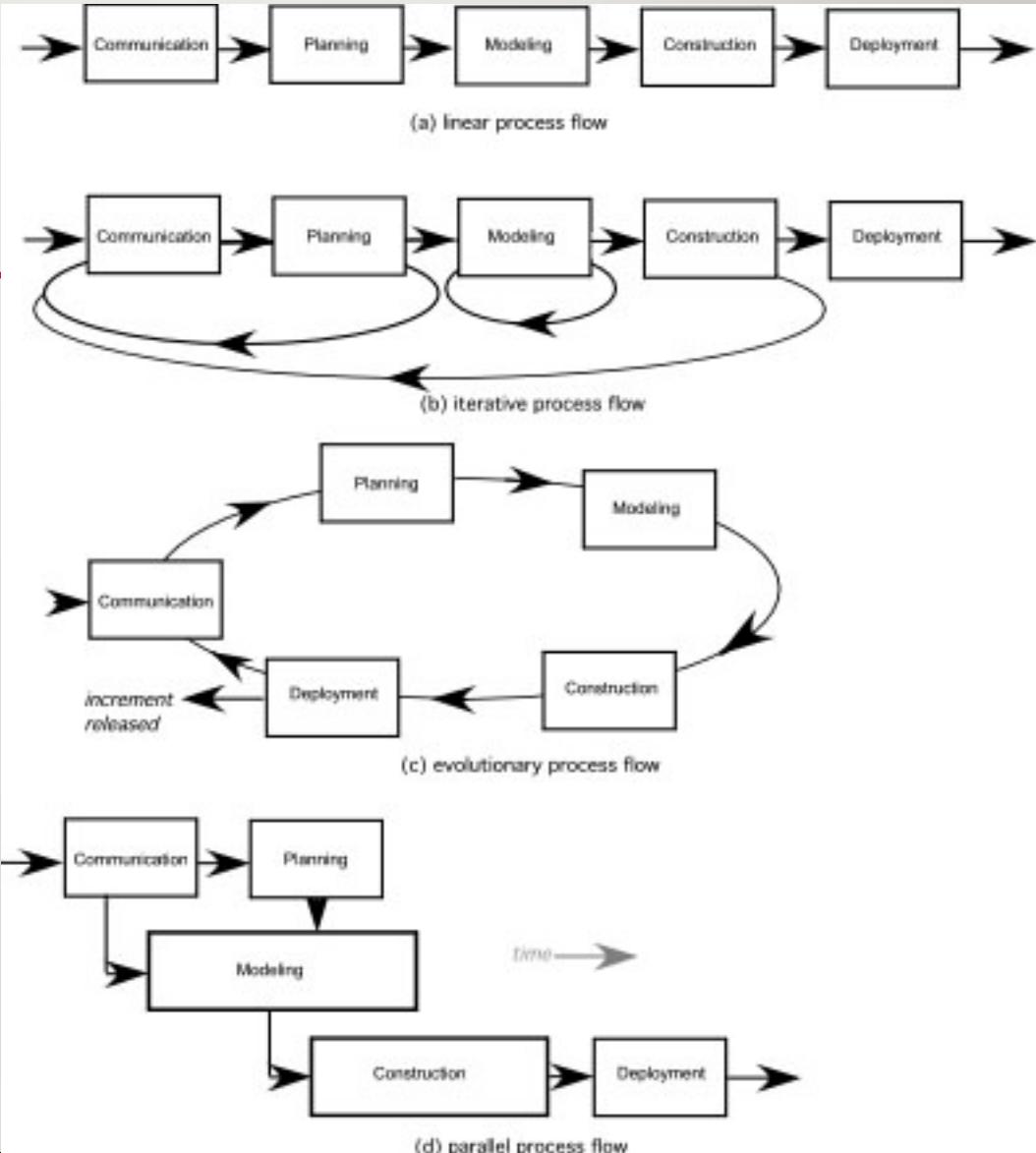
- **Technical reviews** —assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.
- **Software configuration management** — is the task of tracking and controlling changes in the software throughout the software process
- **Reusability management** —defines criteria for work product reuse and establishes mechanisms to achieve reusable components.

A GENERIC PROCESS MODEL

- **Work product preparation and production** —encompasses the activities required to create work products such as models, documents, logs, forms, and lists..
- **Measurement** —defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs

TYPES OF PROCESS FLOW

- 1) Linear Process Flow
- 2) Iterative Process Flow
- 3) Evolutionary Process Flow
- 4) Parallel Process Flow



PROCESS FLOW

CONTD...

- 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

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.

IDENTIFYING A TASK SET

- For example, a small software project requested by one person with simple requirements, the communication activity might encompass little more than a phone call with the stakeholder. Therefore, the only necessary action is phone conversation, the work tasks of this action are:
 1. Make contact with stakeholder via telephone.
 2. Discuss requirements and take notes.
 3. Organize notes into a brief written statement of requirements.
 4. E-mail to stakeholder for review and approval.

EXAMPLE OF A TASK SET FOR ELICITATION

(IS A TECHNIQUE USED TO GATHER
INFORMATION)

- 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.

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

1. Make a list of stakeholders for the project.
2. Interview each stakeholders separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

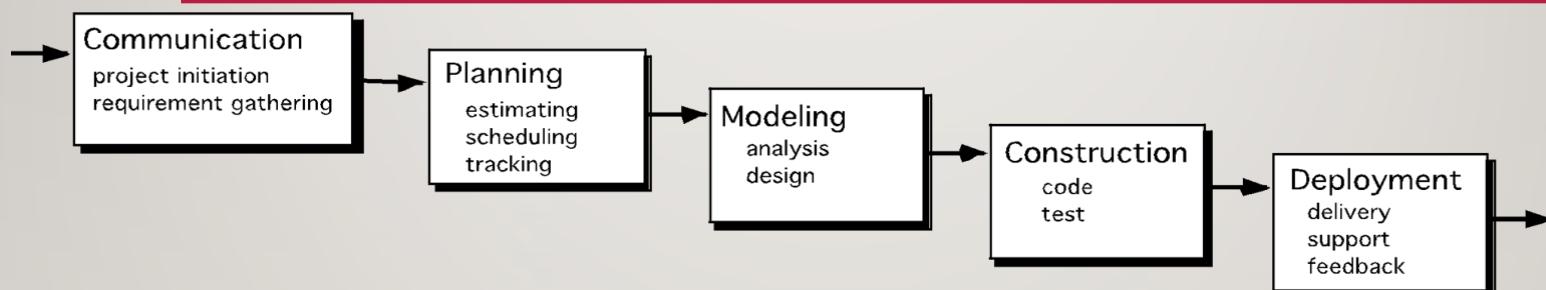
PROCESS MODELS

- **Prescriptive Process Models**
- Specialized Process Models (component based- Not in syllabus)
- **Unified Process**

PRESCRIPTIVE MODELS

- Waterfall/V model
- Incremental
- Evolutionary
 - Prototyping
 - Spiral

THE WATERFALL MODEL



It is the oldest paradigm for SE. 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

THE WATERFALL MODEL

- When to select?
- There are times when the requirements for a problem are well understood—when work flows from **communication through deployment in a reasonably linear fashion.**
- **(problems : 1. rarely linear, iteration needed. 2. hard to state all requirements explicitly . Blocking state. 3. code will not be released until very late.)**

WATERFALL MODEL (CONTD)

When do you choose this model?

- Requirements are well defined
- Well understood
- Well defined enhancements to existing system

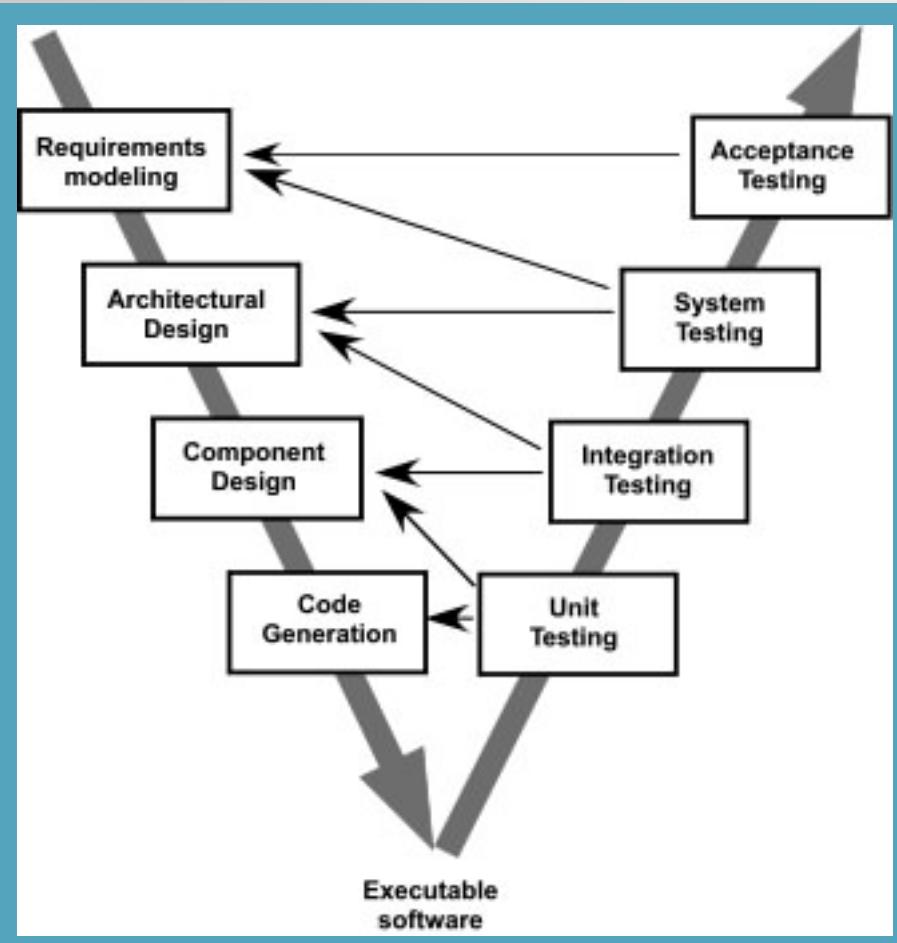
Advantages

1. Simple
2. Easy to understand even for non technical customers
3. Oldest, widely used
4. Base for all other models by including feed back loops, iterations etc.

Disadvantages

- Real projects rarely follow this linear sequence.
- Difficult for customer to state all requirements at one shot
- Customer must have patience.

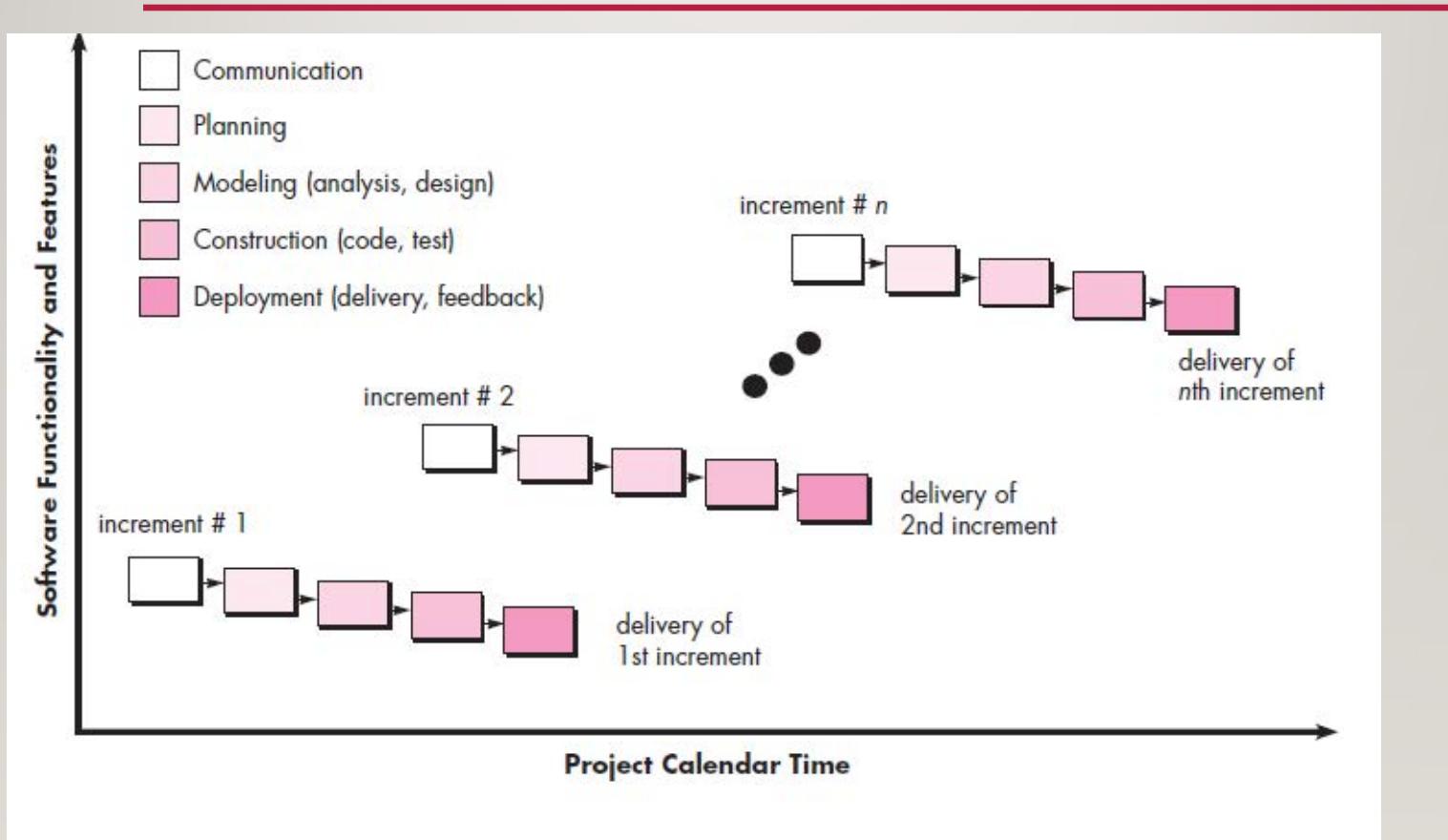
THE V-MODEL



A variation of waterfall model depicts the relationship of quality assurance actions to the 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 INCREMENTAL MODEL



INCREMENTAL MODEL (CONTD)

When do we use this model?

- Requirements are reasonably well defined
- But overall scope precludes linear flow
- Difficult deadlines
- Constraints in staffing
- Manage technical risks.
- Compelling need to provide limited set of functionality to users quickly.

Steps

- First increment is a core product
- Core product used/reviewed by the customer
- A plan for the next increment is laid. Modification of the core product for additional functionality
- Process is repeated following the delivery of each increment, until the complete product is produced.

THE INCREMENTAL MODEL

- When initial requirements are reasonably well defined, but the overall scope of the development effort prevent 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 MODEL

- This process is repeated following the delivery of each increment, until the complete product is produced
- When to use?
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

THE INCREMENTAL MODEL

- Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.
- In addition, increments can be planned to manage technical risks

INCREMENTAL MODEL (CONTD)

Advantages

1. Early feedback is there on the core product.
2. Reduced risk
3. Effective solution since user evaluates core product at each increment level
4. Complexity is reduced

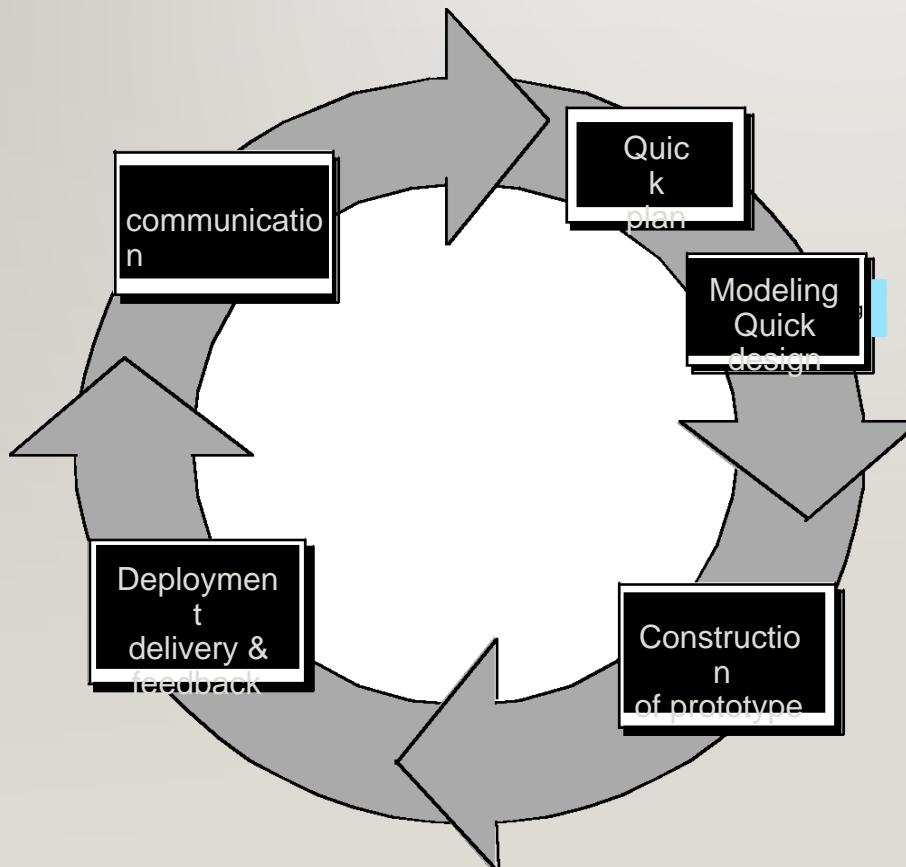
Disadvantages

1. Requires expertise planning both at management and technical level
2. Client dependent, customer should accept phased deliverables

EVOLUTIONARY MODELS

- 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



Steps

- Begins with communication
- A quick plan for prototyping and modeling 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.

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.

EVOLUTIONARY MODELS: PROTOTYPING

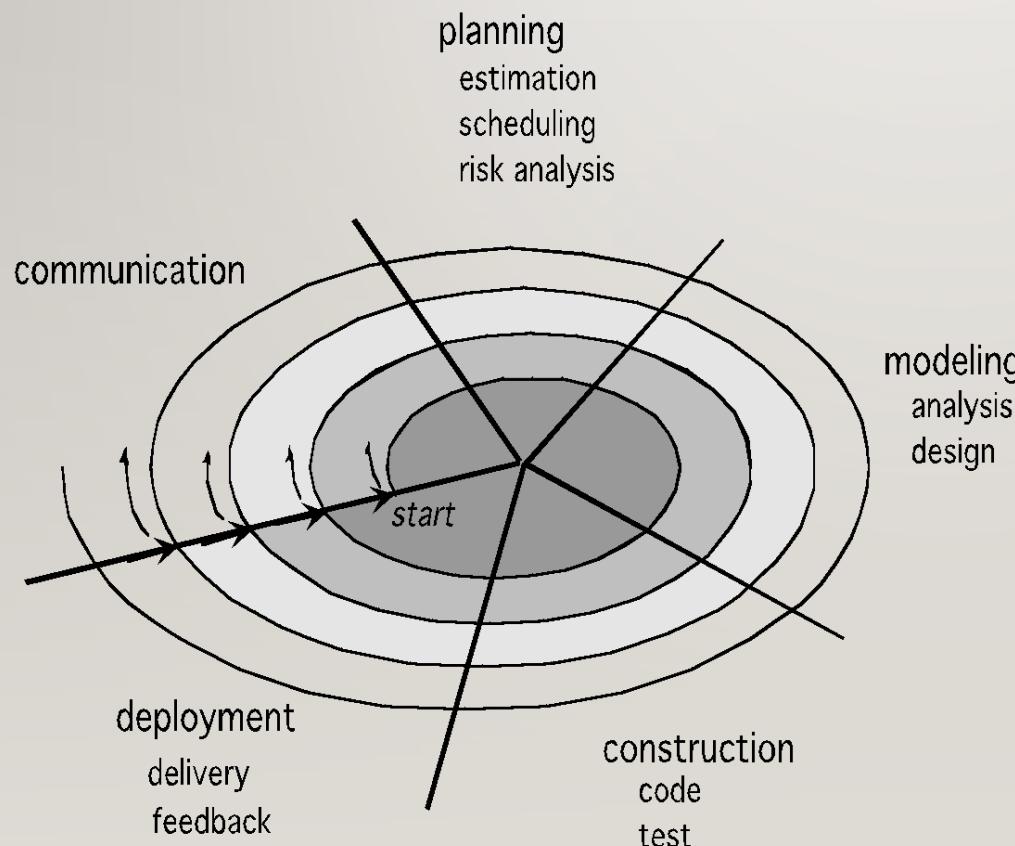
Advantages

- Provides working model.
- Customer is highly satisfied with such a modeling at initial stages
- Developer gains business insight, reducing ambiguity
- Great involvement of users
- Reduce risks

Disadvantages

- Customer - not aware that only interface or appearance is concentrated much and long term quality is at stake
- False expectations from customer that end s/m is finished or will have the same behavior/pace of the prototype
- Inappropriate choices of technology
- Various iterations to a prototype that is to be discarded is expensive

EVOLUTIONARY MODELS: THE SPIRAL



How it works?

- Series of evolutionary releases
 - Earlier releases – prototype, later implementation
 - First circuit - specification, next prototype and sophisticated versions
- Each pass
 - Project plan, Cost, schedule, number of future iterations adjusted
 - Risk is evaluated

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.

EVOLUTIONARY MODELS: THE SPIRAL

Advantages

- Applies throughout lifecycle
 - Concept Development
 - New Prod Development
 - Product Enhancement
- Risk is considered at each pass
- Uses prototyping as risk reduction mechanism
- Customer and developer understand and better react to risks

Disadvantages

- Difficult to convince customers that it is controllable
- Demands considerable risk assessment expertise
- Major risk is not uncovered/managed, problem will occur

THREE CONCERNS ON EVOLUTIONARY PROCESSES

- First concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product.
- Second, it does not establish the maximum speed of the evolution. If the evolution occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected.
- Third, software processes should be focused on flexibility and extensibility rather than on high quality. We should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product when the opportunity niche has disappeared.
55

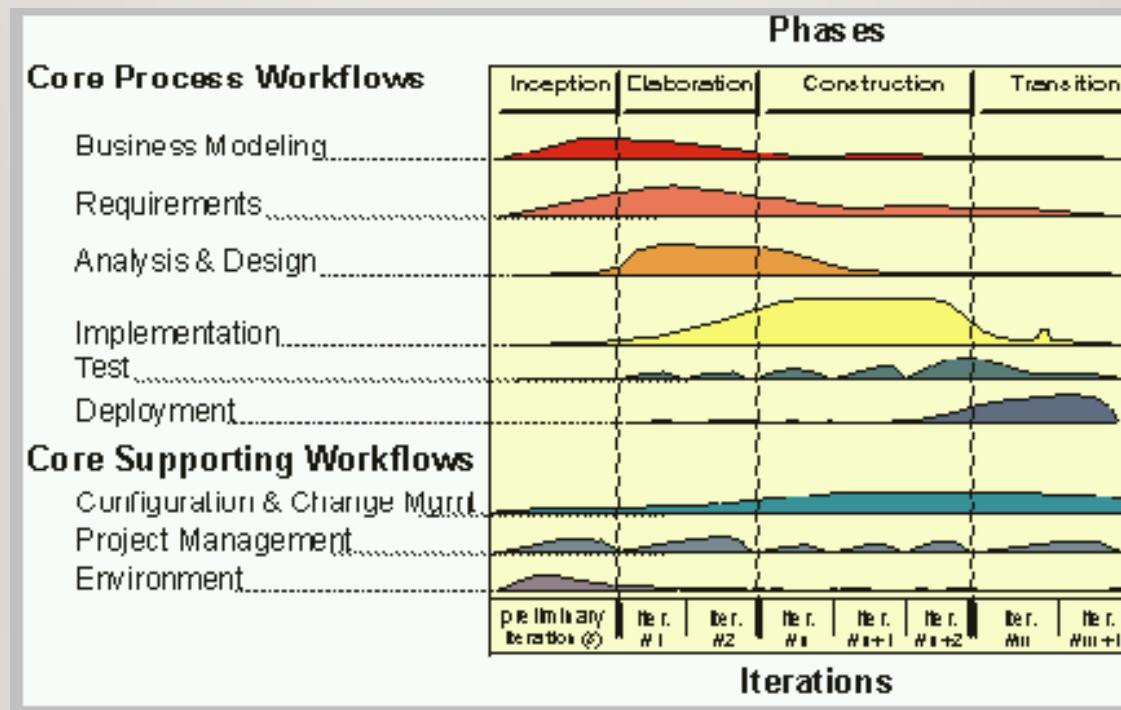
THE RATIONAL UNIFIED PROCESS

- RUP is a method of managing OO Software Development
- It can be viewed as a Software Development Framework which is extensible and features:
 - Iterative Development
 - Requirements Management
 - Component-Based Architectural Vision
 - Visual Modeling of Systems
 - Quality Management
 - Change Control Management

RUP FEATURES

- Online Repository of Process Information and Description in HTML format
- Templates for all major artifacts, including:
 - RequisitePro templates (requirements tracking)
 - Word Templates for Use Cases
 - Project Templates for Project Management
- Process Manuals describing key processes

THE PHASES



AN ITERATIVE DEVELOPMENT PROCESS...

- Recognizes the reality of changing requirements
 - Caspers Jones's research on 8000 projects
 - 40% of final requirements arrived after the analysis phase, after development had already begun
- Promotes early risk mitigation, by breaking down the system into mini-projects and focusing on the riskier elements first
- Allows you to “plan a little, design a little, and code a little”
- Encourages all participants, including testers, integrators, and technical writers to be involved earlier on
- Allows the process itself to modulate with each iteration, allowing you to correct errors sooner and put into practice lessons learned in the prior iteration
- Focuses on component architectures, not final big bang deployments

AN INCREMENTAL DEVELOPMENT PROCESS...

- Allows for software to evolve, not be produced in one huge effort
- Allows software to improve, by giving enough time to the evolutionary process itself
- Forces attention on stability, for only a stable foundation can support multiple additions
- Allows the system (a small subset of it) to actually run much sooner than with other processes
- Allows interim progress to continue through the stubbing of functionality
- Allows for the management of risk, by exposing problems earlier on in the development process

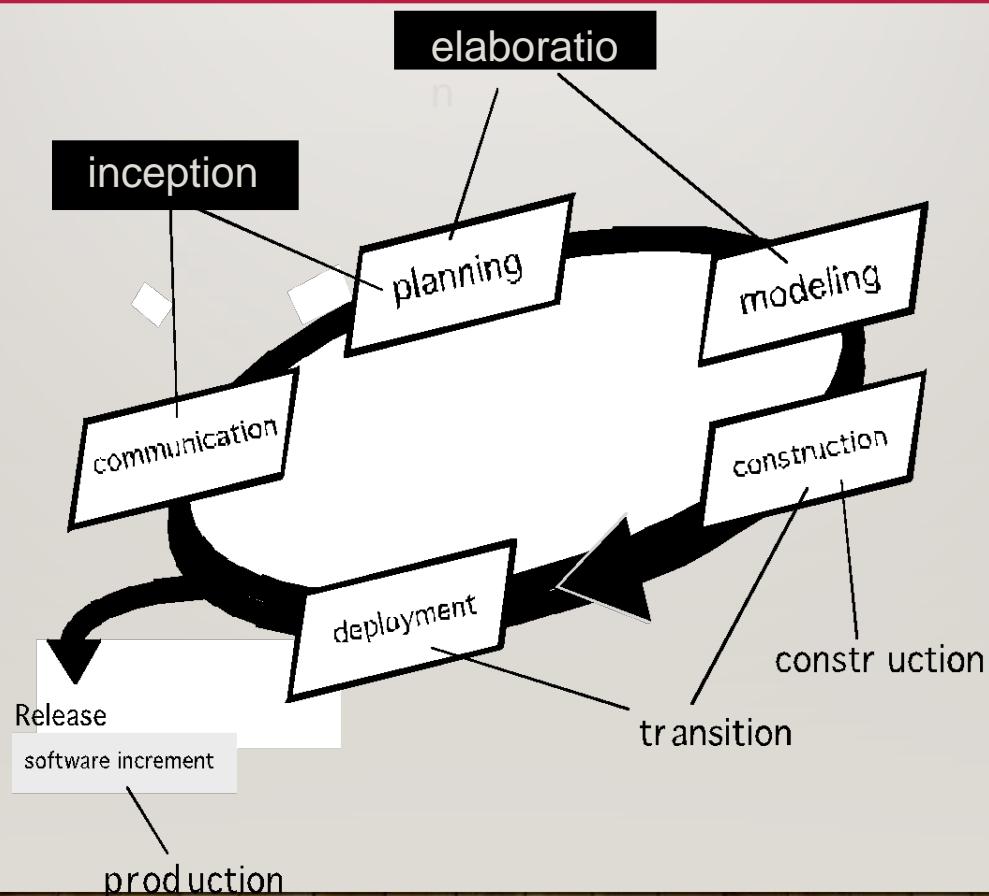
GOALS AND FEATURES OF EACH ITERATION

- The primary goal of each iteration is to slowly chip away at the risk facing the project, namely:
 - performance risks
 - integration risks (different vendors, tools, etc.)
 - conceptual risks (ferret out analysis and design flaws)
- Perform a “miniwaterfall” project that ends with a delivery of something tangible in code, available for scrutiny by the interested parties, which produces validation or correctives
- Each iteration is risk-driven
- The result of a single iteration is an increment--an incremental improvement of the system, yielding an evolutionary approach

THE DEVELOPMENT PHASES

- Inception Phase
- Elaboration Phase
- Construction Phase
- Transition Phase

THE UNIFIED PROCESS (UP)



INCEPTION PHASE

- Overriding goal is obtaining buy-in from all interested parties
- Initial requirements capture
- Cost Benefit Analysis
- Initial Risk Analysis
- Project scope definition
- Defining a candidate architecture
- Development of a disposable prototype
- Initial Use Case Model (10% - 20% complete)
- First pass at a Domain Model

ELABORATION PHASE

- Requirements Analysis and Capture
 - Use Case Analysis
 - Use Case (80% written and reviewed by end of phase)
 - Use Case Model (80% done)
 - Scenarios
 - Sequence and Collaboration Diagrams
 - Class, Activity, Component, State Diagrams
 - Glossary (so users and developers can speak common vocabulary)
 - Domain Model
 - to understand the problem: the system's requirements as they exist within the context of the problem domain
 - Risk Assessment Plan revised
 - Architecture Document

CONSTRUCTION PHASE

- Focus is on implementation of the design:
 - cumulative increase in functionality
 - greater depth of implementation (stubs fleshed out)
 - greater stability begins to appear
 - implement all details, not only those of central architectural value
 - analysis continues, but design and coding predominate

TRANSITION PHASE

- The transition phase consists of the transfer of the system to the user community
- It includes manufacturing, shipping, installation, training, technical support and maintenance
- Development team begins to shrink
- Control is moved to maintenance team
- Alpha, Beta, and final releases
- Software updates
- Integration with existing systems (legacy, existing versions, etc.)

AGILE PROCESS MODEL

THE MANIFESTO FOR AGILE SOFTWARE DEVELOPMENT

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- *Individuals and interactions over processes and tools*
- *Working software over comprehensive documentation*
- *Customer collaboration over contract negotiation*
- *Responding to change over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.”

Kent Beck et al

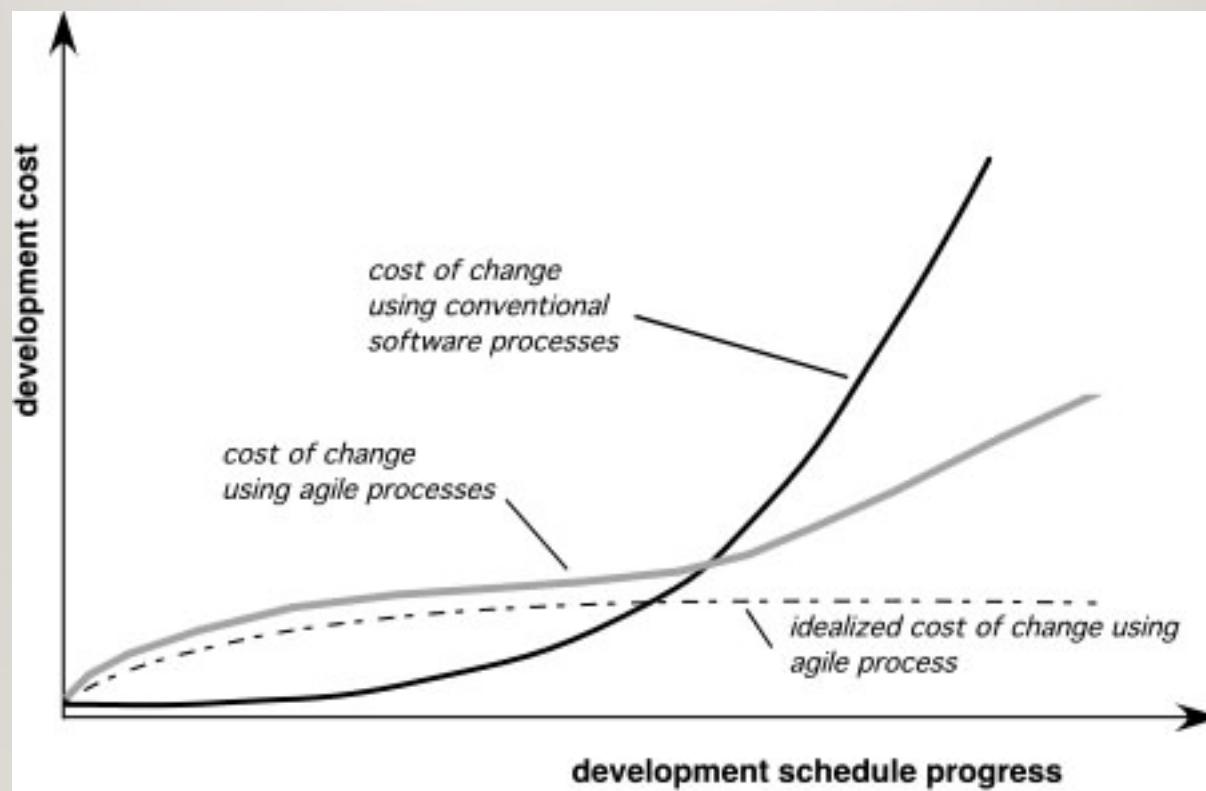
WHAT IS “AGILITY”?

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed

Yielding ...

- Rapid, incremental delivery of software

AGILITY AND THE COST OF CHANGE



AN AGILE PROCESS

- Is driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple ‘software increments’
- Adapts as changes occur

AGILITY PRINCIPLES

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive 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.

AGILITY PRINCIPLES

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

HUMAN FACTORS

- *the process molds to the needs of the people and team, not the other way around*
- key traits must exist among the people on an agile team and the team itself:
 - Competence.
 - Common focus.
 - Collaboration.
 - Decision-making ability.
 - Fuzzy problem-solving ability.
 - Mutual trust and respect.
 - Self-organization.

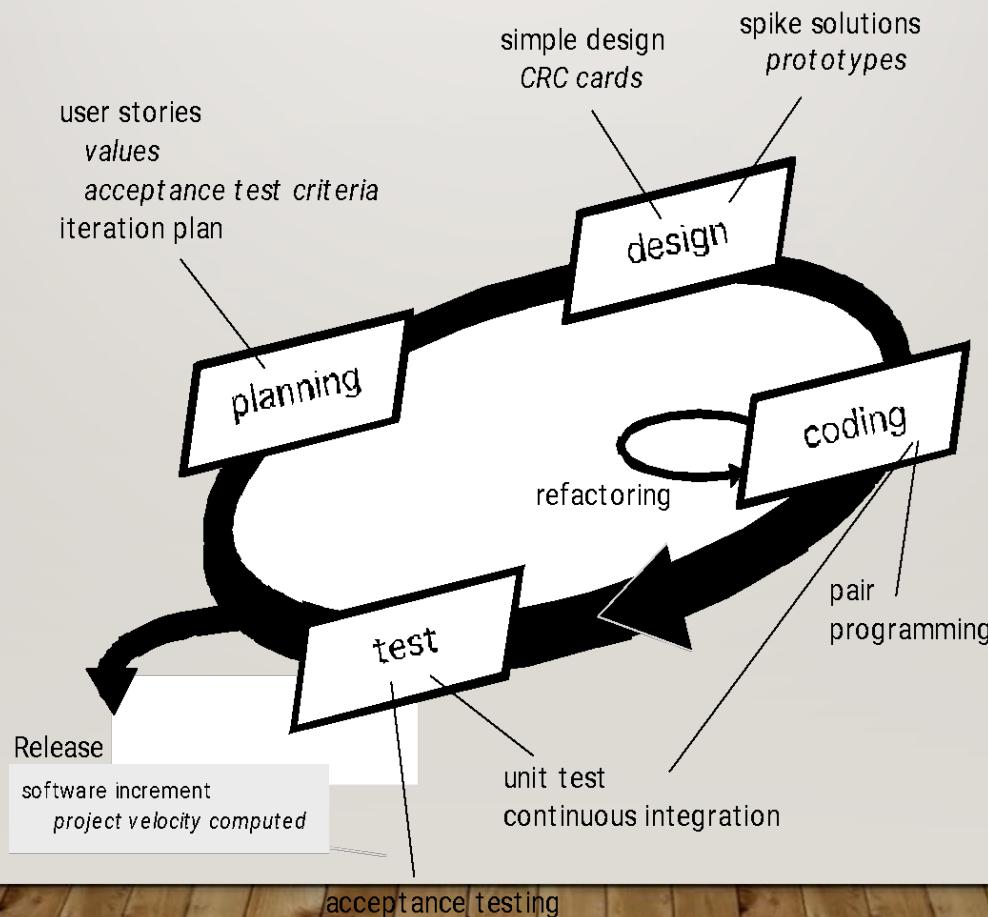
EXTREME PROGRAMMING (XP)

- The most widely used agile process, originally proposed by Kent Beck
- XP Planning
 - Begins with the creation of “**user stories**”
 - Agile team assesses each story and assigns a **cost**
 - Stories are grouped to form a **deliverable increment**
 - A **commitment** is made on delivery date
 - After the first increment “**project velocity**” is used to help define subsequent delivery dates for other increments

EXTREME PROGRAMMING (XP)

- XP Design
 - Follows the **KIS principle**
 - Encourage the use of **CRC cards** (see Chapter 8)
 - For difficult design problems, suggests the creation of “**spike solutions**”—a design prototype
 - Encourages “**refactoring**”—an iterative refinement of the internal program design
- XP Coding
 - Recommends the **construction of a unit test** for a store *before* coding commences
 - Encourages **pair programming**
- XP Testing
 - All **unit tests are executed daily**
 - “**Acceptance tests**” are defined by the customer and executed to assess₇₇ customer visible functionality

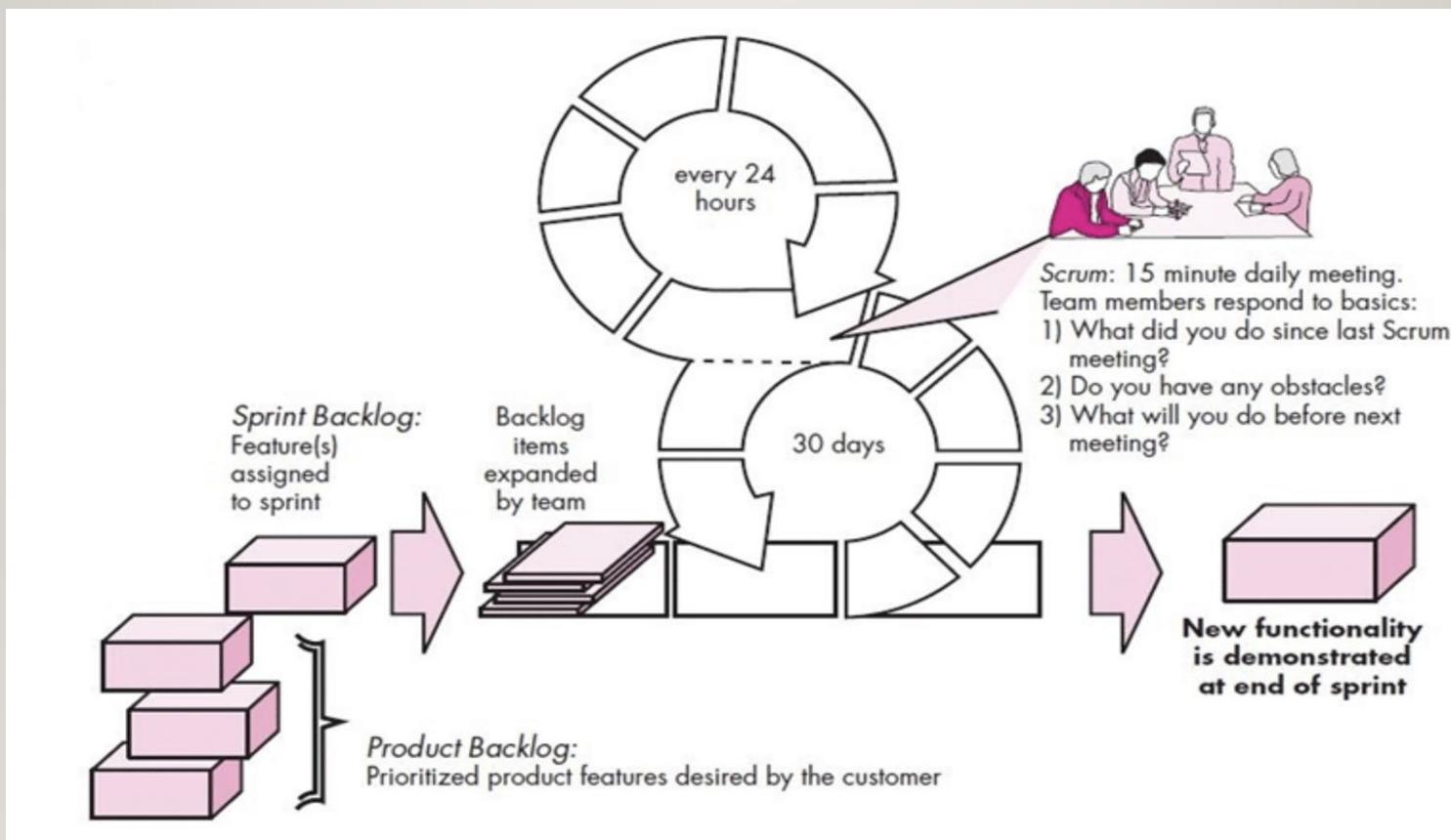
EXTREME PROGRAMMING (XP)



SCRUM

- Originally proposed by Schwaber and Beedle
- Scrum—distinguishing features
 - Development work is partitioned into “packets”
 - Testing and documentation are on-going as the product is constructed
 - Work occurs in “sprints” and is derived from a “backlog” of existing requirements
 - Meetings are very short and sometimes conducted without chairs
 - “demos” are delivered to the customer with the time-box allocated

SCRUM



Project Initiation Management

82 Project Initiation

- Project initiation is the first step in starting a new project. During the project initiation phase, you establish why you're doing the project and what business value it will deliver.
- Project initiation is the first phase of the project management life cycle and in this stage, companies decide if the project is needed and how beneficial it will be for them. The two metrics that are used to judge a proposed project and determine the expectations from it are the business case and feasibility study.

83 Importance of Project Initiation

- Take major decisions that establish the direction and resource requirements, like the project charter and selecting the project stakeholders, are made during this phase. The stakeholders arrive at a clear objective to ensure everyone stays on the same page in terms of how the project should proceed.
- There will be multiple checks during and after project execution to prevent miscommunication and to ensure the project stays on track throughout its course. However, precious time and resources might get wasted which is undesirable.
- Effective project management requires you to maximize benefits and minimize costs while delivering ‘value’ to the customer. Having a clear project objective helps you achieve all this.

1. Creating a business case
2. Conducting a feasibility study
3. Establishing a project charter
4. Identifying stakeholders and making a stakeholder register
5. Assembling the team and establishing a project office
6. Final review

Project Initiation

Follow these six steps to start your project off right



1 Business Case

Explain why the project is necessary and how it will succeed

2 Feasibility Study

Research the reason for the project and determine if it will succeed

3 Project Charter

How will the project be structured and executed?

4 Team

Find the people with the right skills and experience to execute the project

5 Project Office

Where the project manager and support staff are located to assist with projects

6 Review

Review the initiation phase and keep reviewing progress throughout the project

86 Project charter

A project charter demonstrates why your project is important, what it will entail, and who will work on it—all through the following elements:

Why: The project's goals and purpose

What: The scope of the project, including an outline of your project budget

Who: Key stakeholders, project sponsors, and project team members

87 Project Scope

Project scope is a way to set boundaries on your project and define exactly what goals, deadlines, and project deliverables you'll be working towards. By clarifying your project scope, you can ensure you hit your project goals and objectives without delay or overwork.

88 Benefits of defining your project scope

- Ensure all stakeholders have a clear understanding of the boundaries of the project
- Manage stakeholder expectations and get buy-in
- Reduce project risk
- Budget and resource plan appropriately
- Align your project to its main objectives
- Prevent scope creep (when project deliverables exceed the project scope)
- Establish a process for change requests (for complex projects)

89

Steps in defining project scope

- 1: Define the project goal and objectives
- 2: Identify the potential roadblocks
- 3: Incorporate additional project requirements
- 4: List necessary resources
- 5: Chart a project milestone schedule
- 6: Define the tasks and deliverables
- 7: Have a change management and control system
- 8: prepare and share the project scope statement

90 Project Objectives

Project objectives are what you plan to achieve by the end of your project.

This might include deliverables and assets, or more intangible objectives like increasing productivity or motivation.

Your project objectives should be attainable, time-bound, specific goals you can measure at the end of your project.

91

Steps to follow in writing Project Objectives

1. Set your project objectives at the beginning of your project
2. Involve your project team in the goal-setting process
3. Create brief, but clear, project objective statements
4. Make sure your objectives are things you can control (SMART-Specific,Measurable,Achievable,Realistic,Time-bound)
5. Check in on your project objectives during the project's lifecycle

Example 1 Business project objective

Bad: Launch new home page.

This project objective is missing many important characteristics. Though this objective is measurable, achievable, and realistic, it's not specific or time-bound. When should the home page be live? What should the redesign focus on?

Good: Create net-new home page assets and copy, focusing on four customer stories and use cases. Launch refreshed, customer-centric home page by the end of Q2.

This project objective is solid. It's specific (create net-new home page assets and copy), measurable (launch refreshed, customer-centric home page), achievable and realistic (focusing on four customer stories and use cases), and time bound (by the end of Q2).

93 Example 2 Non profit Project Objective

Bad: Increase sustainability in our production process by 5%

Though this project objective is more specific than the previous bad example, it's still lacking several important characteristics. This objective is measurable (by 5%), but it's not specific or time-bound, since we don't specify what "sustainability" means or by when the production process should improve. As a result, we don't really know if it's achievable or realistic.

Good: Reduce operational waste by 5% and increase use of recycled products by 20% in the next 12 weeks.

This project objective builds upon the previous one, because we now have a specific objective. This project objective also includes a way to measure the goal (by 5%... by 20%). The objective is a little ambitious, but the fact that it's time-bound (in the next 12 weeks) makes it

94



Chapter 4 – Requirements Engineering

DR.A.SHANTHINI

ASSOCIATE PROFESSOR

DEPARTMENT OF DSBS

SRM IST

Topics covered

Functional and non-functional requirements

Requirements engineering processes

Requirements elicitation

Requirements specification

Requirements validation

Requirements change / Management



How the customer explained it.



How the Project Manager Understood it.



How the Engineer Designed it.



How the Technician Built it.



How the Customer really wanted it.

Requirements engineering

The process of establishing the services that a customer requires from a system and the constraints under which it operates and is developed.

The system requirements are the descriptions of the system services and constraints that are generated during the requirements engineering process.

What is a requirement?

It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.

This is inevitable as requirements may serve a dual function

- May be the basis for a bid for a contract - therefore must be open to interpretation;
- May be the basis for the contract itself - therefore must be defined in detail;
- Both these statements may be called requirements.

Requirements abstraction (Davis)

“If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not pre-defined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization’s needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.”

Types of requirement

User requirements

- Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.

System requirements

- A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

User and system requirements

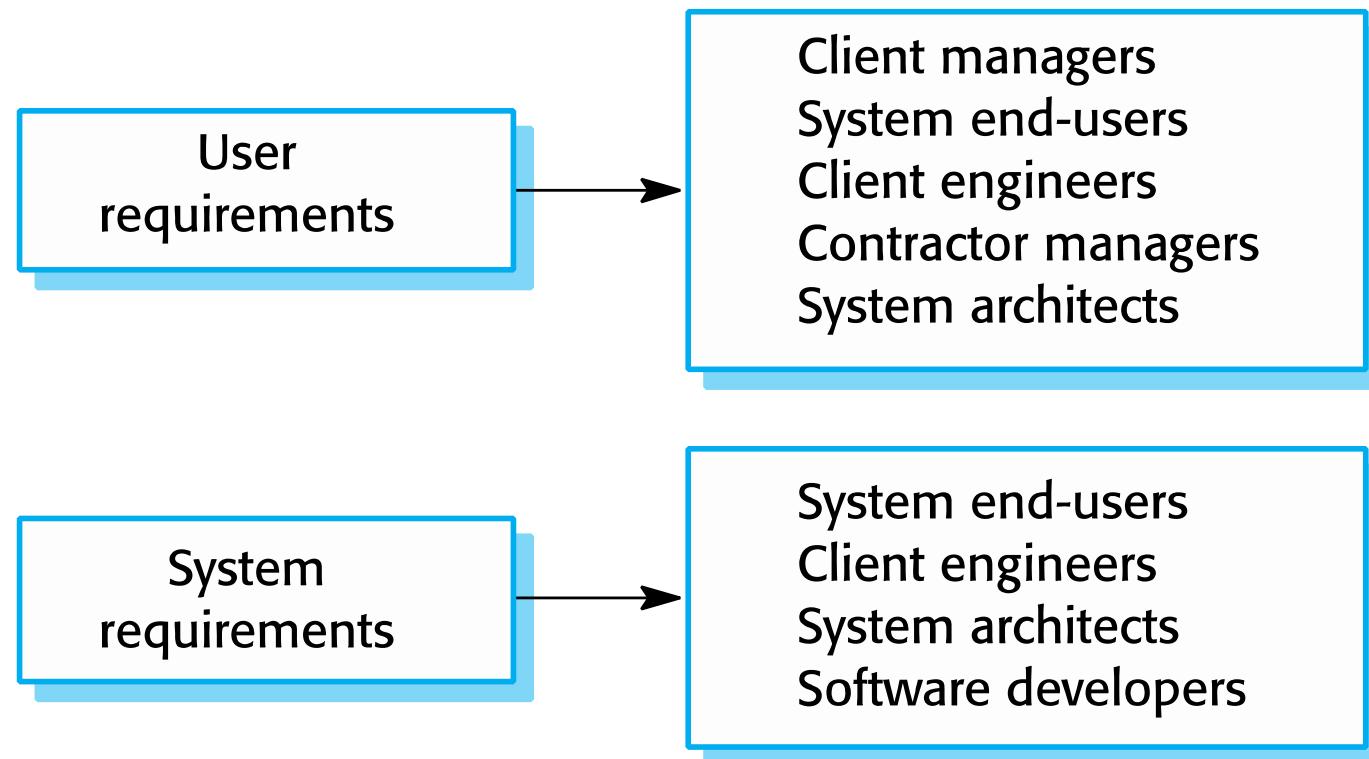
User requirements definition

- 1.** The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

- 1.1** On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
- 1.2** The system shall generate the report for printing after 17.30 on the last working day of the month.
- 1.3** A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
- 1.4** If drugs are available in different dose units (e.g. 10mg, 20mg, etc) separate reports shall be created for each dose unit.
- 1.5** Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Readers of different types of requirements specification



Functional and non-functional requirements

Functional and non-functional requirements

Functional requirements

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- May state what the system should not do.

Non-functional requirements

- Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Often apply to the system as a whole rather than individual features or services.

Domain requirements

- Constraints on the system from the domain of operation

Functional requirements

Describe functionality or system services.

Depend on the type of software, expected users and the type of system where the software is used.

Functional user requirements may be high-level statements of what the system should do.

Functional system requirements should describe the system services in detail.

Mentcare system: functional requirements

A user shall be able to search the appointments lists for all clinics.

The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.

Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

Requirements imprecision

Problems arise when functional requirements are not precisely stated.

Ambiguous requirements may be interpreted in different ways by developers and users.

Consider the term 'search' in requirement 1

- User intention – search for a patient name across all appointments in all clinics;
- Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

Requirements completeness and consistency

In principle, requirements should be both complete and consistent.

Complete

- They should include descriptions of all facilities required.

Consistent

- There should be no conflicts or contradictions in the descriptions of the system facilities.

In practice, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

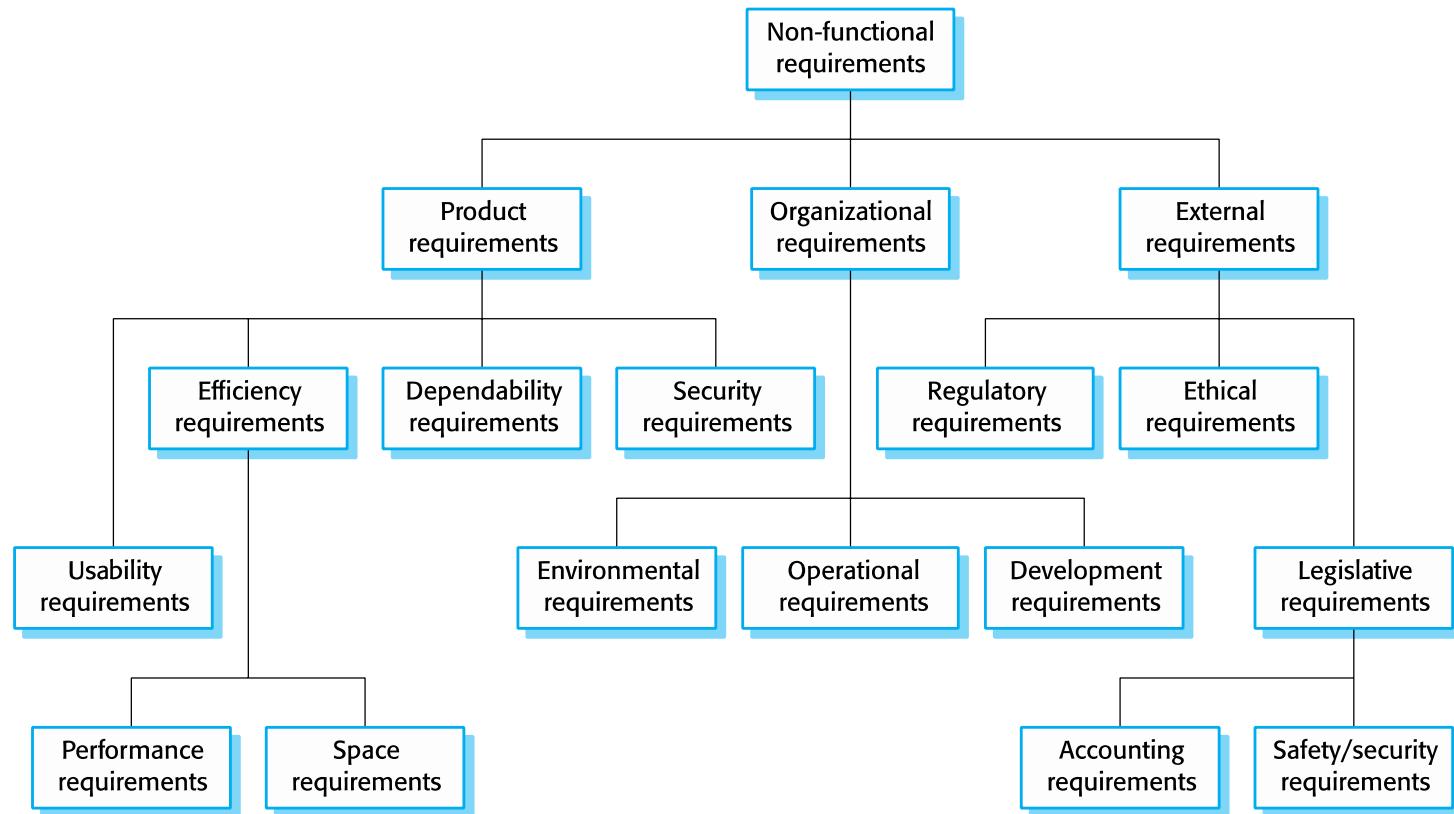
Non-functional requirements

These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.

Process requirements may also be specified mandating a particular IDE, programming language or development method.

Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

Types of nonfunctional requirement



Non-functional requirements implementation

Non-functional requirements may affect the overall architecture of a system rather than the individual components.

- For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.

A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.

- It may also generate requirements that restrict existing requirements.

Non-functional classifications

Product requirements

- Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.

Organisational requirements

- Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.

External requirements

- Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Examples of nonfunctional requirements in the Mentcare system

Product requirement

The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 0830–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement

Users of the Mentcare system shall authenticate themselves using their health authority identity card.

External requirement

The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Goals and requirements

Non-functional requirements may be very difficult to state precisely and imprecise requirements may be difficult to verify.

Goal

- A general intention of the user such as ease of use.

Verifiable non-functional requirement

- A statement using some measure that can be objectively tested.

Goals are helpful to developers as they convey the intentions of the system users.

Usability requirements

The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized. (Goal)

Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.
(Testable non-functional requirement)

Metrics for specifying nonfunctional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Software Requirement Document
(OR)
Software Requirement specification (SRS)

Requirements specification

Requirements specification

The process of writing down the user and system requirements in a requirements document.

User requirements have to be understandable by end-users and customers who do not have a technical background.

System requirements are more detailed requirements and may include more technical information.

The requirements may be part of a contract for the system development

- It is therefore important that these are as complete as possible.

Ways of writing a system requirements specification

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Requirements and design

In principle, requirements should state what the system should do and the design should describe how it does this.

In practice, requirements and design are inseparable

- A system architecture may be designed to structure the requirements;
- The system may inter-operate with other systems that generate design requirements;
- The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
- This may be the consequence of a regulatory requirement.

Natural language specification

Requirements are written as natural language sentences supplemented by diagrams and tables.

Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

Guidelines for writing requirements

Invent a standard format and use it for all requirements.

Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.

Use text highlighting to identify key parts of the requirement.

Avoid the use of computer jargon.

Include an explanation (rationale) of why a requirement is necessary.

Problems with natural language

Lack of clarity

- Precision is difficult without making the document difficult to read.

Requirements confusion

- Functional and non-functional requirements tend to be mixed-up.

Requirements amalgamation

- Several different requirements may be expressed together.

Example requirements for the insulin pump software system

- 3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

- 3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Structured specifications

An approach to writing requirements where the freedom of the requirements writer is limited and requirements are written in a standard way.

This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.

Form-based specifications

Definition of the function or entity.

Description of inputs and where they come from.

Description of outputs and where they go to.

Information about the information needed for the computation and other entities used.

Description of the action to be taken.

Pre and post conditions (if appropriate).

The side effects (if any) of the function.

A structured specification of a requirement for an insulin pump

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r_2); the previous two readings (r_0 and r_1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

A structured specification of a requirement for an insulin pump

Action

CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.

Requirements

Two previous readings so that the rate of change of sugar level can be computed.

Pre-condition

The insulin reservoir contains at least the maximum allowed single dose of insulin.

Post-condition r0 is replaced by r1 then r1 is replaced by r2.

Side effects None.

Tabular specification

Used to supplement natural language.

Particularly useful when you have to define a number of possible alternative courses of action.

For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

Tabular specification of computation for an insulin pump

Condition	Action
Sugar level falling ($r_2 < r_1$)	CompDose = 0
Sugar level stable ($r_2 = r_1$)	CompDose = 0
Sugar level increasing and rate of increase decreasing $((r_2 - r_1) < (r_1 - r_0))$	CompDose = 0
Sugar level increasing and rate of increase stable or increasing $((r_2 - r_1) \geq (r_1 - r_0))$	CompDose = round $((r_2 - r_1)/4)$ If rounded result = 0 then CompDose = MinimumDose

Use cases

Use-cases are a kind of scenario that are included in the UML.

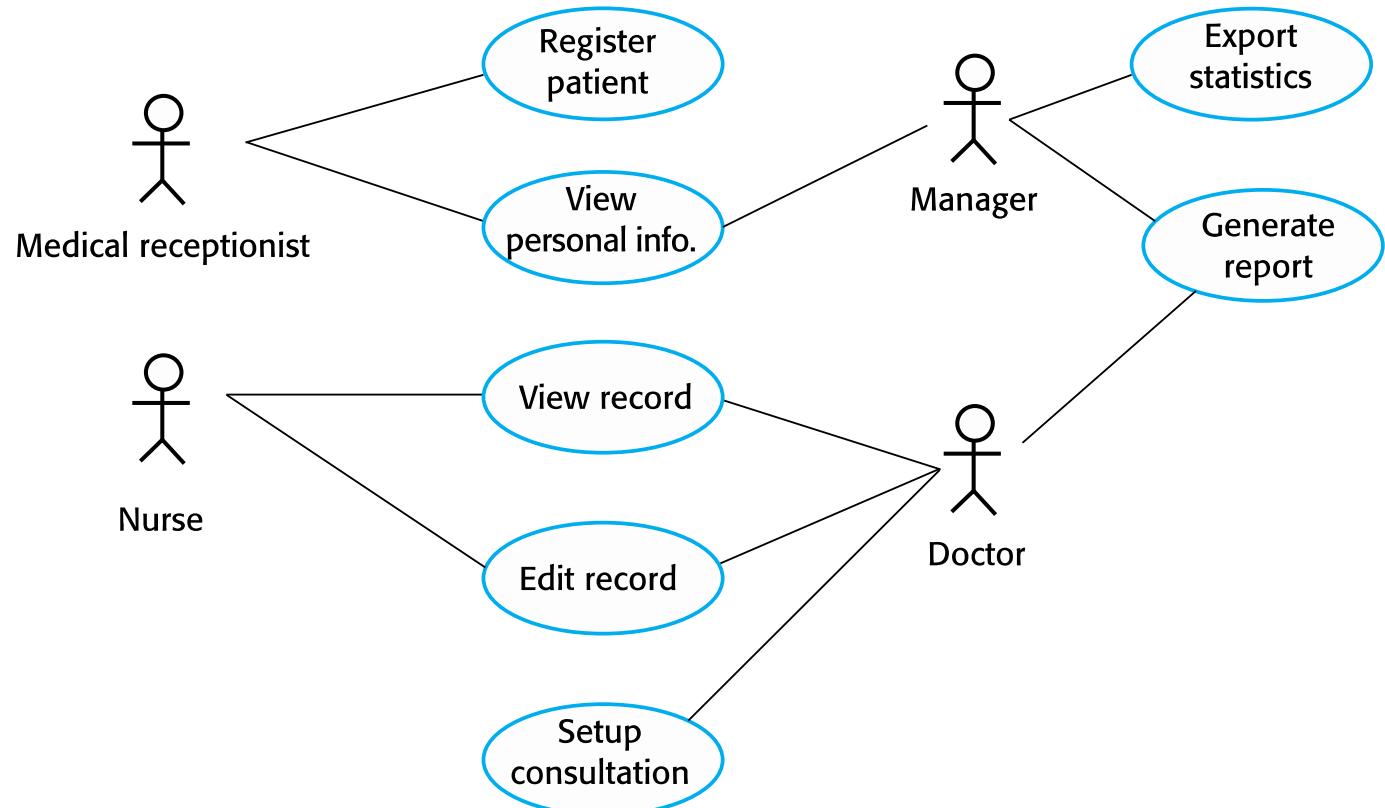
Use cases identify the actors in an interaction and which describe the interaction itself.

A set of use cases should describe all possible interactions with the system.

High-level graphical model supplemented by more detailed tabular description (see Chapter 5).

UML sequence diagrams may be used to add detail to use-cases by showing the sequence of event processing in the system.

Use cases for the Mentcare system



The software requirements document

The software requirements document is the official statement of what is required of the system developers.

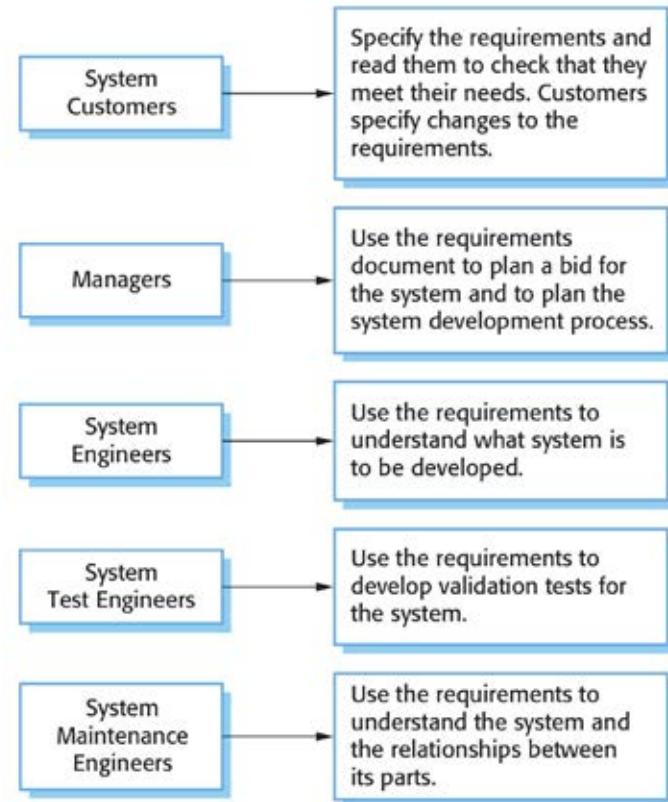
Should include both a definition of user requirements and a specification of the system requirements.

It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

SRS – Users of requirement doc

- official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software



Requirements document variability

Information in requirements document depends on type of system and the approach to development used.

Systems developed incrementally will, typically, have less detail in the requirements document.

Requirements documents standards have been designed e.g. IEEE standard. These are mostly applicable to the requirements for large systems engineering projects.

Structure of requirement doc

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Requirements engineering processes

Requirements engineering processes

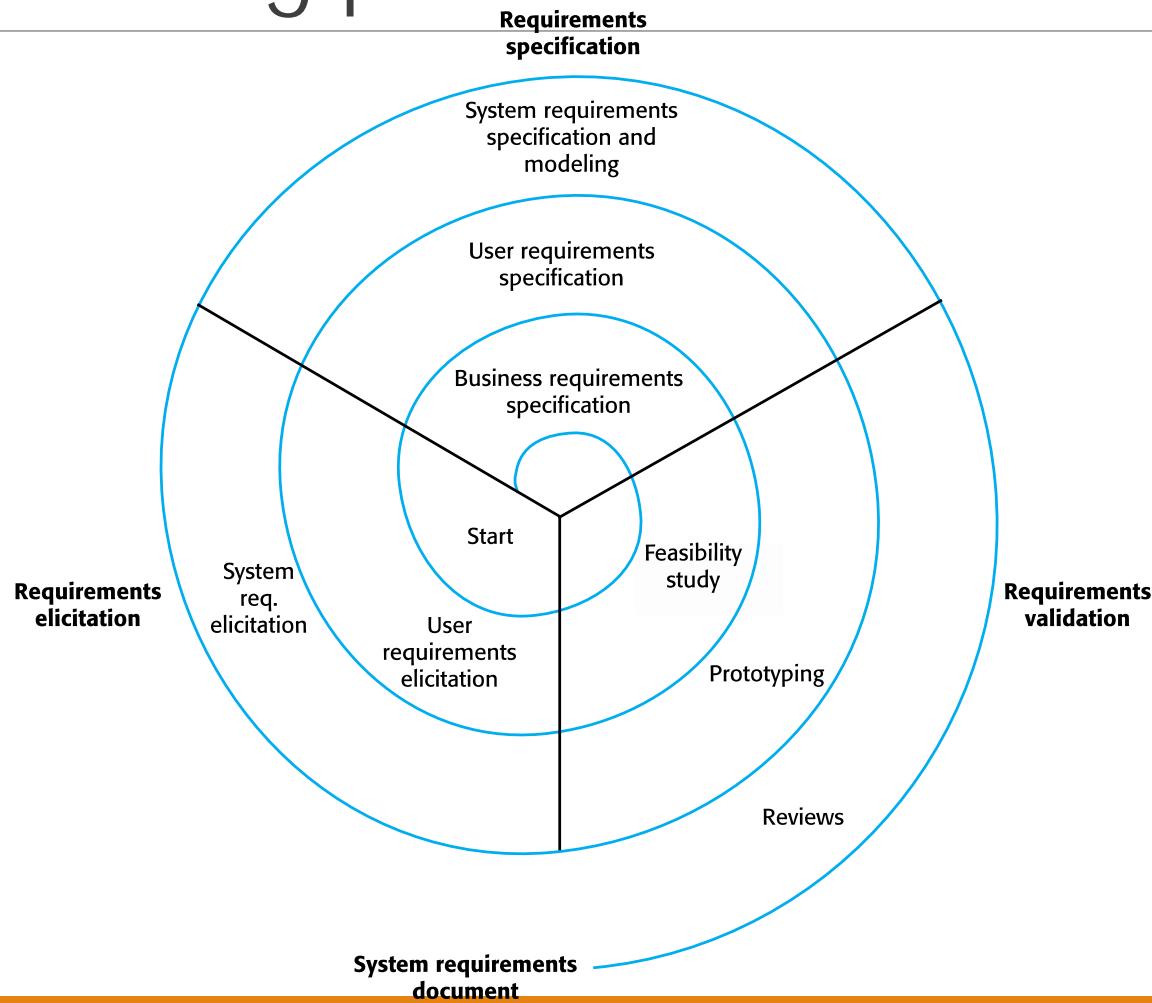
The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.

However, there are a number of generic activities common to all processes

1. Requirements elicitation;
2. Requirements analysis;
3. Requirements validation;
4. Requirements management.

In practice, RE is an iterative activity in which these processes are interleaved.

A spiral view of the requirements engineering process



1. Requirements elicitation

2. Requirements elicitation and analysis

Sometimes called requirements elicitation or requirements discovery.

Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.

May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders*.

Requirements elicitation

Software engineers work with a range of system stakeholders to find out about the application domain, the services that the system should provide, the required system performance, hardware constraints, other systems, etc.

Stages include:

- Requirements discovery,
- Requirements classification and organization,
- Requirements prioritization and negotiation,
- Requirements specification.

Problems of requirements elicitation

Stakeholders don't know what they really want.

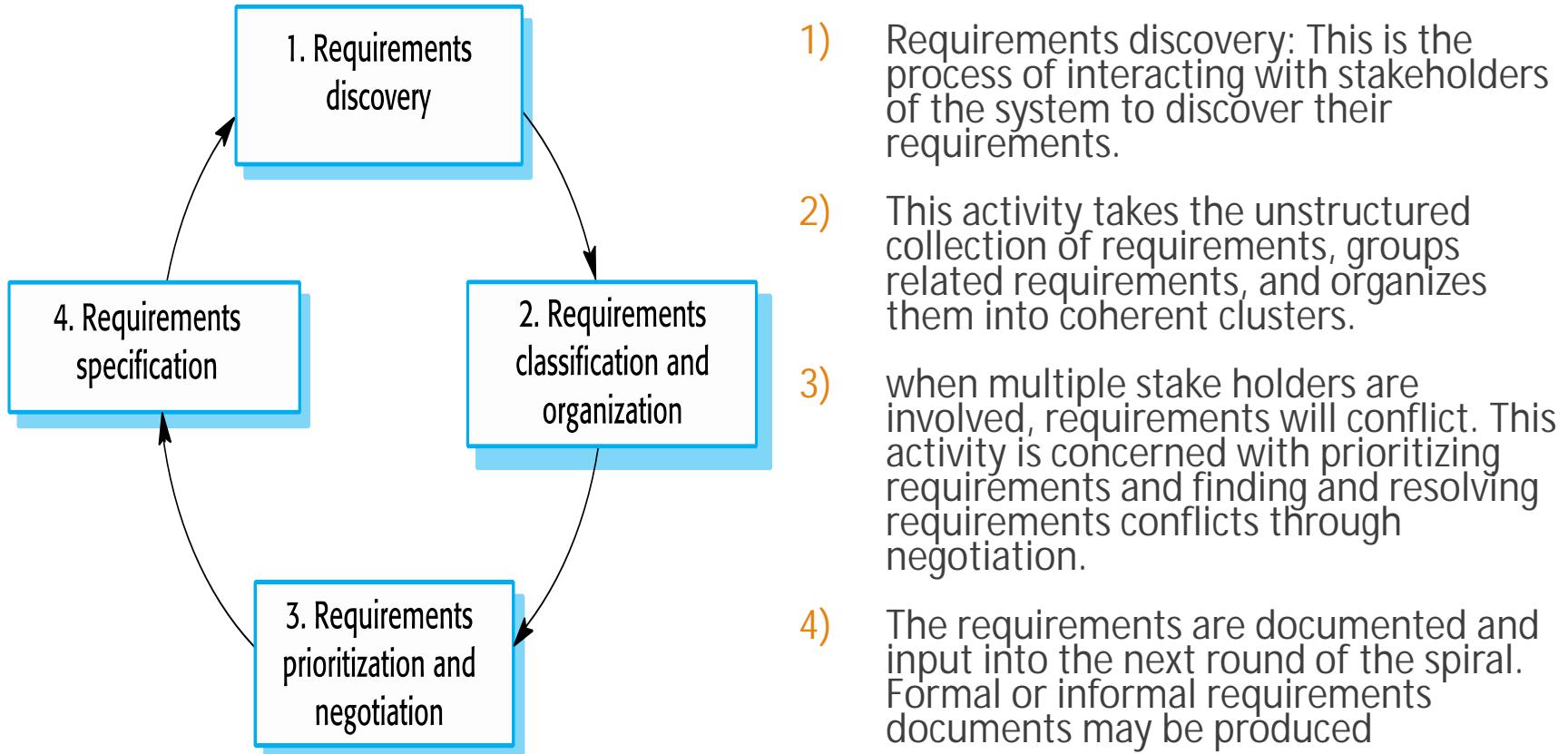
Stakeholders express requirements in their own terms.

Different stakeholders may have conflicting requirements.

Organisational and political factors may influence the system requirements.

The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

The requirements elicitation and analysis process



Process activities

Requirements discovery

- Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

Requirements classification and organisation

- Groups related requirements and organises them into coherent clusters.

Prioritisation and negotiation

- Prioritising requirements and resolving requirements conflicts.

Requirements specification

- Requirements are documented and input into the next round of the spiral.

Requirements discovery

The process of gathering information about the required and existing systems and distilling the user and system requirements from this information.

Interaction is with system stakeholders from managers to external regulators.

Systems normally have a range of stakeholders.

Interviewing

Formal or informal interviews with stakeholders are part of most RE processes.

Types of interview

- Closed interviews based on pre-determined list of questions
- Open interviews where various issues are explored with stakeholders.

Effective interviewing

- Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
- Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

Interviews in practice

Normally a mix of closed and open-ended interviewing.

Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.

Interviewers need to be open-minded without pre-conceived ideas of what the system should do

You need to prompt the user to talk about the system by suggesting requirements rather than simply asking them what they want.

Problems with interviews

Application specialists may use language to describe their work that isn't easy for the requirements engineer to understand.

Interviews are not good for understanding domain requirements

- Requirements engineers cannot understand specific domain terminology;
- Some domain knowledge is so familiar that people find it hard to articulate or think that it isn't worth articulating.

Ethnography

A social scientist spends a considerable time observing and analysing how people actually work.

People do not have to explain or articulate their work.

Social and organisational factors of importance may be observed.

Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Scope of ethnography

Requirements that are derived from the way that people actually work rather than the way in which process definitions suggest that they ought to work.

Requirements that are derived from cooperation and awareness of other people's activities.

- Awareness of what other people are doing leads to changes in the ways in which we do things.

Ethnography is effective for understanding existing processes but cannot identify new features that should be added to a system.

Focused ethnography

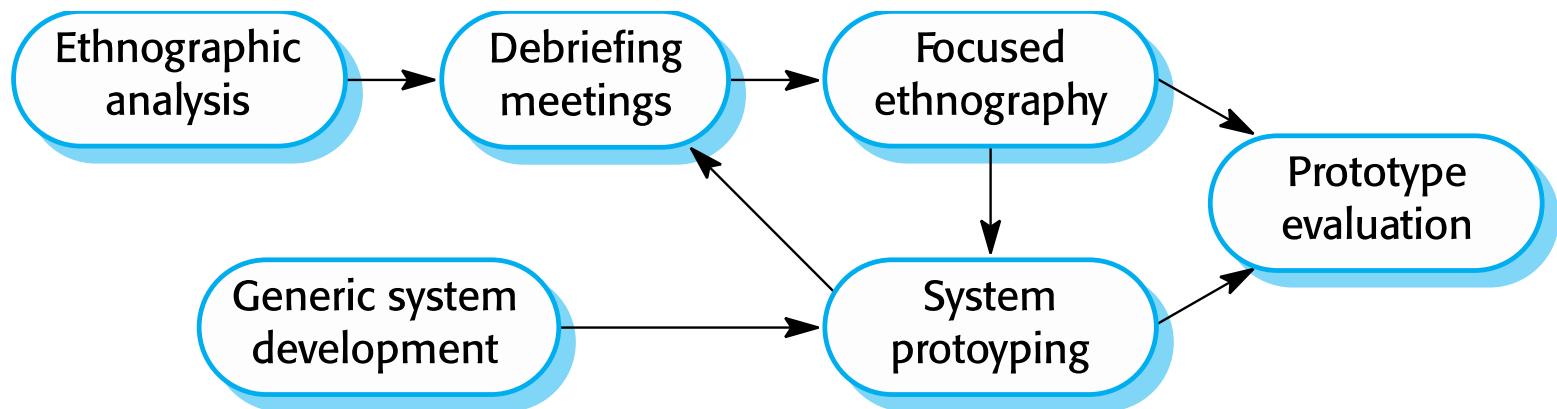
Developed in a project studying the air traffic control process

Combines ethnography with prototyping

Prototype development results in unanswered questions which focus the ethnographic analysis.

The problem with ethnography is that it studies existing practices which may have some historical basis which is no longer relevant.

Ethnography and prototyping for requirements analysis



Stories and scenarios

Scenarios and user stories are real-life examples of how a system can be used.

Stories and scenarios are a description of how a system may be used for a particular task.

Because they are based on a practical situation, stakeholders can relate to them and can comment on their situation with respect to the story.

Photo sharing in the classroom (iLearn)

Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused around the fishing industry in the area, looking at the history, development and economic impact of fishing. As part of this, pupils are asked to gather and share reminiscences from relatives, use newspaper archives and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo sharing site as he wants pupils to take and comment on each others' photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers group, which he is a member of to see if anyone can recommend an appropriate system. Two teachers reply and both suggest that he uses KidsTakePics, a photo sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

Scenarios

A structured form of user story

Scenarios should include

- A description of the starting situation;
- A description of the normal flow of events;
- A description of what can go wrong;
- Information about other concurrent activities;
- A description of the state when the scenario finishes.

Uploading photos iLearn)

Initial assumption: A user or a group of users have one or more digital photographs to be uploaded to the picture sharing site. These are saved on either a tablet or laptop computer. They have successfully logged on to KidsTakePics.

Normal: The user chooses upload photos and they are prompted to select the photos to be uploaded on their computer and to select the project name under which the photos will be stored. They should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.

On completion of the upload, the system automatically sends an email to the project moderator asking them to check new content and generates an on-screen message to the user that this has been done.

Uploading photos

What can go wrong:

No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed that there could be a delay in making their photos visible.

Photos with the same name have already been uploaded by the same user. The user should be asked if they wish to re-upload the photos with the same name, rename the photos or cancel the upload. If they chose to re-upload the photos, the originals are overwritten. If they chose to rename the photos, a new name is automatically generated by adding a number to the existing file name.

Other activities: The moderator may be logged on to the system and may approve photos as they are uploaded.

System state on completion: User is logged on. The selected photos have been uploaded and assigned a status 'awaiting moderation'. Photos are visible to the moderator and to the user who uploaded them.

3. Requirements validation

Requirements validation

Concerned with demonstrating that the requirements define the system that the customer really wants.

Requirements error costs are high so validation is very important

- Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking

Validity. Does the system provide the functions which best support the customer's needs?

Consistency. Are there any requirements conflicts?

Completeness. Are all functions required by the customer included?

Realism. Can the requirements be implemented given available budget and technology

Verifiability. Can the requirements be checked?

Requirements validation techniques

Requirements reviews

- Systematic manual analysis of the requirements.

Prototyping

- Using an executable model of the system to check requirements. Covered in Chapter 2.

Test-case generation

- Developing tests for requirements to check testability.

Requirements reviews

Regular reviews should be held while the requirements definition is being formulated.

Both client and contractor staff should be involved in reviews.

Reviews may be formal (with completed documents) or informal.
Good communications between developers, customers and users can resolve problems at an early stage.

Review checks

Verifiability

- Is the requirement realistically testable?

Comprehensibility

- Is the requirement properly understood?

Traceability

- Is the origin of the requirement clearly stated?

Adaptability

- Can the requirement be changed without a large impact on other requirements?

4. Requirements change & Management

Changing requirements

The business and technical environment of the system always changes after installation.

- New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.

The people who pay for a system and the users of that system are rarely the same people.

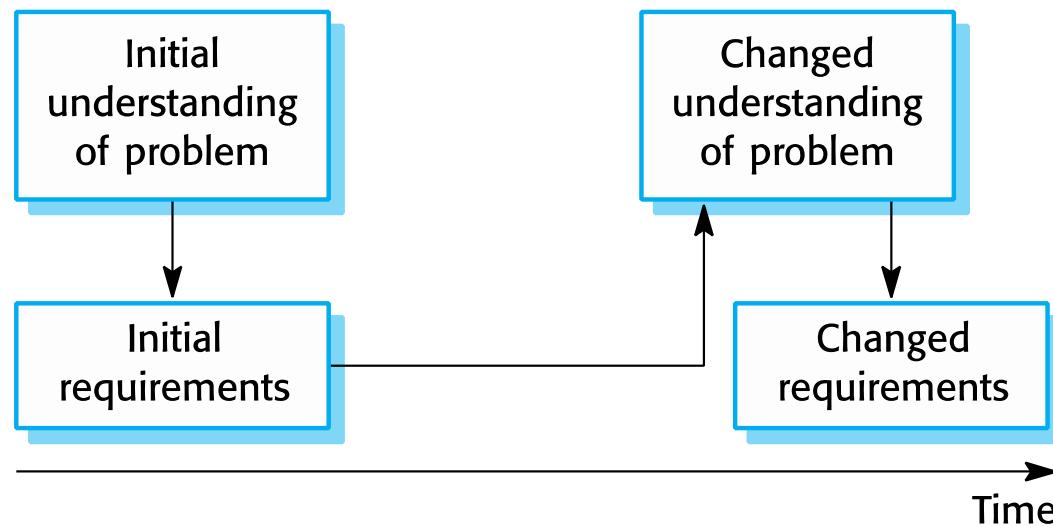
- System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

Changing requirements

Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.

- The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements evolution



Requirements management

Requirements management is the process of managing changing requirements during the requirements engineering process and system development.

New requirements emerge as a system is being developed and after it has gone into use.

You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements.

Requirements management planning

Establishes the level of requirements management detail that is required.

Requirements management decisions:

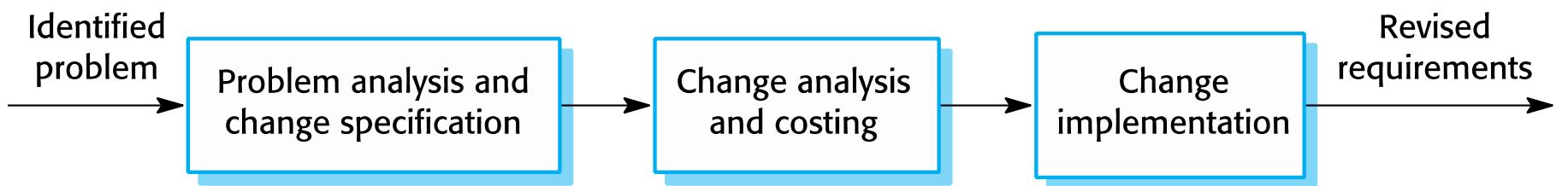
- *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
- *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
- *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
- *Tool support* Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements change management

Deciding if a requirements change should be accepted

- *Problem analysis and change specification*
 - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
- *Change analysis and costing*
 - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
- *Change implementation*
 - The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

Requirements change management



Key points

Requirements for a software system set out what the system should do and define constraints on its operation and implementation.

Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.

Non-functional requirements often constrain the system being developed and the development process being used.

They often relate to the emergent properties of the system and therefore apply to the system as a whole.

Key points

The requirements engineering process is an iterative process that includes requirements elicitation, specification and validation.

Requirements elicitation is an iterative process that can be represented as a spiral of activities – requirements discovery, requirements classification and organization, requirements negotiation and requirements documentation.

You can use a range of techniques for requirements elicitation including interviews and ethnography. User stories and scenarios may be used to facilitate discussions.

Key points

Requirements specification is the process of formally documenting the user and system requirements and creating a software requirements document.

The software requirements document is an agreed statement of the system requirements. It should be organized so that both system customers and software developers can use it.

Key points

Requirements validation is the process of checking the requirements for validity, consistency, completeness, realism and verifiability.

Business, organizational and technical changes inevitably lead to changes to the requirements for a software system. Requirements management is the process of managing and controlling these changes.

Chapter 26

n **Estimation for Software Projects**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 7/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Software Project Planning

The overall goal of project planning is to establish a pragmatic strategy for controlling, tracking, and monitoring a complex technical project.

Why?

So the end result gets done on time, with quality!

Project Planning Task Set-I

- Establish project scope
- Determine feasibility
- Analyze risks
 - Risk analysis is considered in detail in Chapter 25.
- Define required resources
 - Determine require human resources
 - Define reusable software resources
 - Identify environmental resources

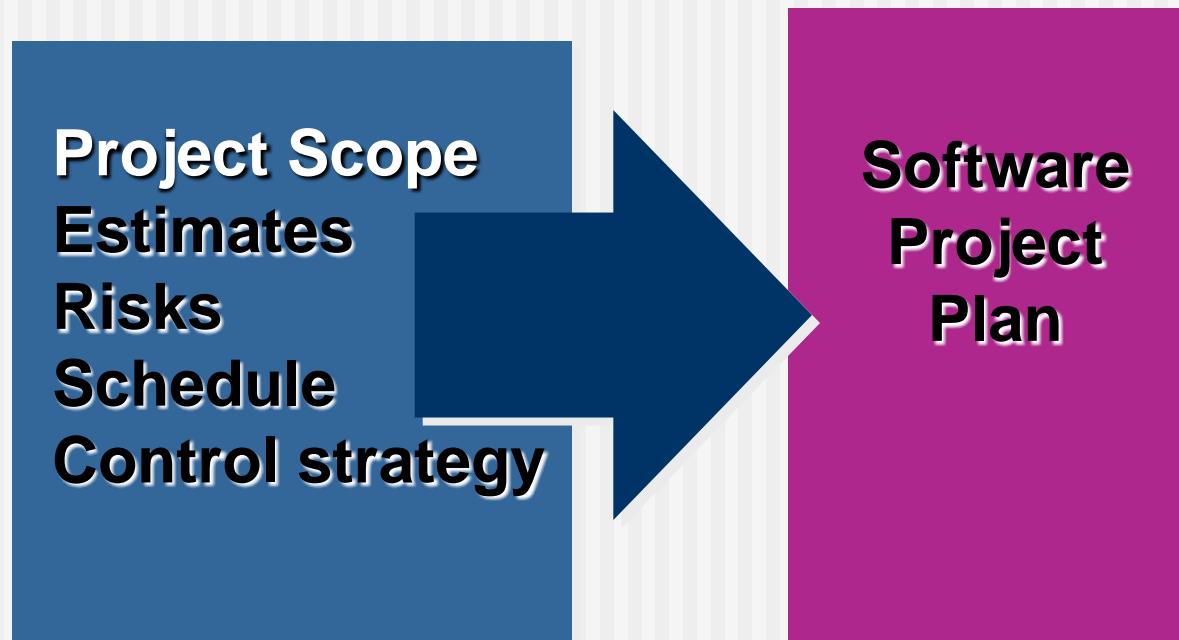
Project Planning Task Set-II

- n Estimate cost and effort
 - n Decompose the problem
 - n Develop two or more estimates using size, function points, process tasks or use-cases
 - n Reconcile the estimates
- n Develop a project schedule
 - n Scheduling is considered in detail in Chapter 27.
 - Establish a meaningful task set
 - Define a task network
 - Use scheduling tools to develop a timeline chart
 - Define schedule tracking mechanisms

Estimation

- n Estimation of resources, cost, and schedule for a software engineering effort requires
 - n experience
 - n access to good historical information (metrics)
 - n the courage to commit to quantitative predictions when qualitative information is all that exists
- n Estimation carries inherent risk and this risk leads to uncertainty

Write it Down!



To Understand Scope ...

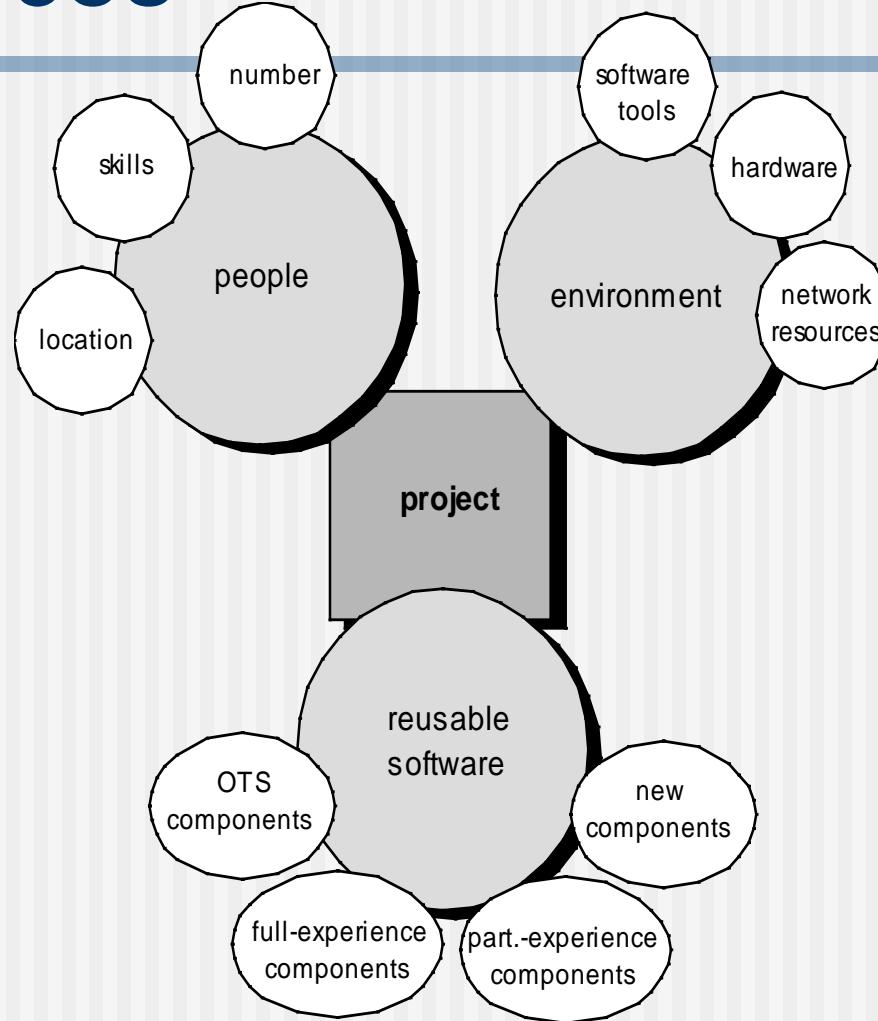
- n Understand the customers needs
- n understand the business context
- n understand the project boundaries
- n understand the customer's motivation
- n understand the likely paths for change
- n understand that ...

***Even when you understand,
nothing is guaranteed!***

What is Scope?

- n **Software scope** describes
 - n the functions and features that are to be delivered to end-users
 - n the data that are input and output
 - n the “content” that is presented to users as a consequence of using the software
 - n the performance, constraints, interfaces, and reliability that *bound* the system.
- n Scope is defined using one of two techniques:
 - A narrative description of software scope is developed after communication with all stakeholders.
 - A set of use-cases is developed by end-users.

Resources



Project Estimation



- Project scope must be understood
- Elaboration (decomposition) is necessary
- Historical metrics are very helpful
- At least two different techniques should be used
- Uncertainty is inherent in the process

Estimation Techniques

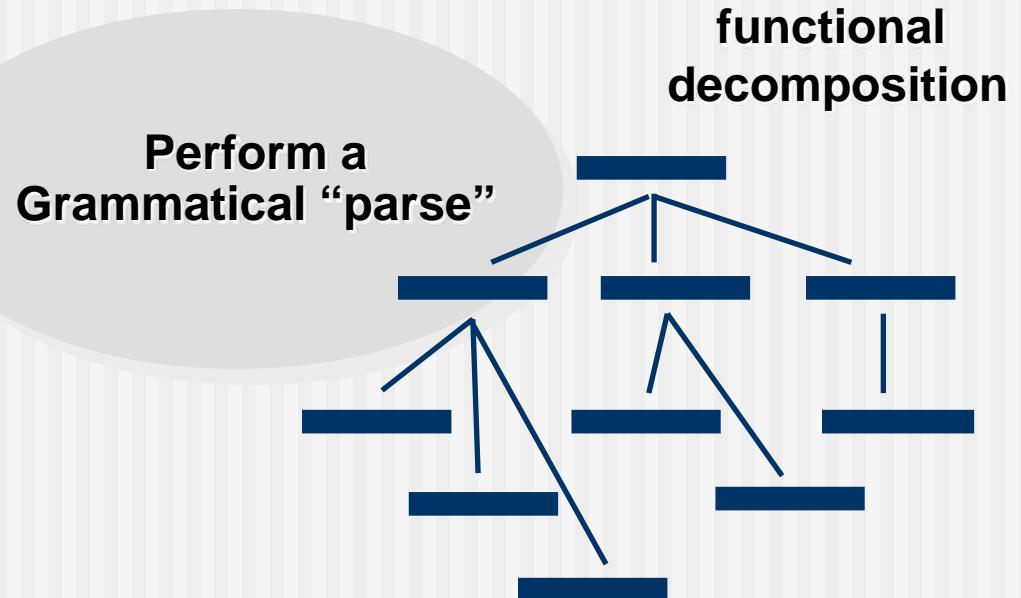
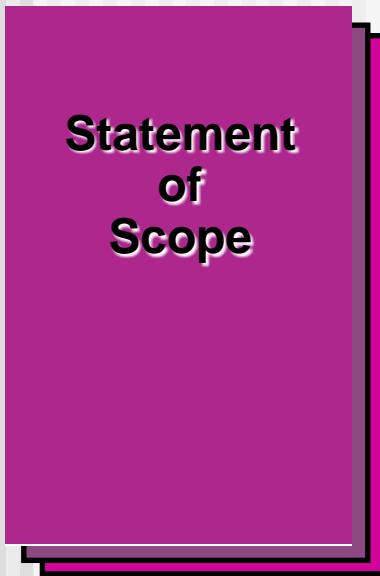
- Past (similar) project experience
- Conventional estimation techniques
 - task breakdown and effort estimates
 - size (e.g., FP) estimates
- Empirical models
- Automated tools



Estimation Accuracy

- Predicated on ...
 - the degree to which the planner has properly estimated the size of the product to be built
 - the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects)
 - the degree to which the project plan reflects the abilities of the software team
 - the stability of product requirements and the environment that supports the software engineering effort.

Functional Decomposition



Conventional Methods: LOC/FP Approach

- compute LOC/FP using estimates of information domain values
- use historical data to build estimates for the project

Example: LOC Approach

Function	Estimated LOC
user interface and control facilities (UI/CF)	2,300
two-dimensional geometric analysis (2D GA)	5,300
three-dimensional geometric analysis (3D GA)	6,800
database management (DBM)	3,300
computer graphics display facilities (CGDF)	4,900
peripheral control (PC)	2,100
design analysis modules (DAM)	8,400
<i>estimated lines of code</i>	33,200

Average productivity for systems of this type = 620 LOC/pm.

Burdened labor rate = \$8000 per month, the cost per line of code is approximately \$13.

Based on the LOC estimate and the historical productivity data, the total estimated project cost is **\$431,000 and the estimated effort is 54 person-months.**

Example: FP Approach

Information Domain Value	opt.	Likely	pess.	est. count	weight	FP-count
number of inputs	20	24	30	24	4	97
number of outputs	12	18	22	16	5	78
number of inquiries	16	22	28	22	5	68
number of files	4	4	8	4	10	42
number of external interfaces	2	2	3	2	7	15
count-total						321

The estimated number of FP is derived:

$$FP_{estimated} = \text{count-total} \cdot [0.65 + 0.01 \cdot \sum S(F_i)]$$

$$FP_{estimated} = 375$$

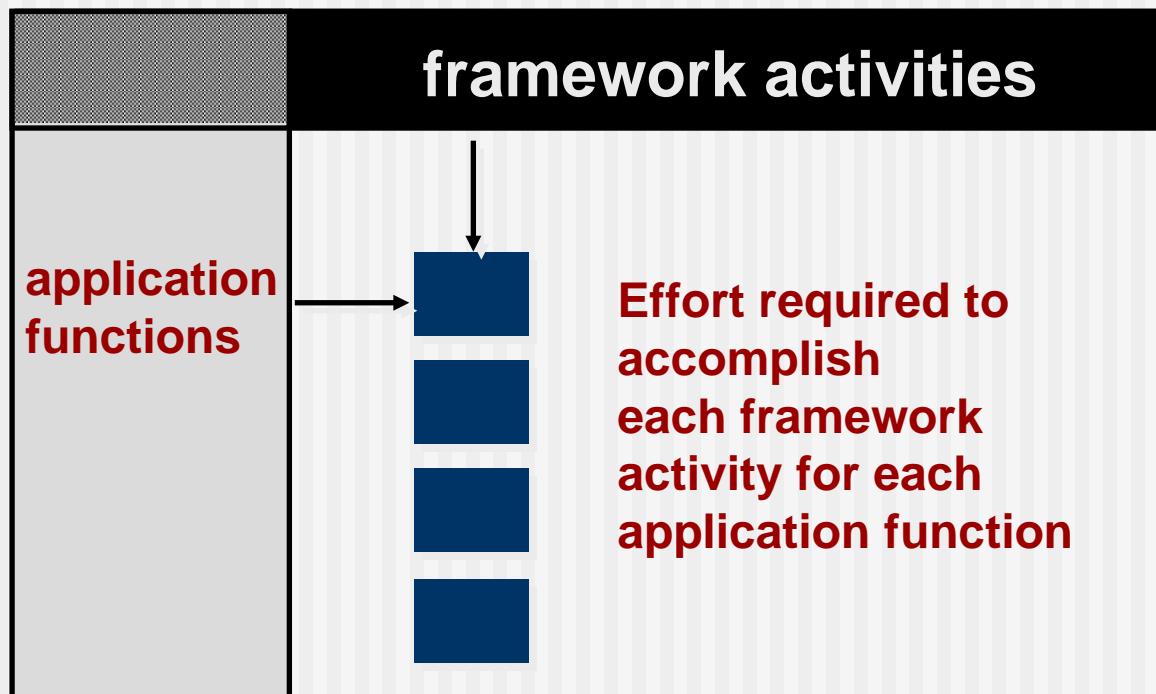
organizational average productivity = 6.5 FP/pm.

burdened labor rate = \$8000 per month, approximately \$1230/FP.

Based on the FP estimate and the historical productivity data, **total estimated project cost is \$461,000 and estimated effort is 58 person-months.**

Process-Based Estimation

Obtained from “process framework”



Process-Based Estimation Example

Activity →	CC	Planning	Risk Analysis	Engineering		Construction Release		CE	Totals
Task →				analysis	design	code	test		
Function									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DSM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

Based on an average burdened labor rate of \$8,000 per month, **the total estimated project cost is \$368,000 and the estimated effort is 46 person-months.**

Tool-Based Estimation

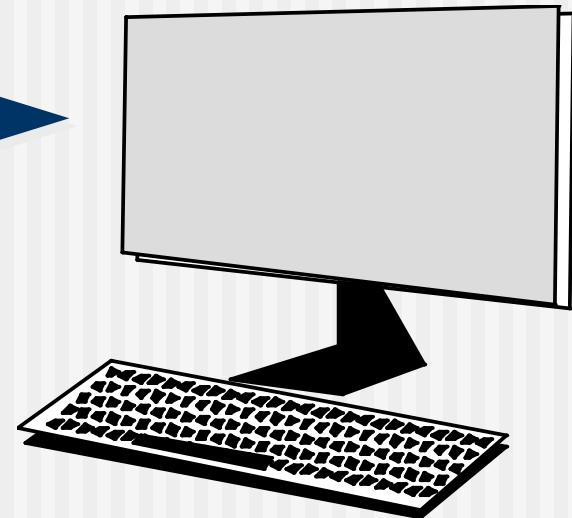
project characteristics



calibration factors



LOC/FP data



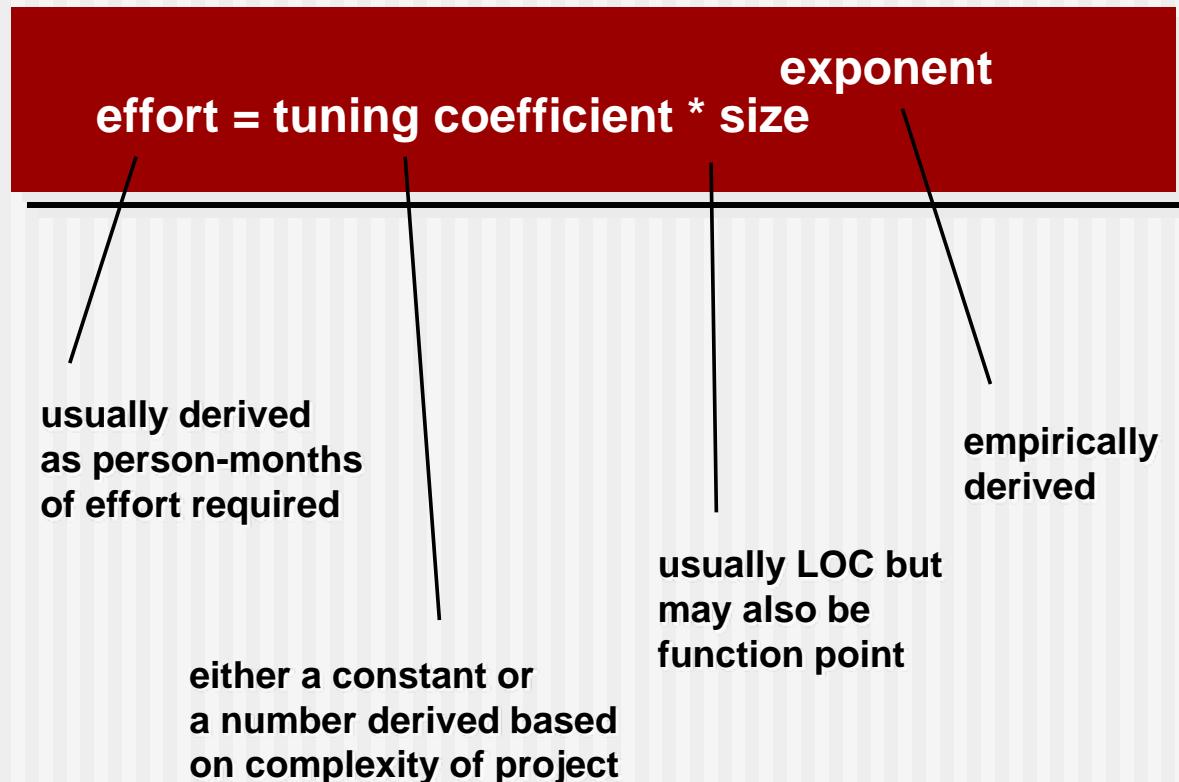
Estimation with Use-Cases

	use cases	scenarios	pages	scenarios	pages	LOC	LOC estimate
User interface subsystem	6	10	6	12	5	560	3,366
Engineering subsystem group	10	20	8	16	8	3100	31,233
Infrastructure subsystem group	5	6	5	10	6	1650	7,970
Total LOC estimate				Ê	Ê	Ê	Ê
				Ê	Ê	Ê	42,568

Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of \$8000 per month, the cost per line of code is approximately \$13. Based on the use-case estimate and the historical productivity data, **the total estimated project cost is \$552,000 and the estimated effort is 68 person-months.**

Empirical Estimation Models

General form:



COCOMO-II

- n COCOMO II is actually a hierarchy of estimation models that address the following areas:
 - *Application composition model.* Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
 - *Early design stage model.* Used once requirements have been stabilized and basic software architecture has been established.
 - *Post-architecture-stage model.* Used during the construction of the software.

The Software Equation

A dynamic multivariable model

$$E = [LOC \times B^{0.333}/P]^3 \times (1/t^4)$$

where

E = effort in person-months or person-years

t = project duration in months or years

B = “special skills factor”

P = “productivity parameter”

Estimation for OO Projects-I

- n Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
- n Using object-oriented requirements modeling (Chapter 6), develop use-cases and determine a count.
- n From the analysis model, determine the number of key classes (called analysis classes in Chapter 6).
- n Categorize the type of interface for the application and develop a multiplier for support classes:

Interface type	Multiplier
n No GUI	2.0
n Text-based user interface	2.25
n GUI	2.5
n Complex GUI	3.0

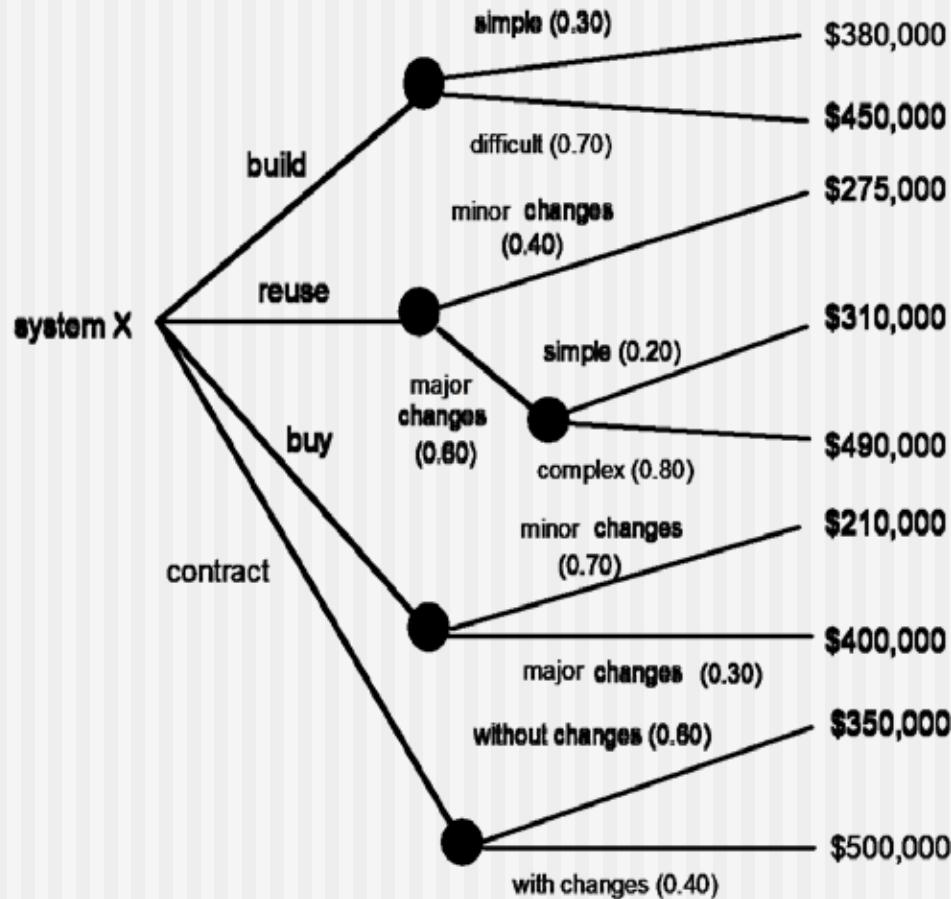
Estimation for OO Projects-II

- Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.
- Multiply the total number of classes (key + support) by the average number of work-units per class. Lorenz and Kidd suggest 15 to 20 person-days per class.
- Cross check the class-based estimate by multiplying the average number of work-units per use-case

Estimation for Agile Projects

- n Each user scenario (a mini-use-case) is considered separately for estimation purposes.
- n The scenario is decomposed into the set of software engineering tasks that will be required to develop it.
- n Each task is estimated separately. Note: estimation can be based on historical data, an empirical model, or “experience.”
 - n Alternatively, the ‘volume’ of the scenario can be estimated in LOC, FP or some other volume-oriented measure (e.g., use-case count).
- n Estimates for each task are summed to create an estimate for the scenario.
 - n Alternatively, the volume estimate for the scenario is translated into effort using historical data.
- n 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.

The Make-Buy Decision



Computing Expected Cost

expected cost =

$$\sum_i (\text{path probability}_i \times \text{estimated path cost}_i)$$

For example, the expected cost to build is:

$$\begin{aligned}\text{expected cost}_{\text{build}} &= 0.30 (\$380K) + 0.70 (\$450K) \\ &= \$429 K\end{aligned}$$

similarly,

$$\text{expected cost}_{\text{reuse}} = \$382K$$

$$\text{expected cost}_{\text{buy}} = \$267K$$

$$\text{expected cost}_{\text{contr}} = \$410K$$

PROBLEMS

Problem 1- wide band delphi method

Given,

Optimistic LOC, $S_{opt} = 2600$

Pessimistic LOC, $S_{pess} = 4000$

Most Likely LOC, $S_m = 3800$

Calculate the expected value for the estimation variable

Problem 1 Solution

The expected value of the estimation variable,

$$\begin{aligned} S &= (S_{\text{opt}} + 4^* S_{\text{m}} + S_{\text{pess}}) / 6 \\ &= (2600 + 4^* 3800 + 4000) / 6 \\ S &= 3633 \end{aligned}$$

Problem 2

Given

- The estimated LOC count is 56,100
- From the review of historical data, below two values were derived
 - Average productivity for this category of systems = 693 LOC/pm
 - Burdened labor rate = \$8000/month

Compute the below,

- Cost per LOC
- Total Estimated Project Cost
- Estimated Effort in person months

Problem 2 Solution

- Cost per LOC = **Labor rate per month/LOC per pm**
 $= 8000/693 = \$11.5$
- Total Estimated Project Cost = **Estimated LOC * Cost per LOC**
 $= 56,100 * 11.5 = \$6,45,150$
- Estimated Effort in pm = **Total Estimated Project Cost/ Labor rate per month**
 $= 6,45,150/8000 = 81 \text{ person months}$

Function Points

Information Domain Value	Count	Weighting factor			=	
		simple	average	complex		
External Inputs (EI)	<input type="text"/>	3	3	4	6	<input type="text"/>
External Outputs (EO)	<input type="text"/>	3	4	5	7	<input type="text"/>
External Inquiries (EQ)	<input type="text"/>	3	3	4	6	<input type="text"/>
Internal Logical Files (ILF)	<input type="text"/>	3	7	10	15	<input type="text"/>
External Interface Files (EIF)	<input type="text"/>	3	5	7	10	<input type="text"/>
Count total						<input type="text"/>

These slides are designed to accompany *Software Engineering: A Practitioner's Approach*, 7/e (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

Problem 3- Function Point

Information Domain Value Count	Opt	Likely	Pess	Estimated Count	Weight Average	FP Count
Inputs	20	24	34	?	4	?
Outputs	12	17	22	?	5	?
Inquiries	16	26	33	?	5	?
Files	4	5	7	?	10	?
Interfaces	2	4	6	?	7	?
Count Total						

Given the above table, with default simple weight for all categories and the Complexity Factor = 1.19.

compute

- The estimated count for each value
- Total Unadjusted FP
- Adjusted FP Count

Problem 3 - Contd

- Also, if the below historical data are given,
- Organizational Average Productivity of this type of system = 5.8 FP/m
- Burdened Labor Rate = \$8000/m
- Compute
 - Cost per FP
 - Total Estimated Project Cost
 - Estimated Effort in person months

Problem 3 - Solution

Information Domain Value Count	Opt	Likely	Pess	Estimated Count	Weight	FP Count
Inputs	20	24	34	24	4	96
Outputs	12	17	22	17	5	85
Inquiries	16	26	33	26	5	130
Files	4	5	7	5	10	50
Interfaces	2	4	6	4	7	28
Count Total						389

$$UFP(\text{Unadjusted FP}) = 389$$

$$VAF(\text{value adjustment factors}) = 0.65 + 0.01 * \sum F_i = \\ 0.65 + 0.01 * 1.19 = 0.66$$

$$AFP(\text{Adjusted FP Count}) = UFP * VAF = 389 * 0.66 \\ = 257$$

Problem 3 – Solution Contd

- Cost per FP = Labor rate per month/FP per month
 $= 8000/5.8 = \$1379$
- Total Estimated Project Cost = AFP * Cost per FP = $257 * 1379 = \$ 3,54,403$
- Estimated Effort in person months = Total Estimated Project Cost/Labor rate per month = $3,54,403/8000 = 44 \text{ pm}$

Problem 4- COCOMO II

For the given data, compute the effort in person months using the COCOMO model

1. Objects' Count

Object Type	Count of each object		
	Simple	Medium	Difficult
Screen	5	4	5
Report	6	7	5
3GL Comp			6

2. Objects' Complexity Weight

Object Type	Complexity Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Comp			10

Also given,
Percentage of
reuse is 30%,
Value of PROD =
7

Compute the
effort in person
months.

Problem 4 - Solution

Object Type	Object Points Calculation			Count
	Simple	Medium	Difficult	
Screen	5 * 1	4 * 2	5 * 3	28
Report	6 * 2	7 * 5	5 * 8	87
3GL Comp			6 * 10	60

Value of Productivity rate
v.Low- 4
Low-7
Normal – 13
High -25
V.High -50

$$\text{Object Points Count} = 28 + 87 + 60 = 175$$

$$\text{NOP} = \text{Object Points} * [(100 - \% \text{ reuse})/100] = 175 * 70/100 = 122.5$$

$$\text{Estimated Effort} = \text{NOP/PROD} = 122.5/7 (\text{LOW}) = 17.5 \text{ pm}$$

Problem 5

Using the simplified **Software Equation Model**, compute the effort in person months, given the below data,

- Estimated LOC = 33580
- Productivity Parameter, P = 12,000
- Special Skills Factor ,B = 0.28

Problem 5 Solution

Simplified Software Equation Estimation Model

$$t_{\min} = 8.14 \cdot (\text{LOC}/P)^{0.43}$$

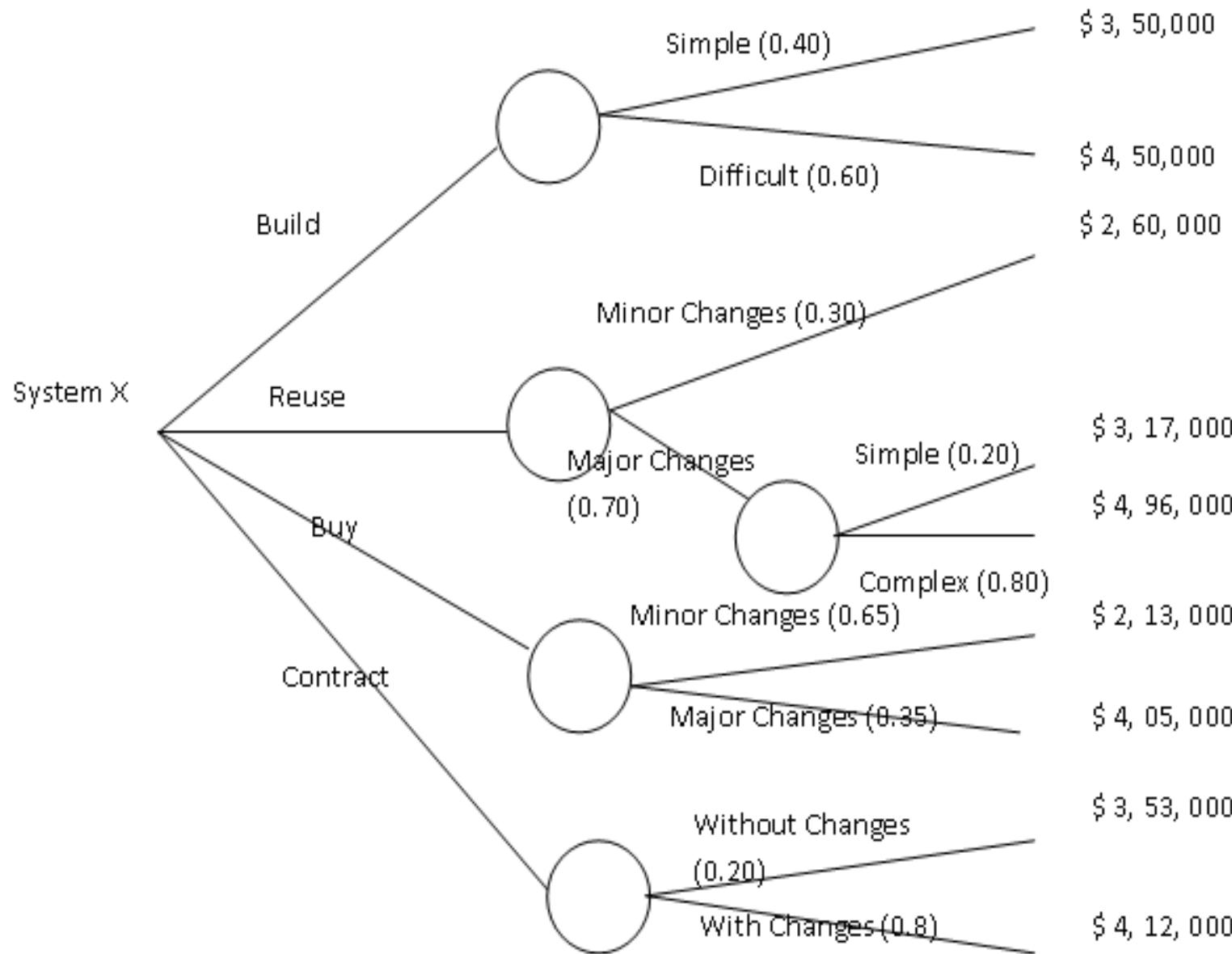
$$E = 180 * B * t^3$$

Substituting,

$$t_{\min} = 8.14 \cdot (33580/12000)^{0.43} = 8.14 \cdot (2.8)^{0.43} = 8.14 \cdot 1.6 = 13 \text{ pm} = 1.08 \text{ person years}$$

$$E = 180 * 0.28 * (1.08)^3 = 63.5 \text{ pm}$$

Problem 6-Make /buy decision



Problem 6 - Solution

Expected Cost = \sum (Path Probability_i * Estimated Path Cost_i)

$$\begin{aligned}\text{Expected Cost}_{\text{build}} &= 1,40,000 + 2,70,000 \\ &= \$ 4,10,000\end{aligned}$$

$$\begin{aligned}\text{Expected Cost}_{\text{reuse}} &= 78000 + 0.70 [0.20 * 3,17,000 + 0.80 \\ &\quad * 4,96,000] = \$ 4,00,140\end{aligned}$$

$$\text{Expected Cost}_{\text{buy}} = 1,38,450 + 1,41,750 = \$ 2,80,200$$

$$\text{Expected Cost}_{\text{contract}} = 70600 + 329600 = \$ 4,00,200$$

The decision would be to buy the software.

Problem 7

Given,

Probability of occurrence of a risk, $P = 0.5$

Cost to the project if the risk occurs,

$C = \$ 20,600$

Calculate Risk Exposure

Problem 7 - Solution

$$\text{Risk Exposure} = P * C$$

$$= 0.5 * 20,600$$

$$= \$10,300$$

Problem 8

Adaptation Criteria	Grade	Weight	New Dev – Entry Point Multiplier
Size	2	1.2	1
No of Users	3	1.1	1
Business Criticality	4	1.1	1
Longevity	3	0.9	1
Req. Stability	4	1.2	1
Ease of Communication	2	0.9	1
Maturity of Tech	2	0.9	1
Perf Constraints	3	0.8	1
Embedded/Non Emb.	3	1.2	1
Project Staffing	1	1.0	1
Interoperability	3	1.1	1
Engg. Factors	0	1.2	0

Given, the below adaptation criteria, graded for a new dev project, with low risk, Calculate TSS and conclude on the degree of rigor that you choose for applying process.

Problem 8 - Solution

Adaptation Criteria	Grade	Weight	New Dev – Entry Point Multiplier	TTS
Size	2	1.2	1	2.4
No of Users	3	1.1	1	3.3
Business Criticality	4	1.1	1	4.4
Longevity	3	0.9	1	2.7
Req. Stability	4	1.2	1	4.8
Ease of Communication	2	0.9	1	1.8
Maturity of Tech	2	0.9	1	1.8
Perf Constraints	3	0.8	1	2.4
Embedded/Non Emb.	3	1.2	1	3.6
Project Staffing	1	1.0	1	1
Interoperability	3	1.1	1	3.3
Engg. Factors	0	1.2	0	0

Average TSS
= 2.7

TSS < 1.24 – Casual
TSS between 1 and 3 – Structured
TSS > 2.4 – Strict

So
structured,
application
is chosen

Problem 9

Task ID	Planned Effort	Actual Effort	Scheduled Cost	Actual Cost
1	17	18	1700	1800
2	10	12	1000	1200
3	15	13	1500	1300
4	16	19	1600	1900
5	11	-	1100	-
6	9	-	900	-
..
..

Given, the above project table, while you are asked to perform the EVA, 6 tasks should have been completed as per schedule. But only 4 tasks have been completed. Calculate Scheduled Performance Index (SPI), Scheduled Variance (SV), Percent scheduled for complete, Percent Complete, Cost Performance Index(CPI), Cost Variance(CV), Total budget is \$15, 000

Problem 9 - Solution

- *budgeted cost of work scheduled(BCWS) = \$ 7800*
- *budgeted cost of work performed(BCWP)= \$ 5800*
- Actual cost of work performed(ACWP) = \$ 6200
- Scheduled performance index(SPI) = BCWP/BCWS
 $= 5800/7800 = 0.75$
- Scheduled Variance (SV) = BCWP – BCWS
 $= 5800 - 7800 = -2000$
- Percent Scheduled = BCWS/BAC = 52%
- Percent Complete = BCWP/BAC = 39%
- CPI = BCWP/ACWP = 0.93
- CV = BCWP – ACWP = -400

UNIT - III

Software Design

Disclaimer:

The lecture notes have been prepared by referring to many books and notes prepared by the teachers. This document does not claim any originality and cannot be used as a substitute for prescribed textbooks.

Topics

- Software Design Fundamentals
- Design Standards - Design Type
- Design Model – Architectural design, Software Architecture
- Software Design Methods
- Top Down, Bottom Up
- Module Division (Refactoring)
- Module Coupling
- Component Level Design
- User Interface Design
- Pattern Oriented Design
- Web Application Design
- Design Reuse
- Concurrent Engineering in Software Design
- Design Life-Cycle Management

Software Design

Introduction

- Software design is the process where **the software architecture is built and refined** using the end user requirements.
- In the waterfall model, the software product will have a **large number of parts** which can be designed separately from each other and later can be joined.
- If you need to break the software product design into many parts then either **top down approach or bottom up approach** can be used.
- In top down approach, the software design for the **entire product is built first** and later its parts are designed.
- In bottom up approach, the **software product parts are designed first and later** they are joined to create design of the entire product.
- With each change request there will be a **different version of the software** design.
- Maintaining these design versions through the development process is very important.
- It is necessary to make sure that the **right design** is used for software construction.
- Software design should be **robust** and should also be able to take change requests without any problems.

Software Design

Introduction

- When a software product is incrementally built then the later versions of the software design should be fully compatible with the original software design.
- There are many software design techniques available to make **appropriate software designs** for the product being built.
- They include prototyping, **structural models, object oriented design, entity relationship models etc.**
- Refactoring is a design technique which is used in iterative models.
- When the design become **bulky after many iterations of development**, it starts getting crumbling against new features being added to the product.
- In such cases, the design is changed so that new factors of the features being added are incorporated.
- Software design development can be likened to **designing a physical product**.
- Suppose a new car model is to be developed.
- The car design is broken down into separate components and in the end assembling them will become a complete design for the car model.
- Various factors are considered during the design of the components.

Software Design

Introduction

- Suppose one factor to be considered is that during a **car accident**, the car body should take most of the impact and the passengers should get the least impact, so that injury to car passengers can be minimized during accidents.
- For this to happen, the car body should be made of material that can collapse on impact and thus take most of the impact.
- So during design, when selecting the material of the car with safety in mind, the **body is one of the prime** considerations.
- Similarly, an aerodynamic **body helps in keeping the car from rolling over during accidents and thus it is a prime safety factor** that the car body should be aerodynamic.
- During design, one consideration is also made that though **each component is developed separately**, after assembly the components should work with each other without any problems.
- That means assembling does not create any problems in the product itself.
- Similar considerations are also done when software products are designed.

Software Design

Introduction

- In fact, in designing software systems, consideration is given to things like how well the **system will be maintained during operation and how easily the system will be actually developed and be tested.**
- Software design is done using modeling languages like **UML** and using **notation methods like use cases and activity diagrams.**
- We will learn all about software design considerations, workflows involved in design, etc.

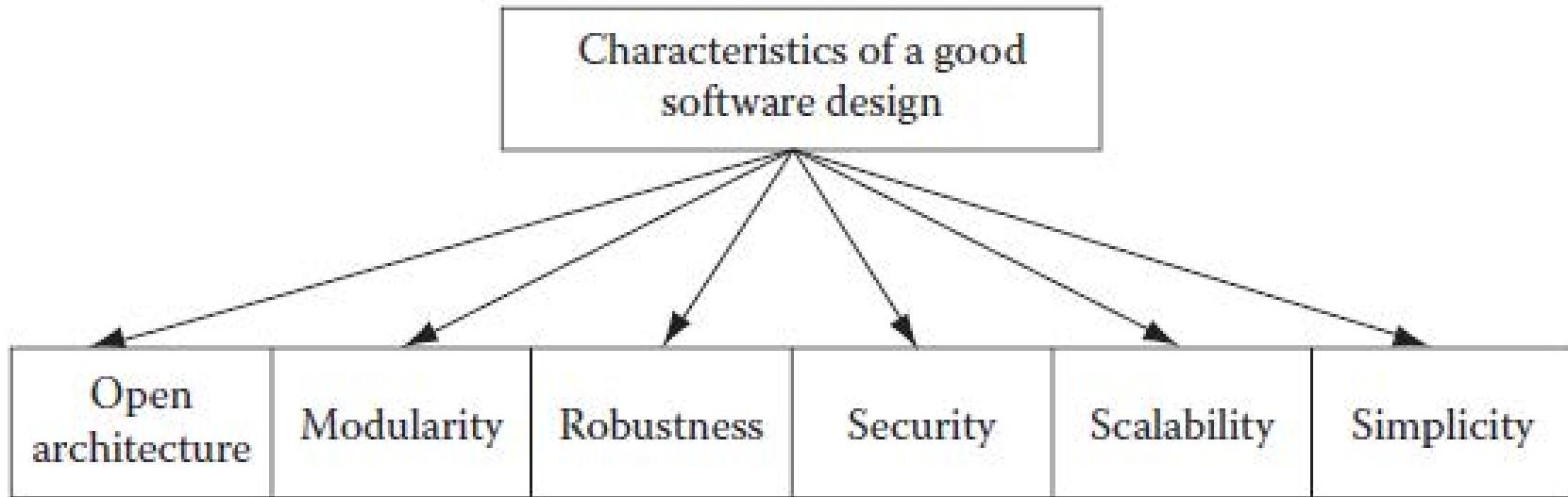
Software Design

Software Design Fundamentals

- When a building is constructed, a good foundation is laid out for the building, so that the building will have a **long lifespan** and will not collapse.
- Similarly, it is given a strong and resilient structure, so that even in case of an earthquake, it will not fall down.
- Similarly, software design provides the foundation and structure upon which the software system is constructed.
- The design should provide a **sound, resilient and scalable structure** to support the software system (Figure below-Characteristics of a good software design).
- In these days, most software systems are built **incrementally**.
- In the beginning, a software system may consist of only a few features.
- The feature set is expanded in future releases as and when it becomes necessary to include them in the system.
- If proper structure is not provided from the very beginning, the addition of these new features will make the system **unstable**.
- To deal with this problem, a technique called refactoring is used on these **agile** projects where **incremental software development** is done.

Software Design

Software Design Fundamentals



- Some of the design techniques that help make good software design include open architecture, modularity and scalability.
- The current trend of service-oriented architecture (SOA) has also helped tremendously in changing the design concepts.
- SOA is built on Web services and loose coupling of software components.
- The asynchronous messaging method of SOA is a vital aspect for developing Web-based applications.

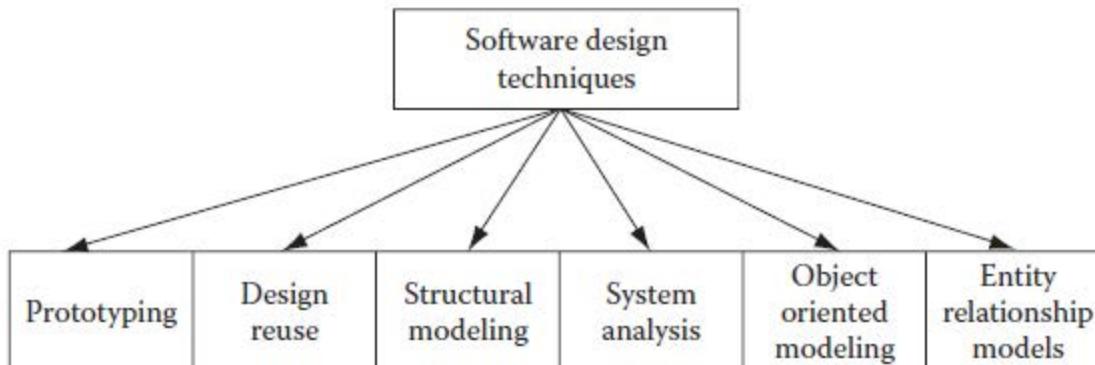
Software Design

Design Standards

- If design standards are implemented on a project, then it will help in streamlining activities that are involved during the software design phase.
- Some industry standards for software design include **operator interface standards, test scenarios, safety standards, design constraints and design tolerances**.

Design Types

- Software design on any **project may consist of many work products**, which together can be termed the software design for the software product that will be built during the software project.
- Some examples include **prototypes, structural models, object-oriented design, systems analysis and entity relationship models**.



Software Design

Design Types

- A good software design not only ensures a smooth transition to the development phase but also ensures that the software product has a good shelf life during operation.
- So what are the keys to a good design? A good design should start from the most possible **abstract** architecture of the software product often termed as "**high level design**".
- Subsequent transition of the **abstract design should lead to platform-specific design often termed as "low level design"**.
- The platform-specific design or low level design will be in terms of a good database model and a good application model (Figure above - Software design techniques).
- Over the years, many software design techniques have evolved with the evolution of different programming paradigms.
- Starting with the early procedural programming paradigms, programming has evolved into present day "**service-oriented architecture**".
- Software design has kept the pace with these evolving paradigms, and thus it has also been evolving. So, we have early structural design paradigms to modern day SOA designs. Let us discuss some of these design techniques.

Software Design

Design Types – Prototypes

- Prototyping is cheap and fast.
- It also gets a buy in from customer at an early stage of the project.
- If not, a full prototype of the application, a partial prototype can contribute to win over the customer.
- There are many automatic code generation tools that allows to drag and drop some components on screens, and the tool generates the code and makes a working prototype of the application that can be demonstrated to the customer.
- A miscommunication or misunderstanding between the customer and the project team gets cleared once the difference of opinion are sorted out early on during the prototype demonstration sessions.
- This greatly helps in reducing the risks of not meeting customer expectations.
- In any case, customers do not care about internal workings of the application.
- They are always concerned about what the application screens look like and how the application behaves with different kind of inputs and events.

Software Design

Design Types – Prototypes

- The downside about prototypes is that many customers assume the prototype is the fully functional application and later on wonder why the application is taking so much time in development when they saw the working demonstration so early in the project.
- Customer expectations become difficult to manage in such instances.
- Prototypes can only show the user interface screens.
- When complex logic is involved in developing applications, that logic cannot be depicted in prototypes, as program logic is mostly not visible and cannot be developed in prototypes.
- Starting with the early procedural programming paradigms, programming has evolved into present day "service-oriented architecture".

Software Design

Design Types – Design Reuse

- For large software products, the **design can be broken into many design parts representing each module of the product.**
- Each of these design modules contain a lot of design information that can be represented as design components.
- Many details inside these design components can be repeated inside different components.
- If a standard method of representing the same information can be used for these components, then it is possible to use these pieces of information in many components by reusing them.
- It will reduce effort in designing the product.
- This method of design reuse is known as internal design reuse.
- A more potent design reuse is becoming available after the advent of the open source paradigm and SOA.
- In the case of open source, the design reuse is in fact a case of copying existing design and then using it exactly as it is or modifying it to suit the needs.
- But in the case of SOA, there is no copying or modifying a software design.
- The existing design is utilized as it is.

Software Design

Design Types – Design Reuse

- In addition, there is **no process of buying the application/component** whose design is required.
- There is simply **buying a service from the owner of the application/component** and using that service in building the application.
- The owner of that application/component publishes full details as to how to integrate with the application for the according application/component.
- The full interface details are provided by the owner.
- Using this information, the design is done for the application.
- There is an assumption that as if the application/component provided as a service is available and the application uses this application/component.
- SOA is indeed leading to a reuse model that is going to transform the world of computing and the lives in years to come.

Software Design

Design Types – Structural Models

- Most software applications are built using components.
- At the bottom are the smallest units of functions and procedures in a software application.
- These functions are contained within classes or packages depending on the programming language used.
- Many classes together build a component.
- Components in turn make modules. Modules in turn make the complete application.
- For ease of working, maintenance, and breaking development tasks to allocate to group of developers, it is essential that an application is broken down into manageable parts.
- Breaking into parts for an application can best be done using a structural analysis.
- From requirement specifications, a feature set is made to decide what features will be in the application.
- This feature set is analyzed and broken down into smaller sets of features, which will go into different modules.
- This is represented in a structural model of the application.

Software Design

Design Types – Object-oriented Design

- It has always been difficult to represent business entities and business information flow in a software model.
- With object-oriented design, this problem was solved.
- Business entities are represented as objects in the object- oriented software design.
- Properties of these objects are made in such a way that they are similar to the properties of the business entities.
- These objects are instantiated from classes in the form of child classes.
- These child classes inherit all the properties of their parent class, and they can have some more properties of their own in addition.
- So if we have a group of similar objects with somewhat different properties, then we can implement classes in such a way that a base parent class has child classes with different properties.
- This concept aligns very much to the real-world scenarios.
- Object-oriented design takes input from use cases, activity diagrams, user interfaces etc.

Software Design

Design Types – Systems Analysis

- System analysis is the process of finding solutions in the form of a system designed from the inputs coming from business needs.
- The fundamental question addressed in system analysis is whether a business scenario can be converted into a software application, so that the user can use the software application to do his routine business tasks.
- For instance, a person may want to access his bank account using an Internet connection to the online web site of the bank.
- This scenario calls for many things that are involved in the whole chain of objects and events.
- The system analysis will be concerned with user activities, what objects on the web site act with user activities, how these objects interact with the underlying software system of the bank, and how connections are made between the user and the website and between the website and the bank system.
- System analysis will analyze all these things.
- Based on the analysis, a system model can be made that will be used in developing the application.

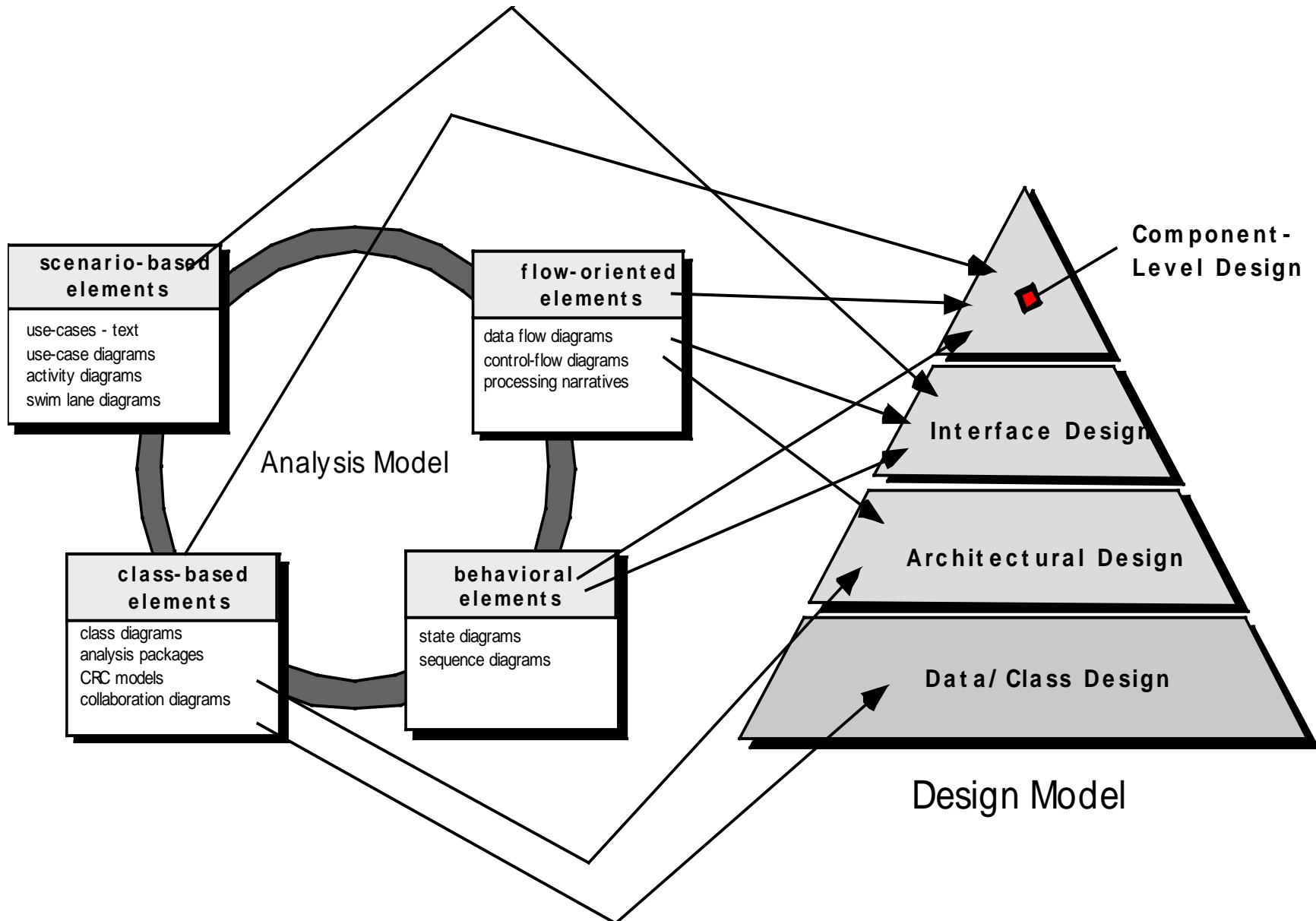
Software Design

Design Types – Entity Relationship Models

- Entity relationship models are one of the ways to represent business entities and their relationships to each other through diagrams.
- These diagrams are used for creating databases and database tables.
- How many tables are needed to fulfill the needs of the software product, how these tables are related to each other, and in what form data are to be kept inside these tables, etc. are decided through these diagrams.
- With object-oriented modeling, it is possible to correlate each object with a corresponding database object.
- This kind of representation helps to make a clean database design.

Design Model

Analysis Model -> Design Model



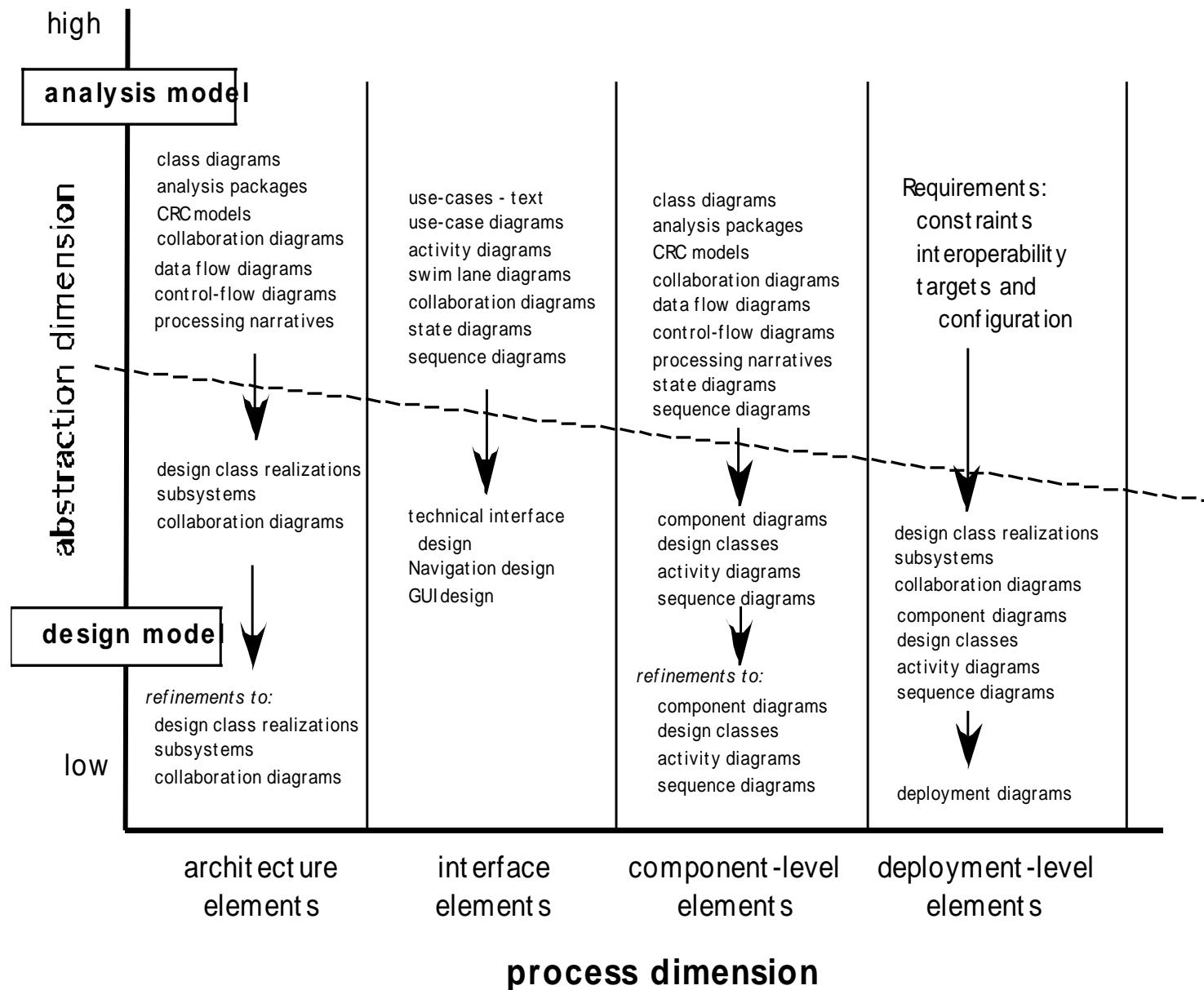
Design Model

Design Model Elements

- Data Elements
 - Data model → Data structures
 - Data model → Database architecture
- Architectural Elements
 - Application domain
 - Analysis classes, their relationships, collaborations and behaviors are transformed into design realizations
 - Patterns and “styles”.
- Interface Elements
 - User Interface (UI)
 - External interfaces to other systems, devices, networks or other producers or consumers of information.
 - Internal interfaces between various design components
- Component Elements
- Deployment Elements

Design Model

Analysis Model -> Design Model



Architectural Design

Architectural Design

- The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.

Structural properties

- This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another.
- For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

Extra-functional properties

- The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems

- The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems.
- In essence, the design should have the ability to reuse architectural building blocks.

Software Architecture

Why Architecture?

- The architecture is not the operational software.
- Rather, it is a representation that enables a software engineer to:
 - (1) Analyze the effectiveness of the design in meeting its stated requirements,
 - (2) Consider architectural alternatives at a stage when making design changes is still relatively easy, and
 - (3) Reduce the risks associated with the construction of the software.

Architecture - Significance

- Representations of software architecture are an enabler for communication between all parties (stakeholders) 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 the ultimate success of the system as an operational entity.
- Architecture “constitutes a relatively small, intellectually graspable mode of how the system is structured and how its components work together”.

Software Architecture

Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1471-2000, Recommended Practice for Architectural Description of Software-Intensive System,
 - To establish a conceptual framework and vocabulary for use during the design of software architecture,
 - To provide detailed guidelines for representing an architectural description, and
 - To encourage sound architectural design practices.
- The IEEE Standard defines an architectural description (AD) as a “a collection of products to document an architecture.”
 - The description itself is represented using multiple views, where each view is “a representation of a whole system from the perspective of a related set of [stakeholder] concerns.”

Architectural Design

Architectural Genres

- Genre implies a specific category within the overall software domain.
- Within each category, there are a number of subcategories.
 - For example, within the genre of buildings, there are following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.
 - Within each general style, more specific styles might apply.
 - Each style would have a structure that can be described using a set of predictable patterns.

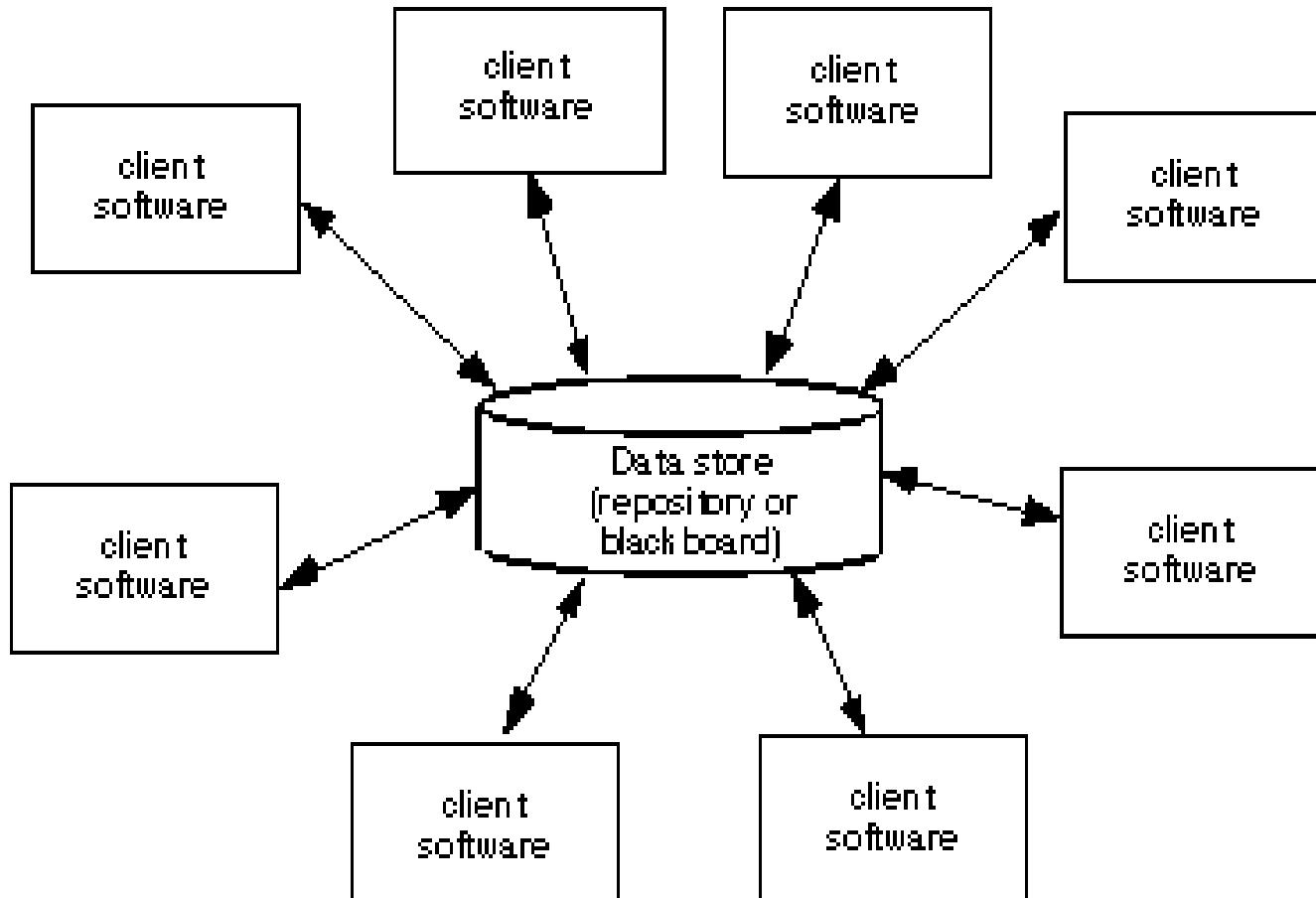
Architectural Styles

- Each style describes a system category that encompasses:
 - (1) a set of components (e.g., a database, computational modules) that perform a function required by a system,
 - (2) a set of connectors that enable “communication, coordination and cooperation” among components,
 - (3) constraints that define how components can be integrated to form the system, and
 - (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

Architectural Design

Architectural Styles – Data Centered Architecture

- 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, delete, or otherwise modify data within the store.
- Figure below illustrates a typical data-centered style.



Architectural Design

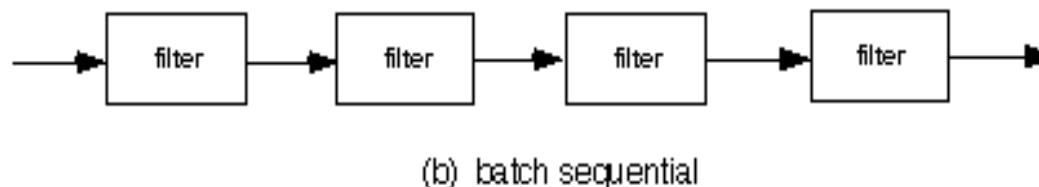
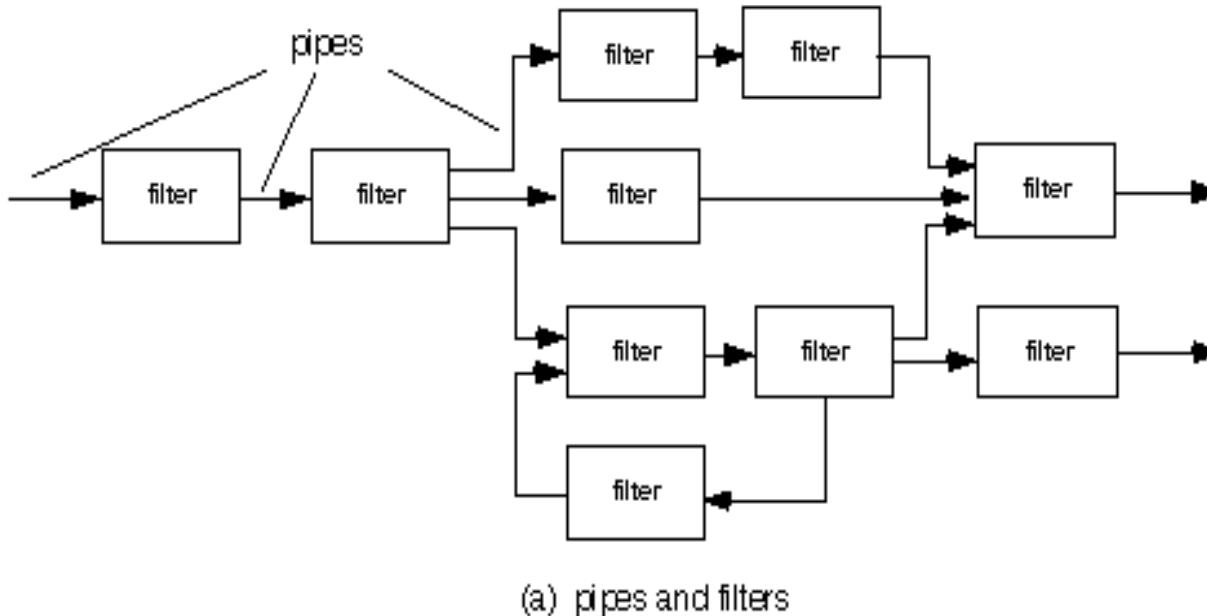
Architectural Styles – Data Centered Architecture

- 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" that sends notifications to client software when data of interest to the client changes.

Architectural Design

Architectural Styles – Data Flow Architecture

- 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 has a set of components called filters, connected by pipes that transmit data from one component to the next.



Architectural Design

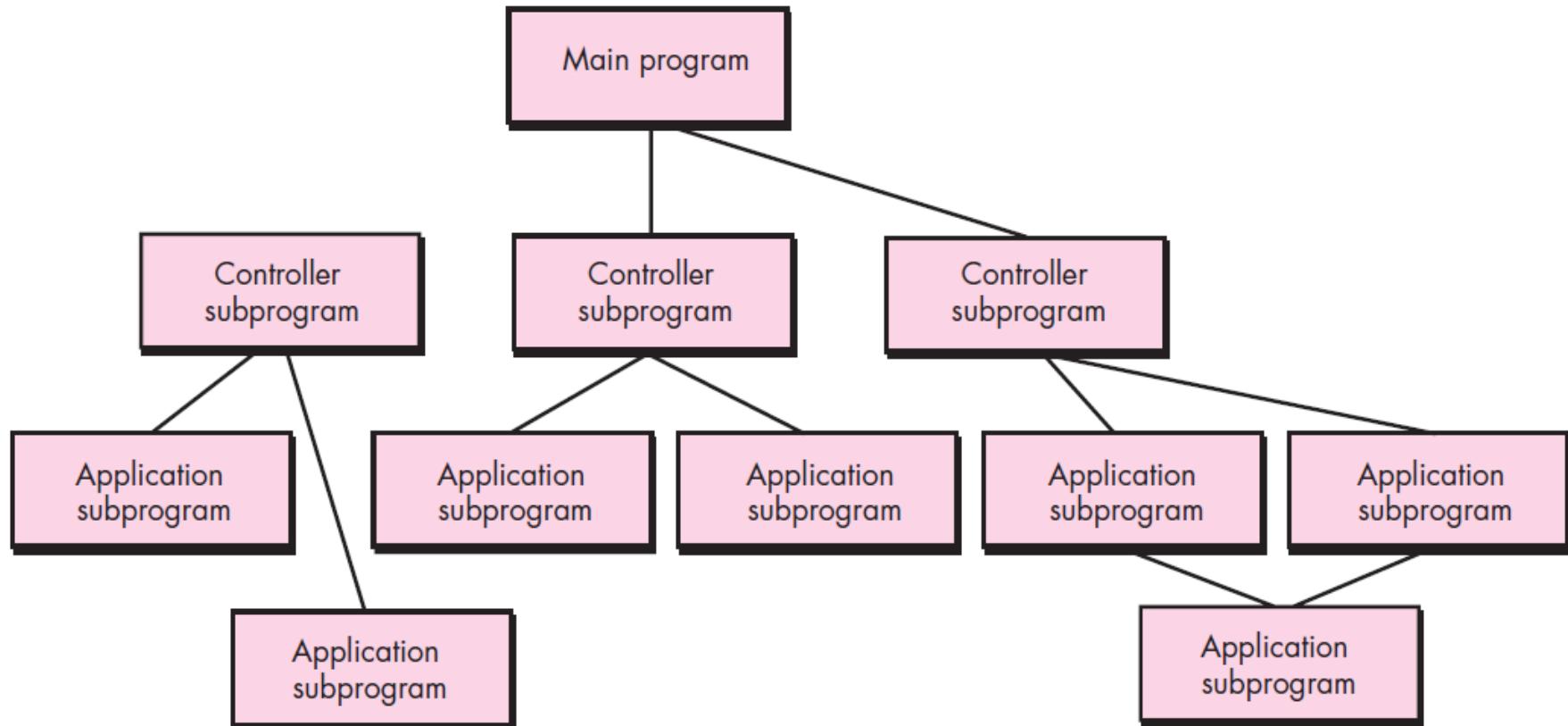
Architectural Styles – Data Flow Architecture

- 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.
- If the data flow degenerates into a single line of transforms, it is termed batch sequential.
- This structure accepts a batch of data and then applies a series of sequential components (filters) to transform it.

Architectural Design

Architectural Styles – Call and Return Architecture

- This architectural style enables you to achieve a program structure that is relatively easy to modify and scale.
- A number of sub-styles 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.



Architectural Design

Architectural Styles – Call and Return Architecture

- The figure above illustrates an architecture of this type.
- Remote procedure call architectures: The components of a main program/subprogram architecture are distributed across multiple computers on a network.

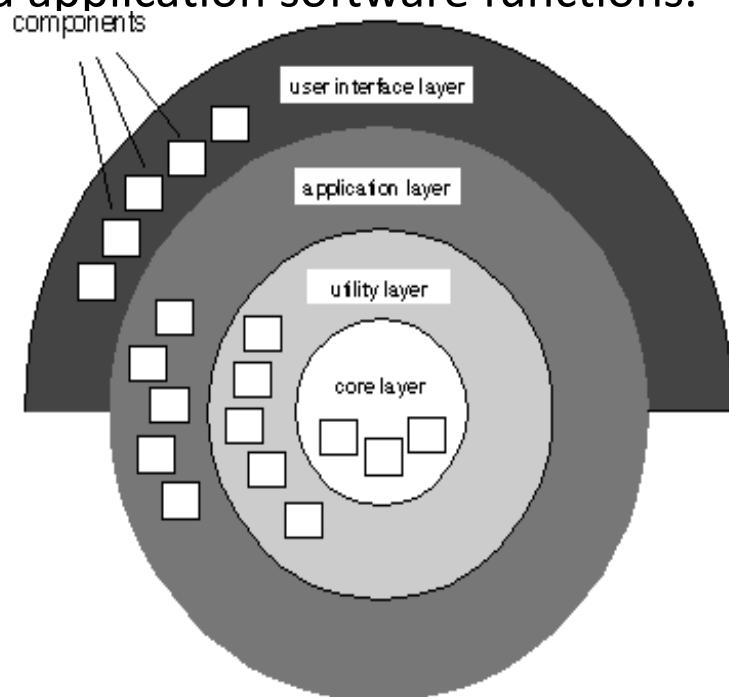
Architectural Styles – Object-oriented Architecture

- 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.

Architectural Design

Architectural Styles – Layered Architecture

- The basic structure of a layered architecture is illustrated below.
- 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.
- These architectural styles are only a small subset of those available.
- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen.



Architectural Design

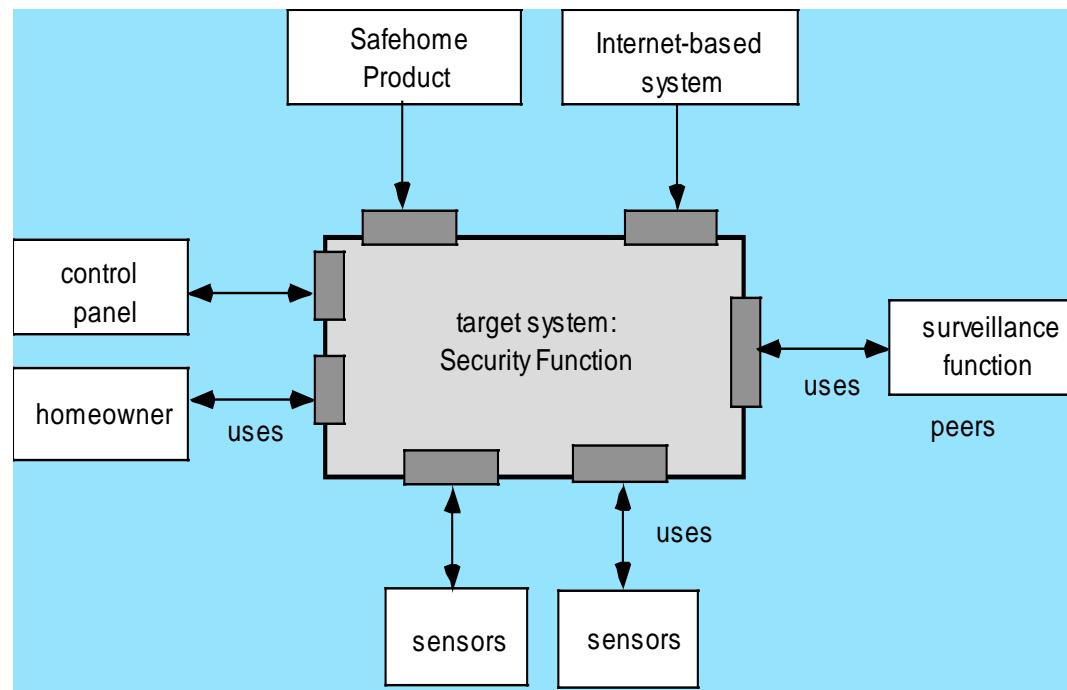
Architectural Patterns

- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
 - Operating system process management pattern
 - Task scheduler pattern
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
 - A **database management system pattern** that applies the storage and retrieval capability of a DBMS to the application architecture.
 - An **application level persistence pattern** that builds persistence features into the application architecture.
- Distribution—the manner in which systems or components within systems communicate with one another in a distributed environment
 - A broker acts as a ‘middle-man’ between the client component and a server component.

Architectural Design

a) Representing the System in Architectural Context

- The software must be placed into context
 - The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.
- A set of architectural archetypes should be identified.
- The designer specifies the structure of the system by defining and refining software components that implements each archetype.
- The below diagram illustrates Architectural context diagram for the SafeHome security function.



architectural context diagram (ACD)

At the architectural design level, a software architect uses an architectural context diagram (ACD) to model the manner in which software interacts with entities external to its boundaries

- Superordinate systems—those systems that use the target system as part of some higher-level processing scheme.
- Subordinate 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—those systems that interact on a peer-to-peer basis (i.e., information is either produced or consumed by the peers and the target system).
- Actors—entities (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing.

Architectural Design

b) Defining Archetypes

- 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.
- In general, a relatively small set of archetypes are required to design even relatively complex systems.
- In relation to the SafeHome home security function, it is necessary to define the following archetypes:
- Node – represents a cohesive collection of input and output elements of the home security function.
- Detector – An abstraction that encompasses all sensing equipment that feeds information into the target system.
- The diagram illustrates the UML relationships for SafeHome security function archetypes.

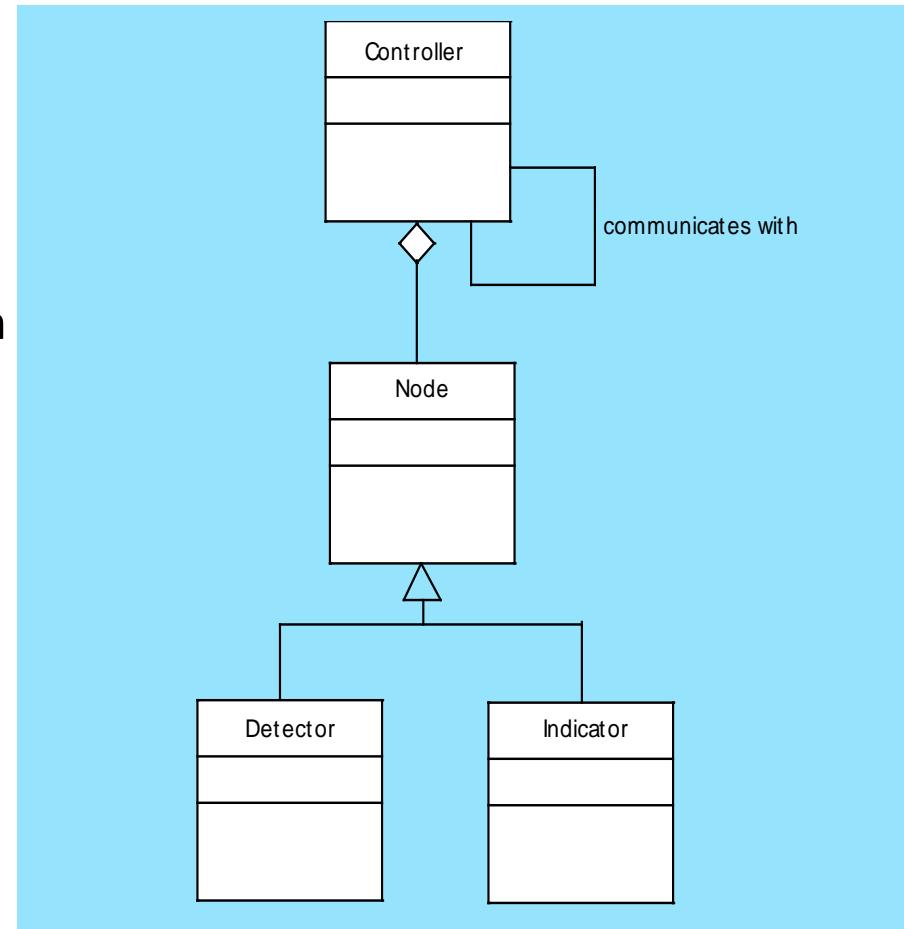


Figure 10.7 UML relationships for SafeHome security function archetypes
(adapted from [BOS00])

Architectural Design

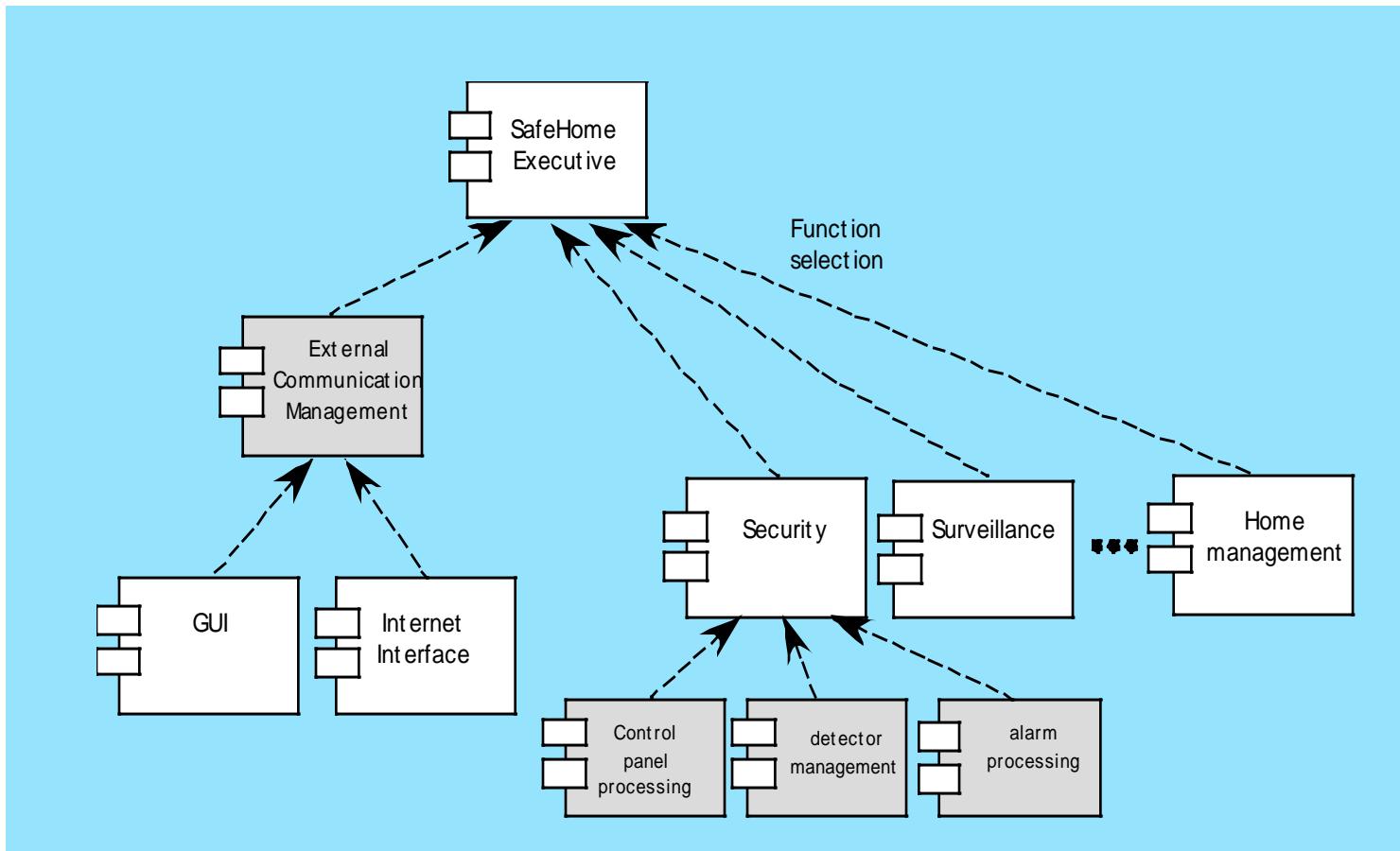
Archetypes

- 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.
- Each of these archetypes is depicted using UML notation as shown in above figure.
- Recall that the archetypes form the basis for the architecture but are abstractions that must be further refined as architectural design proceeds.
- For example, Detector might be refined into a class hierarchy of sensors.

Architectural Design

c) Refining the architecture into Component Structure

- As the software architecture is refined into components, the structure of the system begins to emerge.
- But how are these components chosen? In order to answer this question, it is necessary to begin with the classes that were described as part of the requirements model.



Architectural Design

Component Structure

- The figure above illustrates the overall architectural structure for SafeHome with top-level components.
- These analysis classes represent entities within the application (business) domain that must be addressed within the software architecture.
- Hence, the application domain is one source for the derivation and refinement of components.
- Another source is the infrastructure domain. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain.
- In regards to the SafeHome home security function example, the set of top-level components might be defined to address the following functionality:
 1. External communication management – coordinates communication of the security function with external entities such as other Internet-based systems and external alarm notification.
 2. Control panel processing – manages all control panel functionality.
 3. Detector management – coordinates access to all detectors attached to the system.
 4. Alarm processing – verifies and acts on all alarm conditions.

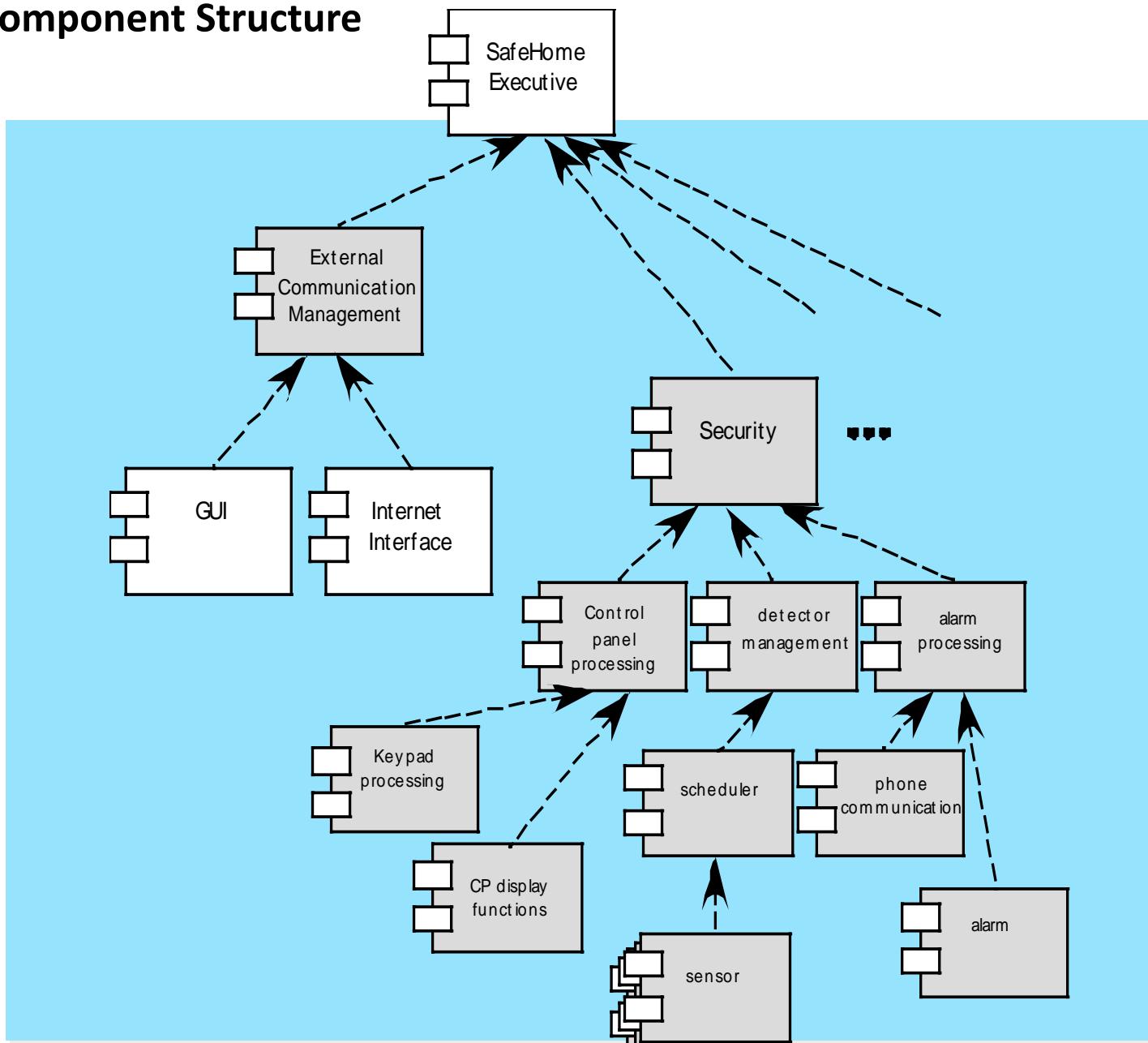
Architectural Design

Refined Component Structure

- 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.
- To accomplish this, an actual instantiation of the architecture is developed.
- The figure below illustrates an instantiation of the SafeHome architecture for the security system.
- Components shown in figure above are elaborated to show additional details.
- For example, the detector management component interacts with a scheduler infrastructure component that implements polling of each sensor object used by the security system.
- Similar elaboration is performed for each of the components represented in figure above.

Architectural Design

Refined Component Structure



Architectural Design

Analyzing Architectural Design

1. Collect scenarios.
2. Elicit requirements, constraints, and environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements:
 - Module view
 - Process view
 - Data flow view
4. Evaluate quality attributes by considering each attribute in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.

Architectural Design

Architectural Complexity

- The overall complexity of a proposed architecture is assessed by considering the dependencies between components within the architecture.
 - Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers.
 - Flow dependencies represent dependence relationships between producers and consumers of resources.
 - Constrained dependencies represent constraints on the relative flow of control among a set of activities.

ADL

- Architectural description language (ADL) provides a semantics and syntax for describing a software architecture
- Provide the designer with the ability to:
 - Decompose architectural components
 - Compose individual components into larger architectural blocks and
 - Represent interfaces (connection mechanisms) between components.

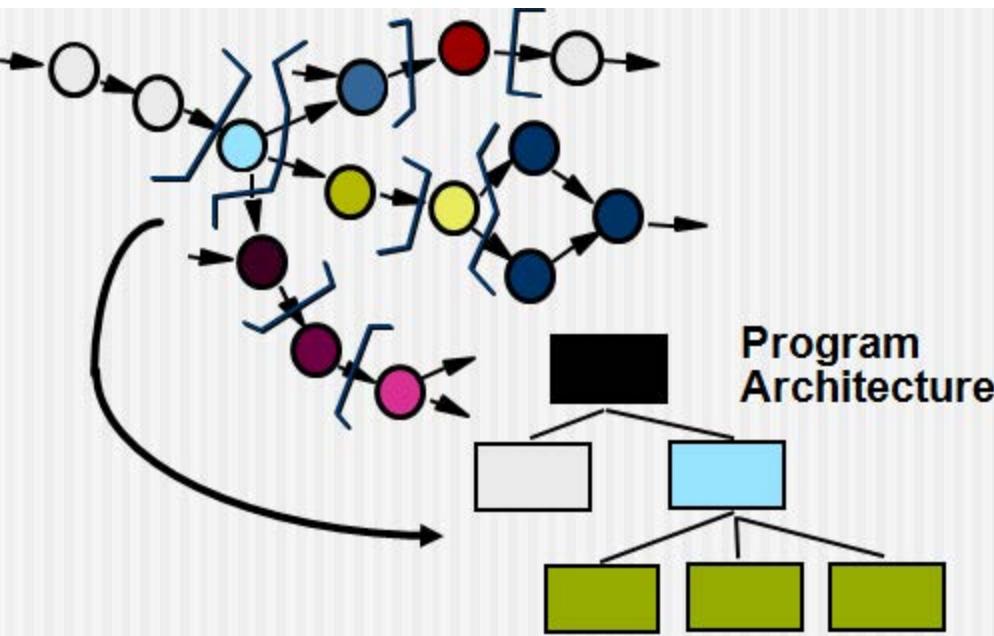
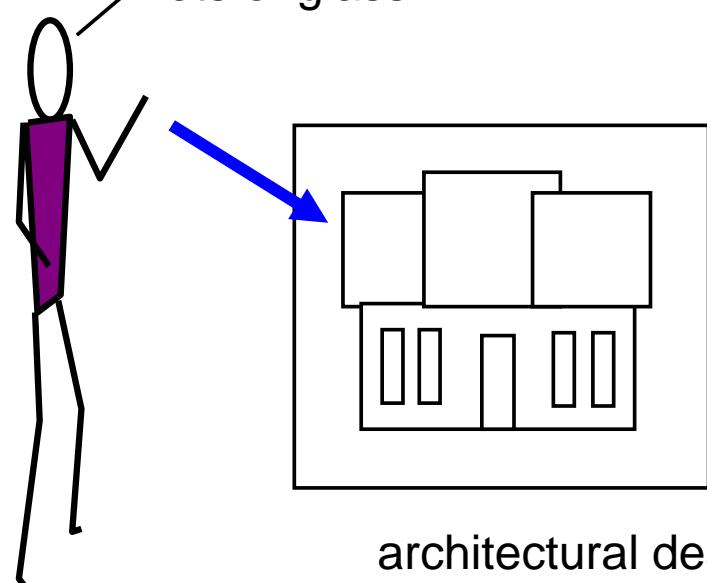
Architectural Design

An Architectural Design Method

- Customer requirements
- Deriving program architecture

customer requirements

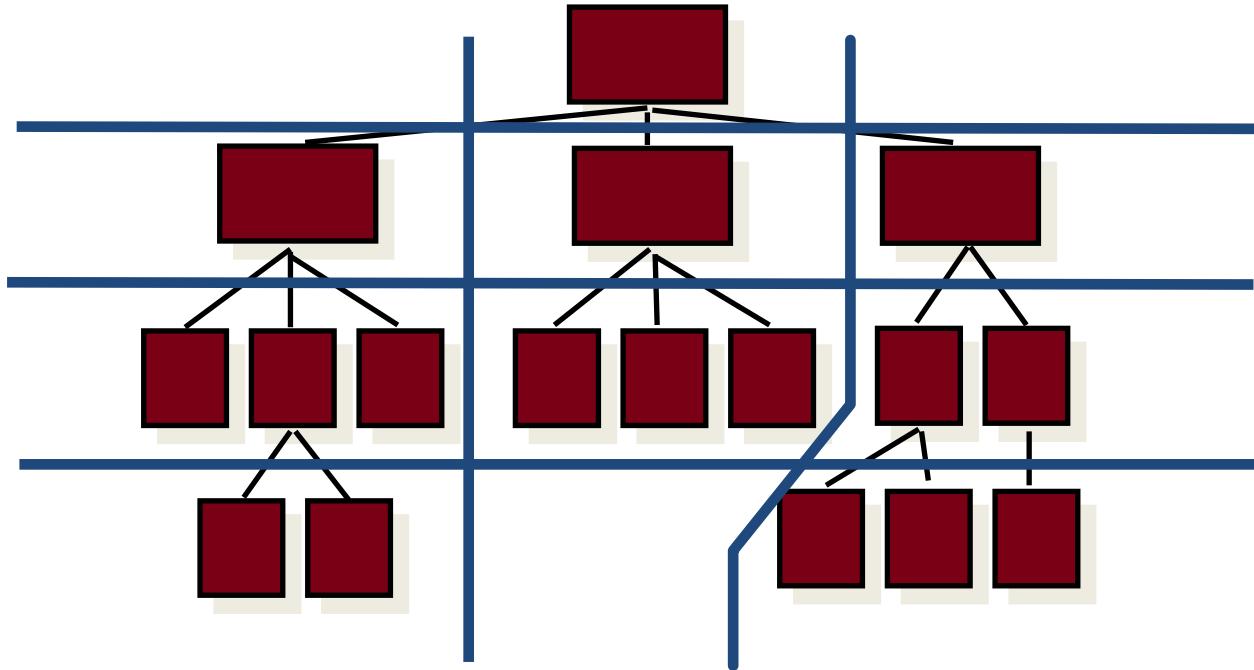
"four bedrooms, three baths,
lots of glass ..."



Architectural Design

Partitioning the Architecture

- The “horizontal” and “vertical” partitioning are required



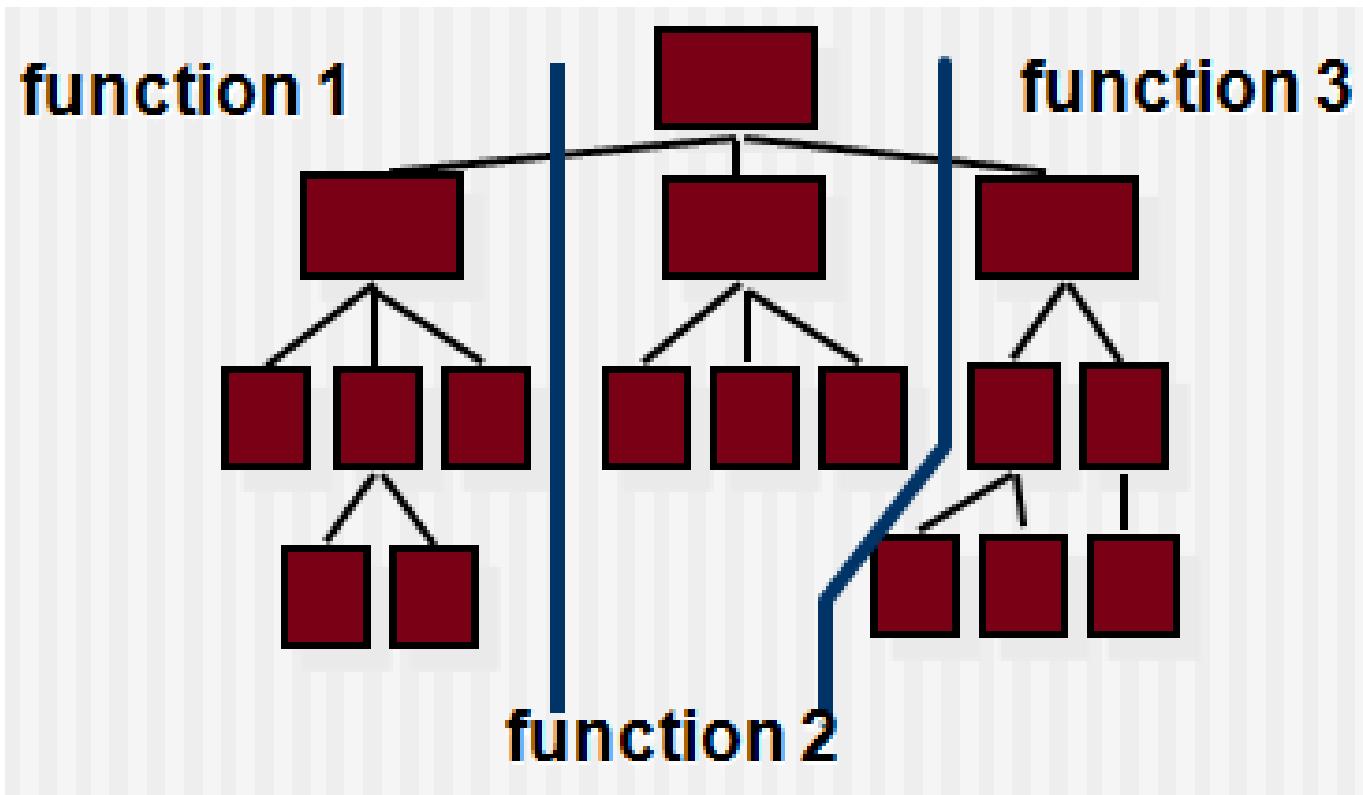
Why Partitioned Architecture?

- Results in software that are easier to test
- Leads to software that are easier to maintain
- Results in propagation of fewer side effects
- Results in software that are easier to extend

Architectural Design

Horizontal Partitioning

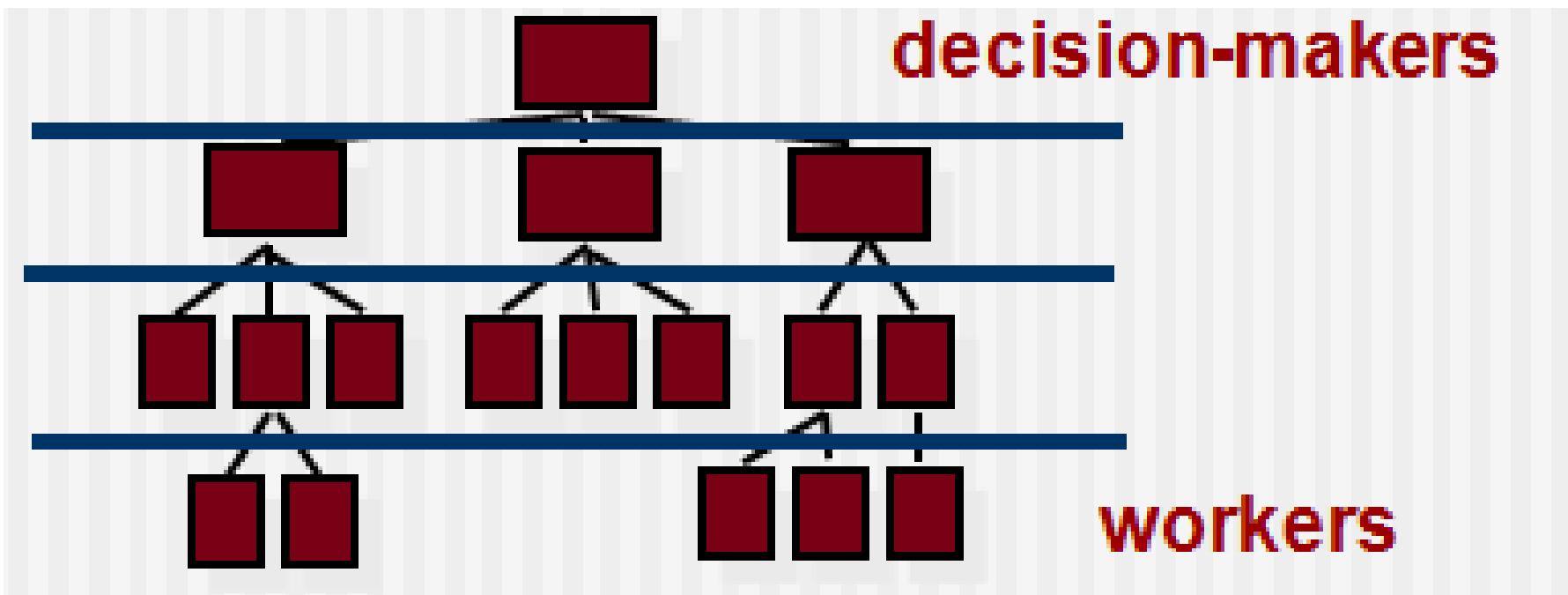
- Define separate branches of the module hierarchy for each major function
- Use control modules to coordinate communication between functions



Architectural Design

Vertical Partitioning – Factoring

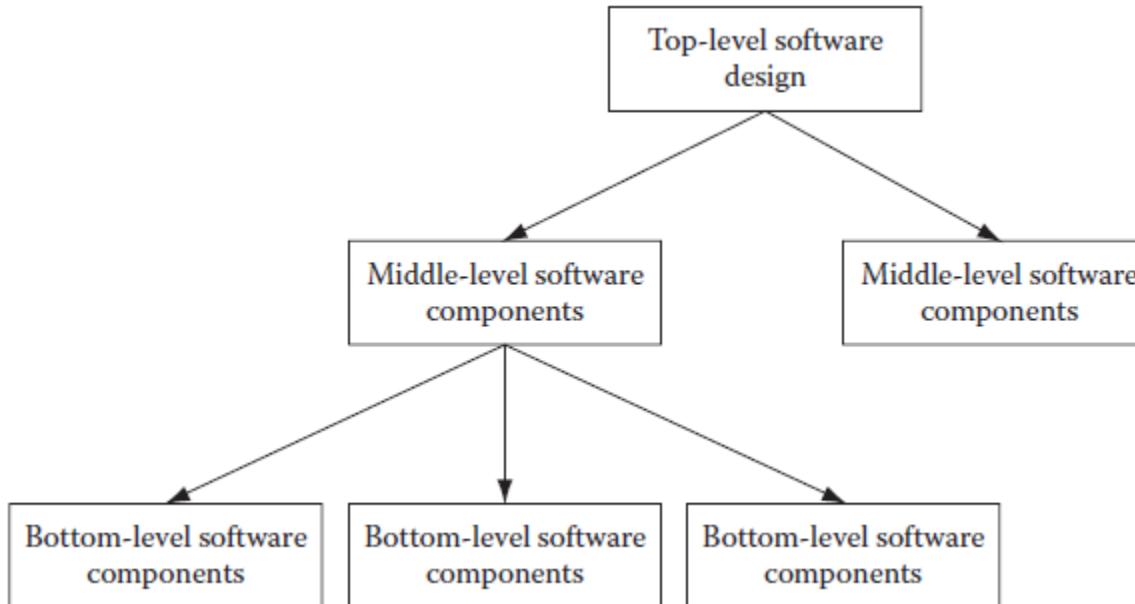
- Design so that decision making and work are stratified
- Decision making modules should reside at the top of the architecture



Software Design Methods

Top Down

- In the top-down approach, the top structure of the product is conceived and designed first.
- Once the structure is perfected, components that will make the product are designed.
- Once the major components are designed, the features that make the component are designed (Figure below – Top-down Software Design).
- Apart from the functional consideration for making the structure, nonfunctional considerations are also considered from the top level for example, how the security, performance, usability, aspects will be provided in the product.



Software Design Methods

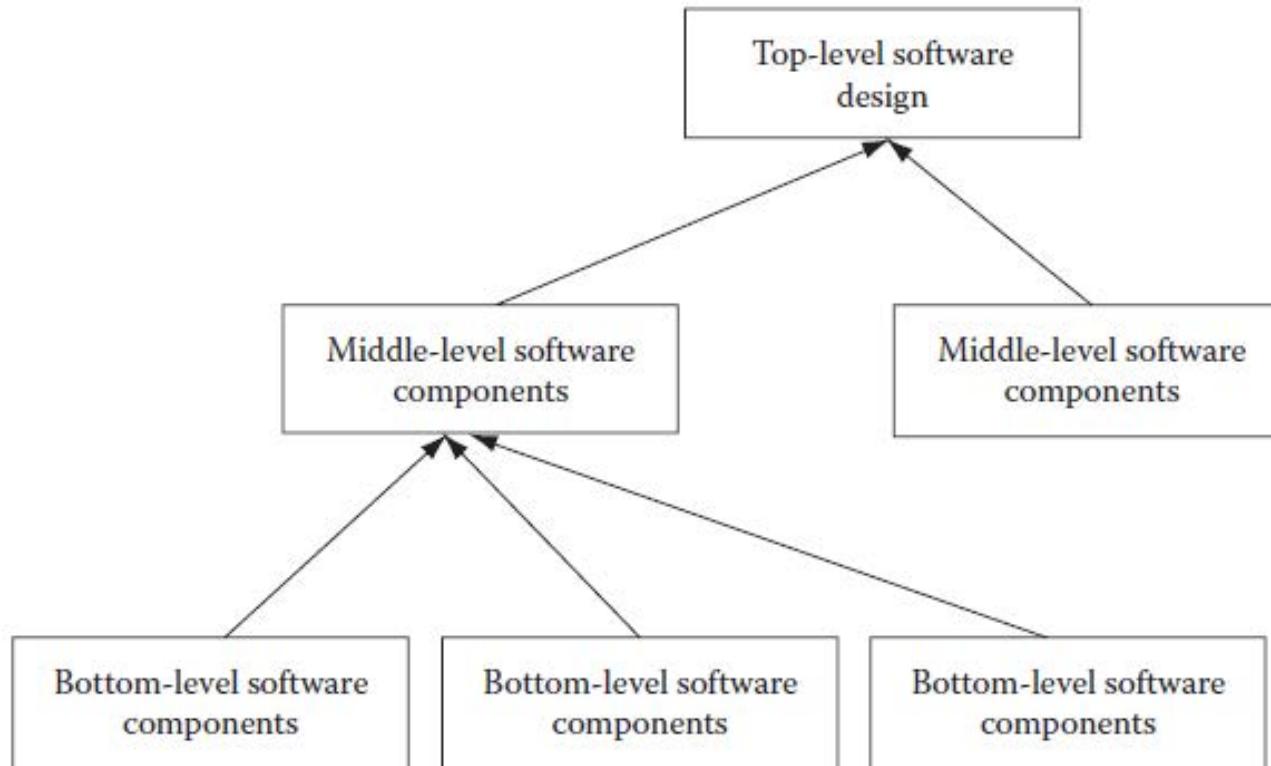
Top Down

- There are many benefits to the top -down approach.
- Nonfunctional aspects are taken care of at the beginning of design, and hence they are an integral part of the product and not an after-thought.
- This makes a secure, robust, and usable product.
- A top- down approach also helps in creating reusable components and hence increases productivity as well as maintainability.
- This approach also promotes integrity, as the whole product is designed inside a single framework.
- So a fragmented and dissimilar approach for designing different parts of the product is avoided.
- The drawback of the top -down approach is that it is a risky model.
- The whole design has to be made in one go instead of making attempts to incrementally building the design, which is relatively a safer option.
- Generally, the top-down design approach is adopted on waterfall model-based projects.

Software Design Methods

Bottom Up

- In the bottom-up approach, first, the minute functions of the software product are structured and designed.
- Then, the middle-level components are designed, and, finally, the top-level structure is designed.
- Once some components are designed, they can be shown to the customer, and a buy in can be made for the project.



Software Design Methods

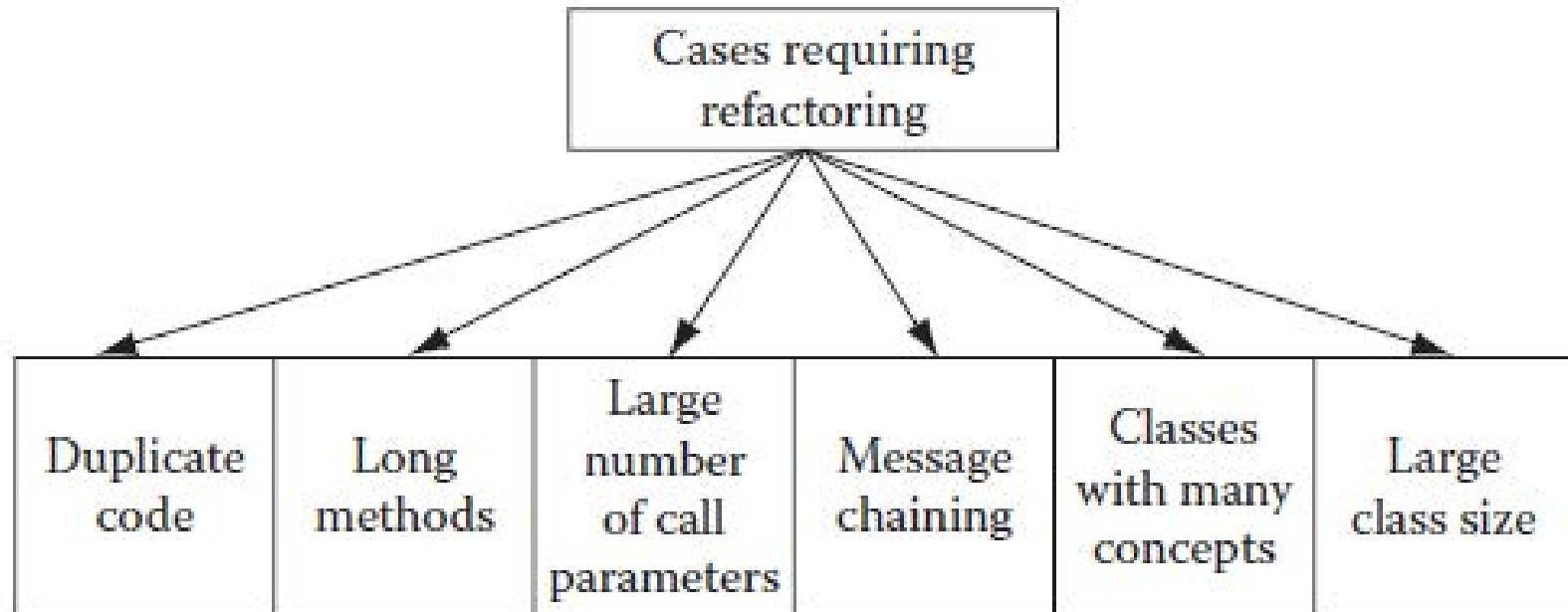
Bottom Up

- There are some benefits to the bottom-up approach.
- It leads to incremental building of design that ensures that any missing information can be accommodated later in the design (Figure above – Bottom-up Software Design).
- With increasing use of incremental and iterative development methodologies, the bottom-up design approach is becoming more popular than the top-down approach.
- In fact, nowadays, agile models do not go for elaborate and complete software design from the beginning of the project.
- In each iteration, a design is thought of for the requirements that are taken during the iteration.
- To compensate for a sturdy and elaborate design upfront, the project team engages in refactoring (discussed next) the design to make sure that it does not become bulgy and unmanageable in later iterations.

Module Division (Refactoring)

- Whenever a software product is designed, it is done with good intentions.
- Care is taken to ensure that the design is extensible, so that when customer needs increase over time, the product can be extended to take care of those increased needs.
- Unfortunately, even this foresight is not enough, and it becomes difficult to extend the product functionality further. In such cases, it becomes necessary to change the internal structure of software code without changing external behavior of the software product.

Figure - Characteristics of a software product code that requires refactoring



Module Division (Refactoring)

- To do this, one technique is employed, which is known as refactoring.
- **Using refactoring, the internal design of a piece of software code is improved by decreasing coupling among classes of objects and increasing cohesion among classes.**
- Refactoring is very similar to the concept of normalization in relational databases.
- Some of the indications of code analysis that may suggest that the code needs refactoring include duplicate code at many places, using long methods, a large class with many concepts, the need to pass a large number of parameters, too much communication between classes resulting from a large number of calls for methods in code, and message chaining by calling one method which in turn calls another method.
- When software code starts having these characteristics, then it is better to go for code cleaning or refactoring.
- Going for refactoring will be justified by savings in time due to better code reuse and make it easier to maintain code and scale up the product.
- Refactoring can be achieved by dividing cumbersome classes into smaller classes that can be managed and used in a better way.

Module Division (Refactoring)

- In the new code, the functions will be the same, but many of the functions will be moved now into new classes.
- On agile projects, the project team builds the software product without making an elaborate design from start.
- One product module is built after another in the subsequent project iterations.
- This fact makes it necessary to adjust the software design as the product evolves in this fashion.
- The adjustment in the software design in such cases is done using refactoring.

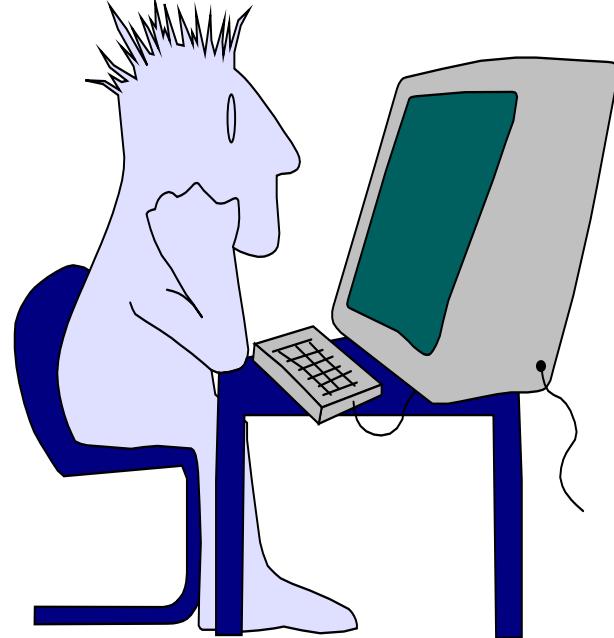
Module Coupling

- One area similar to refactoring is coupling between modules.
- As products mature and more and more lines of code are added to the existing product, coupling between modules tends to increase.
- This has a profound impact when any changes in code are required.
- Changes in code result in more than normal occurrence of defects as dependency between modules keeps increasing with increase in the size of the product.
- To reduce the chances of product defects, it is necessary to reduce the number of calls among different modules and classes.
- Service Oriented Architecture (SOA) architecture provides great help here.
- SOA architecture essentially promotes loose coupling, and this implies more or less self-contained classes having less dependency on other classes.
- Increasing module coupling with increase in size of software product is always a concern.
- Frequent refactoring can help in reducing module coupling among classes.

User Interface Design

Interface Design

- Easy to learn?
- Easy to use?
- Easy to understand?



Typical Design Errors

- Lack of consistency
- Too much memorization
- No guidance / help
- No context sensitivity
- Poor response
- Arcane/unfriendly



User Interface Design

Golden Rules

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

Place the User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with objects that appear on the screen.

User Interface Design

Reduce the User's Memory Load

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real world metaphor.
- Disclose information in a progressive fashion.

Make the Interface Consistent

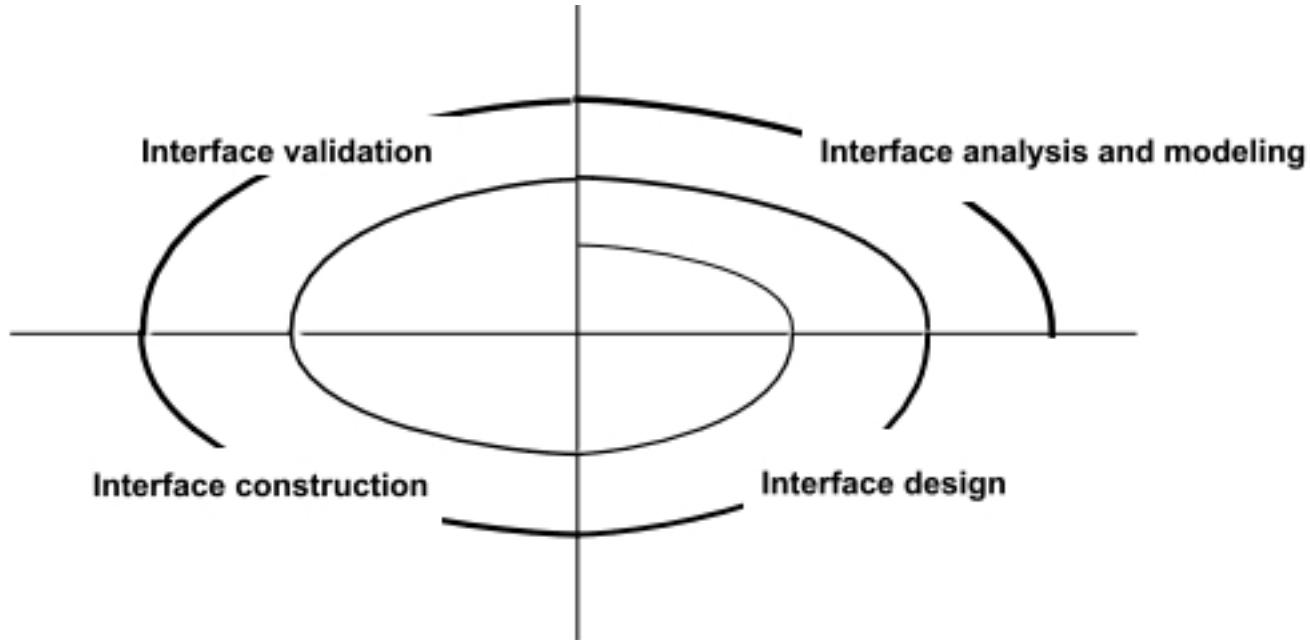
- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

User Interface Design

User Interface Design Models

- User model — A profile of all end users of the system
- Design model — A design realization of the user model
- Mental model (system perception) — The user's mental image of what the interface is
- Implementation model — The interface “look and feel” coupled with supporting information that describe interface syntax and semantics.

User Interface Design Process



User Interface Design

User Interface Design Process – Interface Analysis

- Interface analysis means understanding
 - (1) The people (end-users) who will interact with the system through the interface;
 - (2) The tasks that end-users must perform to do their work,
 - (3) The content that is presented as part of the interface
 - (4) The environment in which these tasks will be conducted.

User Analysis

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?

User Interface Design

User Interface Design Process – Interface Analysis

User Analysis

- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology that sits behind the interface?

Task Analysis and Modeling

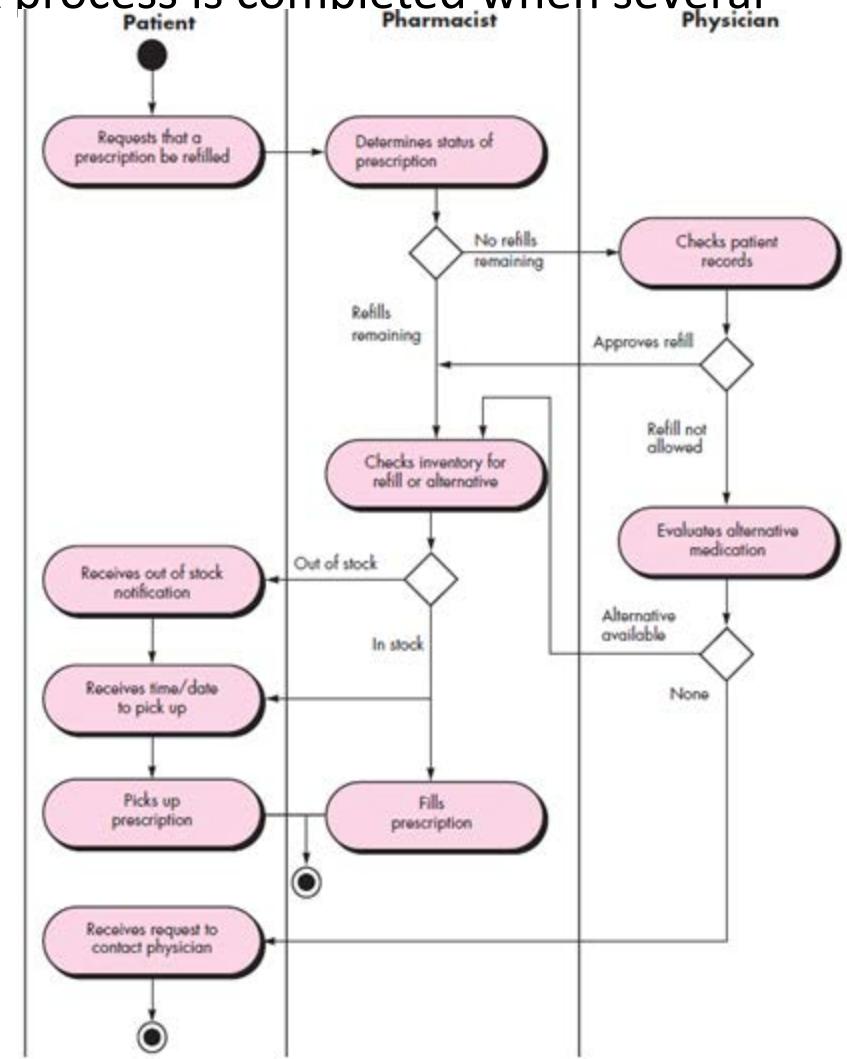
- Answers the following questions ...
 - What work will the user perform in specific circumstances?
 - What tasks and subtasks will be performed as the user does the work?
 - What specific problem domain objects will the user manipulate as work is performed?
 - What is the sequence of work tasks—the workflow?
 - What is the hierarchy of tasks?
- Use-cases define basic interaction
- Task elaboration refines interactive tasks

User Interface Design

User Interface Design Process – Interface Analysis

Task Analysis and Modeling

- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved
 - Swimlane Diagram for Prescription refill function



User Interface Design

Analysis of Display Content

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data?
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warning be presented to the user?

User Interface Design

Interface Design Steps

- Using information developed during interface analysis, define interface objects and actions (operations).
- Define events (user actions) that will cause the state of the user interface to change.
- Model this behavior.
- Depict each interface state as it will actually look to the end-user.
- Indicate how the user interprets the state of the system from information provided through the interface.

Design Issues

- Response time
- Help facilities
- Error handling
- Menu and command labeling
- Application accessibility
- Internationalization

User Interface Design

WebApp Interface Design

- Where am I?

The interface should

- Provide an indication of the WebApp that has been accessed
- Inform the user of her location in the content hierarchy.

- What can I do now?

The interface should always help the user understand his current options

- What functions are available?
- What links are live?
- What content is relevant?

- Where have I been, where am I going?

The interface must facilitate navigation.

- Provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

User Interface Design

Effective WebApp Interfaces

- Bruce Tognazzi suggests...
 - Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control.
 - Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
 - Effective interfaces do not concern the user with the inner workings of the system.
 - Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
 - Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

User Interface Design

Interface Design Principles

- Anticipation—A WebApp should be designed so that it anticipates the user's next move.
- Communication—The interface should communicate the status of any activity initiated by the user
- Consistency—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- Controlled autonomy—The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- Efficiency—The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.
- Focus—The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- Fitt's Law—"The time to acquire a target is a function of the distance to and size of the target."
- Human interface objects—A vast library of reusable human interface objects has been developed for WebApps.

User Interface Design

Interface Design Principles

- Latency reduction—The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- Learnability— A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.
- Maintain work product integrity—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- Readability—All information presented through the interface should be readable by young and old.
- Track state—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- Visible navigation—A well-designed WebApp interface provides “the illusion that users are in the same place, with the work brought to them.”

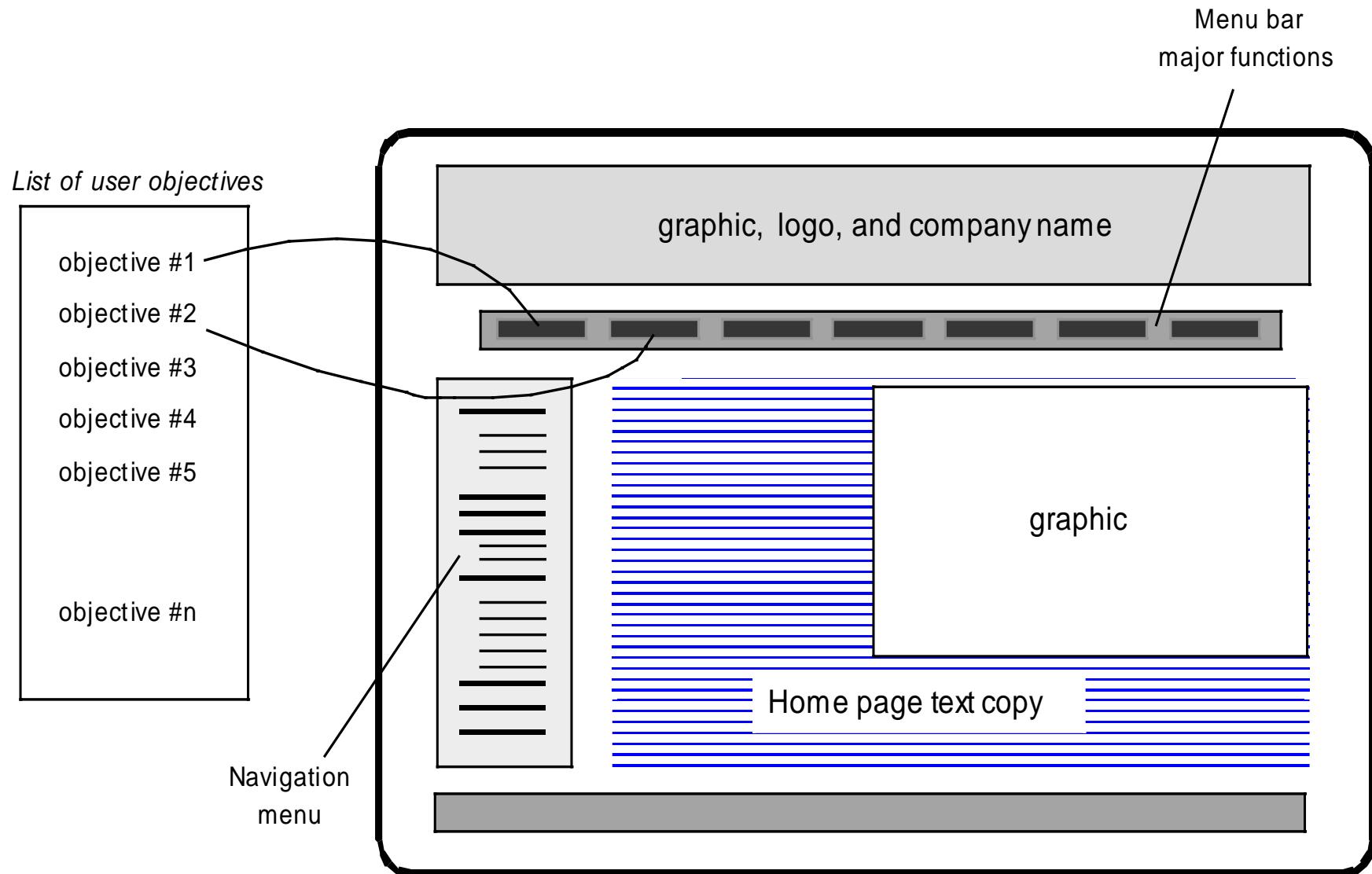
User Interface Design

Interface Design Workflow

- Review information contained in the analysis model and refine as required.
- Develop a rough sketch of the WebApp interface layout.
- Map user objectives into specific interface actions.
- Define a set of user tasks that are associated with each action.
- Storyboard screen images for each interface action.
- Refine interface layout and storyboards using input from aesthetic design.
- Identify user interface objects that are required to implement the interface.
- Develop a procedural representation of the user's interaction with the interface.
- Develop a behavioral representation of the interface.
- Describe the interface layout for each state.
- Refine and review the interface design model.

User Interface Design

Mapping User Objectives



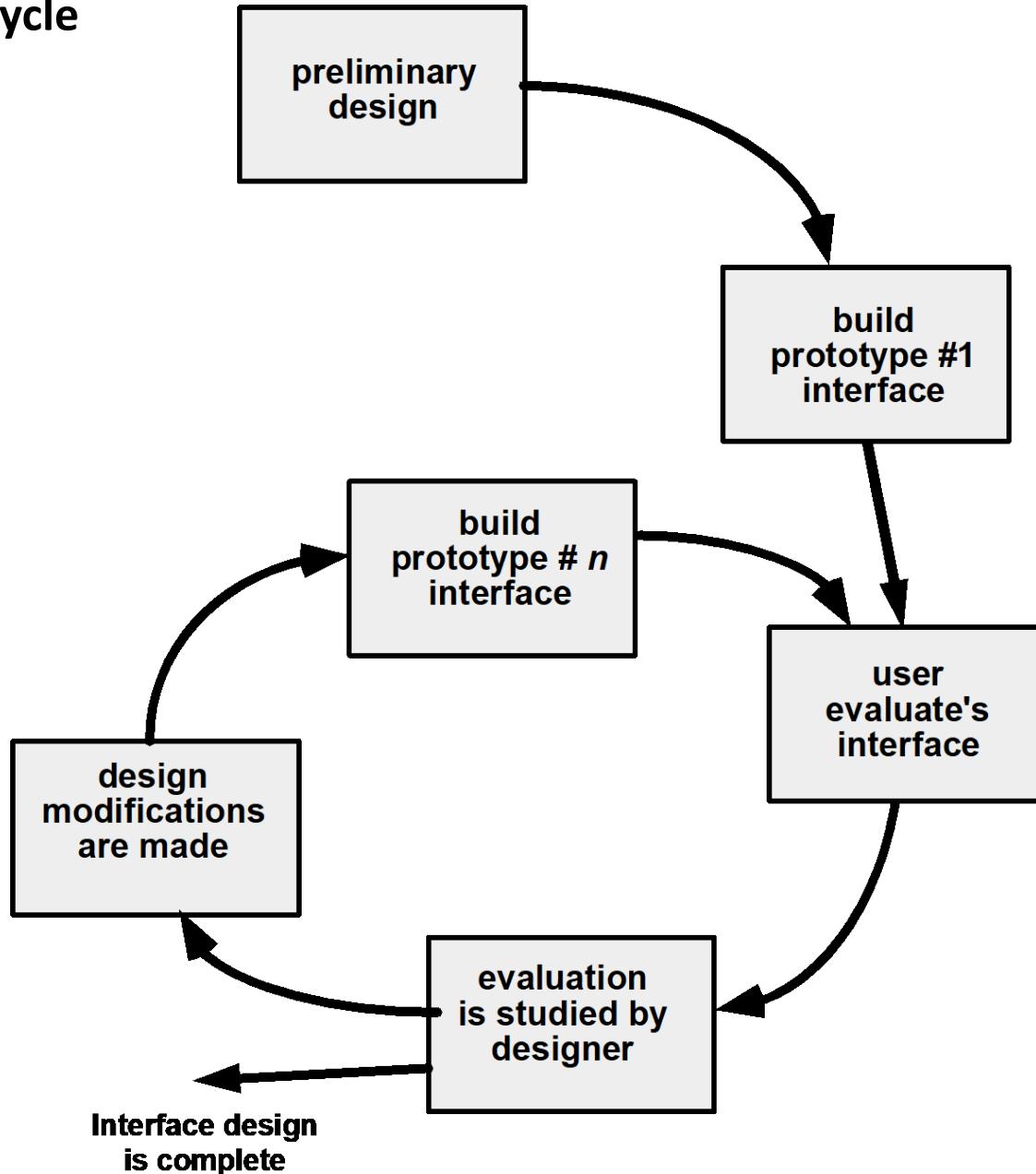
User Interface Design

Aesthetic Design

- Don't be afraid of white space.
- Emphasize content.
- Organize layout elements from top-left to bottom right.
- Group navigation, content, and function geographically within the page.
- Don't extend your real estate with the scrolling bar.
- Consider resolution and browser window size when designing layout.

User Interface Design

Design Evaluation Cycle



Pattern Oriented Design

Patterns for Requirements Modeling

- Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered.
 - Domain knowledge can be applied to a new problem within the same application domain
 - The domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.
- The original author of an analysis pattern does not “create” the pattern, but rather, discovers it as requirements engineering work is being conducted.
- Once the pattern has been discovered, it is documented.

Discovering Analysis Patterns

- The most basic element in the description of a requirements model is the use case.
- A coherent set of use cases may serve as the basis for discovering one or more analysis patterns.
- A semantic analysis pattern (SAP) “is a pattern that describes a small set of coherent use cases that together describe a basic generic application.”

Pattern Oriented Design

An Example

- Consider the following preliminary use case for software required to control and monitor a rear-view camera and proximity sensor for an automobile:
 - **Use case: Monitor reverse motion**
 - **Description:** When the vehicle is placed in reverse gear, the control software enables a video feed from a rear-placed video camera to the dashboard display.
 - The control software superimposes a variety of distance and orientation lines on the dashboard display so that the vehicle operator can maintain orientation as the vehicle moves in reverse.
 - The control software also monitors a proximity sensor to determine whether an object is inside 10 feet of the rear of the vehicle.
 - It will automatically break the vehicle if the proximity sensor indicates an object within 3 feet of the rear of the vehicle.
- This use case implies a variety of functionality that would be refined and elaborated (into a coherent set of use cases) during requirements gathering and modeling.

Pattern Oriented Design

An Example

- Regardless of how much elaboration is accomplished, the use case(s) suggest(s) a simple, yet widely applicable SAP (Semantic Analysis Pattern)—the software-based monitoring and control of sensors and actuators in a physical system.
- In this case, the “sensors” provide information about proximity and video information.
- The “actuator” is the breaking system of the vehicle (invoked if an object is very close to the vehicle).
- But in a more general case, a widely applicable pattern is discovered --> Actuator-Sensor

Pattern Oriented Design

Actuator-Sensor Pattern

- **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 Passive Sensors and a push mechanism (broadcast of information) for the Active Sensors.

Pattern Oriented Design

Actuator-Sensor Pattern

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.

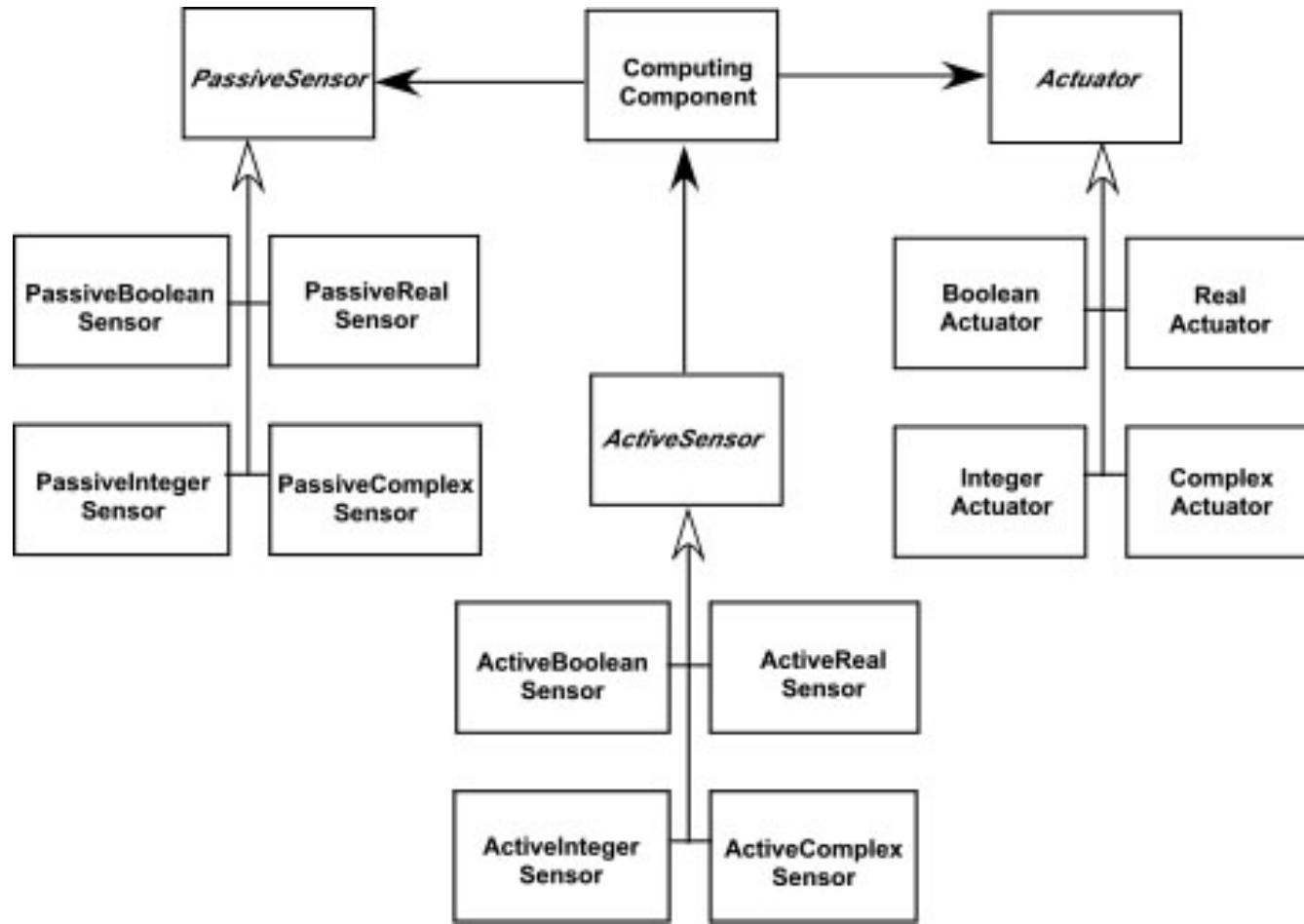
Pattern Oriented Design

Actuator-Sensor Pattern

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 below.



Pattern Oriented Design

Actuator-Sensor Pattern

Structure:

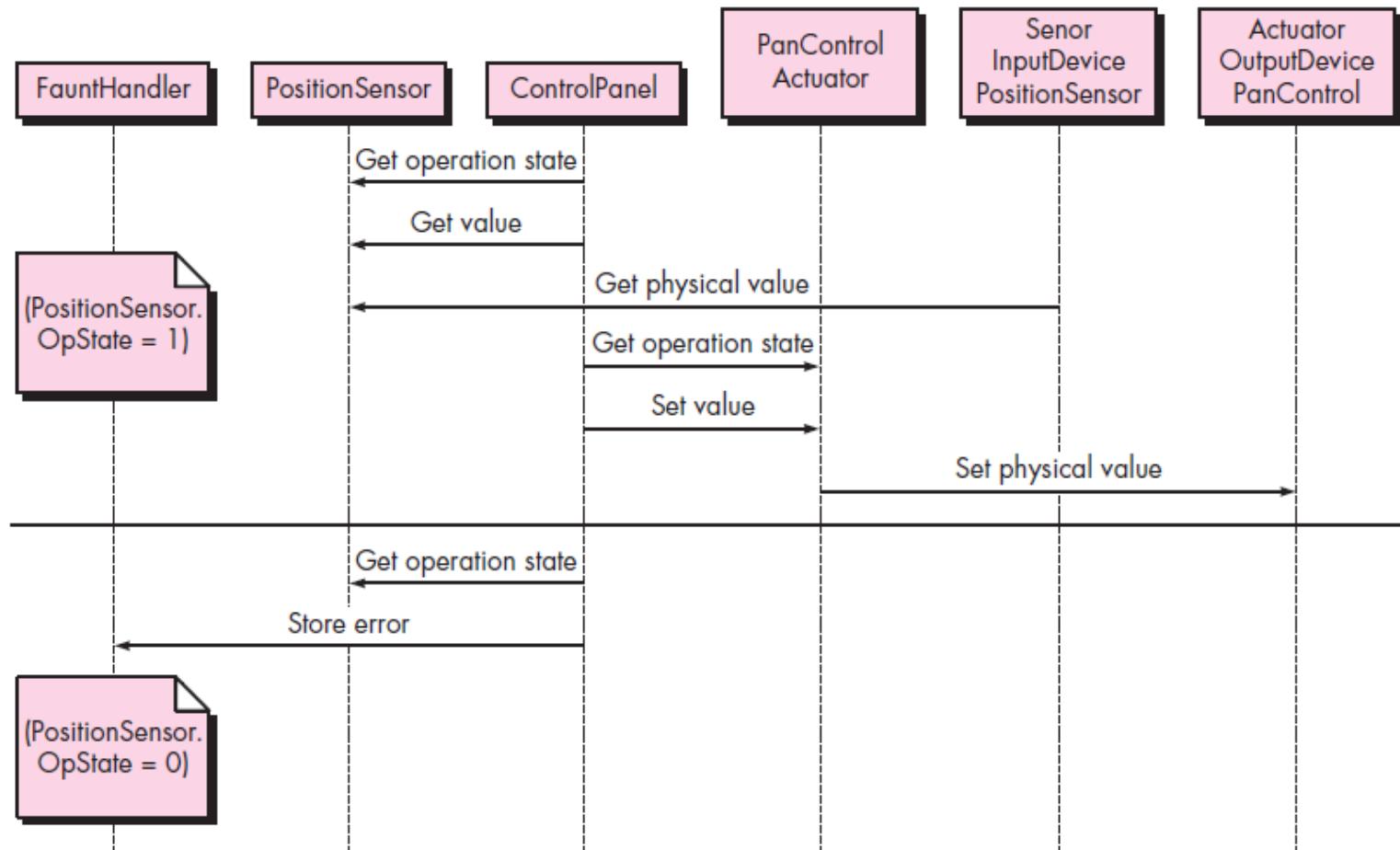
- Actuator, *PassiveSensor* and *ActiveSensor* are abstract classes and denoted in italics.
- There are four different types of sensors and actuators in this pattern.
- The Boolean, integer, and real classes represent the most common types of sensors and actuators.
- The complex classes are sensors or actuators that use values that cannot be easily represented in terms of primitive data types, such as a radar device.
- Nonetheless, these devices should still inherit the interface from the abstract classes since they should have basic functionalities such as querying the operation states.

Pattern Oriented Design

Actuator-Sensor Pattern

Behavior:

- The figure below presents a UML sequence diagram for an example of the Actuator-Sensor Pattern as it might be applied for the SafeHome function that controls the positioning (e.g., pan, zoom) of a security camera.



Pattern Oriented Design

Actuator-Sensor Pattern

Behavior:

- Here, the Control Panel queries a sensor (a passive position sensor) and an actuator (pan control) to check the operation state for diagnostic purposes before reading or setting a value.
- The messages Set Physical Value and Get Physical Value are not messages between objects.
- Instead, they describe the interaction between the physical devices of the system and their software counterparts.
- In the lower part of the diagram, below the horizontal line, the Position Sensor reports that the operation state is zero.
- The ComputingComponent then sends the error code for a position sensor failure to the Fault Handler that will decide how this error affects the system and what actions are required.
- It gets the data from the sensors and computes the required response for the actuators.

Web Application Design

- **Content Analysis** - The full spectrum of content to be provided by the WebApp is identified, including text, graphics and images, video, and audio data. Data modeling can be used to identify and describe each of the data objects.
- **Interaction Analysis** - The manner in which the user interacts with the WebApp is described in detail. Use-cases can be developed to provide detailed descriptions of this interaction.
- **Functional Analysis** - The usage scenarios (use-cases) created as part of interaction analysis define the operations that will be applied to WebApp content and imply other processing functions. All operations and functions are described in detail.
- **Configuration Analysis** - The environment and infrastructure in which the WebApp resides are described in detail.

Web Application Design

When Do We Perform Analysis?

- In some Web situations, analysis and design merge.
- However, an explicit analysis activity occurs when...
 - The WebApp to be built is large and/or complex
 - The number of stakeholders is large
 - The number of Web engineers and other contributors is large
 - The goals and objectives (determined during formulation) for the WebApp will effect the business' bottom line
 - The success of the WebApp will have a strong bearing on the success of the business

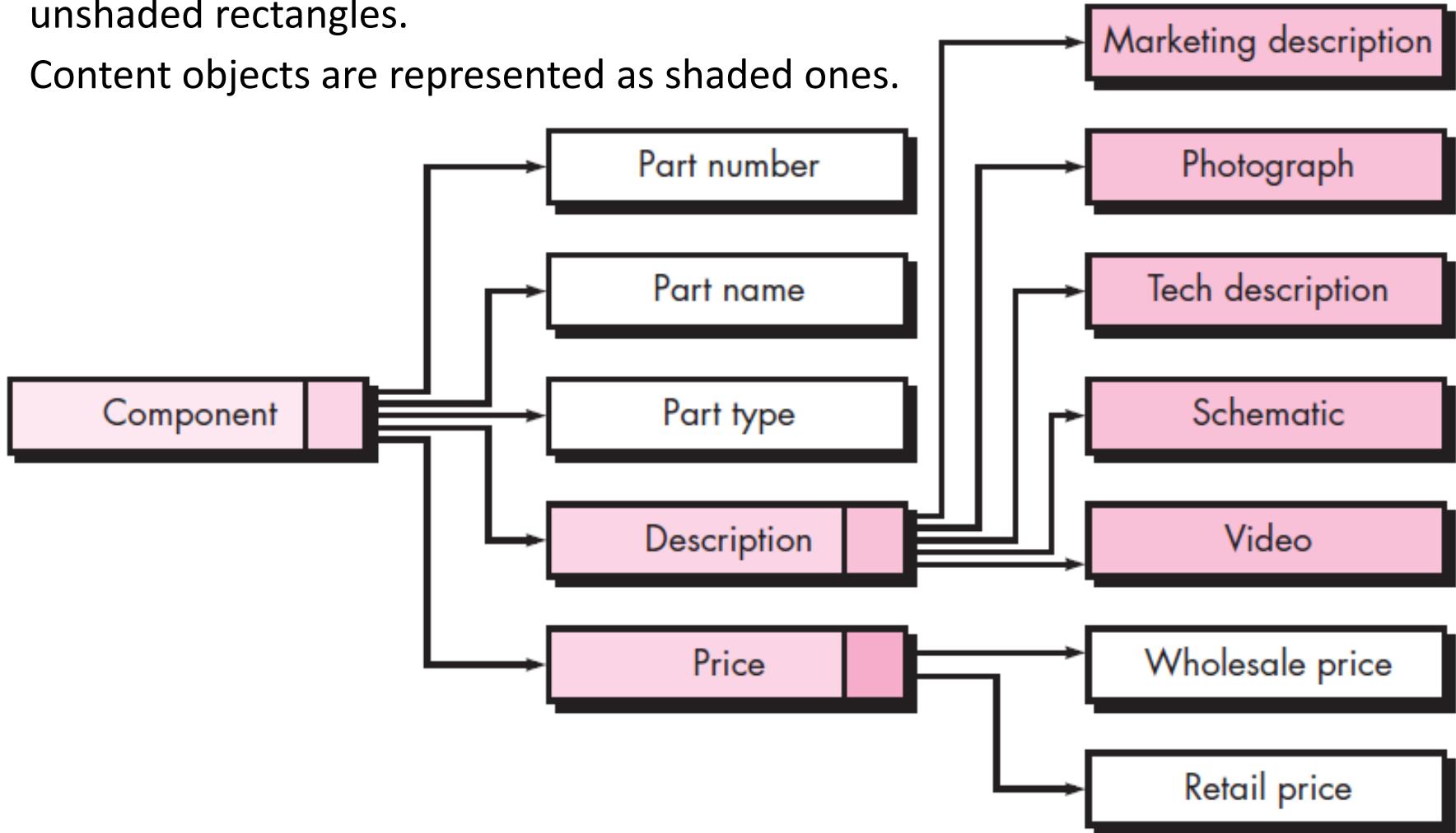
The Content Model

- Content objects are extracted from use-cases
 - Examine the scenario description for direct and indirect references to content
- Attributes of each content object are identified
- The relationships among content objects and/or the hierarchy of content maintained by a WebApp
 - Relationships—Entity-relationship diagram or UML
 - Hierarchy—Data tree or UML

Web Application Design

Data Tree

- Data tree for a SafeHome-Assured.com component – represents a hierarchy of information that is used to describe a component.
- Simple or composite data items (one or more data values) are represented as unshaded rectangles.
- Content objects are represented as shaded ones.



Web Application Design

Interaction Model

Composed of four elements:

- Use-cases
- Sequence diagrams
- State diagrams
- User interface prototype

Web Application Design

Interaction Model - Sequence diagram

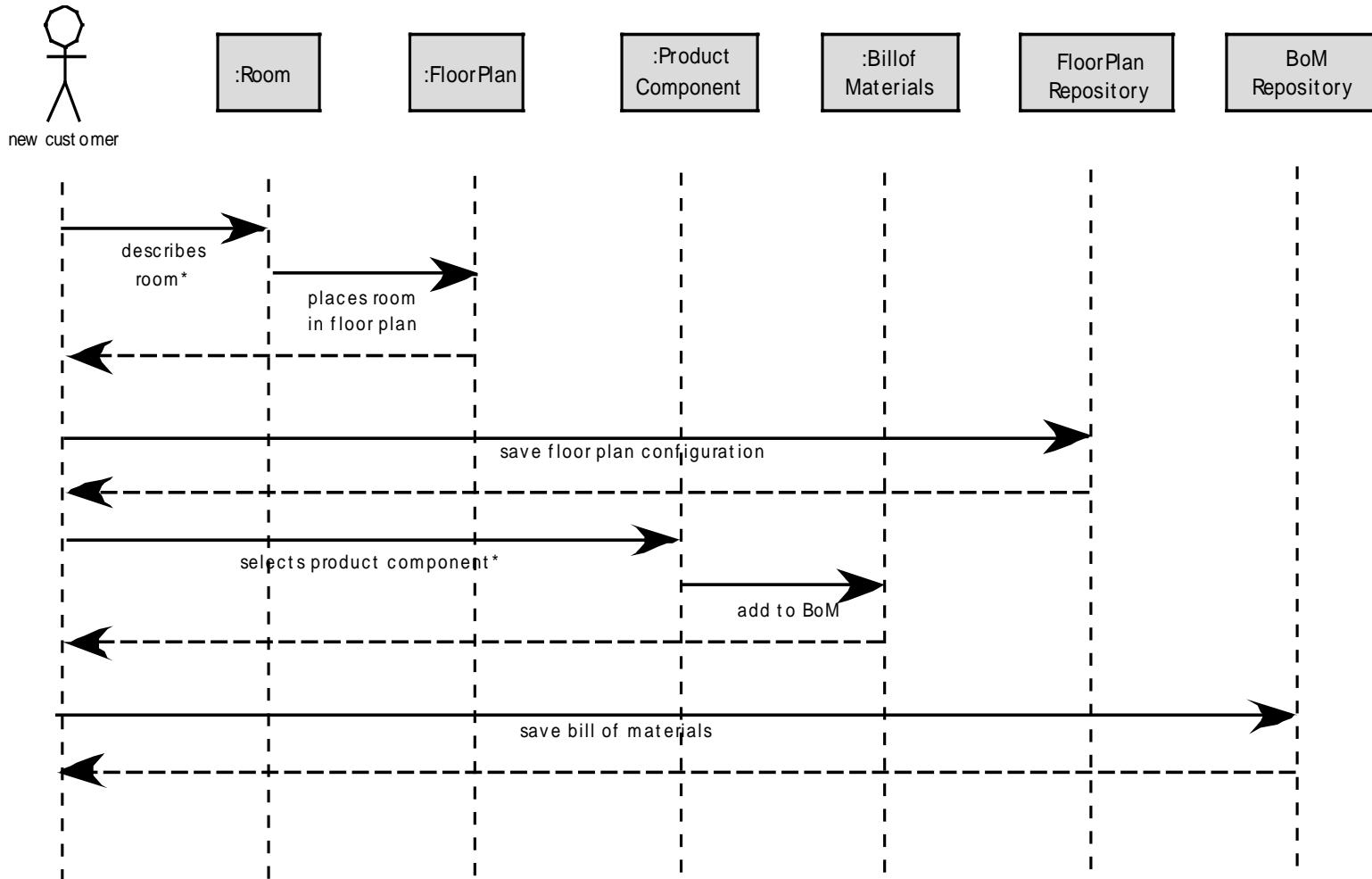


Figure 18.5 Sequence diagram for use-case:select *SafeHome components*

Web Application Design

Interaction Model - State diagram

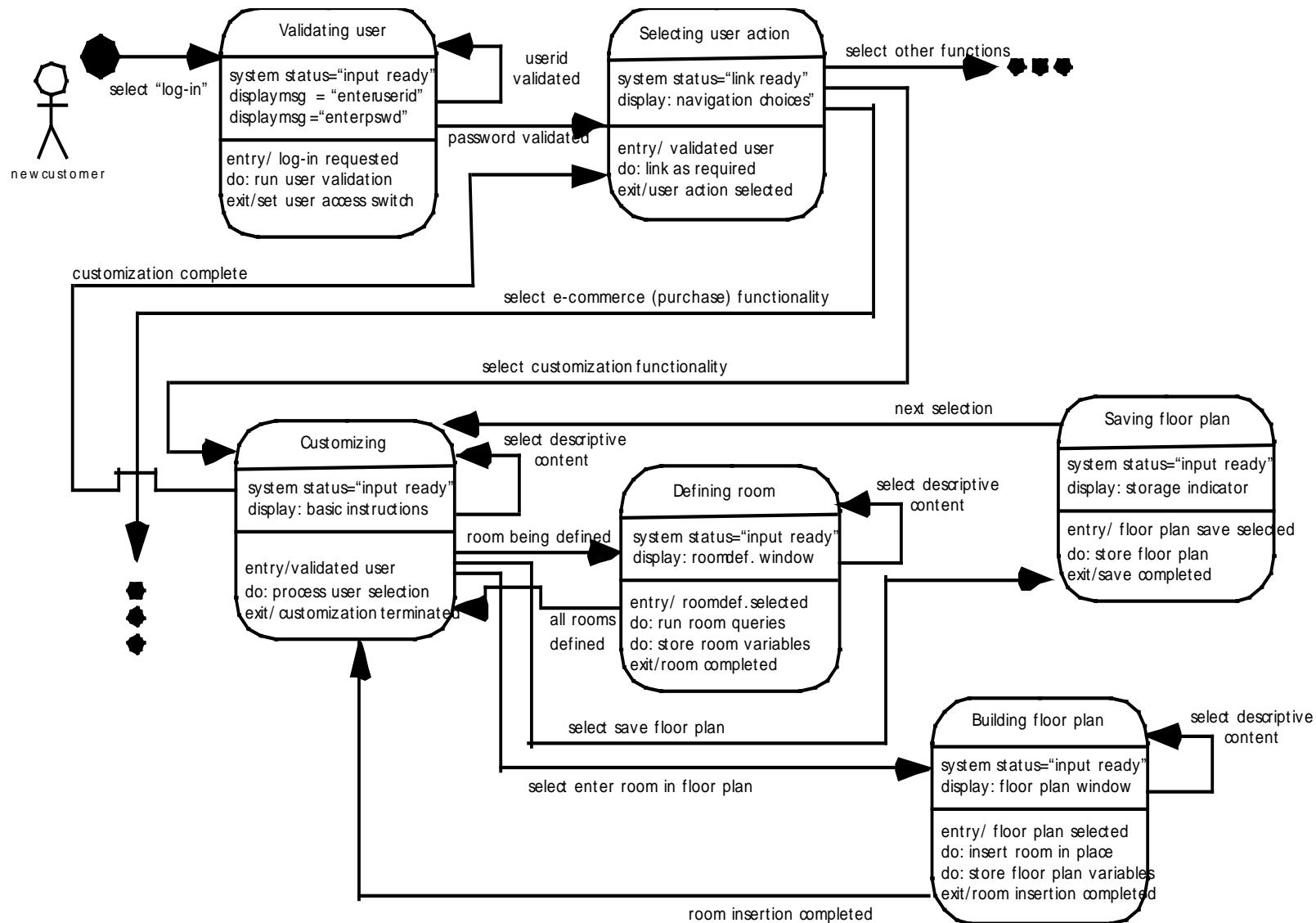


Figure 18.6 Partial state diagram for **new customer** interaction

Web Application Design

Functional Model

- The functional model addresses two processing elements of the WebApp
 - User observable functionality that is delivered by the WebApp to end-users
 - The operations contained within analysis classes that implement behaviors associated with the class.
- An activity diagram can be used to represent processing flow

Web Application Design

Functional Model - Activity diagram

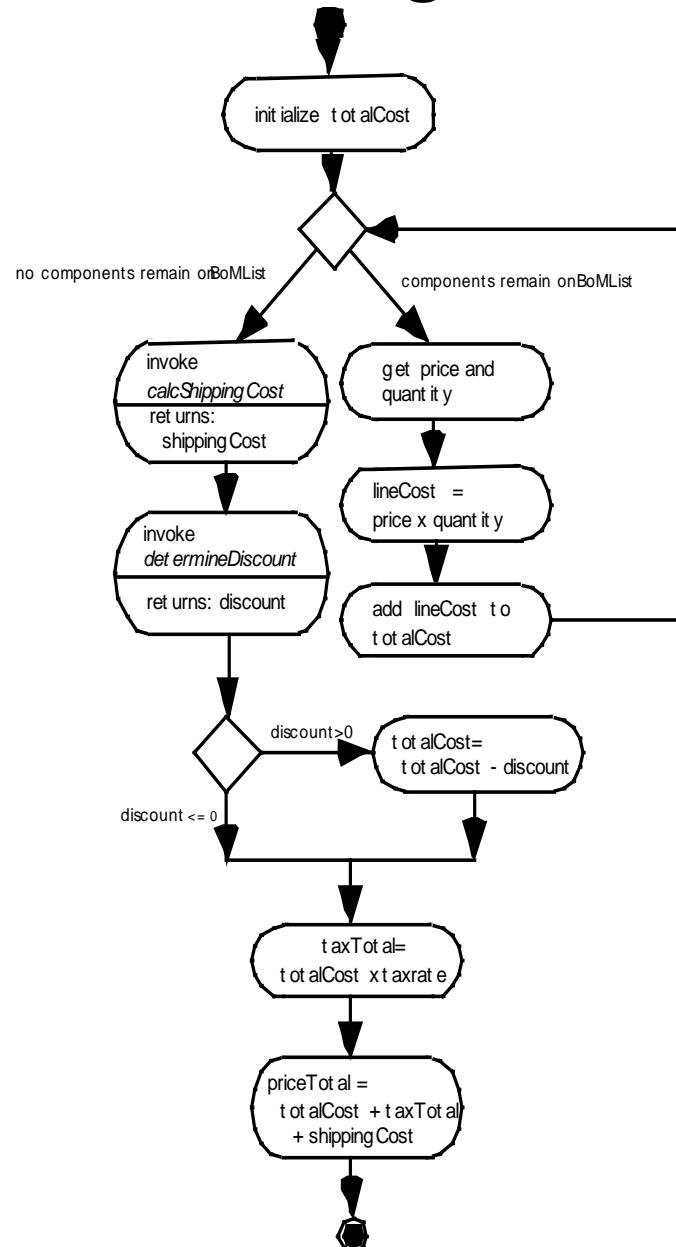


Figure 18.7 Activity diagram for **computePriceOperatic**

Web Application Design

Configuration Model

- Server-side
 - Server hardware and operating system environment must be specified
 - Interoperability considerations on the server-side must be considered
 - Appropriate interfaces, communication protocols and related collaborative information must be specified
- Client-side
 - Browser configuration issues must be identified
 - Testing requirements should be defined

Web Application Design

Navigation Modeling

- Should certain elements be easier to reach (require fewer navigation steps) than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element.
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?

Web Application Design

Navigation Modeling

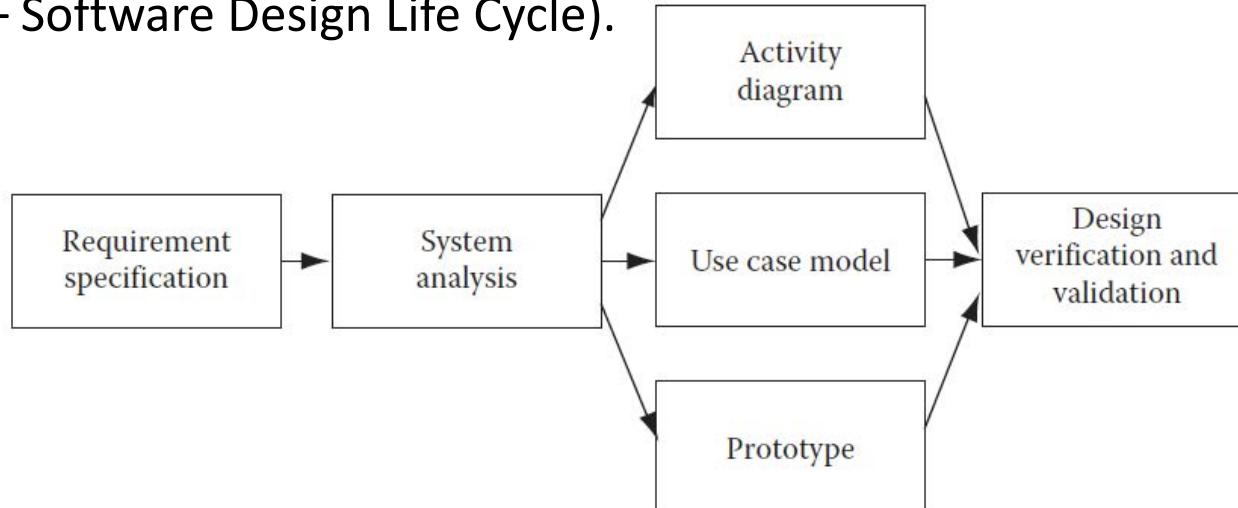
- Should a full navigation map or menu (as opposed to a single “back” link or directed pointer be available at every point in a user’s interaction?)
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user “store” his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled, whether it is by overlaying the existing browser window or as a new browser window or as a separate frame?

Concurrent Engineering in Software Design

- Concurrent engineering deals with taking advance information from an earlier stage for a later stage in project, so that both the stages can be performed simultaneously.
- Though project activities are planned ahead in time, most often there are dependencies between a previous task and the next task in line.
- So, the latter task cannot start until the previous task finishes.
- That is why it is significant to be aware that it is not feasible to initiate developing an application until its design is complete.
- Moreover, the development will depend on the design.
- Until all details about design are made, development cannot be started.
- So, the development team cannot start their job until they have a software design in their hands.
- Still some aspects about latter tasks can be done in advance.
- For instance, what development language will be used and how the application can be partitioned for development work can be decided at the design stage itself.
- Similarly, how maintenance and support functions will be done for the application can be determined at the design stage itself.
- Knowing in advance helps in taking care of issues that may arise in later stages.

Design Life-Cycle Management

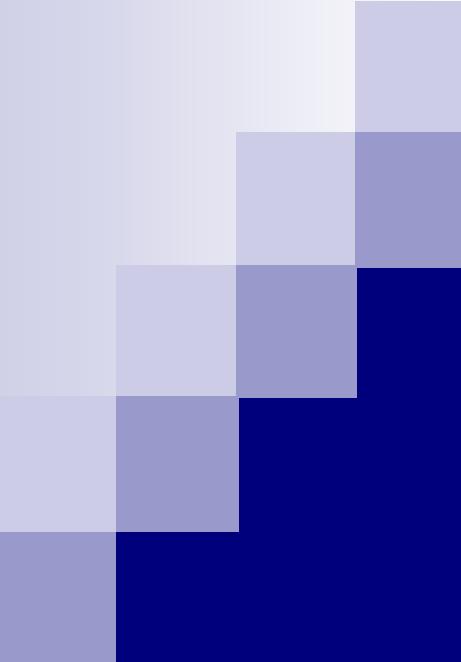
- Software requirements go through design process steps to become a full-fledged software design.
- At the high level, system analysis is performed.
- System analysis includes a study of requirements and finding feasibility of converting them into software design.
- Once the feasibility is done, then the actual software design is made.
- The software design is in the form of activity diagrams, use cases, prototypes, etc.
- Once the design process is complete, these design documents are verified and validated through design reviews.
- Once the design is reviewed and approved, then the design phase is over (Figure below – Software Design Life Cycle).



REFERENCES

- Ashfaque Ahmed, Software Project Management: A Process-driven approach, Boca Raton, Fla: CRC Press, 2012
- Roger S. Pressman, Software Engineering – A Practitioner Approach, 6th ed., McGraw Hill, 2005
- Ian Somerville, Software Engineering, 8th ed., Pearson Education, 2010

THANK YOU



UNIT - IV

Software Construction

Disclaimer:

The lecture notes have been prepared by referring to many books and notes prepared by the teachers. This document does not claim any originality and cannot be used as a substitute for prescribed textbooks.

Topics

- Software Construction
- Coding Standards
- Coding Framework
- Reviews – Desk Checks (Peer Reviews)
- Walkthroughs
- Code Reviews, Inspections
- Coding Methods
- Structured Programming
- Object-Oriented Programming
- Automatic Code Generation
- Software Code Reuse
- Pair Programming
- Test-Driven Development
- Configuration Management
- Software Construction Artefacts

Software Construction

Introduction

- A layman believes that software construction is the entire software development process.
- But, in fact, it is just one of the crucial tasks in software development; software requirement management, software design, software testing and software deployment are all equally crucial tasks.
- Furthermore, the process of software construction itself consists of many tasks; it not only includes software coding but also unit testing, integration testing, reviews and analysis.
- Construction is one of the most labor intensive phases in the software development life cycle.
- It comprises 30% or more of the total effort in software development.
- What a user sees as the product at the end of the software development life cycle is merely the result of the software code that was written during software construction.
- Due to the labor intensive nature of the software construction phase, the work is divided not only among developers, but also small teams are formed to work on parts of the software build.

Software Construction

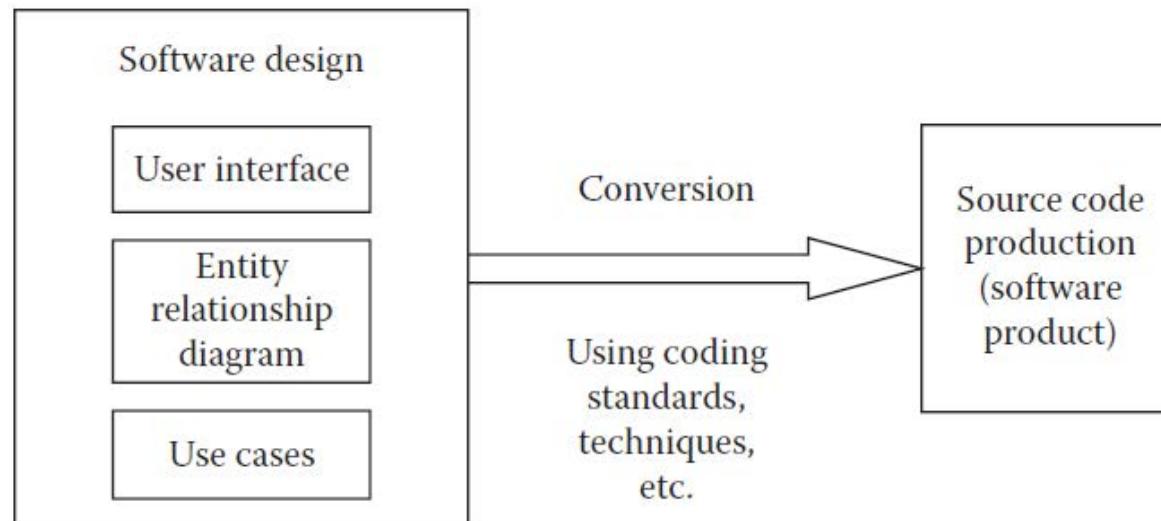
Introduction

- In fact, to **shrink the construction time**, many distributed teams, either internal or through contractors are deployed.
- The advantage to this is that these project teams do the software coding and other construction **work in parallel with each** other and thus the construction phase can be collapsed.
- This parallel development is known as **concurrent engineering**.
- Constructing an industry strength software product of a large size requires stringent coding standards.
- The whole process of construction should follow a proven process so that the produced code is maintainable, testable and reliable.
- The process itself should be efficient so that resource utilization can be optimized and thus cost of construction can be kept at a minimum.

Software Construction

Coding Standards

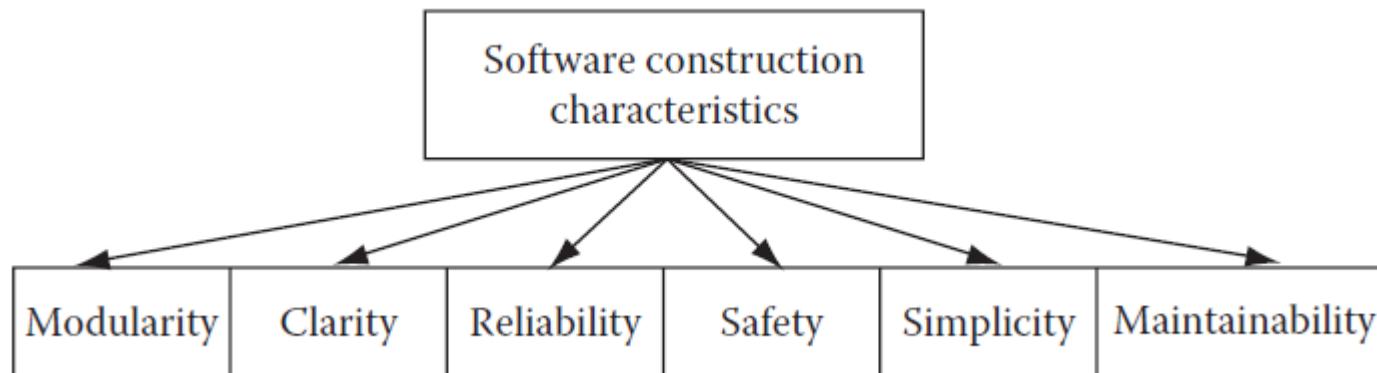
- Developers are given software design specifications in the form of use cases, flow diagrams, UI mock ups, etc., and they are supposed to write a code so that the built software matches these specifications.
- Converting the specifications into software code is totally dependent on the construction team.
- How well they do it depends on their experience, skills and the process they follow to do their job.
- Apart from these facilities, they also need some standards in their coding so that the work is fast as well as has other benefits like maintainability, readability and reusability (*Figure - Source Code Production (Conversion) from Software Design*).



Software Construction

Coding Standards

- At any time, a code written by a developer will always be different from that written by any other developer.
- This poses a challenge in terms of comprehending the code while reusing the code, maintaining it, or simply reviewing it.
- A uniform coding standard across all construction teams working on the same project will make sure that these issues can be minimized if not eliminated (Figure below - Software Construction Characteristics).
- Some of the coding standards include standards for code modularity, clarity, simplicity, reliability, safety and maintainability.



Software Construction

Coding Standards – Modularity

- The produced software code should be modular in nature.
- Each major function should be contained inside a software code module.
- The module should contain not only structure, but it should also process data.
- Each time a particular functionality is needed in the software construction, it can be implemented using that particular module of software code.
- This increases software code reuse and thus enhances productivity of developers and code readability.

Coding Standards – Clarity

- The produced code **should be clear for any person who would read the source code.**
- Standard **naming conventions** should be used so that the code has ample clarity.
- There should be **sufficient documentation** inside the code block, so that anybody reading the code could understand what a piece of code is supposed to do.
- There should also be **ample white spaces** in the code blocks, so that no piece of code should look crammed. White spaces enhance readability of written code.

Software Construction

Coding Standards – Simplicity

- The source code should have simplicity and **no unnecessary complex logic**; improvisation should be involved, if the same functionality can be achieved by a simpler piece of source code.
- Simplicity makes the code readable and will help in removing any defects found in the source code.
- Simplicity of written code can be enhanced by adopting best practices for many programming paradigms.
- For instance, in the case of object-oriented programming, abstraction and information hiding add a great degree of simplicity.
- Similarly, breaking the product to be developed into meaningful pieces that mimic real life parts makes the software product simple.

Coding Standards – Reliability

- Reliability is one of the most important aspects of industry strength software products.
- If the software product is not reliable and contains critical defects, then it will not be of much use for end users.

Software Construction

Coding Standards – Reliability

- Reliability of source code can be **increased by sticking to the standard processes** for software construction.
- During reviews, if any defects are found, they can be fixed easily if the source code is neat, simple, and clear.
- Reliable source code can be achieved by **first designing the software product with future enhancement in consideration** as well as by having a solid structure on which the software product is to be built.
- When writing pieces of source code based on this structure, there will be little chance of defects entering into the source code.
- Generally during enhancements, the existing structure is not able to take load of additional source code and thus the structure becomes shaky.
- If the development team feels that this is the case, then it is far better to restructure the software design and then write a code based on the new structure than to add a spaghetti code on top of a crumbling structure.

Software Construction

Coding Standards – Safety

- Safety is important, considering that software products are used by many industries where human lives are concerned and that human lives could be in danger because of faulty machine operation or exposure to a harmful environment.
- In these industries, the software product must be ensured to operate correctly and chances of error are less than 0.00001%.
- Industries like **medicine and healthcare, road safety**, hazardous material handling need foolproof software products to ensure that either human lives are saved (in case of medicine and healthcare) or human lives are not in danger.
- Here the software code must have inbuilt safety harnesses.

Coding Standards – Maintainability

- As it has been pointed out after several studies, maintenance costs are more than 70% of all costs including software development, implementation, and maintenance.
- To make sure that maintenance costs are under limit during software construction, it should be made sure that the source code is maintainable.
- It will be easy to change the source code for fixing defects during maintenance.

Software Construction

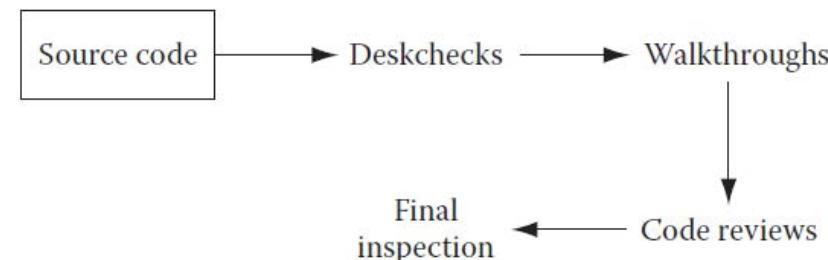
Coding Framework

- Like most construction work, you need to set up an infrastructure based on which construction can take place.
- For software construction, you need to have a coding framework that will ensure a consistent coding production with standard code that will be easy to debug and test.
- In object oriented programming, what base classes are to be made, which will be used throughout construction, is a subject that is part of the coding framework.
- In general, coding frameworks allow construction of the common infrastructure of basic functionality which can be extended later by the developers.
- This way of working increases productivity and allows for a robust and well structured software product.
- It is similar in approach to house building where a structure is built based on a solid foundation.

Software Construction

Reviews (Quality Control)

- It is estimated that almost 70% of software defects arise from faulty software code.
- To compound this problem, software construction is the most labor intensive phase in software development.
- Any construction rework means wasting a lot of effort already put in.
- Moreover, it is also a fact that it is cheaper to fix any defects found during construction at the phase level itself.
- If those defects are allowed to go in software testing (which is the next phase), then fixing those defects will become costlier.
- That is why review of the software code and fixing defects is very important.
- There are some techniques available like deskchecks, walkthroughs, code reviews, inspections, etc. that ensure quality of the written code (Figure below- Source code review methods and their operation sequence).



Software Construction

Reviews (Quality Control)

- These different kinds of reviews are done at different stages in software code writing.
- They also serve different purposes.
- While inspections provide the final go/no go decision for approval of a piece of code, other methods are less formal and are meant for removing defects instead of deciding whether a piece of code is good enough or not.

Reviews – Deskchecks (Peer Reviews)

- Deskchecks are employed when a complete review of the source code is not important.
- Here, the developer sends his piece of **code to the designated team members**.
- These team members review the code and send feedback and comments to the developer as suggestions for improvement in the code.
- The developer reads those feedbacks and may decide to incorporate or to discard those suggestions.
- So this form of review is totally **voluntary**.
- Still, it is a powerful tool to eliminate defects or improve software code.

Software Construction

Reviews – Walkthroughs

- Walkthroughs are formal code reviews initiated by the developer. The developer sends an invitation for **walkthrough to team members**.
- At the meeting, the **developer presents his method of coding** and walks through his piece of code.
- The team members then make **suggestions for improvement**, if any.
- The **developer then can decide to incorporate** those suggestions or discard them.

Reviews – Code Reviews

- Code reviews are one of the **most formal methods of reviews**. The project manager calls for a meeting for code review of a developer.
- At the meeting, team members review the code and point out any code errors, defects, or improper code logic for likely defects. An **error log is also generated and is reviewed by the entire team**.

Reviews – Inspections

- Code inspections are **final reviews of software code** in which it is decided whether to pass a piece of code for inclusion into the main software build.

Software Construction

Coding Methods

- Converting design into optimal software construction is a very serious topic that has generated tremendous interest over the years.
- Many programming and coding methods were devised and evolved as a result.
- As it is well known in the industry, the early software products were of small size due to limited hardware capacity.
- With increasing hardware capacity, the size of software products has been increasing.
- Software product size affects the methods that can be used to construct specific sized software products.
- Advancement in the field of computer science also allows discovery of better construction methods.
- To address needs of different sized software products in tandem with advancement in computer science, different programming techniques evolved.
- These include **structured programming, object-oriented programming, automatic code generation, test-driven development, pair programming**, etc.

Software Construction

Coding Methods – Structured Programming

- Structured programming evolved after mainframe computers became popular.
- Mainframe computers offered vast availability of computing power compared to primitive computers that existed before.
- Using structured programming, large programs could be constructed that could be used for making large commercial and business applications.
- Structured programming enabled programmers to store large pieces of code inside procedures and functions.
- These pieces of code could be called by any other procedures or functions.
- This enabled programmers to structure their code in an efficient way.
- Code stored inside procedures could be reused anywhere in the application by calling it.

Software Construction

Coding Methods – Object-Oriented Programming

- In structured programming, data and structured code are separate and accordingly they are modeled separately.
- This is an unnatural way of converting real life objects into software code because objects contain both data and structure.
- Widely used as an example in object-oriented programming books, a car consists of a chassis, an engine, four wheels, body, and transmission.
- Each of these objects has some specific properties and specific functions.
- When a software system is modeled to represent real-world objects, both data and structure are taken care of in object-oriented programming.
- From outside of a class that is made to represent an object, only the behavior of the object is visible or perceived.
- Unnecessary details about the object are hidden and in fact are not available from outside.
- This kind of representation of objects makes them robust and a system built on using them has relatively few problems.

Software Construction

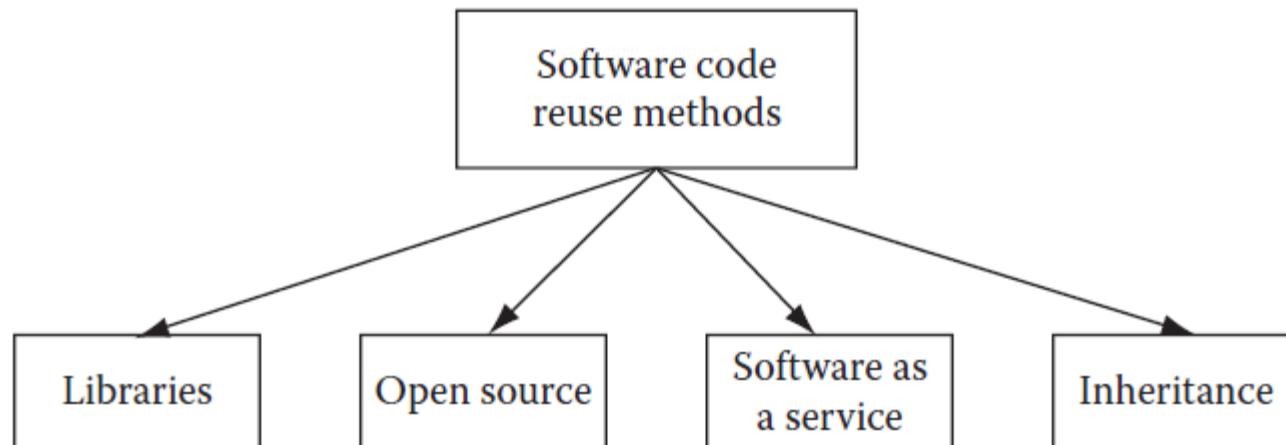
Coding Methods – Automatic Code Generation

- Constructing and generating software code is very labor intensive work. So there has always been fascination about automatic generation of software code.
- Unfortunately, this is still a dream. Some CASE and modeling tools are available that generate software code. But they are not sophisticated. They are also not complete.
- Then there are business analyst platforms developed by many ERP software vendors that generate code automatically when analysts configure the product.
- These analyst platforms are first built using any of the software product development methodologies.
- The generated code is specific to the platform and runs on the device (hardware and software environment) for which the code is generated.
- Generally, any code consists of many construction unit types.
- Some of these code types include control statements such as loop statements, if statements, etc., and database access, etc.
- Generating all of the software code required to build a software application is still difficult.
- But some companies like Sun Microsystems are working to develop such a system.

Software Construction

Coding Methods – Software Code Reuse

- Many techniques have evolved to reduce the labor intensive nature of writing source code.
- Software code reuse is one such technique.
- Making a block of source code to create a functionality or general utility library and using it at all places in the source code wherever this kind of functionality or utility is required is an example of code reuse.
- Code reuse in procedural programming techniques is achieved by creating special functions and utility libraries then using them in the source code.
- In object-oriented programming, code reuse is done at a more advanced level.



Software Construction

Coding Methods – Software Code Reuse

- The classes containing functions and data themselves can not only be reused in the same way as functions and libraries but the classes can also be modified by way of creating child classes and using them in the source code (Figure above – Code reuse methods).
- Apart from creating and using libraries and general purpose classes for code reuse, a more potent code reuse source has evolved recently.
- It is known as “service oriented architecture” (SOA).



Software Construction

Coding Methods – Pair Programming

- Pair programming is a quality driven development technique employed in the eXtreme Programming development model.
- Here, each development task is assigned to two developers.
- While one developer writes the code, the other developer sits behind him and guides him through the requirements (functional, nonfunctional).
- When it is the turn of the other developer to write the code, the first developer sits behind him and guides him on the requirements.
- So developers take turns for the coding and coaching work.
- This makes sure that each developer understands the big picture and helps them to write better code with lesser defects.

Software Construction

Coding Methods – Test-Driven Development

- This concept is used with iteration-based projects especially with eXtreme Programming technique.
- Before developers start writing source code, they create test cases and run the tests to see if they run properly and their logic is working.
- Once it is proved that their logic is perfect, only then they write the source code.
- So here, tests drive software development, and hence it is appropriately named test-driven development.



Testing Strategies

Strategic approach to software testing

- Generic characteristics of strategic software testing:
 - To perform effective testing, a software team should conduct effective *formal technical reviews*. By doing this, many errors will be eliminated before testing start.
 - Testing begins at the *component level* and works "*outward*" toward the integration of the entire computer-based system.
 - Different *testing techniques* are appropriate at different points in *time*.
 - Testing is conducted by the developer of the software and (for large projects) an independent test group.
 - *Testing and debugging are different activities*, but debugging must be accommodated in any testing strategy.

Verification and Validation

- Testing is one element of a broader topic that is often referred to as *verification and validation (V&V)*.
- *Verification* refers to the set of activities that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
- State another way:
 - *Verification*: "Are we building the product right?"
 - *Validation*: "Are we building the right product?"
- The definition of V&V encompasses many of the activities that are similar to *software quality assurance (SQA)*.

- V&V encompasses a wide array of **SQA(sw quality assurance) activities** that include
 - Formal technical reviews,
 - quality and configuration audits,
 - performance monitoring,
 - simulation,
 - feasibility study,
 - documentation review,
 - database review,
 - algorithm analysis,
 - development testing,
 - qualification testing, and installation testing
- Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered.
- Quality is not measure only by no. of error but it is also measure on application methods, process model, tool, formal technical review, etc will lead to quality, that is confirmed during testing.

Verification and Validation

- Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase .
 - Does the product meet its specifications?
 - Are we building the **product right?**
- Methods of Verification : Static Testing
 - Walkthrough
 - Inspection
 - Review

Verification and Validation

- Validation is the process of evaluating a software system or component during, or at the end of, the development cycle in order to determine whether it satisfies specified requirements
- Does the product perform as desired?
- Are we building the **right product**
- Methods of Validation : [Dynamic Testing](#)
 - Testing

Example of verification and validation

- In Software Engineering, consider the following specification

A clickable button with name Submet

- Verification would check the design doc and correcting the spelling mistake.
- Otherwise, the development team will create a button like



Submet

Example for Verification and Validation

A clickable button with name Submit

- Once the code is ready, Validation is done. A Validation test found –

Button NOT Clickable



- Owing to Validation testing, the development team will make the submit button clickable

Difference between Verification and Validation

- | | |
|--|---|
| <p>1. Verification is a static practice of verifying documents, design, code and program.</p> | <p>1. Validation is a dynamic mechanism of validating and testing the actual product.</p> |
| <p>2. It does not involve executing the code.</p> | <p>2. It always involves executing the code.</p> |
| <p>3. It is human based checking of documents and files.</p> | <p>3. It is computer based execution of program.</p> |
| <p>4. Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.</p> | <p>4. Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.</p> |

Difference between Verification and Validation

5. **Verification** is to check whether the software conforms to specifications.

6. It can catch errors that validation cannot catch. It is low level exercise.

7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.

8. **Verification** is done by QA team to ensure that the software is as per the specifications in the SRS document.

9. It generally comes first-done before validation.

5. **Validation** is to check whether software meets the customer expectations and requirements.

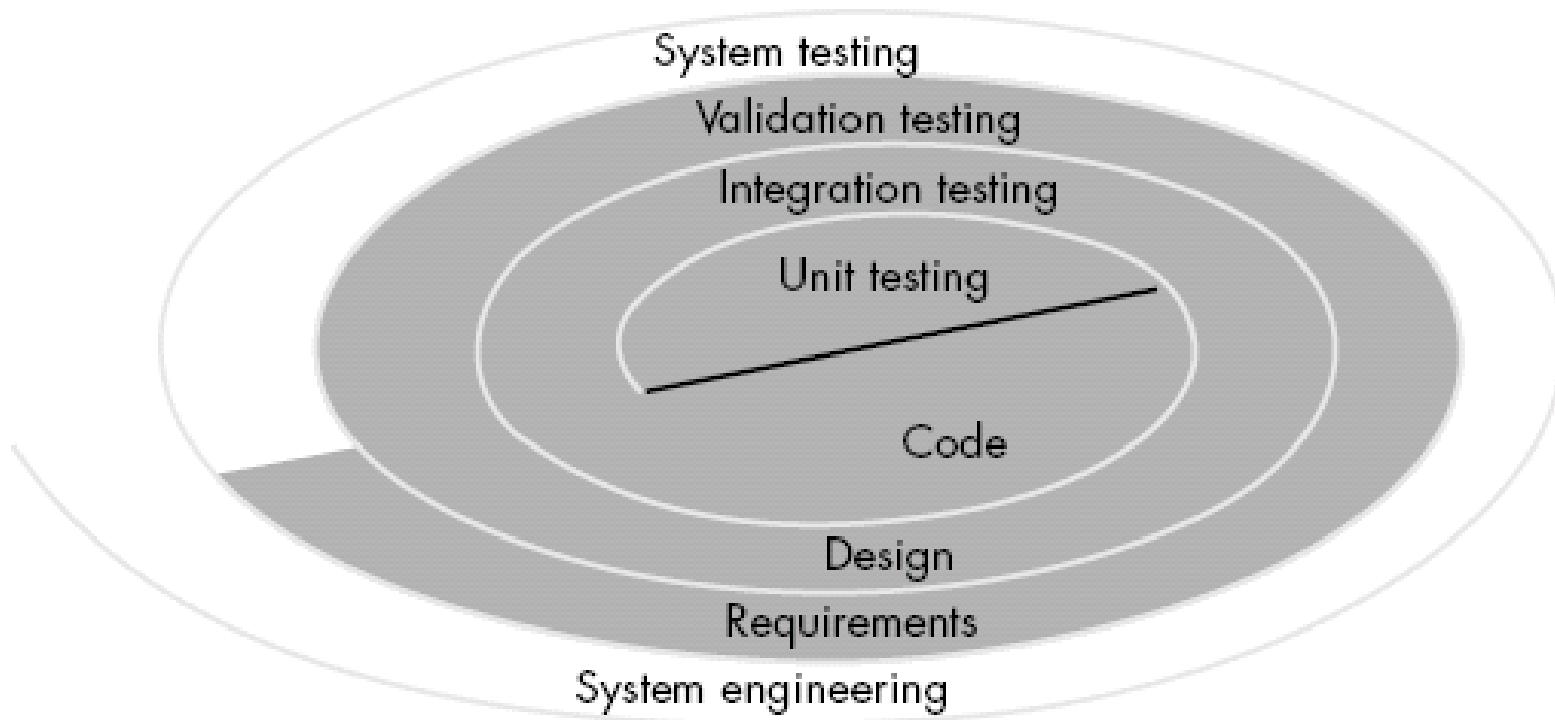
6. *It can catch errors that verification cannot catch. It is High Level Exercise.*

7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.

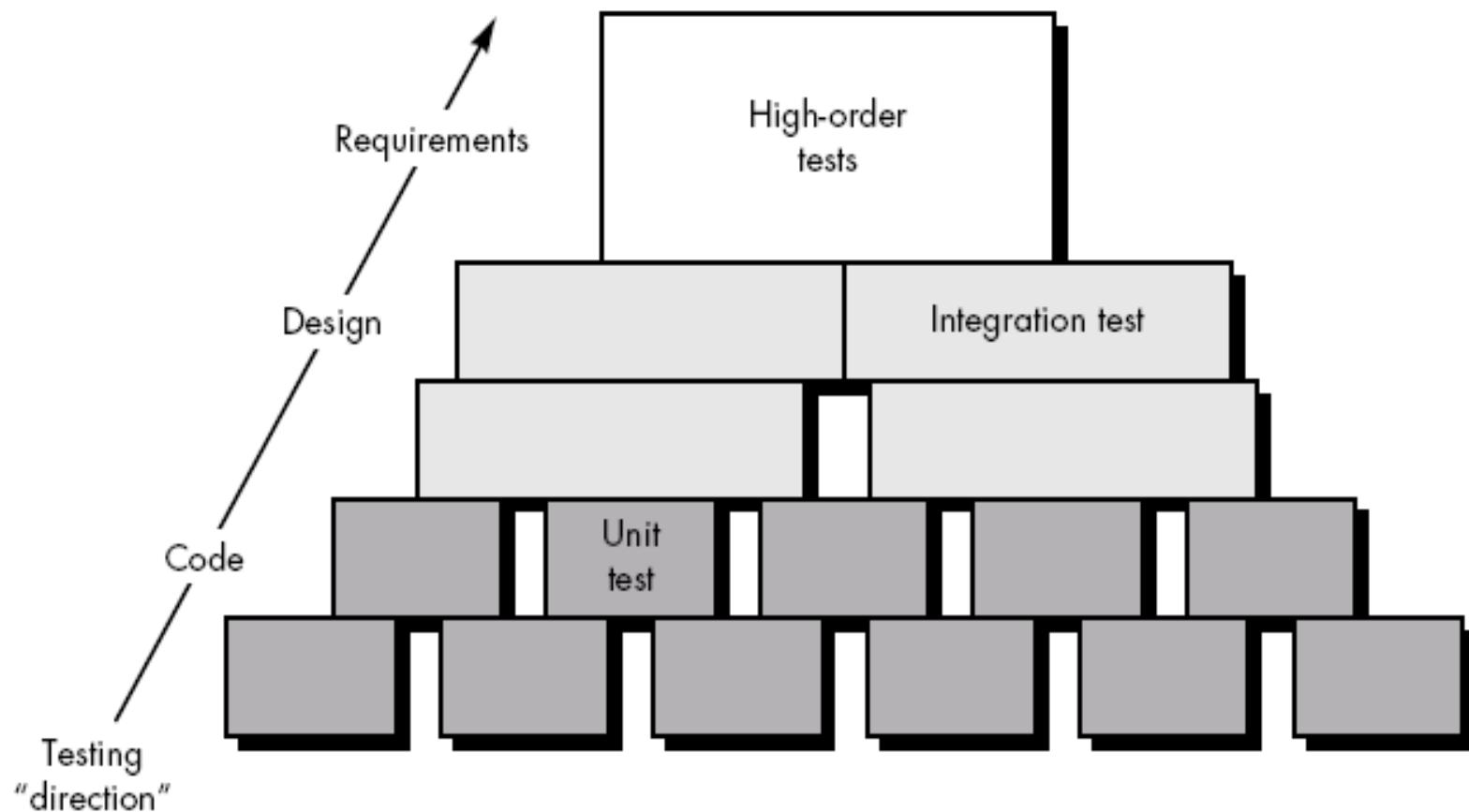
8. **Validation** is carried out with the involvement of testing team.

9. It generally follows after **verification**.

Software Testing Strategy for conventional software architecture



- A *Software process & strategy for software testing* may also be viewed in the context of the *spiral*.
- *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software.
- Testing progresses by moving outward along the spiral to *integration testing*, where the focus is on design and the construction.
- Another turn outward on the spiral, we encounter *validation testing*, where requirements established as part of software requirements analysis are validated against the software.
- Finally, we arrive at *system testing*, where the software and other system elements are tested as a whole.



- Software process from a procedural point of view; a series of four steps that are implemented sequentially.

- Initially, tests focus on each component individually, ensuring that it functions properly as a unit.
- Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure.
- Integration testing addresses the issues associated with the dual problems of verification and program construction.
- Black-box test case design techniques are the most prevalent during integration.
- Now, validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.
- Black-box testing techniques are used exclusively during validation.
- once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function / performance is achieved.

Criteria for Completion of Testing

- There is no definitive answer to state that “we have done with testing”.
- One response to the question is: "You're never done testing, the burden simply shifts from you (the software engineer) to your customer." Every time the customer/ user executes a computer program, the program is being tested.
- Another response is: "You're done testing when you run out of time (deadline to deliver product to customer) or you run out of money (*spend so much money on testing*)."

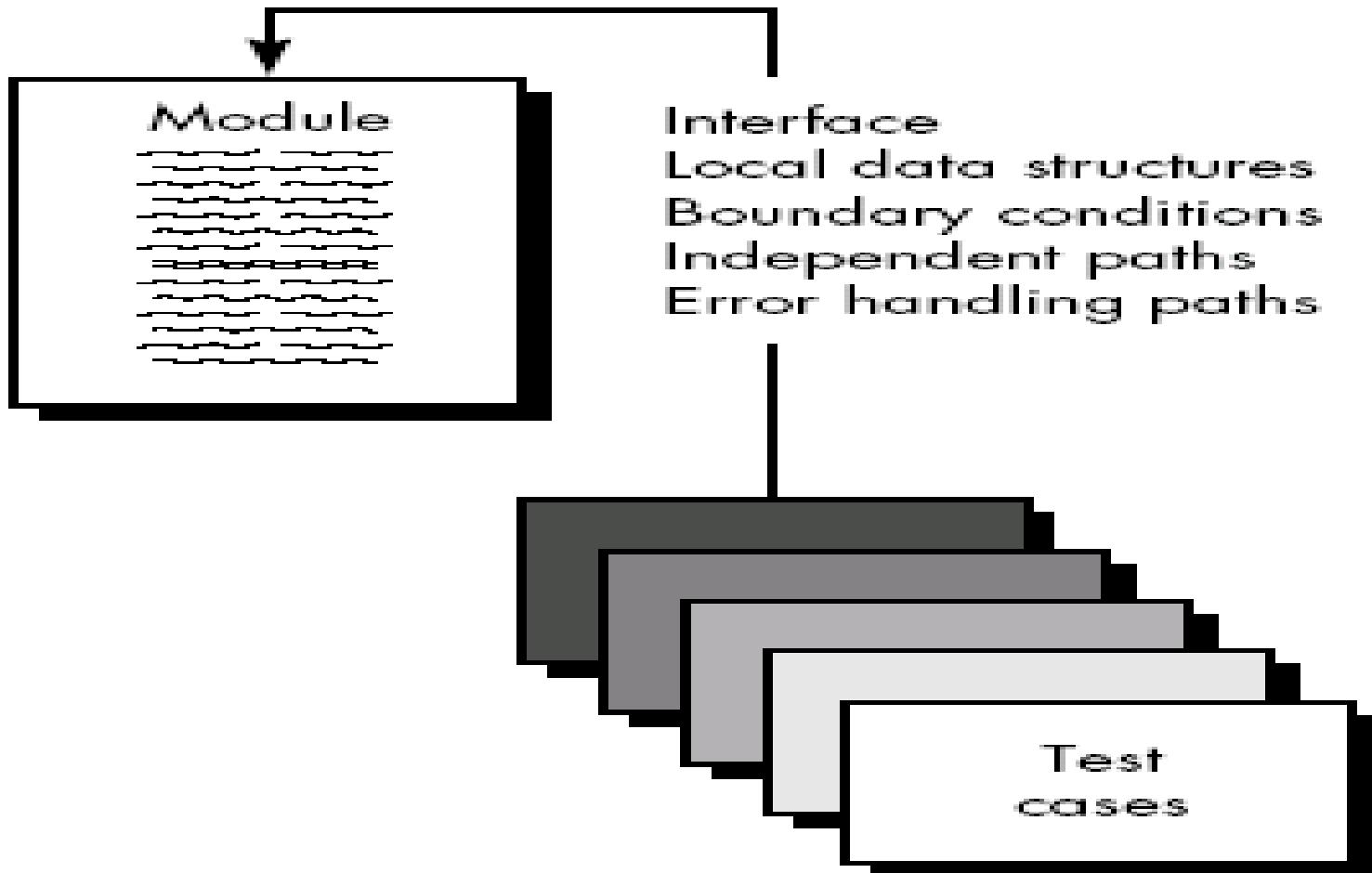
- But few practitioners would argue with these responses, a software engineer needs more rigorous criteria for determining when sufficient testing has been conducted.
- Response that is based on statistical criteria: "No, we cannot be absolutely predict that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95 percent confidence that program will not fail."



Unit testing strategies for conventional software

- Focuses verification effort on the smallest unit of software design – component or module.
- Using the component-level design description as a guide
 - important control paths are tested to uncover errors within the boundary of the module.
- Unit test is white-box oriented, and the step can be conducted in parallel for multiple components.
- Unit test consists of
 - Unit Test Considerations
 - Unit Test Procedures

Unit Test Considerations



Contd.

- *Module interface* - information properly flows into and out of the program unit under test.
- *local data structure* - data stored temporarily maintains its integrity.
- *Boundary conditions* -module operates properly at boundaries established to limit or restrict processing
- Independent paths - all statements in a module have been executed at least once.
- And finally, all *error handling paths* are tested.

- *Module interface* are required before any other test is initiated because If data do not enter and exit properly, all other tests are debatable.
- In addition, *local data structures* should be exercised and the local impact on global data should be discover during unit testing.
- Selective testing of *execution paths* is an essential task during the unit test. Test cases should be designed to uncover errors due to
 - Computations,
 - Incorrect comparisons, or
 - Improper control flow
- *Basis path* and loop testing are effective techniques for uncovering a broad array of *path errors*.

Errors are commonly found during unit testing

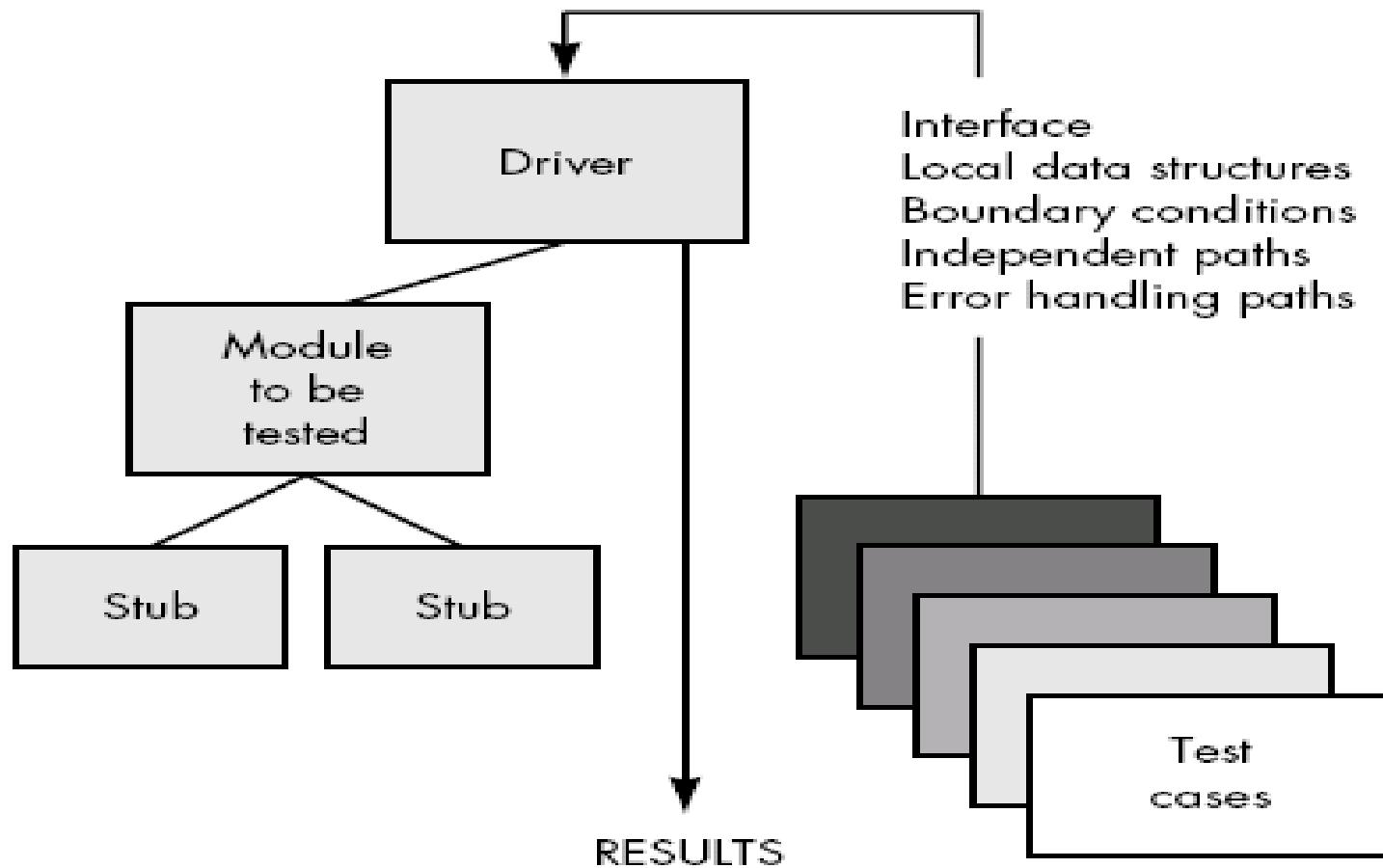
- More common errors in computation are
 - misunderstood or incorrect arithmetic precedence
 - mixed mode operations,
 - incorrect initialization,
 - precision inaccuracy,
 - incorrect symbolic representation of an expression.
- Comparison and control flow are closely coupled to one another
 - Comparison of different data types,
 - Incorrect logical operators or precedence,
 - Incorrect comparison of variables
 - Improper or nonexistent loop termination,
 - Failure to exit when divergent iteration is encountered
 - improperly modified loop variables.

- Potential errors that should be tested when error handling is evaluated are
 - Error description is unintelligible.
 - Error noted does not correspond to error encountered.
 - Error condition causes 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.
- Software often fails at its boundaries. That is, errors often occur when the n th element of an n -dimensional array is processed or when the maximum or minimum allowable value is encountered.
- So BVA test is always be a last task for unit test.
- Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.

Unit Test Procedures

- Perform before coding or after source code has been generated.
- A review of design information provides guidance for establishing test cases. Each test case should be coupled with a set of expected results.
- Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test.
- In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- *Stubs* serve to replace modules that are subordinate the component to be tested.

Unit Test Procedures



Unit Test Environment

- Drivers and stubs represent overhead. That is, both are software that must be written but that is not delivered with the final software product.
- In such cases, complete testing can be postponed until the integration test step
- Unit testing is simplified when a component with high cohesion is designed.
- When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

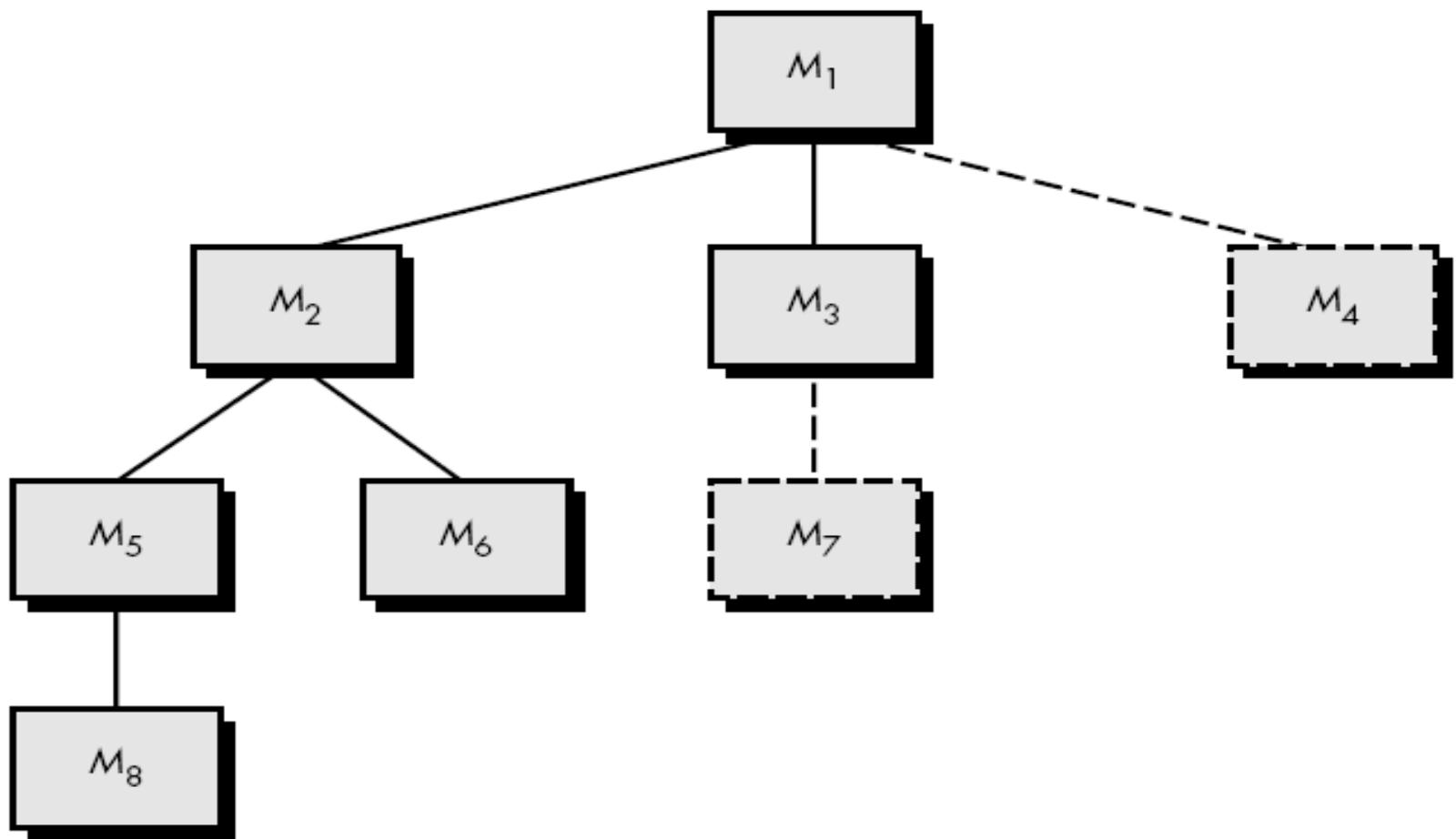
Integration testing

- Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing.
- The objective is to take unit tested components and build a program structure that has been dictated by design.
- There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach.
- A set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.
- Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.
- Incremental integration is the exact opposite of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct;

Top-down Integration

- *Top-down integration testing is an incremental approach to construction of program structure.*
- Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.
- Depth-first integration would integrate all components on a major control path of the structure.
- Selection of a major path is somewhat arbitrary and depends on application-specific characteristics.
- For example, selecting the left hand path,
 - Components M1, M2 , M5 would be integrated first.
 - Next, M8 or M6 would be integrated
 - The central and right hand control paths are built.

Top down integration



- *Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.*
- Step would be:
 - components M2, M3, and M4 would be integrated first
 - next control level, M5, M6, and so on follows.

Top-down Integration process five steps:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

Problem occur in top-down integration

- Logistic problems can arise
- most common problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels.
- No significant data can flow upward in the program structure due to stubs replace low level modules at the beginning of top-down testing. In this case, Tester will have 3 choice
 - Delay many tests until stubs are replaced with actual modules
 - develop stubs that perform limited functions that simulate the actual module
 - integrate the software from the bottom of the hierarchy upward

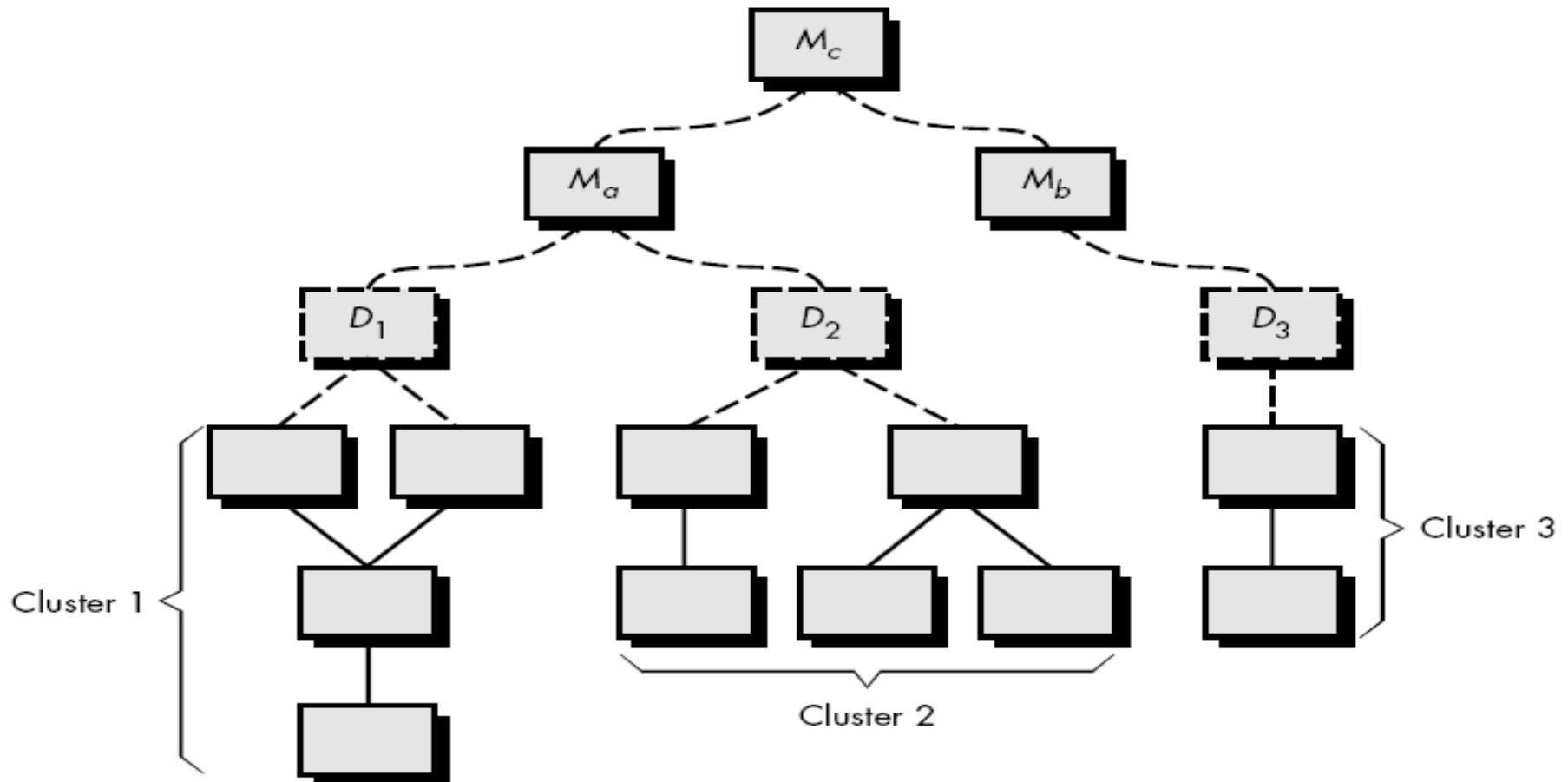
Bottom-up Integration

- *Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure)*
- Because 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.

Bottom up integration process steps

- Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
- A driver (a control program for testing) 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.

Bottom up integration



Example

- Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver.
- Components in clusters 1 and 2 are subordinate to Ma.
- Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb.
- Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

Regression Testing

- Each time a new module is added as part of integration testing
 - New data flow paths are established
 - New I/O may occur
 - New control logic is invoked
- Due to these changes, may cause problems with functions that previously worked flawlessly.
- ***Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.***
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

Contd.

- Regression testing is the activity that helps to ensure that changes do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools*.
- Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.
- Regression testing contains 3 diff. classes of test cases:
 - A representative sample of tests that will exercise all software functions
 - Additional tests that focus on software functions that are likely to be affected by the change.
 - Tests that focus on the software components that have been changed.

Contd.

- As integration testing proceeds, the number of regression tests can grow quite large.
- Regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.
- It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

Smoke Testing

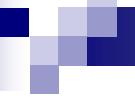
- *Smoke testing* is an integration testing approach that is commonly used when “shrink wrapped” software products are being developed.
- It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis.

Smoke testing approach activities

- Software components that have been translated into code are integrated into a “build.”
 - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
- A series of tests is designed to expose errors that will keep the build from properly performing its function.
 - The intent should be 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.
 - The integration approach may be top down or bottom up.

Smoke Testing benefits

- *Integration risk is minimized.*
 - Smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early
- *The quality of the end-product is improved.*
 - Smoke testing is likely to uncover both functional errors and architectural and component-level design defects. At the end, better product quality will result.
- *Error diagnosis and correction are simplified.*
 - Software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.*
 - Frequent tests give both managers and practitioners a realistic assessment of integration testing progress.



What is a critical module and why should we identify it?

- As integration testing is conducted, the tester should identify *critical modules*.
- A critical module has one or more of the following characteristics:
 - Addresses several software requirements,
 - Has a high level of control (Program structure)
 - Is complex or error prone
 - Has definite performance requirements.
- Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.

Integration Test Documentation

- An overall plan for integration of the software and a description of specific tests are documented in a *Test Specification*
- It contains a test plan, and a test procedure, is a work product of the software process, and becomes part of the software configuration.
- The test plan describes the overall strategy for integration.
- Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software.
- Integration testing might be divided into the following test phases:
 - User interaction
 - Data manipulation and analysis
 - Display processing and generation
 - Database management

Contd.

- Therefore, groups of modules are created to correspond to each phase.
- The following criteria and corresponding tests are applied for all test phases:
- **Interface integrity**- Internal and external interfaces are tested as each module.
- **Functional validity** - Tests designed to uncover functional errors are conducted.
- **Information content** - associated with local or global data structures are conducted.
- **Performance** - to verify performance

Contd.

- A schedule for integration and related topics is also discussed as part of the test plan.
- Start and end dates for each phase are established
- A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort.
- Finally, test environment and resources are described.
- The order of integration and corresponding tests at each integration step are described.
- A listing of all test cases and expected results is also included.
- A history of actual test results, problems is recorded in the *Test Specification*.

Validation Testing

- *Validation testing* succeeds when software functions in a manner that can be *reasonably expected by the customer*.
- Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level— on things that will be immediately apparent to the end-user.
- Reasonable expectations are defined in the *Software Requirements Specification*— a document that describes all user-visible attributes of the software.
- Validation testing comprises of
 - Validation Test criteria
 - Configuration review
 - Alpha & Beta Testing

Validation Test criteria

- It is achieved through a series of tests that demonstrate agreement with requirements.
- A *test plan* outlines the classes of *tests to be conducted* and a *test procedure* defines specific test cases that will be used to demonstrate *agreement with requirements*.
- Both the plan and procedure are designed to ensure that
 - all functional requirements are satisfied,
 - all behavioral characteristics are achieved,
 - all performance requirements are attained,
 - documentation is correct,
 - other requirements are met
- After each validation test case has been conducted, one of two possible conditions exist:
 1. The function or performance characteristics conform to specification and are accepted
 2. A deviation from specification is uncovered and a deficiency list is created

Configuration Review

- The intent of the review is to ensure that all elements of the software configuration have been *properly developed, are cataloged*, and have the necessary detail to the support phase of the software life cycle.
- The configuration review, sometimes called an *audit*.

Alpha and Beta Testing

- 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.
- Most software product builders use a process called alpha and beta testing to uncover errors that only the end-user seems able to find.

Alpha testing

- The *alpha test* is conducted at the developer's site by a customer.
- The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems.
- Alpha tests are conducted in a controlled environment.

Beta testing

- The *beta test* is conducted at one or more customer sites by the end-user of the software.
- beta test is a "live" application of the software in an environment that cannot be controlled by the developer.
- The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.
- As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

System Testing

- System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.
- Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.
- Types of system tests are:
 - Recovery Testing
 - Security Testing
 - Stress Testing
 - Performance Testing

Recovery Testing

- *Recovery testing* is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic (performed by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.
- If recovery requires human intervention, that is mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

Security Testing

- *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper break through .
- During security testing, the tester plays the role(s) of the individual who desires to break through the system.
- Given enough time and resources, good security testing will ultimately penetrate a system.
- The role of the system designer is to make penetration cost more than the value of the information that will be obtained.
- The tester may attempt to acquire passwords through externally, may attack the system with custom software designed to breakdown any defenses that have been constructed; may browse through insecure data; may purposely cause system errors.

Stress Testing

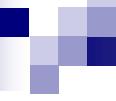
- *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

For example,

1. special tests may be designed that generate ten interrupts per second
 2. Input data rates may be increased by an order of magnitude to determine how input functions will respond
 3. test cases that require maximum memory or other resources are executed
 4. test cases that may cause excessive hunting for disk-resident data are created
- A variation of stress testing is a technique called *sensitivity testing*

Performance Testing

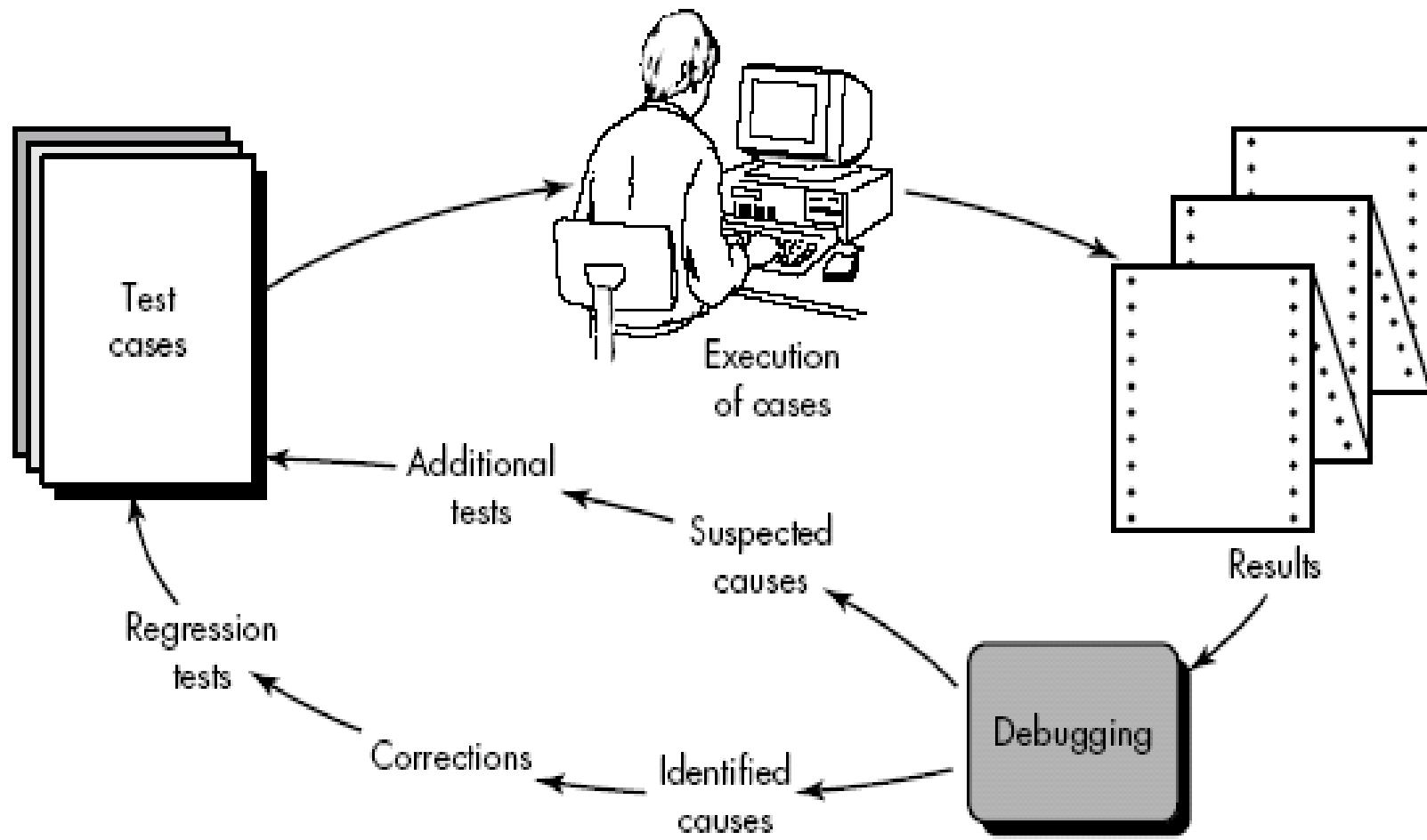
- Performance testing occurs throughout all steps in the testing process.
- Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation
- It is often necessary to measure resource utilization (e.g., processor cycles).



THE ART OF DEBUGGING

- Debugging is the process that results in the removal of the error.
- Although debugging can and should be an orderly process, it is still very much an art.
- Debugging is not testing but always occurs as a consequence of testing.

Debugging Process



Debugging Process

- Results are examined and a lack of correspondence between expected and actual performance is encountered (due to cause of error).
- Debugging process attempts to match symptom with cause, thereby leading to error correction.
- One of two outcomes always comes from debugging process:
 - The cause will be found and corrected,
 - The cause will not be found.
- The person performing debugging may *suspect a cause*, design a test case to help validate that doubt, and work toward error correction in an iterative fashion.

Why is debugging so difficult?

1. The symptom may disappear (temporarily) when another error is corrected.
2. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
3. The symptom may be caused by human error that is not easily traced (e.g. wrong input, wrongly configure the system)
4. The symptom may be a result of timing problems, rather than processing problems.(e.g. taking so much time to display result).
5. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

- 6. The symptom may be intermittent (connection irregular or broken). This is particularly common in embedded systems that couple hardware and software
- 7. The symptom may be due to causes that are distributed across a number of tasks running on different processors

As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

Debugging Approaches or strategies

- Debugging has one overriding objective: to find and correct the cause of a software error.
- Three categories for debugging approaches
 - Brute force
 - Backtracking
 - Cause elimination

Brute Force:

- probably the most common and least efficient method for isolating the cause of a software error.
- Apply brute force debugging methods when all else fails.
- Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE or PRINT statements
- It more frequently leads to wasted effort and time.

Backtracking:

- common debugging approach that can be used successfully in small programs.
- Beginning at the site where a symptom has been open, the source code is traced backward (manually) until the site of the cause is found.

Cause elimination

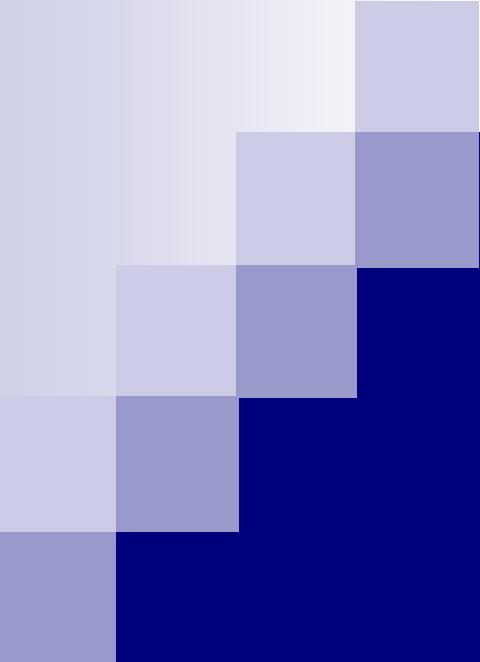
- Is cleared by induction or deduction and introduces the concept of binary partitioning (i.e. valid and invalid).
- A list of all possible causes is developed and tests are conducted to eliminate each.

Correcting the error

- The correction of a bug can introduce other errors and therefore do more harm than good.

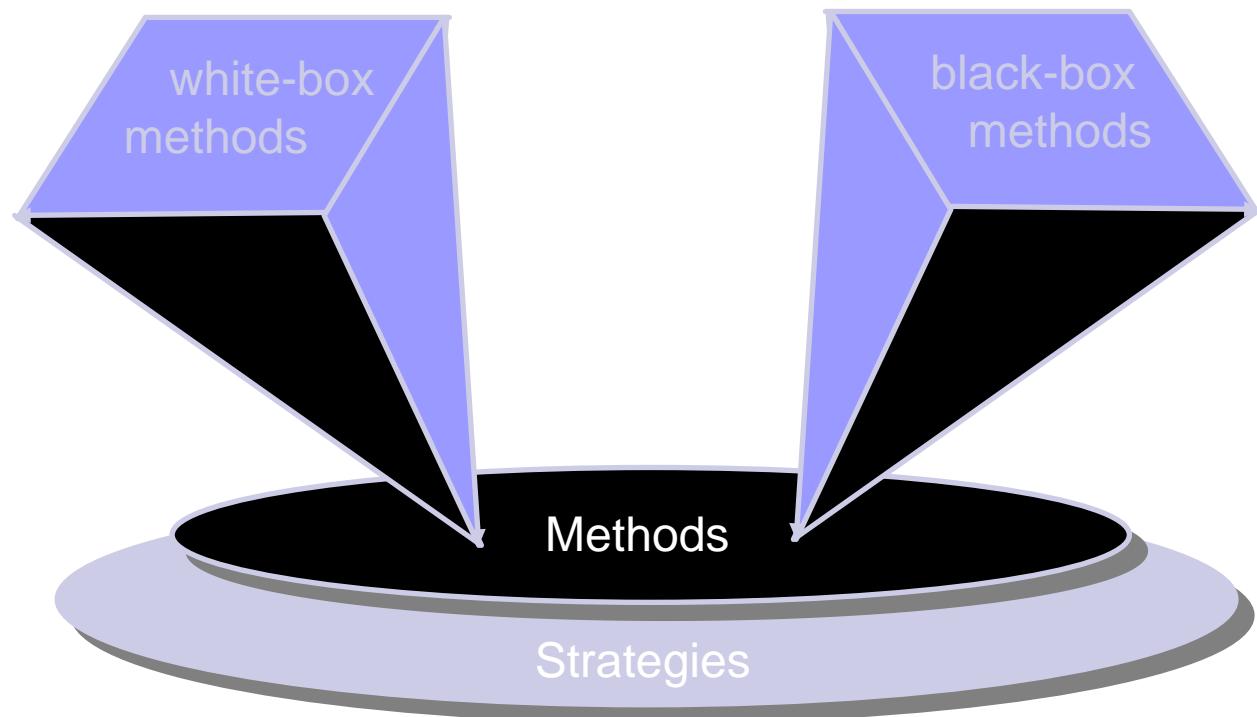
Questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

- **Is the cause of the bug reproduced in another part of the program?** (i.e. cause of bug is logical pattern)
- **What "next bug" might be introduced by the fix I'm about to make?** (i.e. cause of bug can be in logic or structure or design).
- **What could we have done to prevent this kind of bug previously?** (i.e. same kind of bug might generated early so developer can go through the steps)

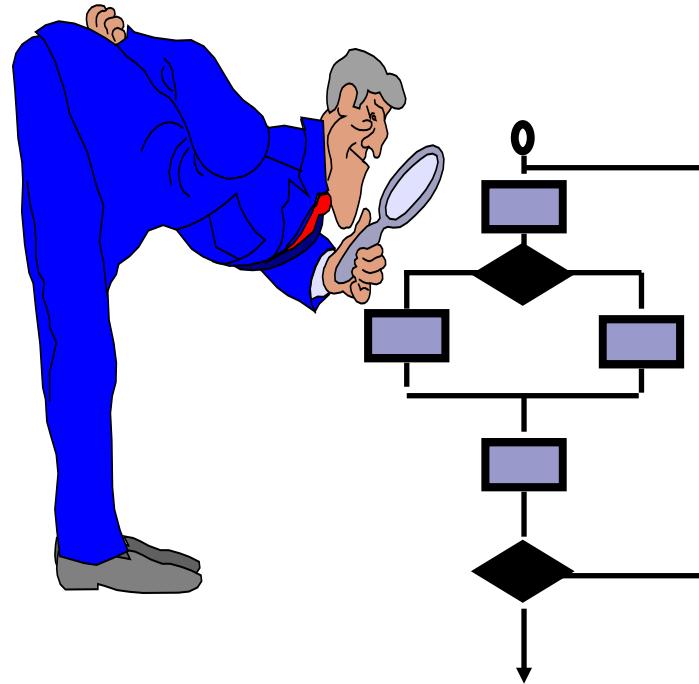


Black box & White box Testing

Software Testing



White-Box Testing



**... our goal is to ensure that all
statements and conditions have
been executed at least once ...**

Why Cover?

- logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- typographical errors are random; it's likely that untested paths will contain some

These slides are designed to accompany *Software Engineering: A Practitioner's Approach, 7/e* (McGraw-Hill 2009). Slides copyright 2009 by Roger Pressman.

White Box Testing

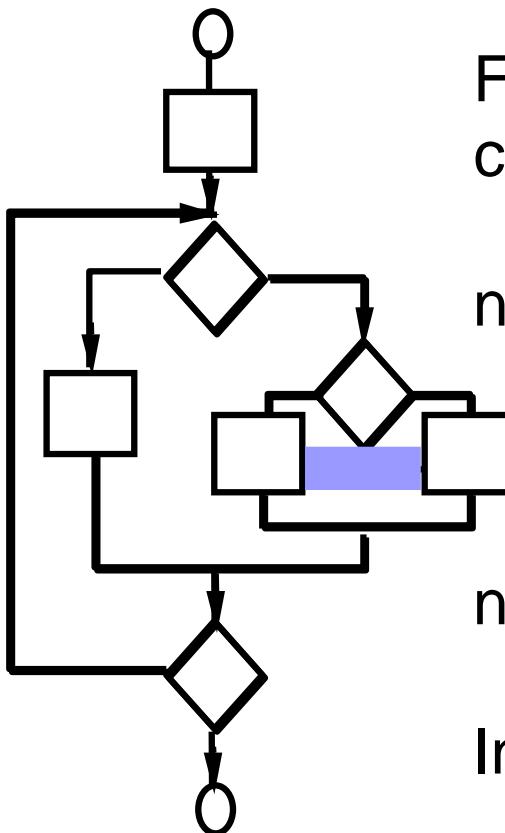
- It is software testing technique in which internal structure, design and coding of software are tested to verify flow of input-output and to improve design, usability and security.
- In white box testing, code is visible to testers so it is also called **Clear box testing, Open box testing, Transparent box testing, Code-based testing and Glass box testing**.
- It is one of two parts of the Box Testing approach to software testing.
- Its counterpart, Blackbox testing, involves testing from an external or end-user type perspective.
- On the other hand, Whitebox testing is based on the inner workings of an application and revolves around internal testing.

- The term "WhiteBox" was used because of the see-through box concept.
- The clear box or WhiteBox name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings.
- Likewise, the "black box" in "Black Box Testing" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested.

White Box Testing Techniques

- **Statement Coverage** - This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage** - This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage** - This technique corresponds to testing all possible paths which means that each statement and branch is covered

Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

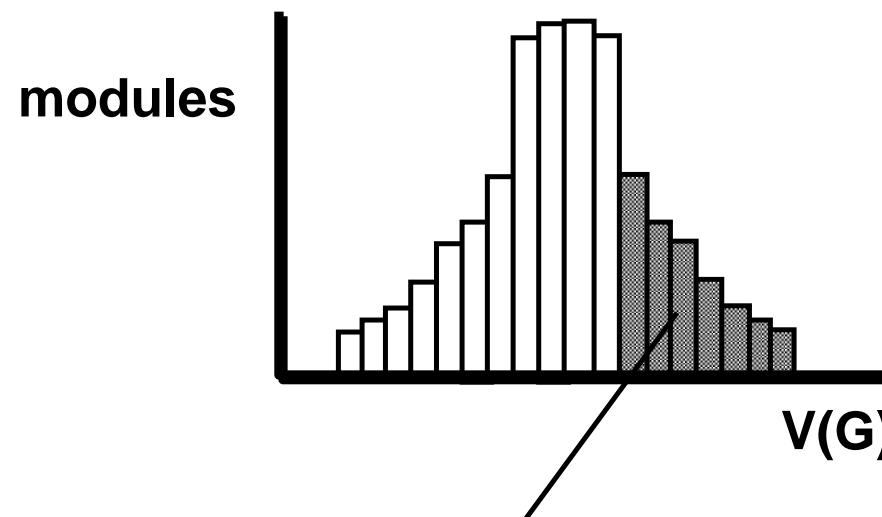
or

number of enclosed areas + 1

In this case, $V(G) = 4$

Cyclomatic Complexity

A number of industry studies have indicated that the higher $V(G)$, the higher the probability of errors.



modules in this range are
more error prone

Cyclomatic Complexity

- It is a software metric that measures the logical complexity of the program code.
- It counts the number of decisions in the given program code.
- It measures the number of linearly independent paths through the program code.
- Cyclomatic complexity indicates several information about the program code-

Cyclomatic Complexity	Meaning
1 – 10	<ul style="list-style-type: none">• Structured and Well Written Code• High Testability• Less Cost and Effort
10 – 20	<ul style="list-style-type: none">• Complex Code• Medium Testability• Medium Cost and Effort
20 – 40	<ul style="list-style-type: none">• Very Complex Code• Low Testability• High Cost and Effort
> 40	<ul style="list-style-type: none">• Highly Complex Code• Not at all Testable• Very High Cost and Effort

Importance of Cyclomatic Complexity

- It helps in determining the software quality.
- It is an important indicator of program code's readability, maintainability and portability.
- It helps the developers and testers to determine independent path executions.
- It helps to focus more on the uncovered paths.
- It evaluates the risk associated with the application or program.
- It provides assurance to the developers that all the paths have been tested at least once.

Properties of Cyclomatic Complexity

- It is the maximum number of independent paths through the program code.
- It depends only on the number of decisions in the program code.
- Insertion or deletion of functional statements from the code does not affect its cyclomatic complexity.
- It is always greater than or equal to 1.

Calculating Cyclomatic Complexity

- Cyclomatic complexity is calculated using the control flow representation of the program code
- In control flow representation of the program code,
 - Nodes represent parts of the code having no branches.
 - Edges represent possible control flow transfers during program execution
- There are 3 commonly used methods for calculating the cyclomatic complexity-

Calculating Cyclomatic Complexity

Method-01:

Cyclomatic Complexity = Total number of closed regions in the control flow graph + 1

Method-02:

Cyclomatic Complexity = $E - N + 2$

Here-

- E = Total number of edges in the control flow graph
- N = Total number of nodes in the control flow graph

Calculating Cyclomatic Complexity

Method-03:

$$\text{Cyclomatic Complexity} = P + 1$$

Here,

P = Total number of predicate nodes contained in the control flow graph

Note-

- Predicate nodes are the conditional nodes.
- They give rise to two branches in the control flow graph.

PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Problem-01

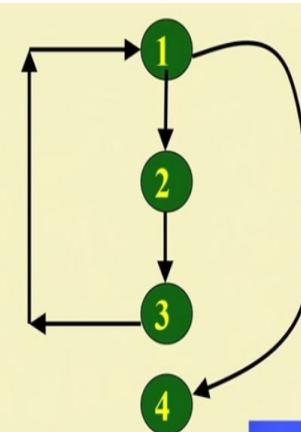
Calculate cyclomatic complexity
for the given code-

Method-01:

Cyclomatic Complexity
= Total number of closed
regions in the control flow
graph + 1
= 1+ 1
= 2

- Iteration:

```
1 while(a>b){  
2   b=b*a;  
3   b=b-1;}  
4 c=b+d;
```



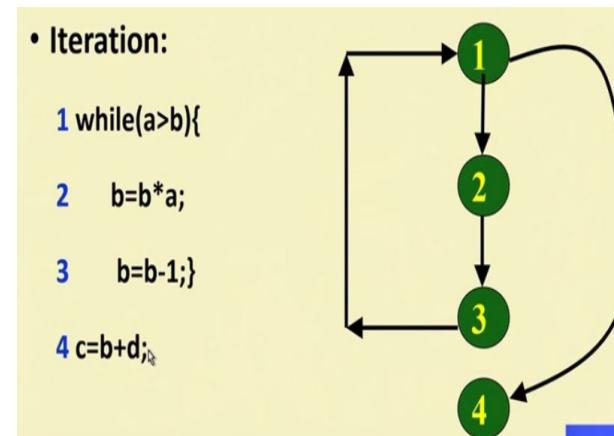
PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Method-02:

Cyclomatic Complexity
 $= 4 - 4 + 2$
 $= 4 - 4 + 2$
 $= 2$

- Iteration:

```
1 while(a>b){  
2     b=b*a;  
3     b=b-1;}  
4 c=b+d;
```

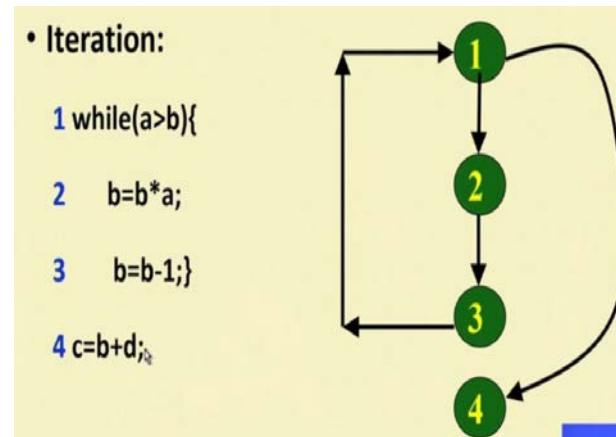


PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Method-03:

Cyclomatic Complexity
 $= P + 1$
 $= 1 + 1$
 $= 2$

- Iteration:
1 while($a > b$) {
2 $b = b * a$;
3 $b = b - 1$;
4 $c = b + d$;

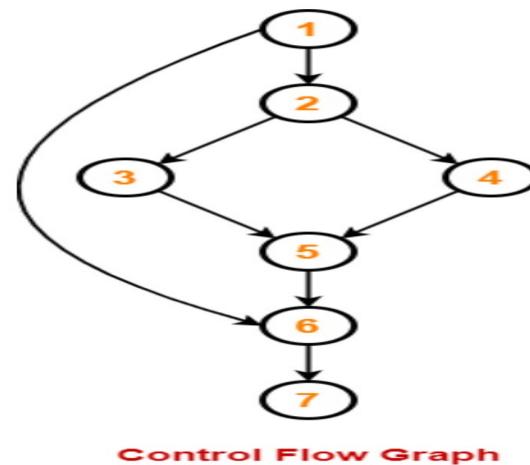


PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Problem-02

Calculate cyclomatic complexity
for the given code-

1. IF A = 354
2. THEN IF B > C
3. THEN A = B
4. ELSE A = C
5. END IF
6. END IF
7. PRINT A

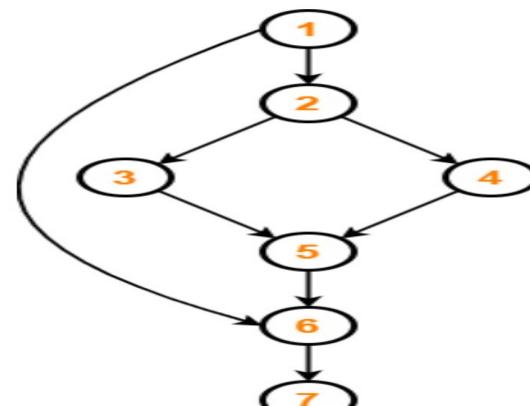


PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

Method-01:

Cyclomatic Complexity

$$\begin{aligned} &= \text{Total number of closed regions in the} \\ &\text{control flow graph} + 1 \\ &= 2 + 1 \\ &= 3 \end{aligned}$$



Control Flow Graph

PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

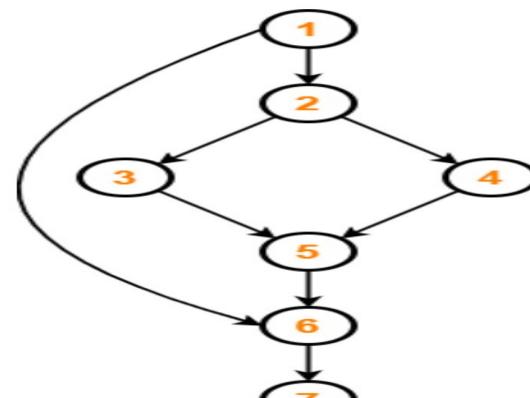
Method-02:

Cyclomatic Complexity

$$= E - N + 2$$

$$= 8 - 7 + 2$$

$$= 3$$



Control Flow Graph

PRACTICE PROBLEMS BASED ON CYCLOMATIC COMPLEXITY

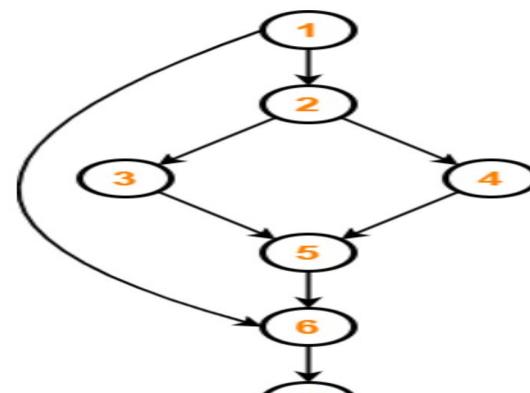
Method-03:

Cyclomatic Complexity

$$= P + 1$$

$$= 2 + 1$$

$$= 3$$



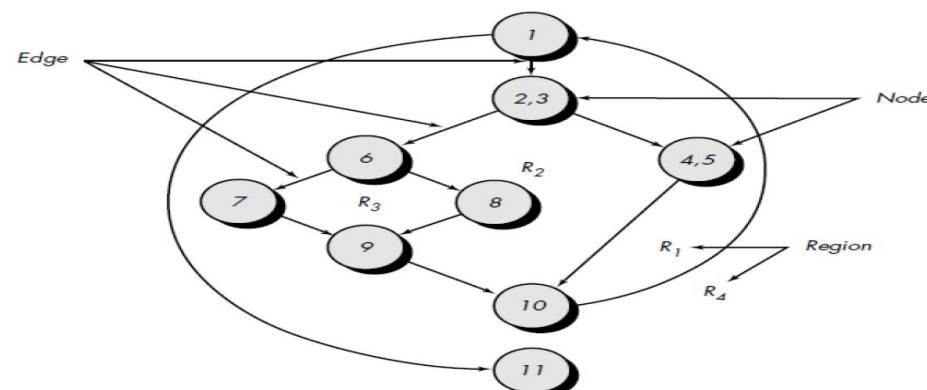
Control Flow Graph

Problem 4

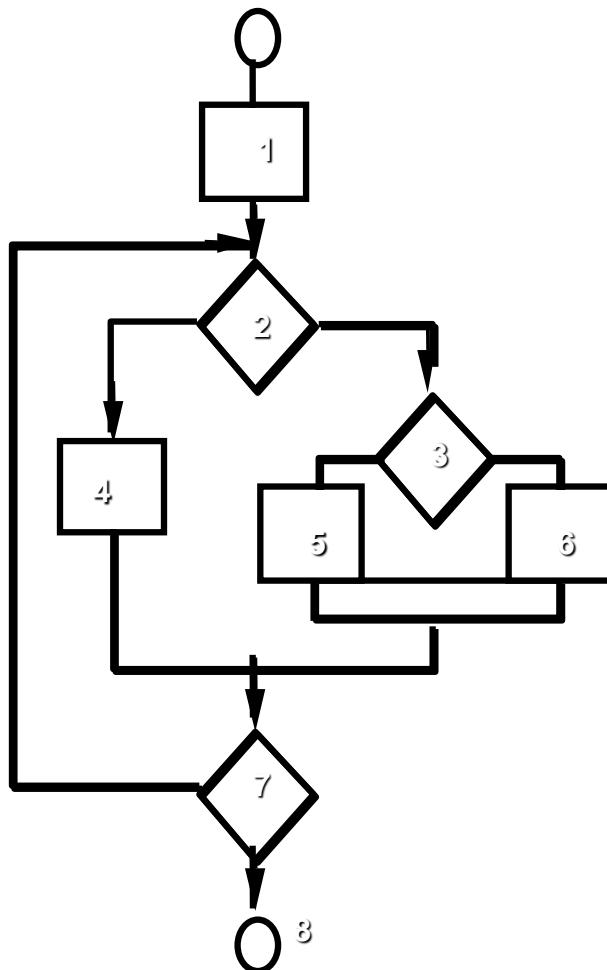
**Find the Cyclomatic Complexity
for the Control flow graph**

Paths

- path 1: 1-11
- path 2: 1-2-3-4-5-10-1-11
- path 3: 1-2-3-6-8-9-10-1-11
- path 4: 1-2-3-6-7-9-10-1-11



Basis Path Testing



Next, we derive the independent paths:

Since $V(G) = 4$,
there are four paths

Path 1: 1,2,3,6,7,8

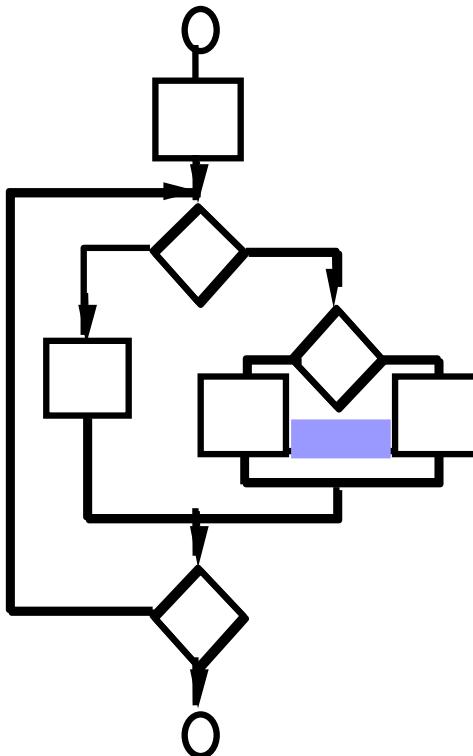
Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

Basis Path Testing Notes



- you don't need a flow chart,
but the picture will help when
you trace program paths
- count each simple logical test,
compound tests count as 2 or
more
- basis path testing should be
applied to critical modules

Deriving Test Cases

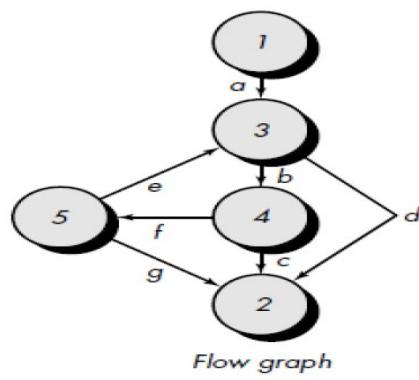
■ *Summarizing:*

- Using the design or code as a foundation, draw a corresponding flow graph.
- Determine the cyclomatic complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.

Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

Graph Matrix

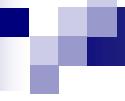


Node	Connected to node				
	1	2	3	4	5
1			<i>a</i>		
2					
3		<i>d</i>		<i>b</i>	
4	<i>c</i>				<i>f</i>
5	<i>g</i>	<i>e</i>			

Graph matrix

Connection Matrix

Node	Connected to node					Connections $1 - 1 = 0$
	1	2	3	4	5	
1			1			$1 - 1 = 0$
2						$2 - 1 = 1$
3		1		1		$2 - 1 = 1$
4	1				1	$2 - 1 = 1$
5		1	1			$2 - 1 = 1$
Graph matrix					$\overline{3 + 1} = 4$ ← Cyclomatic complexity	



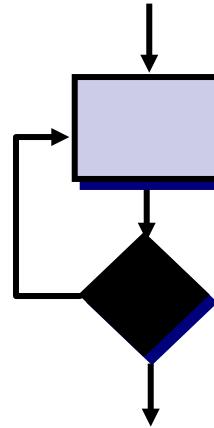
Control Structure Testing

- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

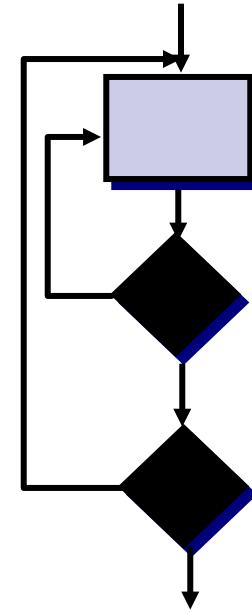
Data Flow Testing

- The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.
 - 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
 - $\text{DEF}(S) = \{X \mid \text{statement } S \text{ contains a definition of } X\}$
 - $\text{USE}(S) = \{X \mid \text{statement } S \text{ contains a use of } X\}$
 - A *definition-use (DU) chain* of variable X is of the form $[X, S, S']$, where S and S' are statement numbers, X is in $\text{DEF}(S)$ and $\text{USE}(S')$, and the definition of X in statement S is live at statement S'

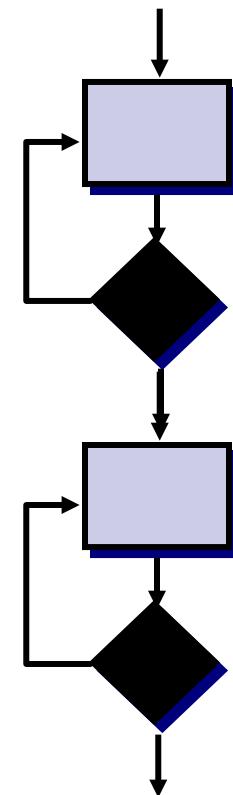
Loop Testing



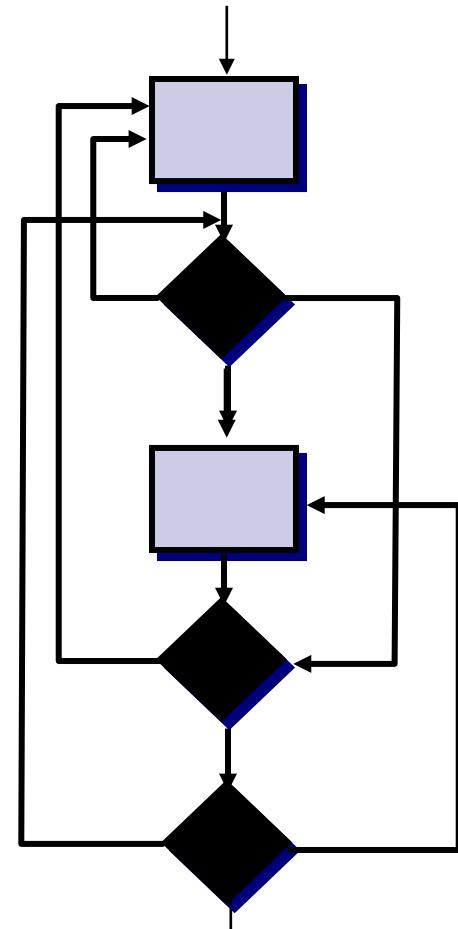
Simple
loop



Nested
Loops



Concatenated
Loops



Unstructured
Loops

Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop $m < n$
5. $(n-1)$, n, and $(n+1)$ passes through the loop

where n is the maximum number
of allowable passes

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

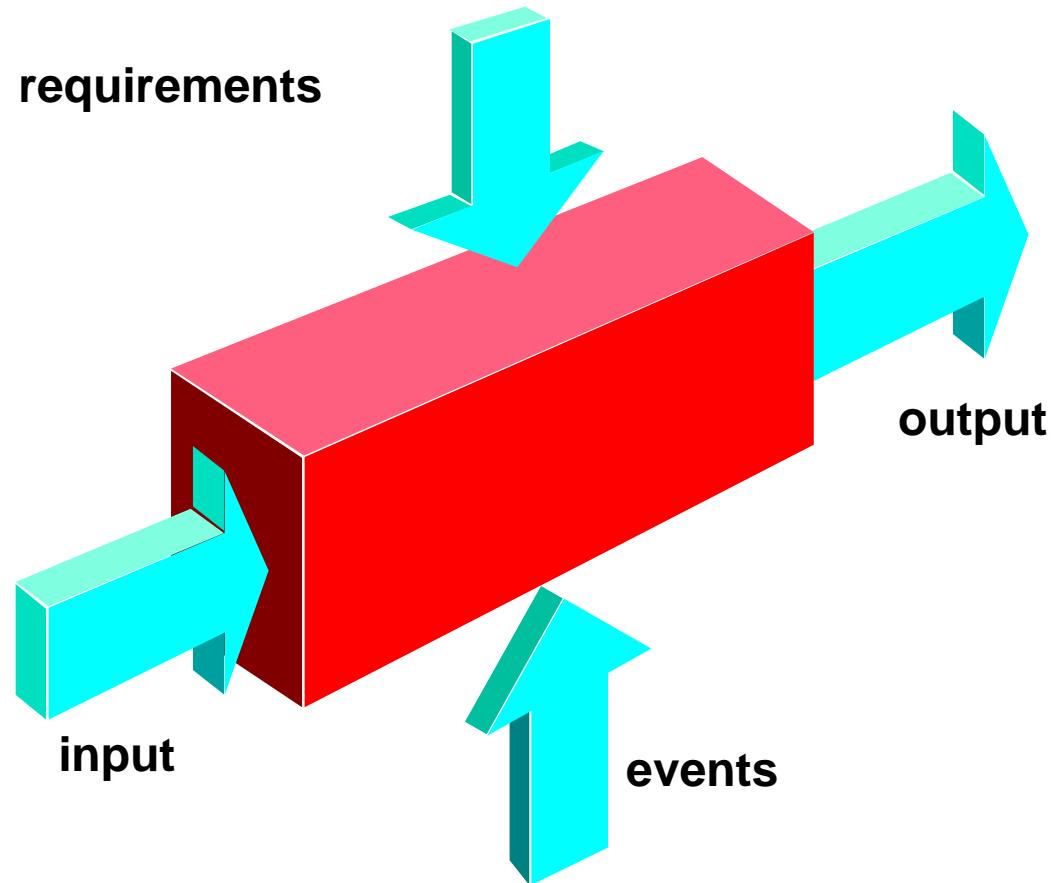
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

If the loops are independent of one another
then treat each as a simple loop
else* treat as nested loops
endif*

for example, the final loop counter value of loop 1 is used to initialize loop 2.

Black-Box Testing



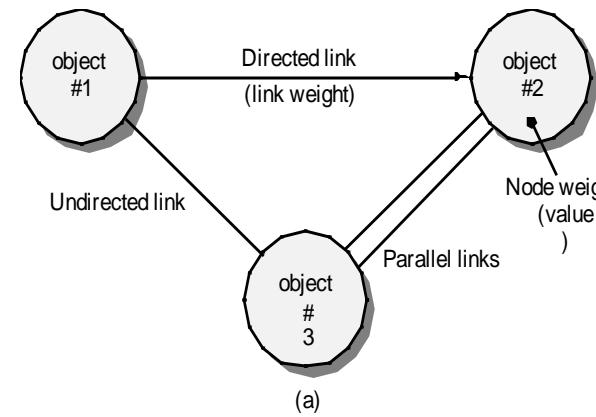
Black-Box Testing

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

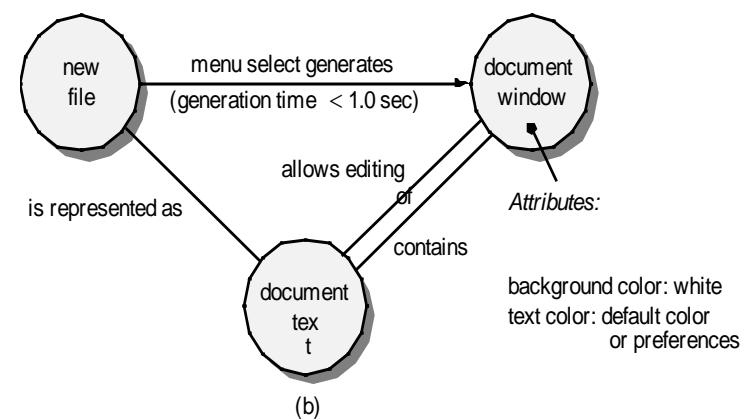
Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term “objects” in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.

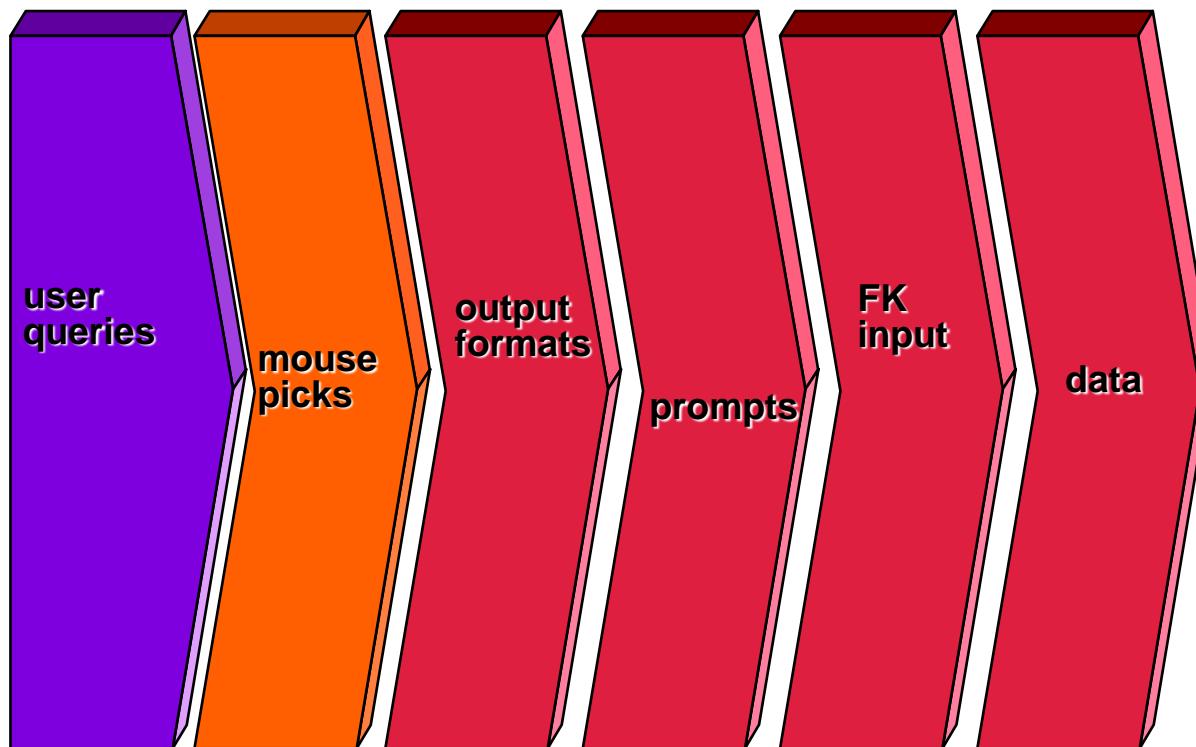


(a)



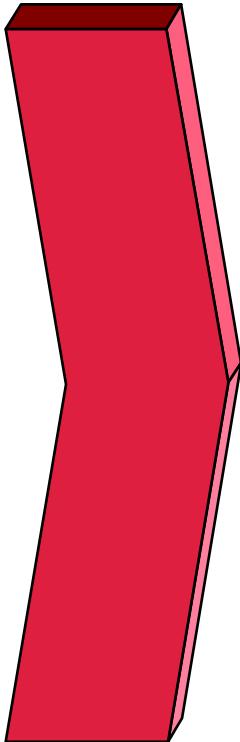
(b)

Equivalence Partitioning



Sample Equivalence Classes

Valid data

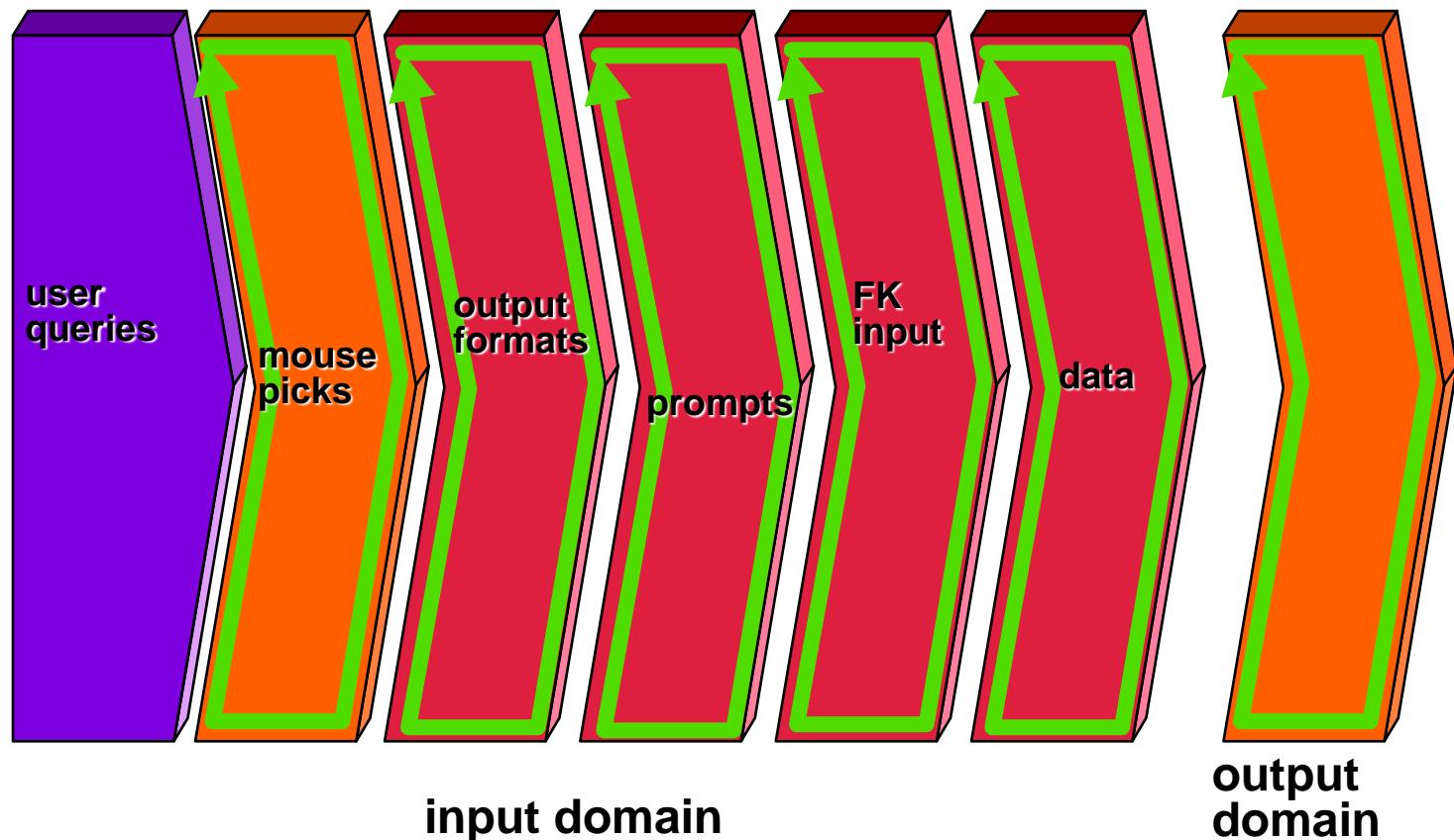


- user supplied commands
- responses to system prompts
- file names
- computational data
- physical parameters
- bounding values
- initiation values
- output data formatting
- responses to error messages
- graphical data (e.g., mouse picks)

Invalid data

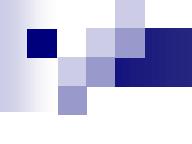
- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place

Boundary Value Analysis



Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
 - Recall that a *behavioral model* indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
 - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.



THANK YOU



21CSC303J – SEPM Unit 5 Risk Management

Risk Management

Introduction

- A risk is a potential problem – it might happen or it might not
- Conceptual definition of risk
 - Risk concerns future happenings
 - Risk involves change in mind, opinion, actions, places, etc.
 - Risk involves choice and the uncertainty that choice entails
- Two characteristics of risk
 - Uncertainty – the risk may or may not happen, that is, there are no 100% risks. (those, instead are called constraints)
 - Loss – the risk becomes a reality and unwanted consequences or losses occur

Risk Management

Risk Categorization – Approach #1

- Project risks
 - They threaten the project plan
 - If they become real, it is likely that the project schedule will slip and that costs will increase
- Technical risks
 - They threaten the quality and timeliness of the software to be produced
 - If they become real, implementation may become difficult or impossible
- Business risks
 - They threaten the viability of the software to be built
 - If they become real, they jeopardize the project or the product
- Sub-categories of Business risks
 - Market risk – building an excellent product or system that no one really wants
 - Strategic risk – building a product that no longer fits into the overall business strategy for the company
 - Sales risk – building a product that the sales force doesn't understand how to sell
 - Management risk – losing the support of senior management due to a change in focus or a change in people
 - Budget risk – losing budgetary or personnel commitment

Risk Management

Risk Categorization – Approach #2

- Known risks
 - Those risks that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date).
- Predictable risks
 - Those risks that are extrapolated from past project experience (e.g., past turnover).
- Unpredictable risks
 - Those risks that can and do occur, but are extremely difficult to identify in advance.

Risk Strategies

- Reactive risk strategies
 - "Don't worry, I'll think of something"
 - The majority of software teams and managers rely on this approach
 - Nothing is done about risks until something goes wrong
 - The team then flies into action in an attempt to correct the problem rapidly
 - (Fire fighting)
 - Crisis management is the choice of management techniques
- Proactive risk strategies
 - Steps for risk management are followed (see next slide)
 - Primary objective is to avoid risk and to have a contingency plan in place to handle unavoidable risks in a controlled and effective manner

Steps for Risk Management

- Identify possible risks; recognize what can go wrong.
- Analyze each risk to estimate the probability that it will occur and the impact (i.e., damage) that it will do if it does occur.
- Rank the risks by probability and impact
 - Impact may be negligible, marginal, critical, and catastrophic
- Develop a contingency plan to manage those risks having high probability and high impact.

Background

Risk Identification

- Risk identification is a systematic attempt to specify threats to the project plan
- By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary
- Generic risks
 - Risks that are a potential threat to every software project
- Product-specific risks
 - Risks that can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built
 - This requires examination of the project plan and the statement of scope
 - "What special characteristics of this product may threaten our project?"

Risk Item Checklist

- Used as one way to identify risks
- Focuses on known and predictable risks in specific subcategories (see next slide)
- Can be organized in several ways
 - A list of characteristics relevant to each risk subcategory
 - Questionnaire that leads to an estimate on the impact of each risk
 - A list containing a set of risk component and drivers along with their probability of occurrence

Known & Predictable Risk Categories

- **Product size** – risks associated with overall size of the software to be built
- **Business impact** – risks associated with constraints imposed by management or the marketplace
- **Customer characteristics** – risks associated with sophistication of the customer and the developer's ability to communicate with the customer in a timely manner
- **Process definition** – risks associated with the degree to which the software process has been defined and is followed
- **Development environment** – risks associated with availability and quality of the tools to be used to build the project
- **Technology to be built** – risks associated with complexity of the system to be built and the "newness" of the technology in the system
- **Staff size and experience** – risks associated with overall technical and project experience of the software engineers who will do the work

Questionnaire on Project Risk

(Questions are ordered by their relative importance to project success)

- 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) Are requirements fully understood by the software engineering team and its customers?
- 4) Have customers been involved fully in the definition of requirements?
- 5) Do end-users have realistic expectations?
- 6) Is the project scope stable?
- 7) Does the software engineering team have the right mix of skills?
- 8) Are project requirements stable?
- 9) Does the project team have experience with the technology to be implemented?
- 10) Is the number of people on the project team adequate to do the job?
- 11) Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

Risk Components and Drivers

- The project manager identifies the risk drivers that affect the following risk components
 - **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit 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 levels
 - Negligible, marginal, critical, and catastrophic
- Risk drivers can be assessed as impossible, improbable, probable, and frequent

Risk Projection (Estimation)

- Risk projection (or estimation) attempts to rate each risk in two ways
 - The probability that the risk is real
 - The consequence of the problems associated with the risk, should it occur
- The project planner, managers, and technical staff perform four risk projection steps
- The intent of these steps is to consider risks in a manner that leads to prioritization
- By prioritizing risks, the software team can allocate limited resources where they will have the most impact
- Risk Projection / Estimation Steps
 1. Establish a scale that reflects the perceived likelihood of a risk (e.g., 1-low, 10-high)
 2. Delineate the consequences of the risk
 3. Estimate the impact of the risk on the project and product
 4. Note the overall accuracy of the risk projection so that there will be no misunderstandings

Risk Table

- A risk table provides a project manager with a simple technique for risk projection.
- It consists of five columns
 - Risk Summary – short description of the risk
 - Risk Category – one of seven risk categories
 - Probability – estimation of risk occurrence based on group input
 - Impact – (1) catastrophic (2) critical (3) marginal (4) negligible
 - RMMM – Pointer to a paragraph in the Risk Mitigation, Monitoring, and Management Plan

Risk Summary	Risk Category	Probability	Impact (1-4)	RMMM

Developing a Risk Table

- List all risks in the first column (by way of the help of the risk item checklists)
- Mark the category of each risk
- Estimate the probability of each risk occurring
- Assess the impact of each risk based on an averaging of the four risk components to determine an overall impact value
- Sort the rows by probability and impact in descending order
- Draw a horizontal cutoff line in the table that indicates the risks that will be given further attention

Assessing Risk Impact

- Three factors affect the consequences that are likely if a risk does occur
 - Its nature – This indicates the problems that are likely if the risk occurs
 - Its scope – This combines the severity of the risk (how serious was it) with its overall distribution (how much was affected)
 - Its timing – This considers when and for how long the impact will be felt
- The overall risk exposure formula is $RE = P \times C$
 - P = the probability of occurrence for a risk
 - C = the cost to the project should the risk actually occur
- Example
 - P = 80% probability that 18 of 60 software components will have to be developed
 - C = Total cost of developing 18 components is \$25,000
 - $RE = .80 \times \$25,000 = \$20,000$

Risk Mitigation, Monitoring and Management

- An effective strategy for dealing with risk must consider three issues
 - (Note: these are not mutually exclusive)
 - Risk mitigation (i.e., avoidance)
 - Risk monitoring
 - Risk management and contingency planning
- Risk mitigation (avoidance) is the primary strategy and is achieved through a plan.
 - Example: Risk of high staff turnover

Risk Mitigation, Monitoring and Management

- Strategy for Reducing Staff Turnover
 - Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market)
 - Mitigate those causes that are under our control before the project starts
 - Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave
 - Organize project teams so that information about each development activity is widely dispersed
 - Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner
 - Conduct peer reviews of all work (so that more than one person is "up to speed")
 - Assign a backup staff member for every critical technologist

Risk Mitigation, Monitoring and Management

- During risk monitoring, the project manager monitors factors that may provide an indication of whether a risk is becoming more or less likely.
- Risk management and contingency planning assume that mitigation efforts have failed and that the risk has become a reality.
- RMMM steps incur additional project cost
 - Large projects may have identified 30 – 40 risks
- Risk is not limited to the software project itself
 - Risks can occur after the software has been delivered to the user
- Software safety and hazard analysis
 - These are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.
 - If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

RMMM Plan

- The RMMM plan may be a part of the software development plan or may be a separate document.
- Once RMMM has been documented and the project has begun, the risk mitigation, and monitoring steps begin
 - Risk mitigation is a problem avoidance activity
 - Risk monitoring is a project tracking activity
- Risk monitoring has three objectives
 - To assess whether predicted risks do, in fact, occur
 - To ensure that risk aversion steps defined for the risk are being properly applied
 - To collect information that can be used for future risk analysis
- The findings from risk monitoring may allow the project manager to ascertain what risks caused which problems throughout the project.

Seven Principles of Risk Management

- Maintain a global perspective
 - View software risks within the context of a system and the business problem that is intended to solve
- Take a forward-looking view
 - Think about risks that may arise in the future; establish contingency plans
- Encourage open communication
 - Encourage all stakeholders and users to point out risks at any time
- Integrate risk management
 - Integrate the consideration of risk into the software process
- Emphasize a continuous process of risk management
 - Modify identified risks as more becomes known and add new risks as better insight is achieved
- Develop a shared product vision
 - A shared vision by all stakeholders facilitates better risk identification and assessment.
- Encourage teamwork when managing risk
 - Pool the skills and experience of all stakeholders when conducting risk management activities



Thank You