Unit-1

Chapter-1:

1) Explain about software engineering role & characteristics of a software.

Ans: Software is defined as a collection of programs, documentation and operating procedures. The **Institute of Electrical and Electronic Engineers (IEEE)** defines software as a 'collection of computer programs, procedures, rules and associated documentation and data.'

Software controls, integrates, and manages the hardware components of a computer system. It also instructs the computer what needs to be done to perform a specific task and how it is to be done. For example, software instructs the hardware how to print a document, take input from the user, and display the output.

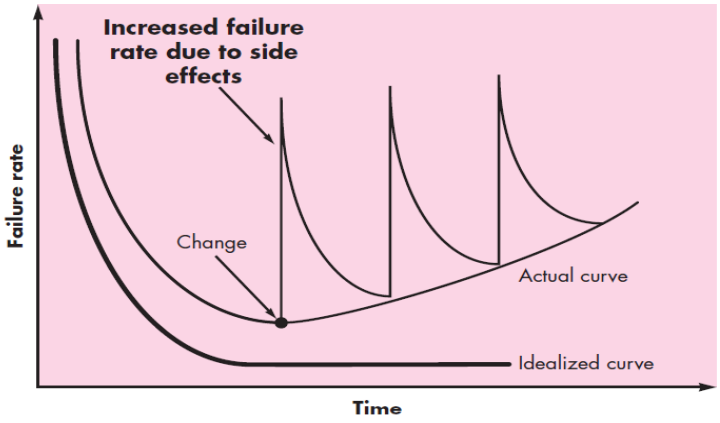
Typical formal **definitions** of **software engineering** are: "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of **software**". "an **engineering** discipline that is concerned with all aspects of **software** production".

Software engineering role:

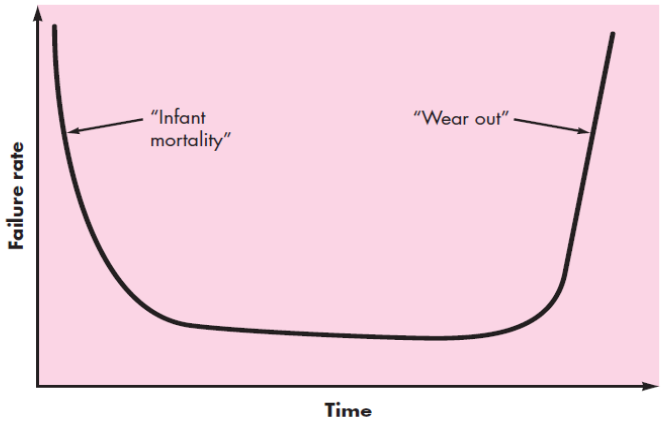
- Today, software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product. As a product, it delivers the computing potential embodied by computer hardware . It resides within a mobile phone or operates inside a mainframe computer, software is an information transformer—producing, managing, acquiring, modifying, displaying, or transmitting information.
  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).
- Software delivers the most important product of our time—information.
- It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context;
- It manages business information to enhance competitiveness;
- It provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms.

characteristics of a software:

- Software is developed or engineered; it is not manufactured in the classical sense.



- Software doesn't "wear out (exhaust)."



- Although the industry is moving toward component-based construction, most software continues to be custom built (made to a particular customer's order).

• **Functionality:** Refers to the degree of performance of the software against its intended purpose.

• **Reliability:** Refers to the ability of the software to provide desired functionality under the given conditions.

• **Usability:** Refers to the extent to which the software can be used with ease.

• **Efficiency:** Refers to the ability of the software to use system resources in the most effective and efficient manner.

• **Maintainability:** Refers to the ease with which the modifications can be made in a software system to extend its functionality, improve its performance, or correct errors.

• **Portability:** Refers to the ease with which software developers can transfer software from one platform to another, without (or with minimum) changes. In simple terms, it refers to the ability of

software to function properly on different hardware and software platforms without making any changes in it.

2) Explain in detail about changing nature of software.

Ans: Today, seven broad categories of computer software present continuing challenges for software engineers:

- System software
- Application software
- Engineering and scientific software
- Embedded software
- Product-line software
- Web-applications
- Artificial intelligence software

1) System software is a type of computer **program** that is designed to run a computer's hardware and application **programs.** (e.g., compilers, editors, and file management utilities).

2) Application software is a program or group of programs designed for end users. Stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making. (e.g.. Microsoft Office, Excel, Google Chrome, Mozilla Firefox and Skype**)**

3) Engineering/scientific software—has been characterized by "number crunching" algorithms. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing.

4) Embedded software—resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems).

5) Product-line software—designed to provide a specific capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer markets (e.g., word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications).

6) Web applications—called "Web-Apps" this network-centric software category spans a wide array of applications. In their simplest form, Web-Apps can be little more than a set of linked hypertext files that present information using text and limited graphics.

7) Artificial intelligence software—makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within

this area include robotics, expert systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing.

3) Define Legacy software and explain in detail about quality of legacy software evaluation.

Ans: Legacy software systems were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve.

- Many legacy systems remain supportive to core business functions and are absolutely to business.
- Unfortunately, there is sometimes one additional characteristic that is present in legacy software—poor quality.
- Legacy systems sometimes have inextensible designs, convoluted code, poor or nonexistent documentation, test cases and results that were never archived, a poorly managed change history—the list can be quite long.
- If the legacy software meets the needs of its users and runs reliably, it isn't broken and does not need to be fixed. However, as time passes, legacy systems often evolve for one or more of the following reasons:
  - The software must be adapted to meet the needs of new computing environments or technology.
  - The software must be enhanced to implement new business requirements.
  - The software must be extended to make it interoperable with other more modern systems or databases.
  - The software must be re-architected to make it viable within a network environment.

  When these modes of evolution occur, a legacy system must be reengineered so that it remains viable into the future. The goal of modern software engineering is to "devise methodologies that are founded on the notion of evolution"; that is, the notion that software systems continually change, new software systems are built from the old ones, and . . . all must interoperate and cooperate with each other".
  Regardless of its application domain, size, or complexity, computer software will evolve Over time.

Change (software maintenance) drives this process and occurs when error are corrected, when the software is adapted to a new environment, when the customer requests new features or functions, and when the application is reengineered to provide benefit in a modern context.

A Unified Theory for SW Evolution [Lehman 1997]

1. **Continuing change –** software must be continually adapted or it will become less and less satisfactory
2. **Increasing complexity –** as software is changed it becomes increasingly complex unless work is done to mitigate the complexity
3. **Relationship to organization –** the software exists within a framework of people, management, rules and goals which create a system of checks and balances which shape software evolution
4. **Invariant work rate –** over the lifetime of a system the amount of work performed on it is essentially the same as external factors beyond anyone's control drive the evolution
5. **Conservation of familiarity –** developers and users of the software must maintain mastery of its content in order to use and evolve it; excessive growth reduces mastery and acts as a brake
6. **Continuing growth –** seemingly similar to the first law, this observation states that additional growth is also driven by the resource constraints that restricted the original scope of the system
7. **Declining quality –** the quality of the software will decline unless steps are taken to keep it in accord with operational changes
8. **Feedback system –** the evolution in functionality and complexity of software is governed by a multi-loop, multi-level, multi-party feedback system

4) Explain in detail about software myths.

Ans: Software myths—erroneous beliefs about software and the process that is used to build it—can be traced to the earliest days of computing. Myths have a number of attributes that make them insidious (proceeding in a gradual, subtle way, but with very harmful effects).

Today, most knowledgeable software engineering professionals recognize myths for what they are—misleading attitudes that have caused serious problems for managers and practitioners alike. However, old attitudes and habits are difficult to modify, and remnants of software myths remain.

Managers, who own software development responsibility, are often under strain and pressure to maintain a software budget, time constraints, improved quality, and many other considerations.

Table Management Myths

The members of an organization can acquire all-the information, they require from a manual, which contains standards, procedures, and principles;	Standards are often incomplete, inadaptable, and outdated. Developers are often unaware of all the established standards. Developers rarely follow all the known standards because not all the standards tend to decrease the delivery time of software while maintaining its quality.
If the project is behind schedule, increasing the number of programmers can reduce the time gap.	Adding more manpower to the project, which is already behind schedule, further delays the project. New workers take longer to learn about the project as compared to those already working on the project.
If the project is outsourced to a third party, the management can relax and let the other firm develop software for them.	Outsourcing software to a third party does not help the organization, which is incompetent in managing and controlling the software project internally. The organization invariably suffers when it out sources the software project.

Customer myths: A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department or an outside company that has requested software under contract. In many cases, the customer believes myths about software because software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and, ultimately, dissatisfaction with the developer.

Table User Myths

Brief requirement stated in the initial process is enough to start development; detailed requirements can be added at the later stages.	Starting development with incomplete and ambiguous requirements often lead to software failure. Instead, a complete and formal description of requirements is essential before starting development. Adding requirements at a later stage often requires repeating the entire development process.

Software is flexible; hence software requirement changes can be added during any phase of the development process.	Incorporating change requests earlier in the development process costs lesser than those that occurs at later stages. This is because incorporating changes later may require redesigning and extra resources.

In the early days of software development, programming was viewed as an art, but now software development has gradually become an engineering discipline. However, developers still believe in some myths-. Some of the common developer myths are listed in Table.

Table Developer Myths

Software development is considered complete when the code is delivered.	50% to 70% of all the efforts are expended after the software is delivered to the user.
The success of a software project depends on the quality of the product produced.	The quality of programs is not the only factor that makes the project successful instead the documentation and software configuration also play a crucial role.
Software engineering requires unnecessary documentation, which slows down the project.	Software engineering is about creating quality at every level of the software project. Proper documentation enhances quality which results in reducing the amount of rework.
The only product that is delivered after the completion of a project is the working program(s).	The deliverables of a successful project includes not only the working program but also the documentation to guide the users for using the software.

2nd chapter

1) Explain in detail about Software Engineering layered technology

Ans: Software development is totally a layered technology. That means, to develop software one will have to go from one layer to another. The layers are related and each layer demands the fulfillment of the previous layer. Figure below is the upward flowchart of the layers of software development.

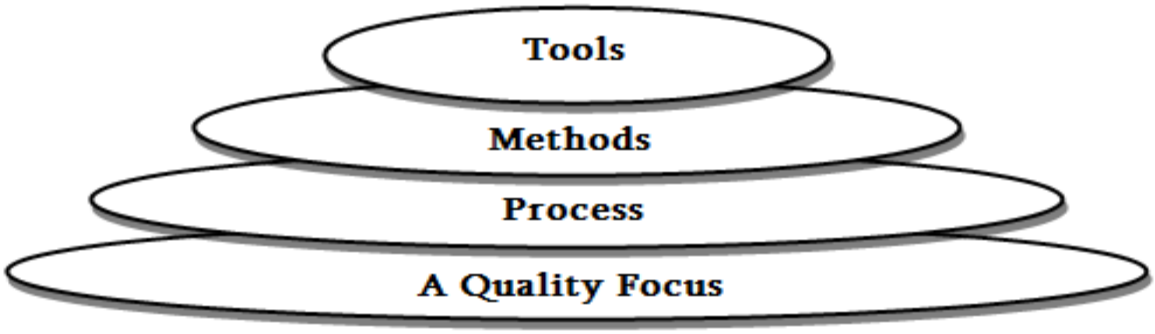


Figure: Flowchart of the Layers of Software Development

01. A Quality Focus: Software engineering must rest on an organizational commitment to quality. Total quality management and similar philosophies foster a continuous process improvement culture, and this culture ultimately leads to the development of increasingly more mature approaches to software engineering. The bedrock that supports software engineering is a quality focus.

02. Process: The foundation for software engineering is the process layer. Process defines a framework for a set of Key Process Areas (KPAs) that must be established for effective delivery of software engineering technology. This establishes the context in which technical methods are applied, work products such as models, documents, data, reports, forms, etc. are produced, milestones are established, quality is ensured, and change is properly managed.

03. Methods: Software engineering *methods* provide the technical how-to's for building software. Methods will include requirements analysis, design, program construction, testing, and support. This relies on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

04. Tools: Software engineering *tools* provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

2) Define Process Framework. Discuss in detail about 5 Generic process framework activities.

Ans:

Software Process

Process defines a framework for a set of Key Process Areas (KPAs) that must be established for effective delivery of Software engineering technology. This establishes the context in which technical methods are applied, work products such as models, documents, data, reports, forms, etc. are produced, milestones are established, quality is ensured, and change is properly managed.

Software Process Framework:

A process framework establishes the foundation for a complete **software process** by identifying a small **number of framework activities** that are applicable to all software projects, regardless of size or complexity. It also includes a set of **Umbrella activities** that are applicable across the entire software process. Some most applicable framework activities are described below.

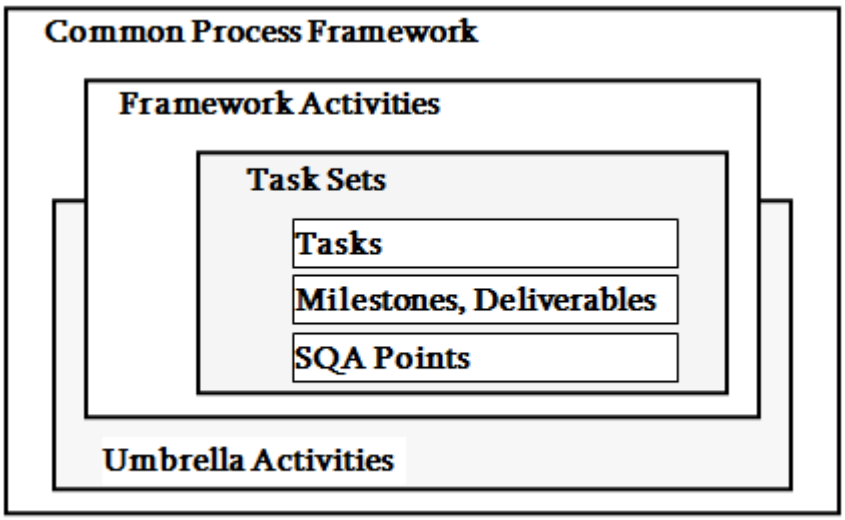


Figure: Chart of Process Framework

Communication:

This activity involves heavy communication with customers and other stakeholders in order to gather requirements and other related activities

Planning:

Here a plan to be followed will be created which will describe the technical tasks to be conducted, risks, required resources, work schedule etc.

Modeling:

A model will be created to better understand the requirements and design to achieve these requirements.

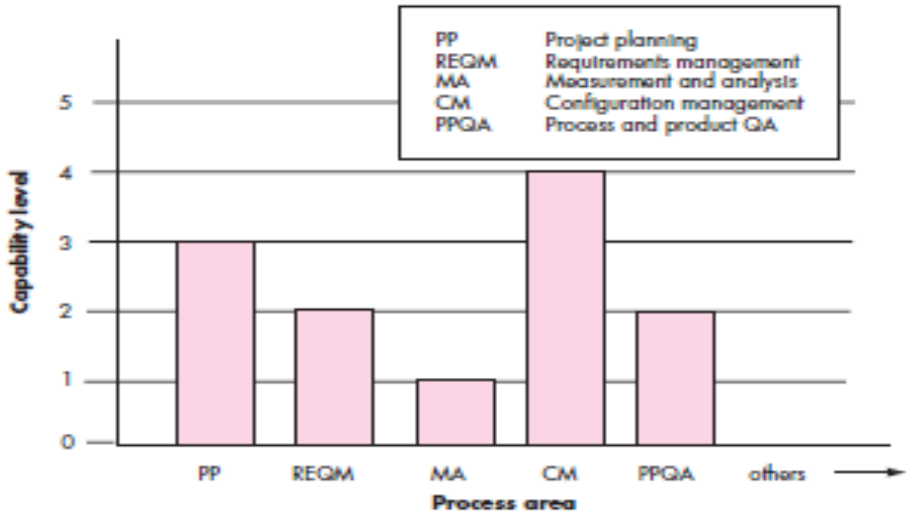
Construction:
Here the code will be generated and tested.

Deployment:
Here, a complete or partially complete version of the software is represented to the customers to evaluate and they give feedbacks based on the evaluation.

These above described five activities can be used in any kind of software development. The details of the software development process may become a little different, but the framework activities remain the same.

3) Explain about CMMI model.

Ans: **CMMI (The Capability Maturity Model Integration):**
The Software Engineering Institute (SEI) has developed a comprehensive model predicated on a set of software engineering capabilities that should be present as organizations reach different levels of process maturity. To determine an organization's current state of process maturity the CMMI uses two ways, (i) as a continuous model and (ii) as a staged model. In the case of a continuous model each process area is rated according to the following capability levels.



Level 0 : Incomplete:
The process are is either not performed or does not achieve all goals and objectives defined by the CMMI for level 1 capability.

Level 1: Performed:
All the specific goals has been achieved and work tasks required to produce defined products has been conducted.

Level 2: Managed:
All level 1 criteria have been achieved. All work associated with the process area conforms to an organizationally defined policy, all people are doing their jobs, stakeholders are actively involved in the process area, all work tasks and work products are monitored, controlled and reviewed.

Level 3: Defined:
All level 2 criteria have been achieved. In addition, the process is "tailored from the organization's set of standard processes according to the organization's tailoring guidelines and contributes work products, measures and other process improvement information to the organizational process assets."

Level 4: Quantitatively Managed:
All level 3 criteria have been achieved. In addition, the process area is controlled and improved using measurement and quantitative assessment. "Quantitative objectives for quality and process performance are established and used as a criteria in managing the process."

Level 5: Optimized:
All capability level 4 criteria have been achieved. In addition, the process area is adapted and optimized using quantitative means to meet changing customer needs and to continually improve the efficacy of the process area under consideration.

The staged CMMI model defines the same process areas, goals and practices as the continuous model. The primary difference is that the staged model defines five maturity levels rather than five capability levels.

Process areas require to achieve maturity levels has been showed in the table below :

Level	Focus	Process Areas
Optimizing	*Continuous process improvement*	Organizational innovation and deployment Causal analysis and resolution
Quantitatively managed	*Quantitative management*	Organizational process performance Quantitative project management
Defined	*Process standardization*	Requirements development Technical solution Product integration Verification Validation Organizational process focus Organizational process definition Organizational training Integrated project management Integrated supplier management Risk management Decision analysis and resolution Organizational environment for integration Integrated teaming
Managed	*Basic project management*	Requirements management Project planning Project monitoring and control Supplier agreement management Measurement and analysis Process and product quality assurance Configuration management
Performed		

4) Define Process Pattern & Explain about various forms of Process patterns.

Ans: The software process can be defined as a collection of patterns that define a set of activities, actions, work tasks, work products and/or related behaviors required to develop computer software.

Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a complete process (e.g., prototyping). In other situations, patterns can be used to describe an important framework activity (e.g., planning) or a task within framework activity (e.g., project-estimating). Ambler has proposed the following template for describing a process pattern. **Pattern Name.** The pattern is given a meaningful name that describes its functions within the software-process-(e.g.,customer-communication).

Intent. The objective of the pattern is described briefly. For example, the intent of customer-communication is "to establish a collaborative relationship with the customer in an effort to define project-scope,business-requirements,and-other_projectsconstraints".

Type.: The_pattern-type-is-specified.Ambler_suggests-three-types:

• Task pattern define a software engineering action or work task that is part of the process and relevant to successful software engineering practice (e.g., requirements gathering is a task pattern).

• Stage patterns represent a framework activity for the process. Since a framework activity encompasses multiple work tasks, a stage pattern incorporates multiple task patterns that are relevant to the stage. An example of a stage pattern might be communication. This pattern would incorporate-the-task_pattern_requirements_gathering_and_others.

• Phase patterns define the sequence of framework activities is iterative in nature. An example of aphase-pattern-might_be_a_spiral_model(or)prototyping.

Initial context. The conditions under which the pattern applies are described. Prior to the initiation of the pattern, we ask (1) what organizational team created activities have already occurred (2) what is the entry state for the process? And (3) what software engineering information or project information_already_exists?

For example, the planning pattern (a stage pattern) requires that (1) customers and software engineers have established a collaborative communication: (2) successful completion of a number of task patterns (specified) for the customer-communication pattern has occurred: and (3) project scope, basic business requirements, and project constraints are known.

An_Example-Process_Pattern:

The following abbreviated process pattern describes an approach that may be applicable when stakeholders have a general idea of what must be done, but are unsure of specific software requirements.

Pattern-name.Prototyping.

Intent: The objective of the pattern is to build a model (a prototype) that can be assessed iteratively by stakeholders in an effort to identify or solidity software requirements.

Type: Phase-pattern.

Initial context: The following conditions must be met prior to the initiation of this pattern:(1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding problem to be solved has been identified by stakeholders: (4) an initial understanding of project scope, basic business requirements, and project constraints-has-been_developed.

Problem: Requirements are hazy or nonexistent, yet there is clear recognition that there is a problem, and the problem must be addressed with a software solution. Stakeholders are unsure of what they want; that is, they cannot describe software requirements in any detail.

Solutions: A description of the prototype that identifies basic requirements (e.g., modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, (1) the prototype may evolve through a series of increments to become the production software or (2) the prototype may be discarded and the production software built using some other process-pattern.

Related Patterns: The following patterns are related to this pattern: **customer-communication; iterative design; iterative development, customer assessment; requirement extraction**.

Known uses/examples: Prototyping is recommended when requirements are uncertain.

5) Explain in detail about personal & Team process models and describe Process assessment.

Ans: The PSP and TSP both use proven techniques to improve individual performance and team in software development. Used together they help reduce costs, produce defect-free and inside deadline by empowering development teams. These technologies are based on the premise that a defined and structured process can improve individual work quality and efficiency. PSP focus is on individuals and TSP focus is on teams.

- The **Personal Software Process** (PSP) provides engineers with a disciplined personal framework for doing software work. The PSP process consists of a set of methods, forms, and scripts that show software engineers how to plan, measure, and manage their work.

  The PSP model defines five framework activities:

  Planning. This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.

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

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

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

Postmortem. Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

- The **Team Software Process** (TSP) guides engineering teams that are developing software-intensive products. Using TSP helps organizations establish a mature and disciplined engineering practice that produces secure, reliable software in less time and at lower costs.

  The TSP was introduced in 1998, and builds upon the foundation of PSP to enable engineering teams to build software-intensive products more predictably and effectively. The goal of TSP is to build a "self-directed" project team that organizes itself to produce high-quality software. Humphrey [Hum98] defines the following objectives for TSP:

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

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

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

  • Provide improvement guidance to high-maturity organizations.

  • Facilitate university teaching of industrial-grade team skills.

  TSP defines the following framework activities: project launch, high-level design, implementation, integration and test, and postmortem.

- The **process assessment** is to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering. A number of different approaches to software process assessment have been proposed over the past few decades:

  1. Standard CMMI Assessment Method for Process Improvement (SCAMPI) provides a five—step process assessment model that incorporates initiating, diagnosing, establishing, acting, and learning. The SCAMPI methods use the SEI CMMI as-the-basis-for-assessment.

  i. **SPICE (ISO/IEC 15504)** Standard defines a set of requirements for software process assessment. The intent of the standard is to assist organization in developing an objective evaluation of the efficacy of any defined software process.

  ii. **ISO9001: 2000 for Software** is a generic standard that applies to any organization that wants to improve overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

Chapter-3:

1) Discuss in detail about prescriptive process models.

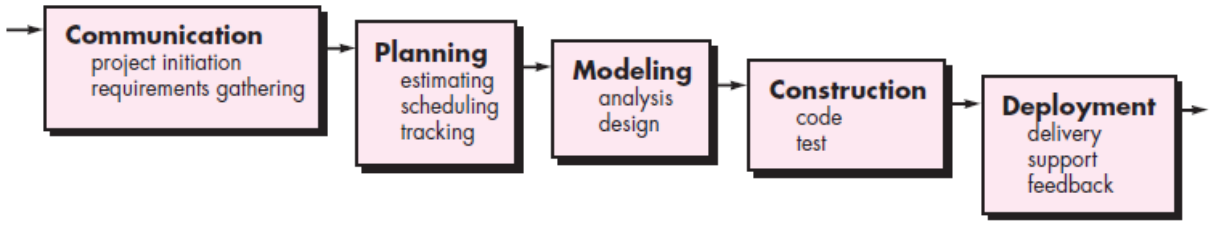
Ans: The following framework activities are carried out irrespective of the process model chosen by-the-organization.
1.Communication
2.Planning
3.Modeling
4.Construction
5.Deployment
The name 'prescriptive' is given because the model prescribes a set of activities, actions, tasks, quality assurance and change the mechanism for every project.

There are three types of prescriptive process models. They are:
1.The-Waterfall-Model
2.Incremental-Process-model
3. RAD model

1. The Waterfall Model

- The waterfall model is also called as **'Linear sequential model'** or **'Classic life cycle model'.**
- In this model, each phase is fully completed before the beginning of the next phase.
- This model is used for the small projects.

- In this model, feedback is taken after each phase to ensure that the project is on the right path.
- Testing part starts only after the development is complete.



NOTE: The description of the phases of the waterfall model is same as that of the process model.

An alternative design for 'linear sequential model' is as follows:

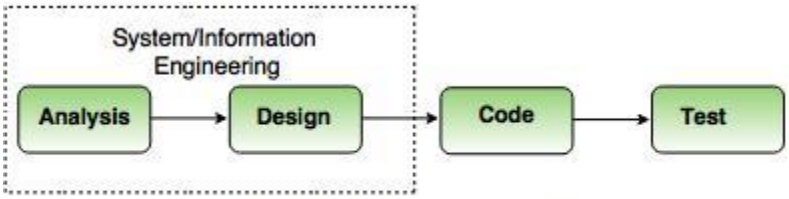


Fig. - The linear sequential model

Advantages of waterfall model

- The waterfall model is simple and easy to understand, implement, and use.
- All the requirements are known at the beginning of the project, hence it is easy to manage.
- It avoids overlapping of phases because each phase is completed at once.
- This model works for small projects because the requirements are understood very well.
- This model is preferred for those projects where the quality is more important as compared to the cost of the project.
- **Disadvantages of the waterfall model**
- This model is not good for complex and object oriented projects.
- It is a poor model for long projects.
- The problems with this model are uncovered, until the software testing.
- The amount of risk is high.

2. Incremental Process model

- The incremental model combines the elements of waterfall model and they are applied in an iterative fashion.
- The first increment in this model is generally a core product.
- Each increment builds the product and submits it to the customer for any suggested modifications.
- The next increment implements on the customer's suggestions and add additional requirements in the previous increment.
- This process is repeated until the product is finished.

For example, the word-processing software is developed using the incremental model.

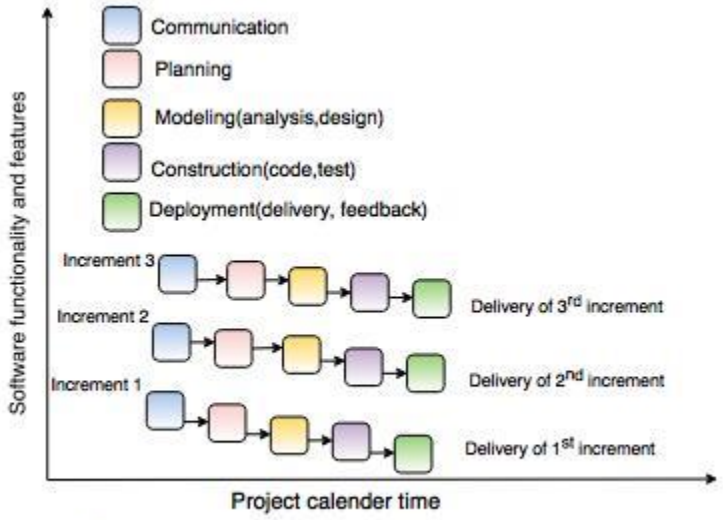


Fig. - Incremental Process Model

Advantages of incremental model

- This model is flexible because the cost of development is low and initial product delivery is faster.
- It is easier to test and debug during the smaller iteration.
- The working software generates quickly and early during the software life cycle.
- The customers can respond to its functionalities after every increment.

Disadvantages of the incremental model

- The cost of the final product may cross the cost estimated initially.
- This model requires a very clear and complete planning.
- The planning of design is required before the whole system is broken into small increments.

- The demands of customer for the additional functionalities after every increment causes problem during the system architecture.

3. RAD model

- RAD is a Rapid Application Development model.
- Using the RAD model, software product is developed in a short period of time.
- The initial activity starts with the communication between customer and developer.
- Planning depends upon the initial requirements and then the requirements are divided into groups.
- Planning is more important to work together on different modules.

The-RAD-model-consist-of-following-phases:

1. Business Modeling

- Business modeling consist of the flow of information between various functions in the project.
- For example what type of information is produced by every function and which are the functions to handle that information.
- A complete business analysis should be performed to get the essential business information.

2. Data modeling

- The information in the business modeling phase is refined into the set of objects and it is essential for the business.
- The attributes of each object are identified and define the relationship between objects.

3. Process modeling

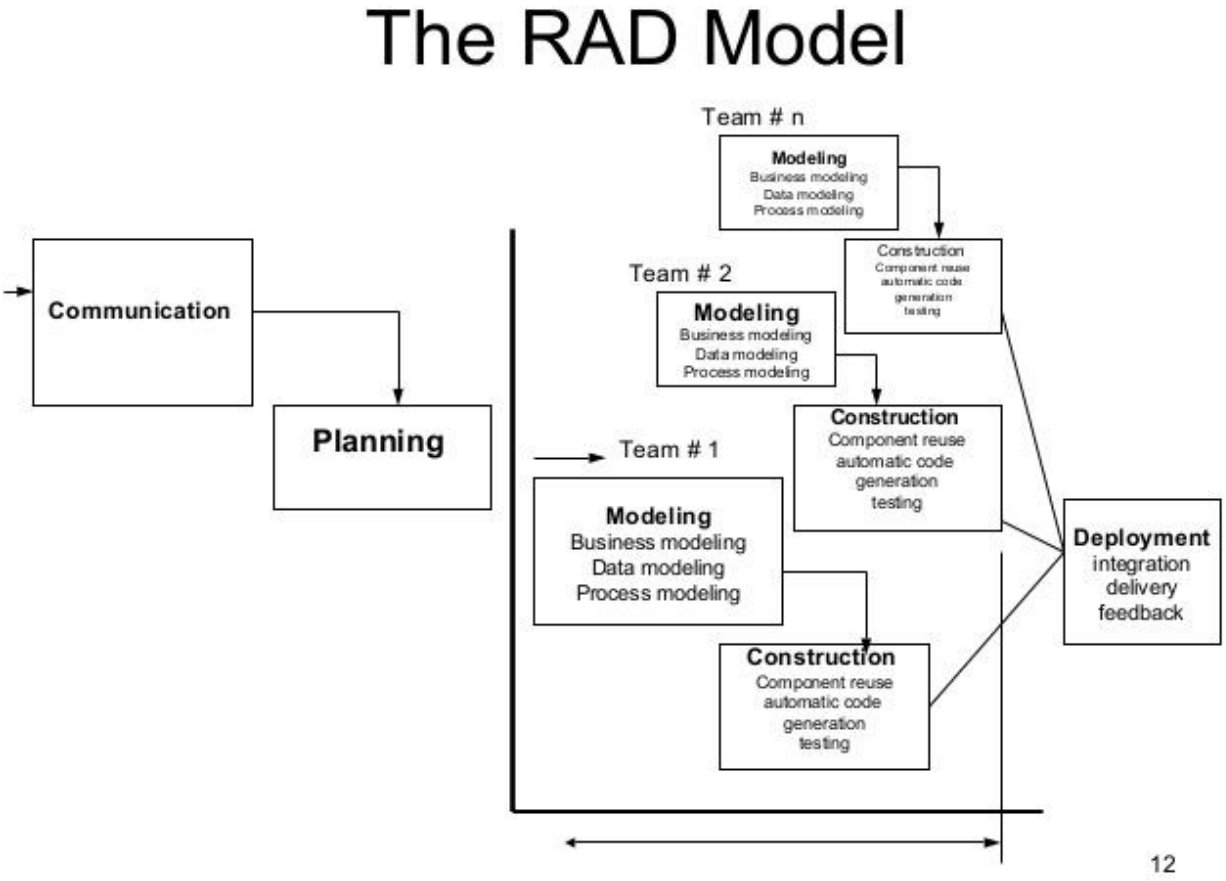
- The data objects defined in the data modeling phase are changed to fulfil the information flow to implement the business model.
- The process description is created for adding, modifying, deleting or retrieving a data object.

4. Application generation

- In the application generation phase, the actual system is built.
- To construct the software the automated tools are used.

5. Testing and turnover

- The prototypes are independently tested after each iteration so that the overall testing time is reduced.
- The data flow and the interfaces between all the components are are fully tested. Hence, most of the programming components are already tested.



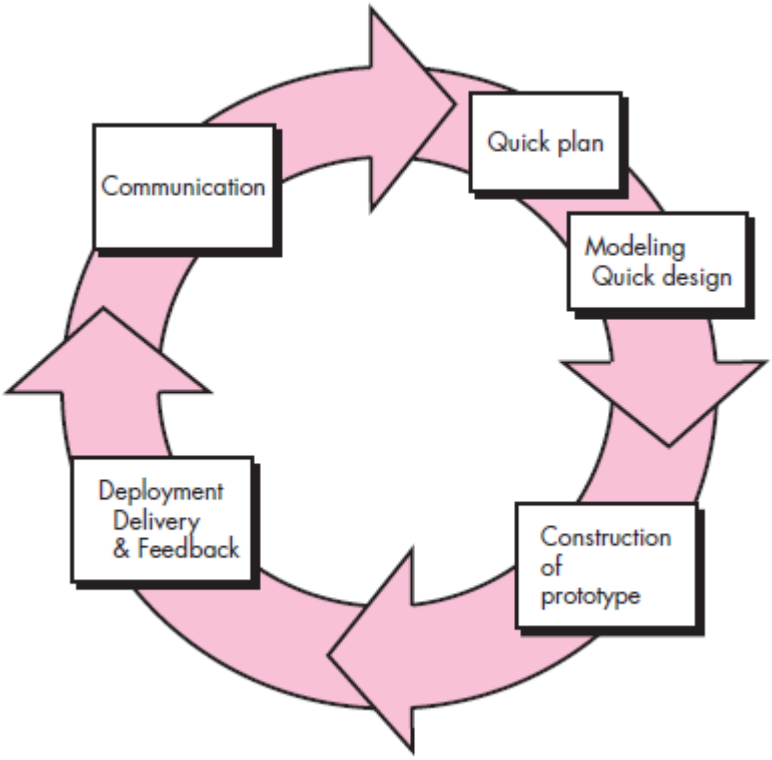
2) Explain about evolutionary process models with a neat diagram.

Ans: Evolutionary software models are iterative. They are characterized in manner that enables the software engineers to develop increasingly more complete version of a software. That is, initially a rapid version of the product is being developed and then the product is developed to more accurate version with the help of the reviewers who review the product after each release and submit improvements. Specification, development and validation are interleaved rather than separate in evolutionary software process model.

These models are applied because as the requirements often change so the end product will be unrealistic, where a complete version is impossible due to tight market deadlines it is better to introduce a limited version to meet the pressure. Thus the software engineers can follow a process

model that has been explicitly designed to accommodate a product that gradually complete over time.

- **Prototyping Model** : Often, a customer defines a set of general objectives for software but does not identify detailed input, processing, or output requirements. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human/machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.



(i) **Communication** : Firstly, developer and customer meet and define the overall objectives, requirements, and outline areas where further definition is mandatory.

(ii) **Quick Plan :** Based on the requirements and others of the communication part a quick plane is made to design the software.

(iii) **Modeling Quick Design** : Based on the quick plane, 'A Quick Design' occurs. The quick design focuses on a representation of those aspects of the software that will be visible to the customer/user, such as input approaches and output formats.

(iv) **Construction of Prototype** : The quick design leads to the construction of a prototype.

(v) **Deployment, delivery and feedback** : The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. All these steps are repeated to tune

the prototype to satisfy user's need. At the same time enable the developer to better understand what needs to be done.

Problems With Prototype Model :

(i) In the rush to get the software working the overall software quality or long-term maintainability will not get consideration. So software, in that way, gets the need of excessive modification to ensure quality.

(ii) The developer may choose inappropriate operating system, algorithms, language in the rush to make the prototype working.

Prototyping is an effective paradigm for software engineering. It necessary to build the prototype to define requirements and then to engineer the software with a eye toward quality.

- **Spiral Model** : The spiral model is an evolutionary software process model that combines the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. Using the spiral model, software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a number of framework activities, also called task regions. Typically, there are between three and six task regions. Given figure is of a spiral model that contains five task regions.

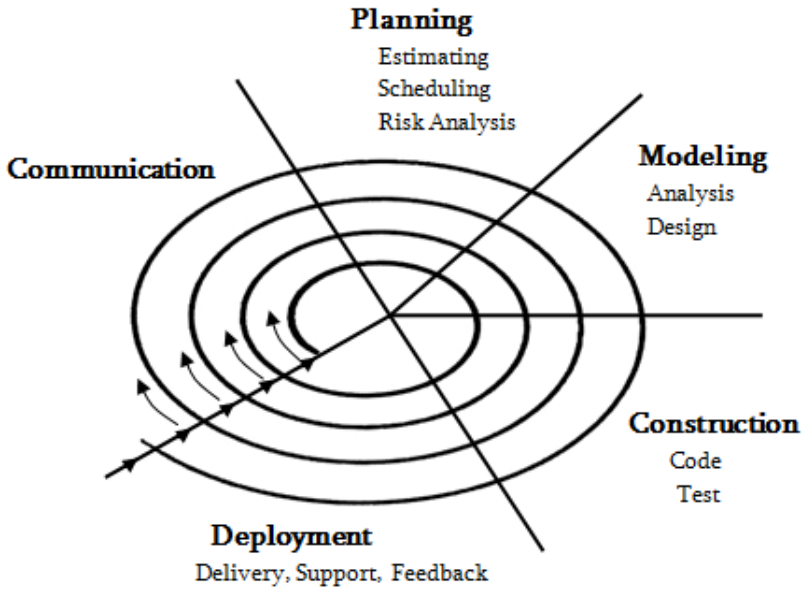


Figure : Spiral Model

model that contains five task regions.

(i) Customer communication — Tasks required to establish effective communication between developer and customer.

(ii) Planning — Tasks required to define resources, timelines, and other project related information.

(iii) Modeling — Tasks required in building one or more representations of the application.

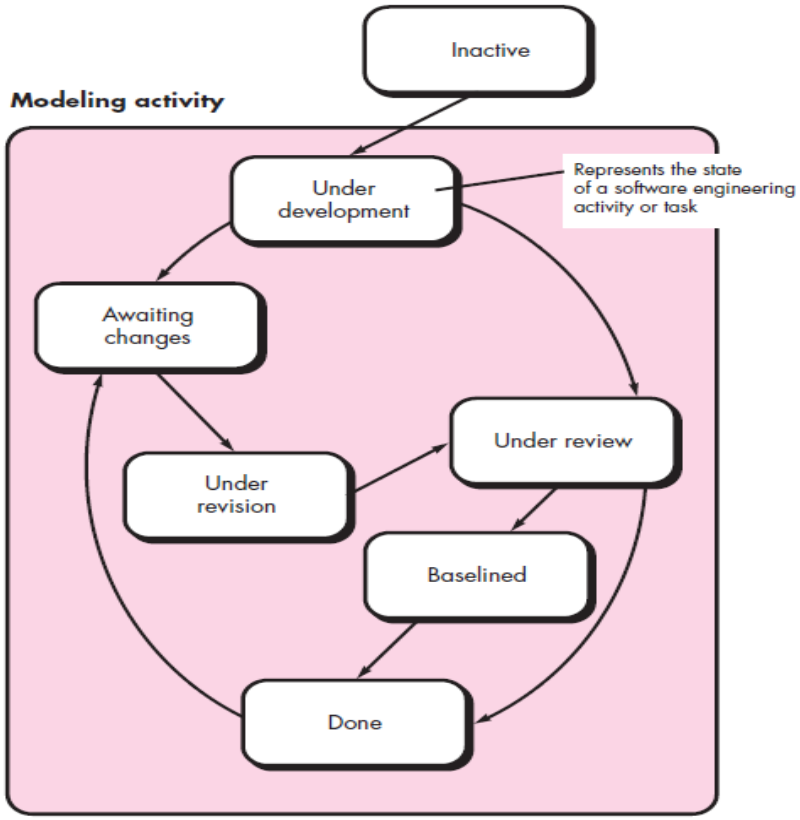
(iv) Construction and release — Tasks required to construct, test, install.

(v) Deployment — Tasks required to deliver the software, getting feedbacks etc.

Software engineering team moves around the spiral in a clockwise direction, beginning at the center. The first circuit around the spiral might result in the development of a 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 through the planning region results in adjustments to the project plan. Cost and schedule are adjusted based on feedback derived from customer evaluation. In addition, the project manager adjusts the planned number of iterations required to complete the software.

The spiral model is a realistic approach to the development of large-scale systems and software. The spiral model enables the developer to apply the prototyping approach at any stage in the evolution of the product. The spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic. It demands considerable risk assessment expertise and relies on this expertise for success.

- **Concurrent process model:**



The concurrent development model, sometimes called concurrent engineering, allows

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

Figure provides a schematic representation of one software engineering activity within the modeling activity using a concurrent modeling approach. The activity—modeling—may be in any one of the states noted at any given time. Similarly, other activities, actions, or tasks (e.g., communication or construction) can be represented in an analogous manner. All software engineering activities exist concurrently but reside in different states.

For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the awaiting changes state. The modeling activity (which existed in the inactive state while initial communication was completed, now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the under development state into the awaiting changes state.

Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks. For example, during early stages of

design (a major software engineering action that occurs during the modeling activity), an inconsistency in the requirements model is uncovered. This generates the event analysis model correction, which will trigger the requirements analysis action from the done state into the awaiting changes state.

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

3) Explain in detail about specialized process models.

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

- **Component based model:**

The component-based development (CBD) model incorporates many of the characteristics of the spiral model. It is evolutionary in nature, demanding an iterative approach to the creation of software. However, the component-based development model composes applications from prepackaged software components, called classes.

Modeling and construction activities begin with the identification of candidate components. Regardless of the technology that has been used to create the components, the model incorporates the following steps

(i) Available component-based products are researched and evaluated for the application domain in question.

(ii) Component integration issues are considered.

(**iii)** A software architecture is designed to accommodate the components.

(iv) Components are integrated into the architecture.

(v) Comprehensive testing is conducted to ensure proper functionality.
The component-based development model leads to software reuse, and re-usability provides software engineers with a number of measurable benefits.

- **Formal model:**

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation.

When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Uncertainty, incompleteness, and inconsistency can be discovered and corrected more easily through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might go undetected.

The concerns raised with the formal model are:

• The development of formal models is currently quite time consuming and expensive.

• Because few software developers have the necessary background to apply formal methods, extensive training is required.

• It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

- **Aspect oriented process model:**

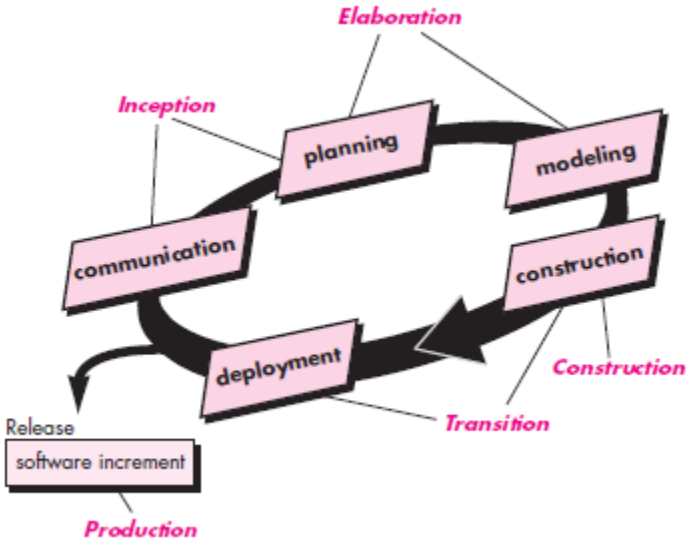
  Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions and information content. These localized software characteristics are modeled as components and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated and complex, certain concerns, customer required properties or areas of technical interest, span the entire architecture. Some concerns are high-level properties of a system; others affect functions or are systemic. When concerns cut across multiple system functions, features, and information they are often referred to as crosscutting concerns. Aspectual requirements define those crosscutting concerns that have impact across the software architecture. Aspects are mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concerns. Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects. A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both the spiral and concurrent process models. The evolutionary nature of the spiral is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components.

4) Explain in detail about unified process model and write all of its 8 phases with a neat diagram.

Ans:

In some ways the unified process (UP) is an attempt to draw on the best features and characteristics of conventional software process models, but characterize them in a way that implements many of the best principles of agile software development. The unified process

recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. It emphasizes the important role of software architecture and helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.



The above dig represents the unified process model. The life of a software system can be represented as a series of cycles. A cycle ends with the release of a version of the system to customers.

Within the Unified Process, each cycle contains four phases. A phase is simply the span of time between two major milestones, points at which managers make important decisions about whether to proceed with development and, if so, what's required concerning project scope, budget, and schedule.

Inception phase:

The primary goal of the Inception phase is to establish the case for the viability of the proposed system. The tasks that a project team performs during Inception include the following:

- Defining the scope of the system (that is, what's in and what's out)
- Outlining a candidate architecture, which is made up of initial versions of six different models
- Identifying critical risks and determining when and how the project will address them
- Starting to make the business case that the project is worth doing, based on initial estimates of cost, effort, schedule, and product quality

Elaboration phase:

The primary goal of the Elaboration phase is to establish the ability to build the new system given the financial constraints, schedule constraints, and other kinds of constraints that the development project faces.

The tasks that a project team performs during Elaboration include the following:

- Capturing a healthy majority of the remaining functional requirements
- Expanding the candidate architecture into a full architectural baseline, which is an internal release of the system focused on describing the architecture
- Addressing significant risks on an ongoing basis
- Finalizing the business case for the project and preparing a project plan that contains sufficient detail to guide the next phase of the project (Construction)

Construction

The primary goal of the **Construction phase** is to build a system capable of operating successfully in beta customer environments.

During Construction, the project team performs tasks that involve building the system iteratively and incrementally (see "Iterations and Increments" later in this chapter), making sure that the viability of the system is always evident in executable form.

The major milestone associated with the Construction phase is called **Initial Operational Capability**. The project has reached this milestone if a set of beta customers has a more or less fully operational system in their hands.

Transition

The primary goal of the **Transition phase** is to roll out the fully functional system to customers. During Transition, the project team focuses on correcting defects and modifying the system to correct previously unidentified problems. The major milestone associated with the Transition phase is called **Product Release**.

Production Phase

- Encompasses the last part of the deployment activity of the generic process
- On-going use of the software is monitored
- Support for the operating environment (infrastructure) is provided
- Defect reports and requests for changes are submitted and evaluated

Chapter-3

AGILE process models:

AGILE is a methodology that promotes **continuous iteration** of development and testing throughout the software development life cycle of the project. Both development and testing activities are concurrent unlike the Waterfall model

The agile software development emphasizes on four core values.

1. Individual and team interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

The Agile Alliance defines **12 agility principles** for those who want to achieve agility:

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.

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

Proponents of agile software development take great pains to emphasize the importance of "people factors." As Cockburn and High-smith [Coc01a] state, "Agile development focuses on the talents and skills of individuals, molding the process to specific people and teams." The key point in this statement is that the process molds to the needs of the people and team, not the other way around.2

If members of the software team are to drive the characteristics of the process that is applied to build software, a number of key traits must exist among the people on an agile team and the team itself:

Competence. In an agile development (as well as software engineering) context, "competence" encompasses innate talent, specific software-related skills, and overall knowledge of the process that the team has chosen to apply. Skill and knowledge of process can and should be taught to all people who serve as agile team members.

Common focus. Although members of the agile team may perform different tasks and bring different skills to the project, all should be focused on one goal—to deliver a working software increment to the customer within the time promised. To achieve this goal, the team will also focus on continual adaptations (small and large) that will make the process fit the needs of the team.

Collaboration. Software engineering (regardless of process) is about assessing, analyzing, and using information that is communicated to the software team; creating information that will help all stakeholders understand the work of the team; and building information (computer software and relevant databases) that provides business value for the customer. To accomplish these tasks, team members must collaborate—with one another and all other stakeholders.

Decision-making ability. Any good software team (including agile teams) must be allowed the freedom to control its own destiny. This implies that the team is given autonomy—decision-making authority for both technical and project issues.

Fuzzy problem-solving ability. Software managers must recognize that the agile team will continually have to deal with ambiguity and will continually be buffeted by change. In some cases, the team must accept the fact that the problem they are solving today may not be the problem that needs to be solved tomorrow. However, lessons learned from any problem-solving activity (including those that solve the wrong problem) may be of benefit to the team later in the project.

Mutual trust and respect. The agile team must become what De-Marco and Lister [DeM98] call a "jelled" team (Chapter 24). A jelled team exhibits the trust and respect that are necessary to make them "so strongly knit that the whole is greater than the sum of the parts." [DeM98]

Self-organization. In the context of agile development, self-organization implies three things: (1) the agile team organizes itself for the work to be done, (2) the team organizes the process to best accommodate its local environment, (3)the team organizes the work schedule to best achieve delivery of the software increment. Self-organization has a number of technical benefits, but more importantly, it serves to improve collaboration and boost team morale. In essence, the team serves as its own management

Agile process models:

-> XP model (Extreme programming model):

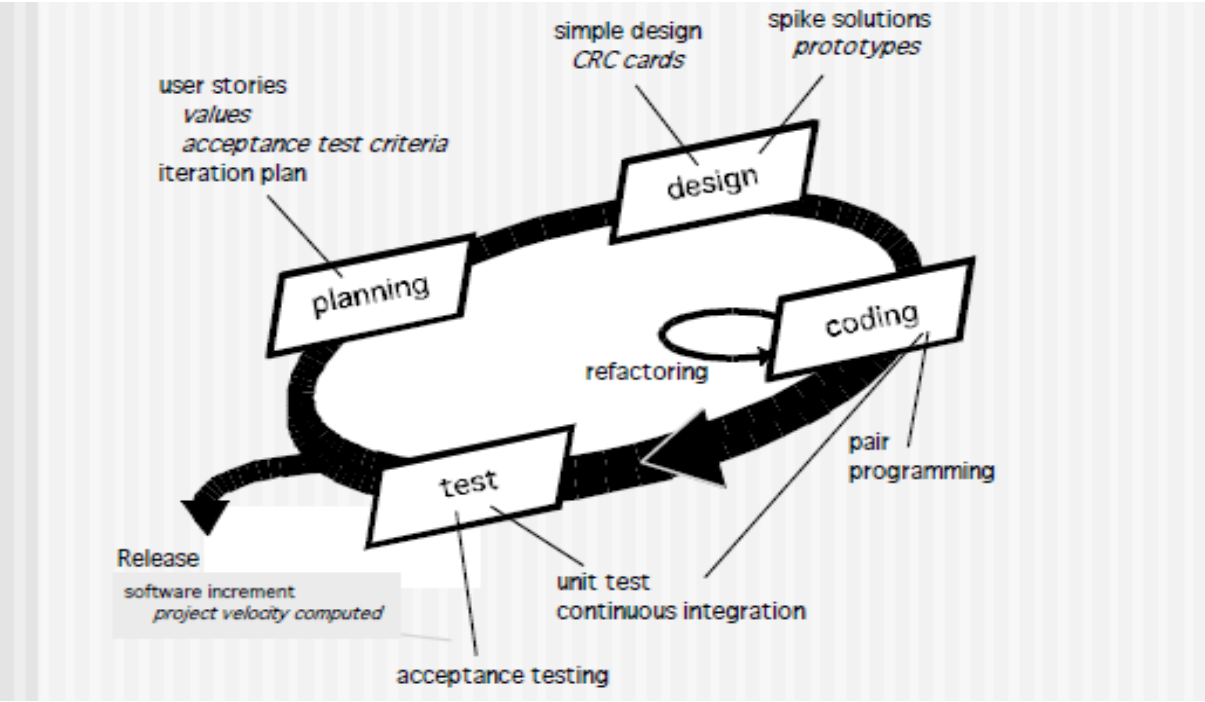
The most widely used agile process, originally proposed by Kent Beck in 2004. It uses an object-oriented approach. !

 XP Planning!

 Begins with the listening, leads to creation of "user stories" that describes required output, features, and functionality. Customer assigns a value (i.e., a priority) to each story. !

 Agile team assesses each story and assigns a cost (development weeks. If more than 3 weeks, customer asked to split into smaller stories)!

Working together, stories are grouped for a deliverable increment next release. !



 A commitment (stories to be included, delivery date and other project matters) is made. Three ways: 1. Either all stories will be implemented in a few weeks, 2. High priority stories first, or 3. the riskiest stories will be implemented first. !

 After the first increment "project velocity", namely number of stories implemented during the first release is used to help define subsequent delivery dates for other increments. Customers can add stories, delete existing stories, change values of an existing story, split stories as development work proceeds.

 XP Design (occurs both before and after coding as refactoring is encouraged)!

 Follows the KIS principle (keep it simple) Nothing more nothing less than the story. !

 Encourage the use of CRC (class-responsibility-collaborator) cards in an object-oriented context. The only design work product of XP. They identify and organize the classes that are relevant to the current software increment.

 For difficult design problems, suggests the creation of "spike solutions"—a design prototype for that portion is implemented and evaluated. !

 Encourages "refactoring"—an iterative refinement of the internal program design. Does not alter the external behavior yet improve the internal structure. Minimize chances of bugs. More efficient, easy to read. !

 XP Coding!

 Recommends the construction of a unit test for a story before coding commences. So implementer can focus on what must be implemented to pass the test. !

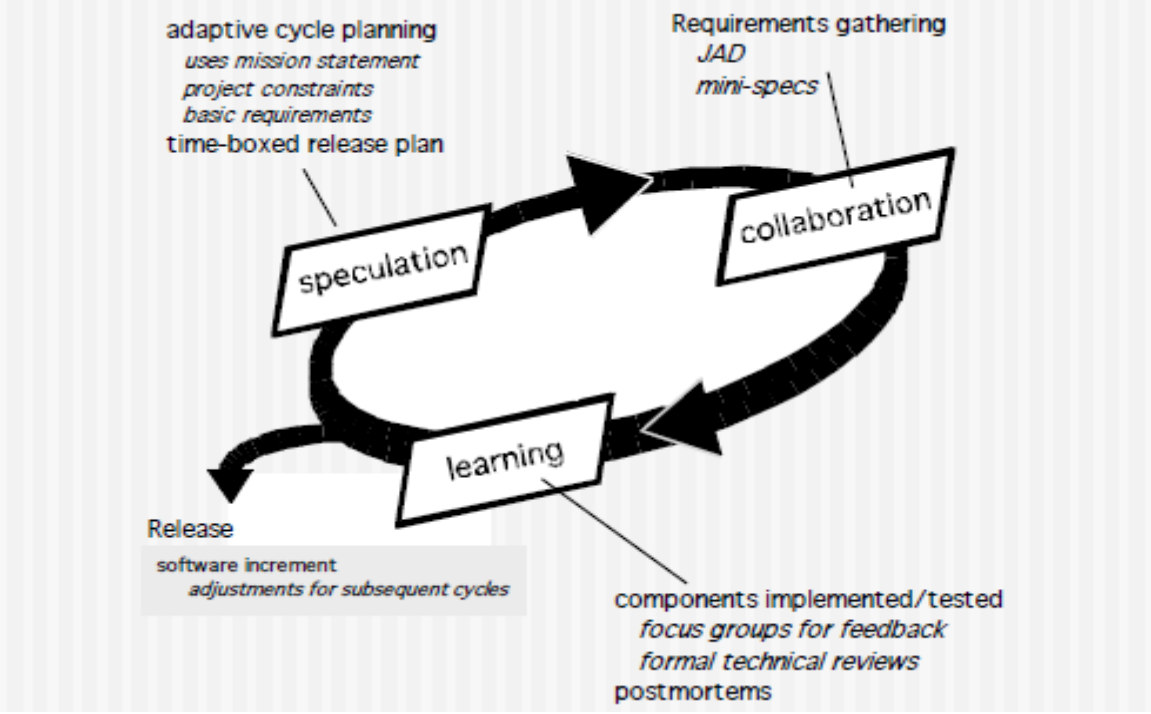
 Encourages "pair programming". Two people work together at one workstation. Real time problem solving, real time review for quality assurance. Take slightly different roles. !

 XP Testing!

 All unit tests are executed daily and ideally should be automated. Regression tests are conducted to test current and previous components. !

 "Acceptance tests" are defined by the customer and executed to assess customer visible functionality!

-> Adaptive software development:



 Originally proposed by Jim High-smith (2000) focusing on human collaboration and team self-organization as a technique to build complex software and system. !

 ASD — distinguishing features!

 Mission-driven planning!

 Component-based focus!

 Explicit consideration of risks!

 Emphasizes collaboration for requirements gathering!

 Emphasizes “learning” throughout the process

Three phase of ASD are:

 1. **Speculation:** project is initiated and adaptive cycle planning is conducted. Adaptive cycle planning uses project initiation information- the customer’s mission statement, project constraints (e.g. delivery date), and basic requirements to define the set of release cycles (increments) that will be required for the project. Based on the information obtained at the completion of the first cycle, the plan is reviewed and adjusted so that planned work better fits the reality.

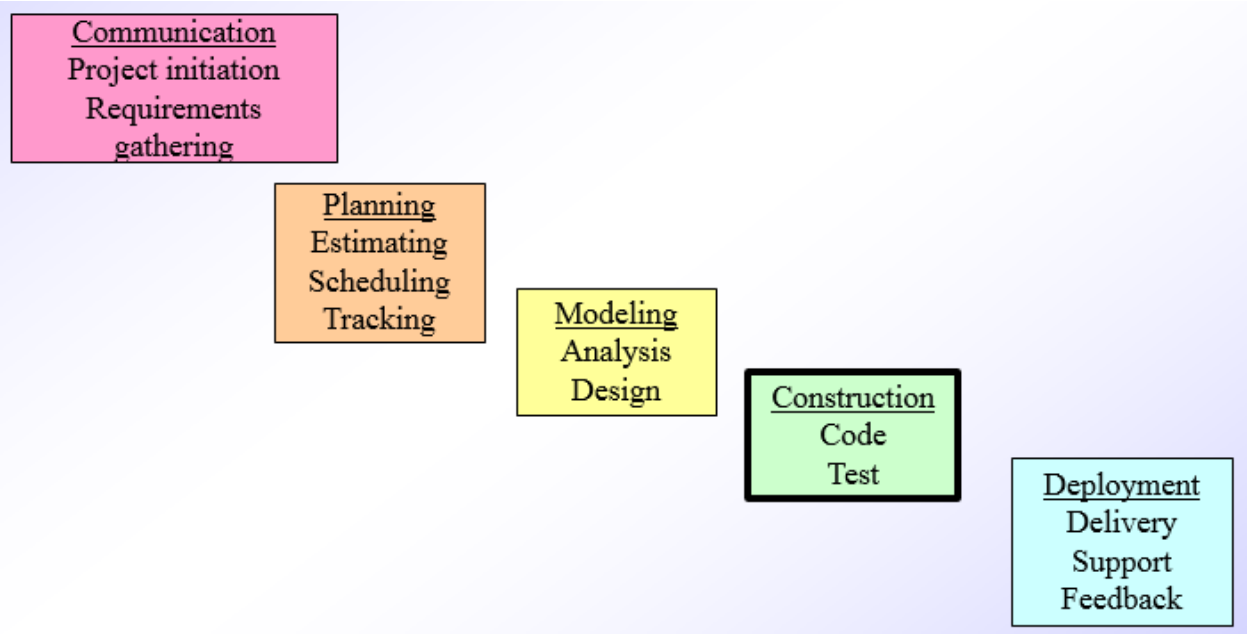
 2. **Collaborations** are used to multiply their talent and creative output beyond absolute number (1+1>2). It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. !

 It is a matter of trust. 1)criticize without animosity, 2) assist without resentments, 3) work as hard as or harder than they do. 4)have the skill set to contribute to the work at hand, 5) communicate problems or concerns in a way that leads to effective action. !

 3. **Learning:** As members of ASD team begin to develop the components, the emphasis is on “learning”. High-smith argues that software developers often overestimate their own understanding of the technology, the process, and the project and that learning will help them to improve their level of real understanding. Three ways: focus groups, technical reviews and project postmortems.

2nd unit:

Software Engineering practice



1) Explain about software engineering practices and core principles

Software engineering practices:

- Consists of a collection of concepts, principles, methods, and tools that a software engineer calls upon on a daily basis
- Equips managers to manage software projects and software engineers to build computer programs
- Provides necessary technical and management how to's in getting the job done
- Transforms a haphazard unfocused approach into something that is more organized, more effective, and more likely to achieve success

The following set of core principles can be applied to the framework, and by extension, to every software process.

Principle 1. Be agile. Whether the process model you choose is prescriptive or agile, the basic tenets of agile development should govern your approach. Every aspect of the work you do should emphasize economy of action—keep your technical approach as simple as possible, keep the work products you produce as concise as possible, and make decisions local whenever possible.

Principle 2. Focus on quality at every step. The exit condition for every process activity, action, and task should focus on the quality of the work product that has been produced.

Principle 3. Be ready to adapt. Process is not a religious experience, and dogma has no place in it. When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.

Principle 4. Build an effective team. Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team that has mutual trust and respect.

Principle 5. Establish mechanisms for communication and coordination. Projects fail because important information falls into the cracks and/or stakeholders fail to coordinate their efforts to create a successful end product. These are management issues and they must be addressed.

Principle 6. Manage change. The approach may be either formal or informal, but mechanisms must be established to manage the way changes are requested, assessed, approved, and implemented.

Principle 7. Assess risk. Lots of things can go wrong as software is being developed. It's essential that you establish contingency plans.

Principle 8. Create work products that provide value for others. Create only those work products that provide value for other process activities, actions, or tasks. Every work product that is produced as part of software engineering practice will be passed on to someone else. A list of required functions and features will be passed along to the person (people) who will develop a design, the design will be passed along to those who generate code, and so on. Be sure that the work product imparts the necessary information without ambiguity or omission.

2) Explain in detail about communication and planning practices.

Ans:

Communication principles:

1) Listen to the speaker and concentrate on what is being said
2) Prepare before you meet by researching and understanding the problem
3) Someone should facility the meeting and have an agenda
4) Face-to-face communication is best, but also have a document or presentation to focus the discussion
5) Take notes and document decisions
6) Strive for collaboration and consensus
7) Stay focused on a topic; modularize your discussion
8) If something is unclear, draw a picture
9) Move on to the next topic a) after you agree to something, b) if you cannot agree to something, or c) if a feature or function is unclear and cannot be clarified at the moment

10) Negotiation is not a contest or a game; it works best when both parties win

Planning principles:

1) Understand the scope of the project
2) Involve the customer in the planning activity
3) Recognize that planning is iterative; things will change
4) Estimate based only on what you know
5) Consider risk as you define the plan
6) Be realistic on how much can be done each day by each person and how well
7) Adjust granularity as you define the plan
8) Define how you intend to ensure quality
9) Describe how you intend to accommodate change
10) Track the plan frequently and make adjustments as required

3)Explain about different types of models and explain about design modelling principles.

Ans: In software engineering work, two classes of models can be created: requirements models and design models. Requirements models (also called analysis models) represent customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral domain. Design models represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Modelling principles are classified into Analysis modelling principles and design modelling principles.

Analysis Modeling Principles

1) The information domain of a problem (the data that flows in and out of a system) must be represented and understood
2) The functions that the software performs must be defined
3) The behavior of the software (as a consequence of external events) must be represented
4) The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion
5) The analysis task should move from essential information toward implementation detail

Design modelling principles

1) The design should be traceable to the analysis model
2) Always consider the software architecture of the system to be built
3) Design of data is as important as design of processing functions
4) Interfaces (both internal and external) must be designed with care
5) User interface design should be tuned to the needs of the end-user and should stress ease of use
6) Component-level design should be functionally independent (high cohesion)
7) Components should be loosely coupled to one another and to the external environment
8) Design representations (models) should be easily understandable
9) The design should be developed iteratively; with each iteration, the designer should strive for greater simplicity

Note:

External quality factors: those properties that can be readily observed

Internal quality factors: those properties that lead to a high-quality design from a technical perspective

4)Explain about construction practices and deployment practices

Ans: The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end user. In modern software engineering work, coding may be

(1) The direct creation of programming language source code (e.g., Java),

(2) The automatic generation of source code using an intermediate design-like representation of the component to be built, or

(3) The automatic generation of executable code using a "fourth-generation programming

language" (e.g., Visual C).

The initial focus of testing is at the component level, often called unit testing. Other levels of testing include (1) integration testing (conducted as the system is constructed), validation testing that assesses whether requirements have been met for the complete system (or software increment), and (3) acceptance testing that is conducted by the customer in an effort to exercise all required features and functions. The following set of fundamental principles and concepts are applicable to coding and testing

Coding Principles. The principles that guide the coding task are closely aligned with programming style, programming languages, and programming methods. However, there are a number of fundamental principles that can be stated:

(Preparation before coding):

1) Understand the problem you are trying to solve
2) Understand basic design principles and concepts
3) Pick a programming language that meets the needs of the software to be built and the environment in which it will operate
4) Select a programming environment that provides tools that will make your work easier
5) Create a set of unit tests that will be applied once the component you code is completed

(As you begin coding)

1) Constrain your algorithms by following structured programming practices
2) Select data structures that will meet the needs of the design
3) Understand the software architecture and create interfaces that are consistent with it
4) Keep conditional logic as simple as possible
5) Create nested loops in a way that makes them easily testable
6) Select meaningful variable names and follow other local coding standards
7) Write code that is self-documenting
8) Create a visual layout (e.g., indentation and blank lines) that aids code understanding

(After completing the first round of code)

1) Conduct a code walkthrough
2) Perform unit tests (black-box and white-box) and correct errors you have uncovered
3) Refactor the code

Testing:

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

Testing principles:

1) All tests should be traceable to the software requirements
2) Tests should be planned long before testing begins
3) The Pareto principle applies to software testing
    - 80% of the uncovered errors are in 20% of the code
4) Testing should begin "in the small" and progress toward testing "in the large"
    - Unit testing --> integration testing --> validation testing --> system testing
5) Exhaustive testing is not possible

Deployment principles:

1) Customer expectations for the software must be managed
    - Be careful not to promise too much or to mislead the user
2) A complete delivery package should be assembled and tested
3) A support regime must be established before the software is delivered
4) Appropriate instructional materials must be provided to end users
5) Buggy software should be fixed first, delivered later.

Chapter-7: Requirement Engineering

The Problems with our Requirements Practices:

- We have trouble understanding the requirements that we do acquire from the customer
- We often record requirements in a disorganized manner
- We spend far too little time verifying what we do record
- We allow change to control us, rather than establishing mechanisms to control change
- Most importantly, we fail to establish a solid foundation for the system or software that the user wants built
- Many software developers argue that

1) Building software is so compelling that we want to jump right in (before having a clear understanding of what is needed)
2) Things will become clear as we build the software
3) Project stakeholders will be able to better understand what they need only after examining early iterations of the software
4) Things change so rapidly that requirements engineering is a waste of time
5) The bottom line is producing a working program and that all else is secondary

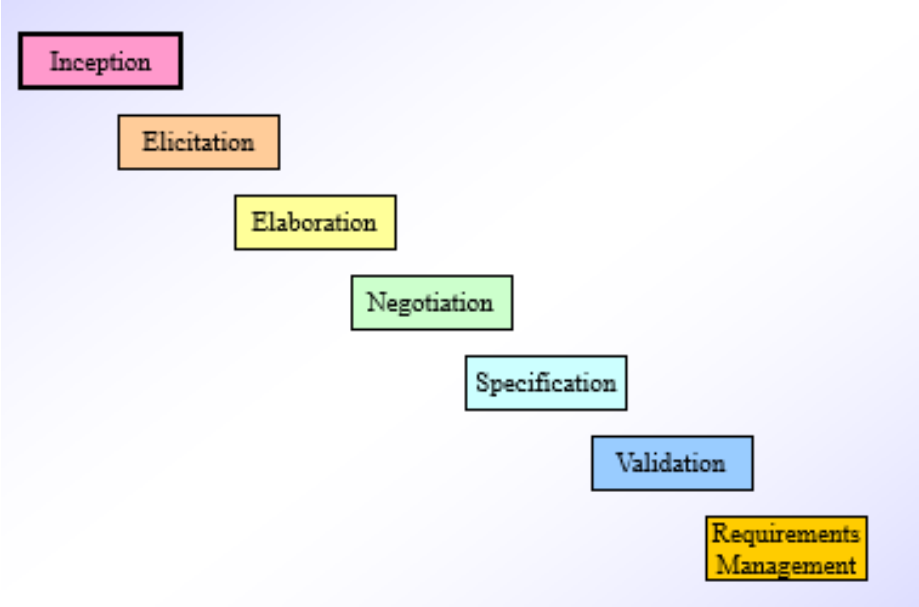
- All of these arguments contain some truth, especially for small projects that take less than one month to complete
- However, as software grows in size and complexity, these arguments begin to break down and can lead to a failed software project

A Solution: Requirements Engineering

- Begins during the communication activity and continues into the modeling activity
- Builds a bridge from the system requirements into software design and construction
- Allows the requirements engineer to examine
  - the context of the software work to be performed
  - the specific needs that design and construction must address
  - the priorities that guide the order in which work is to be completed
  - the information, function, and behavior that will have a profound impact on the resultant design

Requirements Engineering Tasks:

- Seven distinct tasks
  - Inception
  - Elicitation
  - Elaboration
  - Negotiation
  - Specification
  - Validation
  - Requirements Management
- Some of these tasks may occur in parallel and all are adapted to the needs of the project
- All strive to define what the customer wants
- All serve to establish a solid foundation for the design and construction of the software.



Inception Task:

- During inception, the requirements engineer asks a set of questions to establish…
  - A basic understanding of the problem
  - The people who want a solution
  - The nature of the solution that is desired

  - The effectiveness of preliminary communication and collaboration between the customer and the developer

- Through these questions, the requirements engineer needs to…
  - Identify the stakeholders
  - Recognize multiple viewpoints
  - Work toward collaboration
- Break the ice and initiate the communication

Elicitation:

- Eliciting requirements is difficult because of
  - Problems of scope in identifying the boundaries of the system or specifying too much technical detail rather than overall system objectives
  - Problems of understanding what is wanted, what the problem domain is, and what the computing environment can handle (Information that is believed to be "obvious" is often omitted)
  - Problems of volatility because the requirements change over time
- Elicitation may be accomplished through two activities
  - Collaborative requirements gathering
  - Quality function deployment

Basic Guidelines of Collaborative Requirements Gathering:

- Meetings are conducted and attended by both software engineers, customers, and other interested stakeholders
- Rules for preparation and participation are established
- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas
- A "facilitator" (customer, developer, or outsider) controls the meeting
- A "definition mechanism" is used such as work sheets, flip charts, wall stickers, electronic bulletin board, chat room, or some other virtual forum
- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.

Quality Function Deployment:

- This is a technique that translates the needs of the customer into technical requirements for software
- It emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process through functions, information, and tasks
- It identifies three types of requirements
  - Normal requirements: These requirements are the objectives and goals stated for a product or system during meetings with the customer
  - Expected requirements: These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them
  - Exciting requirements: These requirements are for features that go beyond the customer's expectations and prove to be very satisfying when present.

Elicitation Work Products:

The work products will vary depending on the system, but should include one or more of the following items:

- A statement of need and feasibility
- A bounded statement of scope for the system or product
- A list of customers, users, and other stakeholders who participated in requirements elicitation
- A description of the system's technical environment
- A list of requirements (organized by function) and the domain constraints that apply to each
- A set of preliminary usage scenarios (in the form of use cases) that provide insight into the use of the system or product under different operating conditions
- Any prototypes developed to better define requirements

Elaboration Task:

- During elaboration, the software engineer takes the information obtained during inception and elicitation and begins to expand and refine it
- Elaboration focuses on developing a refined technical model of software functions, features, and constraints
- It is an analysis modeling task

  - Use cases are developed
  - Domain classes are identified along with their attributes and relationships
  - State machine diagrams are used to capture the life on an object
- The end result is an analysis model that defines the functional, informational, and behavioral domains of the problem.

Developing Use Cases:

"A use case captures a contract ... [that] describes the system's behavior under various conditions as the system responds to a request from one of its stakeholders . . ."

- Step One – Define the set of actors that will be involved in the story
  - Actors are people, devices, or other systems that use the system or product within the context of the function and behavior that is to be described
  - Actors are anything that communicate with the system or product and that are external to the system itself
- Step Two – Develop use cases, where each one answers a set of questions

Questions Commonly Answered by a Use Case:

Actors are the different people (or devices) that use the system or product within the context of the function and behavior that is to be described. Actors represent the roles that people (or devices) play as the system operates.

Secondary actors support the system so that primary actors can do their work.

- Who is the primary actor(s), the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the scenario begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the scenario is described?
- What variations in the actor's interaction are possible?
- What system information will the actor acquire, produce, or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?

Example of Use case working [Home security system]:

Recalling basic Safe Home requirements, we define four actors: homeowner (a user), setup manager (likely the same person as homeowner, but playing a different role), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors the Safe Home home security function). For the purposes of this example, we consider only the homeowner actor. The homeowner actor interacts with the home security function in a number of different ways using either the alarm control panel or a PC:

• Enters a password to allow all other interactions.

• Inquires about the status of a security zone.

• Inquires about the status of a sensor.

• Presses the panic button in an emergency.

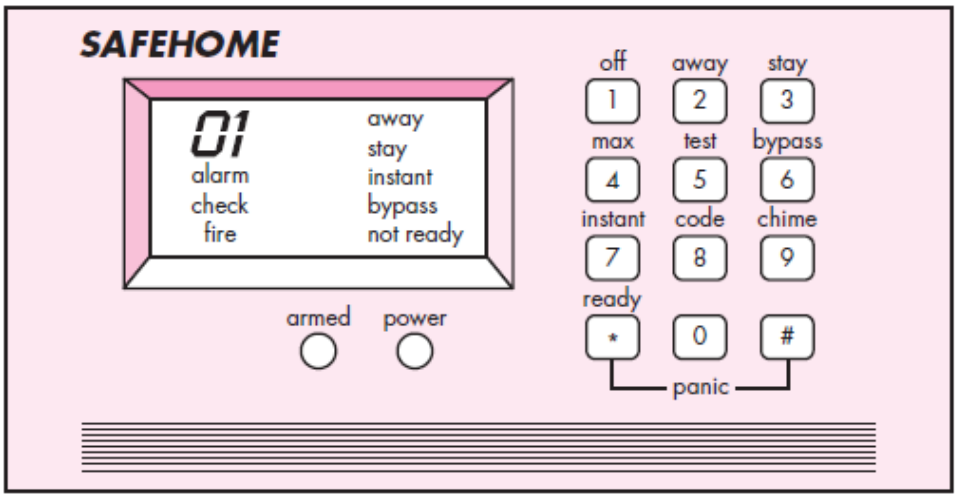
• Activates/deactivates the security system.

Considering the situation in which the homeowner uses the control panel, the basic

Use case for system activation follows

1. The homeowner observes the SafeHome control panel (Figure 5.1) to determine if the system is ready for input. If the system is not ready, a not ready message is displayed on the LCD display, and the homeowner must physically close windows or doors so that the not ready message disappears. [A not ready message implies that a sensor is open; i.e., that a door or window is open.]

FIGURE 5.1
SafeHome control panel



2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

3. The homeowner selects and keys in stay or away (see Figure 5.1) to activate the system. Stay activates only perimeter sensors (inside motion detecting sensors are deactivated). Away activates all sensors.

4. When activation occurs, a red alarm light can be observed by the homeowner.

The basic use case presents a high-level story that describes the interaction between the actor and the system.

In many instances, uses cases are further elaborated to provide considerably more detail about the interaction. For example, Cockburn [Coc01b] suggests the following template for detailed descriptions of use cases:

Use case: Initiate Monitoring

Primary actor: Homeowner.

Goal in context: To set the system to monitor sensors when the homeowner leaves the house or remains inside.

Preconditions: System has been programmed for a password and to recognize various sensors.

Trigger: The homeowner decides to "set" the system, i.e., to turn on the alarm functions.

Scenario:

1. Homeowner: observes control panel

2. Homeowner: enters password

3. Homeowner: selects "stay" or "away"

4. Homeowner: observes read alarm light to indicate that SafeHome has been armed

Exceptions:

1. Control panel is not ready: homeowner checks all sensors to determine which are open; closes them.

2. Password is incorrect (control panel beeps once): homeowner reenters correct password.

3. Password not recognized: monitoring and response subsystem must be contacted to reprogram password.

4. Stay is selected: control panel beeps twice and a stay light is lit; perimeter sensors are activated.

5. Away is selected: control panel beeps three times and an away light is lit; all sensors are activated.

Priority: Essential, must be implemented

When available: First increment

Frequency of use: Many times per day

Channel to actor: Via control panel interface

Secondary actors: Support technician, sensors

Channels to secondary actors:

Support technician: phone line

Sensors: hardwired and radio frequency interfaces

Open issues:

1. Should there be a way to activate the system without the use of a password or with an abbreviated password?

2. Should the control panel display additional text messages?

3. How much time does the homeowner have to enter the password from the time the first key is pressed?

4. Is there a way to deactivate the system before it actually activates?

Use cases for other home owner interactions would be developed in a similar manner.

It is important to review each use case with care. If some element of the interaction is ambiguous, it is likely that a review of the use case will indicate a problem.

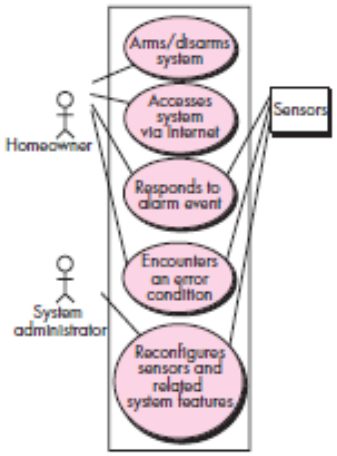


FIGURE 5.2
UML use case diagram for *SafeHome* home security function

Negotiation Task:

- During negotiation, the software engineer reconciles the conflicts between what the customer wants and what can be achieved given limited business resources
- Requirements are ranked (i.e., prioritized) by the customers, users, and other stakeholders

- Risks associated with each requirement are identified and analyzed
- Rough guesses of development effort are made and used to assess the impact of each requirement on project cost and delivery time
- Using an iterative approach, requirements are eliminated, combined and/or modified so that each party achieves some measure of satisfaction

The Art of Negotiation

- Recognize that it is not competition
- Map out a strategy
- Listen actively
- Focus on the other party's interests
- Don't let it get personal
- Be creative
- Be ready to commit

Specification Task

- A specification is the final work product produced by the requirements engineer
- It is normally in the form of a software requirements specification
- It serves as the foundation for subsequent software engineering activities
- It describes the function and performance of a computer-based system and the constraints that will govern its development
- It formalizes the informational, functional, and behavioral requirements of the proposed software in both a graphical and textual format

Typical Contents of a Software Requirements Specification:

- Requirements
  - Required states and modes
  - Software requirements grouped by capabilities (i.e., functions, objects)
  - Software external interface requirements
  - Software internal interface requirements
  - Software internal data requirements

  - Other software requirements (safety, security, privacy, environment, hardware, software, communications, quality, personnel, training, logistics, etc.)
  - Design and implementation constraints
- Qualification provisions to ensure each requirement has been met
  - Demonstration, test, analysis, inspection, etc.
- Requirements traceability
  - Trace back to the system or subsystem where each requirement applies

Validation Task:

- During validation, the work products produced as a result of requirements engineering are assessed for quality
- The specification is examined to ensure that
  - all software requirements have been stated unambiguously
  - inconsistencies, omissions, and errors have been detected and corrected
  - the work products conform to the standards established for the process, the project, and the product
- The formal technical review serves as the primary requirements validation mechanism
  - Members include software engineers, customers, users, and other stakeholders.

Questions to ask when Validating Requirements

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?

-----Approaches: Demonstration, actual test, analysis, or inspection

- Does the requirements model properly reflect the information, function, and behavior of the system to be built?
- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system.

Requirements Management Task:

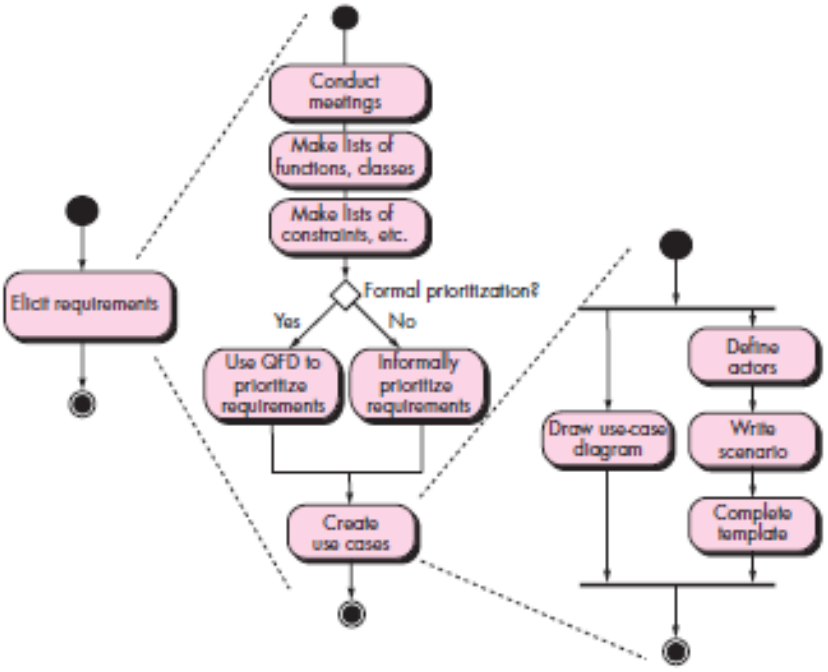
- During requirements management, the project team performs a set of activities to identify, control, and track requirements and changes to the requirements at any time as the project proceeds
- Each requirement is assigned a unique identifier
- The requirements are then placed into one or more traceability tables
- These tables may be stored in a database that relate features, sources, dependencies, subsystems, and interfaces to the requirements
- A requirements traceability table is also placed at the end of the software requirements specification.

>>Elements of the Analysis Model

- Scenario-based elements
  - Describe the system from the user's point of view using scenarios that are depicted in use cases and activity diagrams
- Class-based elements
  - Identify the domain classes for the objects manipulated by the actors, the attributes of these classes, and how they interact with one another; they utilize class diagrams to do this
- Behavioral elements
  - Use state diagrams to represent the state of the system, the events that cause the system to change state, and the actions that are taken as a result of a particular event; can also be applied to each class in the system
- Flow-oriented elements
  - Use data flow diagrams to show the input data that comes into a system, what functions are applied to that data to do transformations, and what resulting output data are produced.

FIGURE 5.3

UML activity diagrams for eliciting requirements



Requirement Analysis results in the specification of software's operational characteristics; indicates software's interface with other system element; and establishes constraints that software must meet. Throughout analysis modeling the software engineer's primary focus is on *what* not *how*.
RA allows the software engineer (called an *analyst* or *modeler* in this role) to:

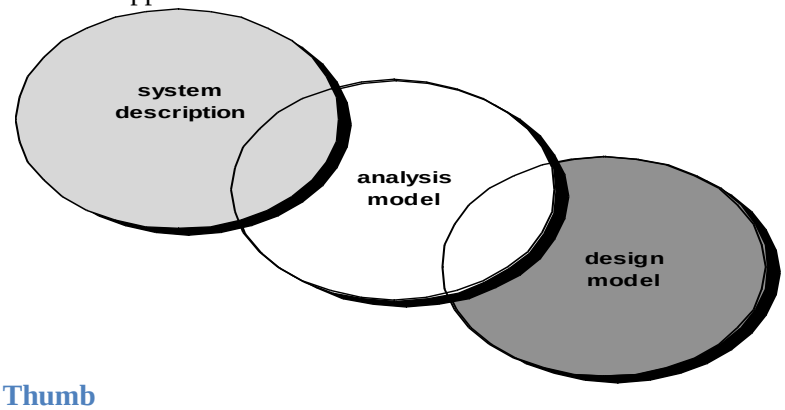
- Elaborate on basic requirements established during earlier requirement engineering tasks
- Build models that depict user scenarios, functional activities, problem classes and their relationships, system and class behavior, and the flow of data as it is transformed.

8.1.1 Overall Objectives and Philosophy

The analysis model must achieve three primary objectives:

1. To describe what the customer requires
2. To establish a basis for the creation of a software design, and
3. To define a set of requirements that can be validated once the software is built.

The analysis model bridges the gap between a system-level description that describes overall functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software application architecture.



8.1.2 Analysis Rules of Thumb

- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the information domain, function and behavior of the system.
- Delay consideration of infrastructure and other non-functional models until design.
    - For example, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.
- Minimize coupling throughout the system.
    - The level of interconnectedness between classes and functions should be reduced to a minimum.
- Be certain that the analysis model provides value to all stakeholders.
    - Each constituency has its own use for the model.
- Keep the model as simple as it can be.
    - Ex: Don't add additional diagrams when they provide no new information.
    - Only modeling elements that have values should be implemented.

8.1.3 Domain Analysis

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks . . . ***Donald Firesmith***

- Define the domain to be investigated.
- Collect a representative sample of applications in the domain.
- Analyze each application in the sample.
- Develop an analysis model for the objects

8.2 Analysis Modeling Concepts and Approaches

One view of analysis modeling, called structural analysis, considers data and the processes that transform the data as separate entities.

Data objects are modeled in a way that defines attributes and relationships.

Processes that manipulate data objects are in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeling called *object-oriented analysis* focuses on the definition of classes and the manner in which they collaborate with one another to affect customer requirements. UML and the Unified Process are predominantly Object Oriented.

8.3 Data Modeling Concepts

8.3. 1. Data Objects

A *data object* is a representation of almost any composite information that must be processed by software. By composite, we mean something that has a number of different properties and attributes.

- "Width" (a single value) would not be a valid data object, but **dimensions** (incorporating height, width and depth) could be defined as object.

A data object encapsulates data only – there is no reference within a data object to operations that act on the data. Therefore, the data can be represented as a table below.

object: automobile
attributes:
- **make**
- **model**
- **body type**
- **price**
- **options code**

8.3.2 Data Attributes

Data attributes define the properties of a data object and take one of three different characteristics. They can be used to:

1. Name an instance of the data object.
2. Describe the instance, or
3. Make reference to another instance in another table.

In addition, one or more of the attributes, must be defined as an identifier, i.e., the identifier attribute becomes a "key" when we want to find an instance of the data object. Values for the identifier(s) are unique, although this is not a requirement.
Referring to the data object **car**, a reasonable identifier might be the ID number.

8.3.3 Relationships

Indicates "connectedness"; a "fact" that must be "remembered" by the system and cannot or is not computed or derived mechanically

- several instances of a relationship can exist
- objects can be related in many different ways

We can define a set of object/relationship pairs that define the relevant relationships. For example:

- A person *owns* a car.
- A person is *insured to drive* a car.

The relationship *owns* and *insured to drive* define the relevant connections between **person** and **car**.

8.4 Object-Oriented Analysis

The intent of Object Oriented Analysis (OOA) is to define all classes (and the relationships and behavior associated with them that are relevant to the problem to be solved.
To accomplish this, a number of tasks must occur:

1. Basic user requirements must be communicated between the customer and the software engineer.
2. Classes must be defined.
3. A class hierarchy is defined
4. Object-to-object relationships should be represented.
5. Object behavior must be modeled.
6. 1 – 5 are repeated iteratively until the model is complete.

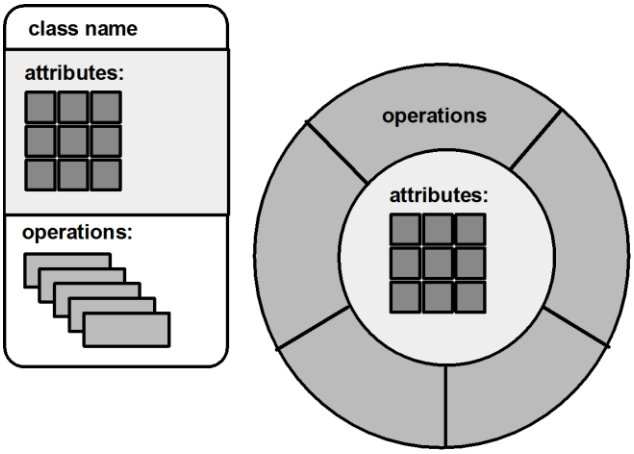
OOA builds a class-oriented model that relies on an understanding of OO concepts.

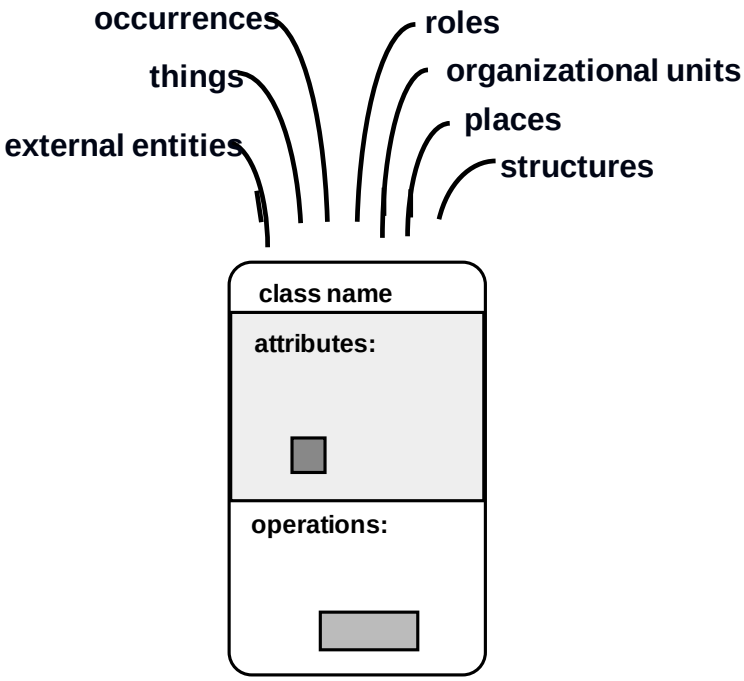
- Classes and objects
- Attributes and operations
- Encapsulation and instantiation
- Inheritance

Object-Oriented thinking begins with the definition of a class, often defined as:

- template
- generalized description
- "blueprint" ... describing a collection of similar items
- a metaclass (also called a superclass) establishes a hierarchy of classes once a class of items is defined, a specific instance of the class can be identified.

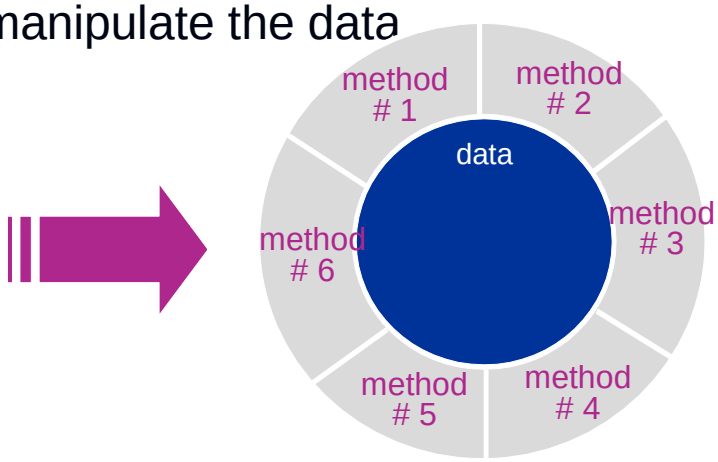
Building a class



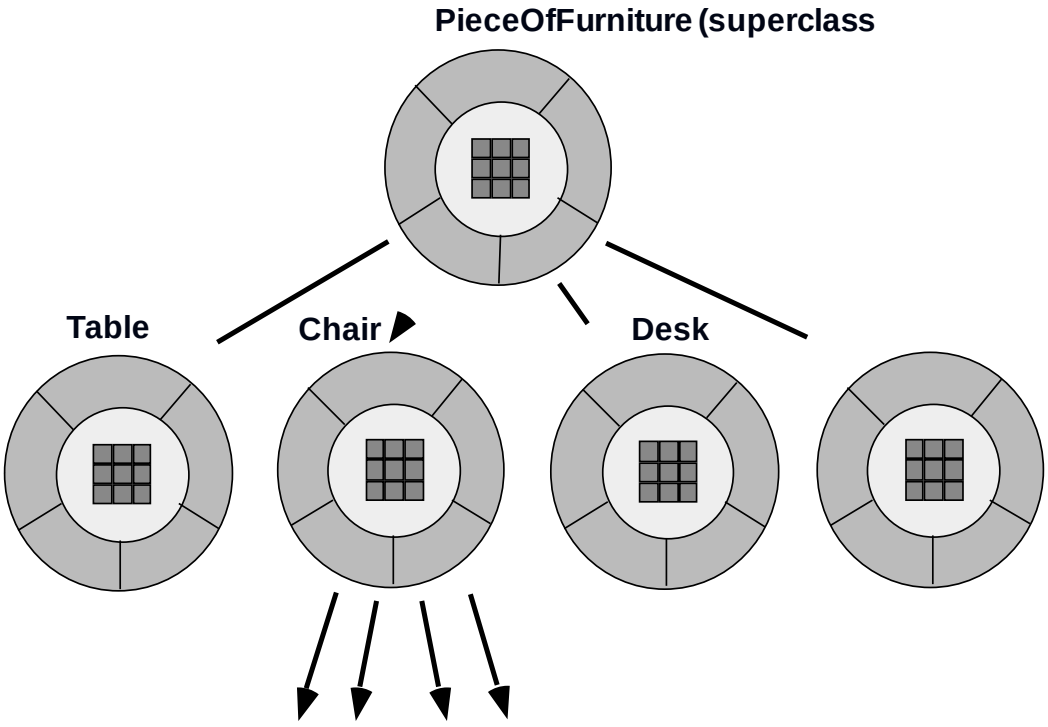


Encapsulating and Hiding

The object encapsulates both data and the logical procedures required To manipulate the data



Class Hierarchy



Methods (a.k.a. Operations, Services)

An executable procedure that is encapsulated in a class and is designed to operate on one or more data attributes that are defined as part of the class.
A method is invoked via message passing.

8.5 Scenario-Based Modeling

"[Use-cases] are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use-cases)." Ivar Jacobson

The concept is relatively easy to understand- describe a specific usage scenario in straightforward language from the point of view of a defined actor.

8.5.1 Writing Use-Cases

(1) What should we write about?
Inception and elicitation provide us the information we need to begin writing use cases.
(2) How much should we write about it?
(3) How detailed should we make our description?
(4) How should we organize the description?

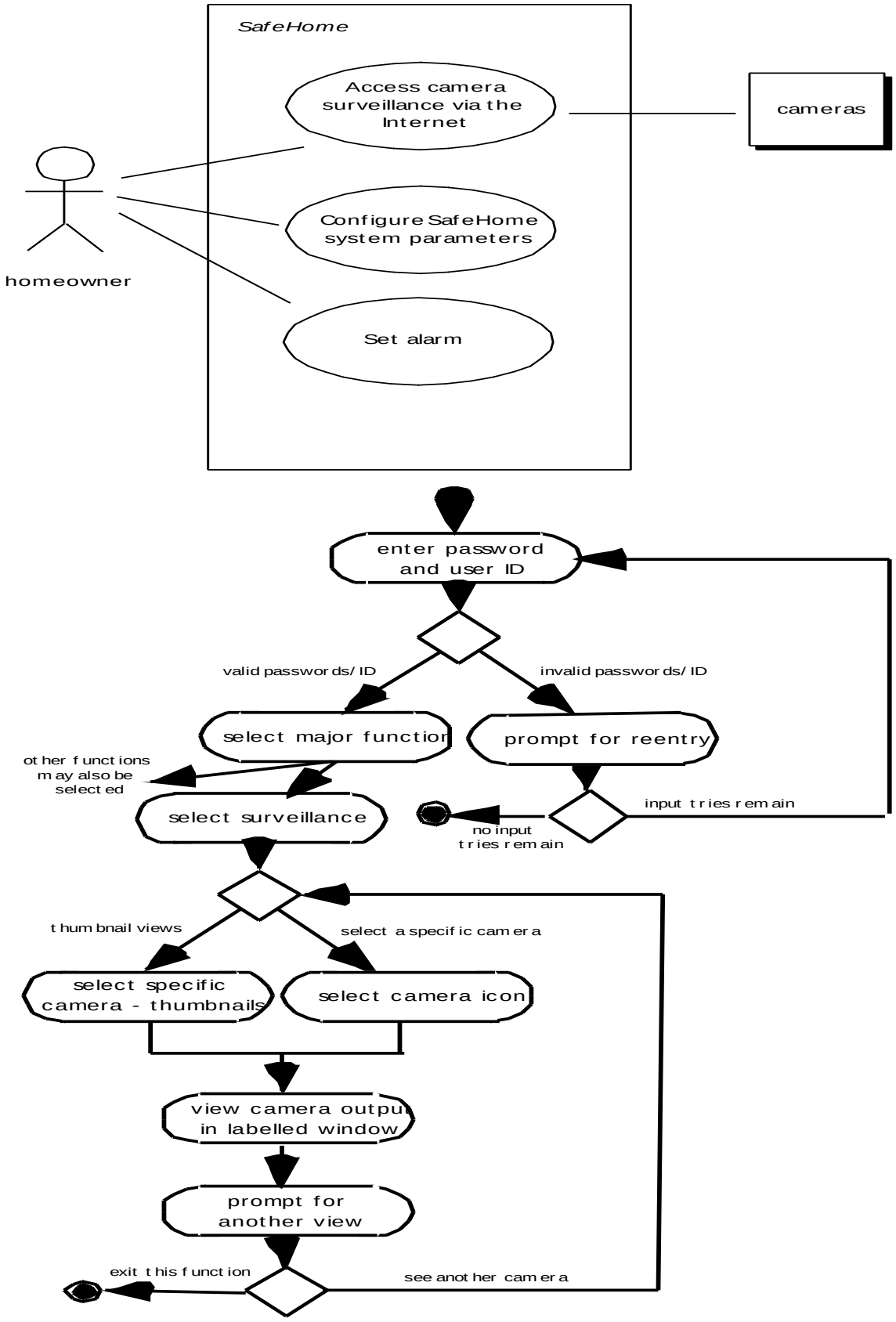
Use Cases:

- A scenario that describes a "thread of usage" for a system.
- ***Actors*** represent roles people or devices play as the system functions.
- ***Users*** can play a number of different roles for a given scenario.

Quality **F**unction **D**eployment and other R.E. mechanisms are used to identify stakeholders, define the scope of the problem, specify overall operational goals, outline all known functional requirements, and describe the object that will be manipulated by the system.

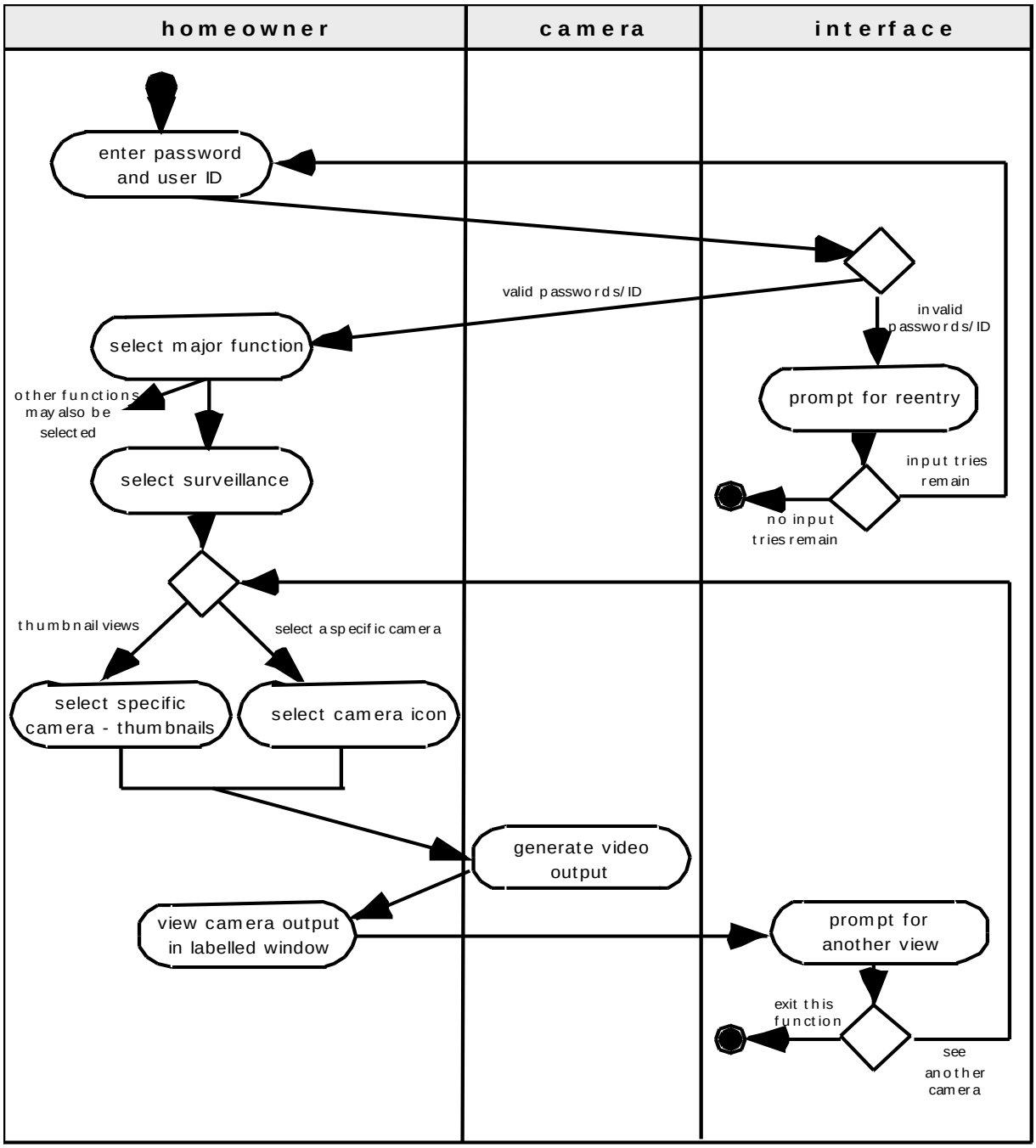
8.5.2 Developing an Activity Diagram

- What are the main tasks or functions that are performed by the actor?
- What system information will the actor acquire, produce or change?
- Will the actor have to inform the system about changes in the external environment?
- What information does the actor desire from the system?
- Does the actor wish to be informed about unexpected changes?


SafeHome
Access camera surveillance via the Internet
cameras
homeowner
Configure SafeHome system parameters
Set alarm
enter password and user ID
valid passwords/ ID
invalid passwords/ ID
select major function
prompt for reentry
other functions may also be selected
input tries remain
select surveillance
no input tries remain
thumbnail views
select a specific camera
select specific camera - thumbnails
select camera icon
view camera output in labelled window
prompt for another view
exit this function
see another camera

8.5.3 Swimlane Diagrams

The UML *swimlane diagram* is a useful variation of the activity diagram and allows the modeler to represent the flow of activities described by the user-case and at the same time indicate which actor or analysis class has responsibility for the action described by an activity rectangle.
Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.



8.6 Flow-Oriented Modeling

Represents how data objects are transformed as they move through the system.
A *data flow diagram* (DFD) is the diagrammatic form that is used to complement UML diagrams.
Considered by many to be an 'old school' approach, flow-oriented modeling continues to provide a view of the system that is unique.
The DFD takes an input-process-output insight into system requirements and flow.
Data objects are represented by labeled arrows and transformations are represented by circles (called *bubbles)*.

8.6.1 Creating a Data Flow Model

The DFD diagram enables the software engineer to develop models of the information domain and functional domain at the same time.
As the DFD is refined into greater levels of detail, the analyst performs an implicit functional decomposition of the system.
Guidelines:

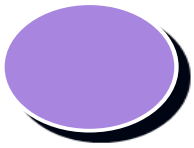
1. The level 0 data DFD should depict the software/system as a single bubble.
2. Primary I/O should be carefully noted.
3. Refinement should begin by isolating candidate processes, data objects, and data stores.
4. All arrows and bubbles should be labeled with meaningful names.
5. Information flow continuity must be maintained from level to level.
6. One bubble at one time should be refined.

Information continuity must be maintained at each level as DFD level is refined. This mean that input and output at one level must be the same as input and output at a refined level. Figure 8.10 and 8.11 show how DFD works.

The flow Model

Flow Modeling Notations

external entity



process

data flow

data store

External Entity

A producer or consumer of data

Example: computer-based system

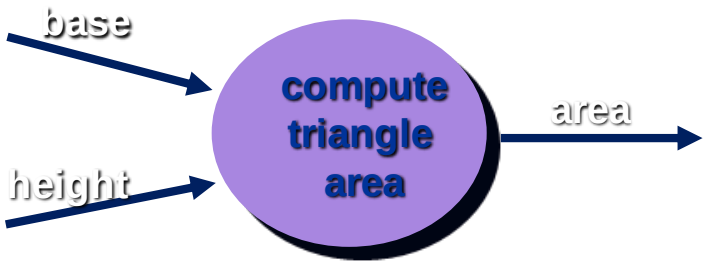
Data must always originate somewhere and must always be sent to something

Process

A data transformer (changes input to output)
Examples: compute taxes, determine area, format report, display graph
Data must always be processed in some way to achieve system function

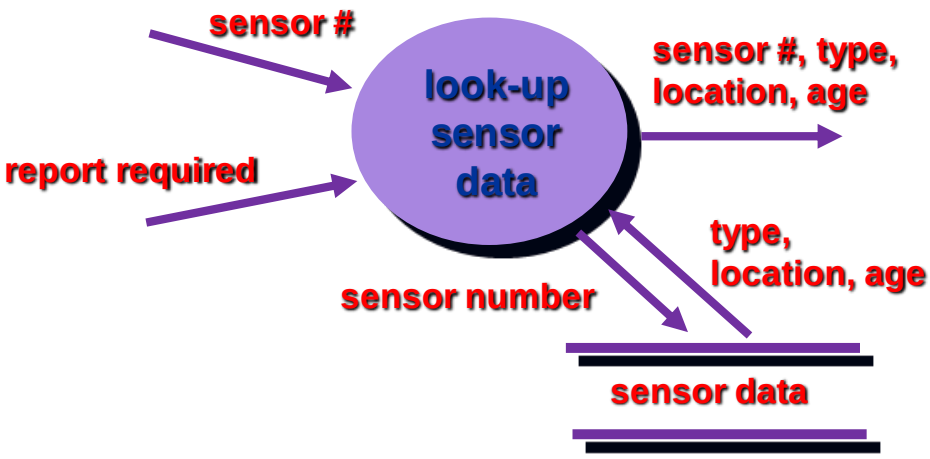
Data Flow

Data flows through a system, beginning as input and be transformed into output.



Data Stores

Data is often stored for later use.

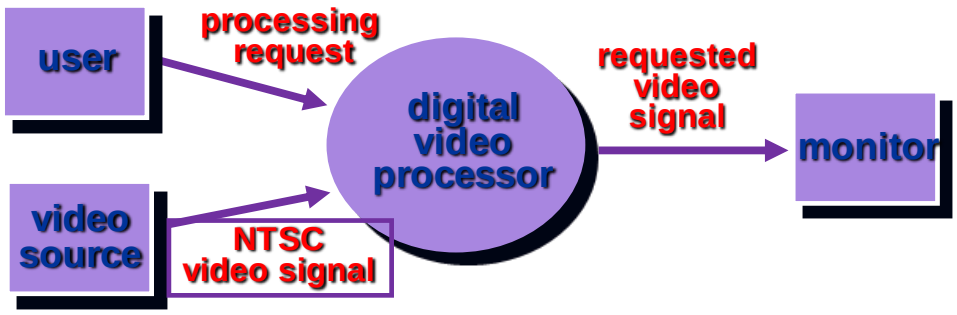

sensor #
look-up sensor data
sensor #, type, location, age
report required
type, location, age
sensor number
sensor data

Data Flow Diagramming:

Constructing a DFD—I

- Review the data model to isolate data objects and use a grammatical parse to determine "operations"
- Determine external entities (producers and consumers of data)
- Create a level 0 DFD

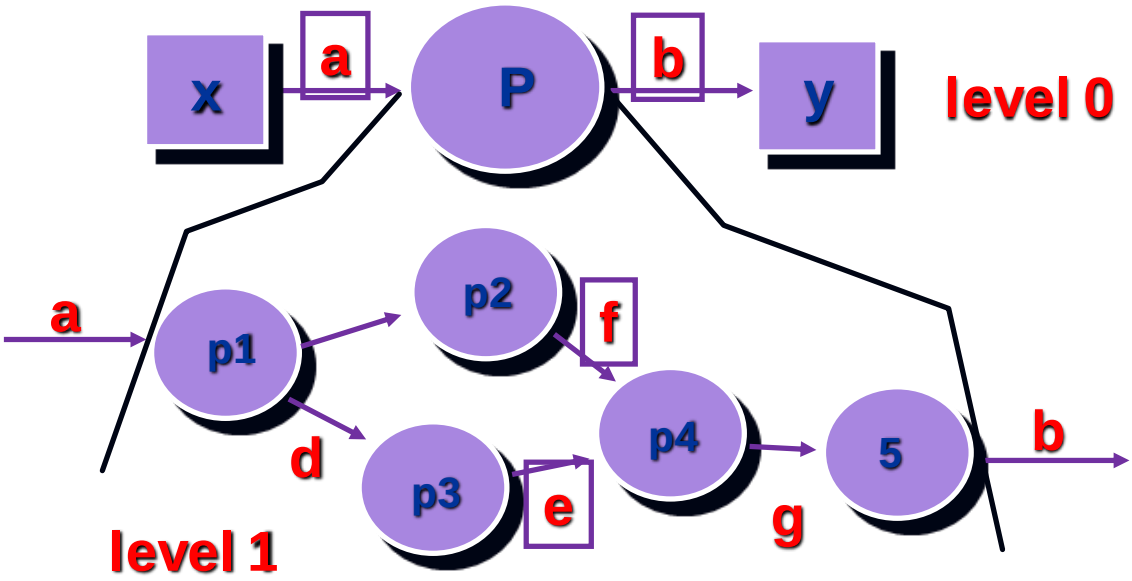
Level 0 DFD Example



Constructing a DFD—II

- Write a narrative describing the transform
- Parse to determine next level transforms
- "Balance" the flow to maintain data flow continuity
- Develop a level 1 DFD
- Use a 1:5 (approx.) expansion ratio

The Data Flow Hierarchy

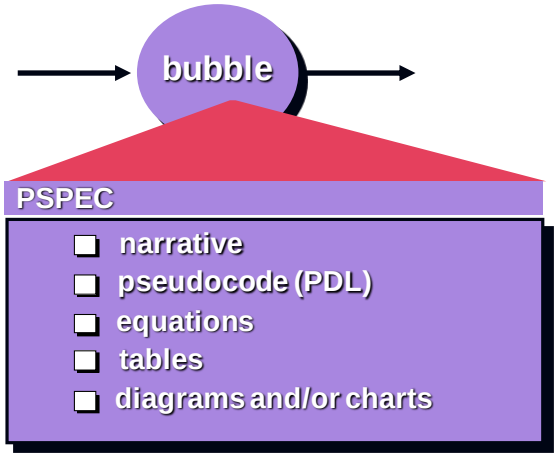


Flow Modeling Notes

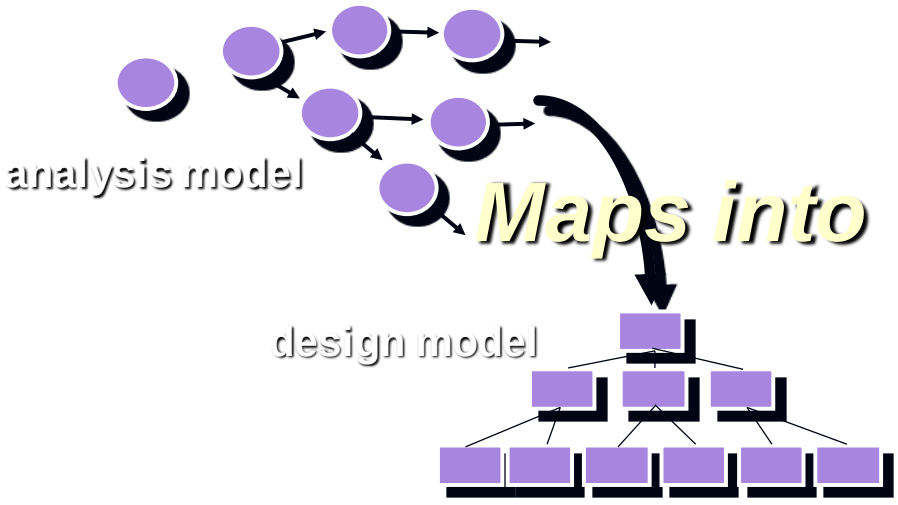
- Each bubble is refined until it does just one thing
- The expansion ratio decreases as the number of levels increase
- Most systems require between 3 and 7 levels for an adequate flow model
- A single data flow item (arrow) may be expanded as levels increase (data dictionary provides information)

8.6.4 The Process Specification

The *Process Specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. It is a "mini" specification for each transform at the lowest refined of a DFD.



DFDs: A Look Ahead



Control Flow Diagrams

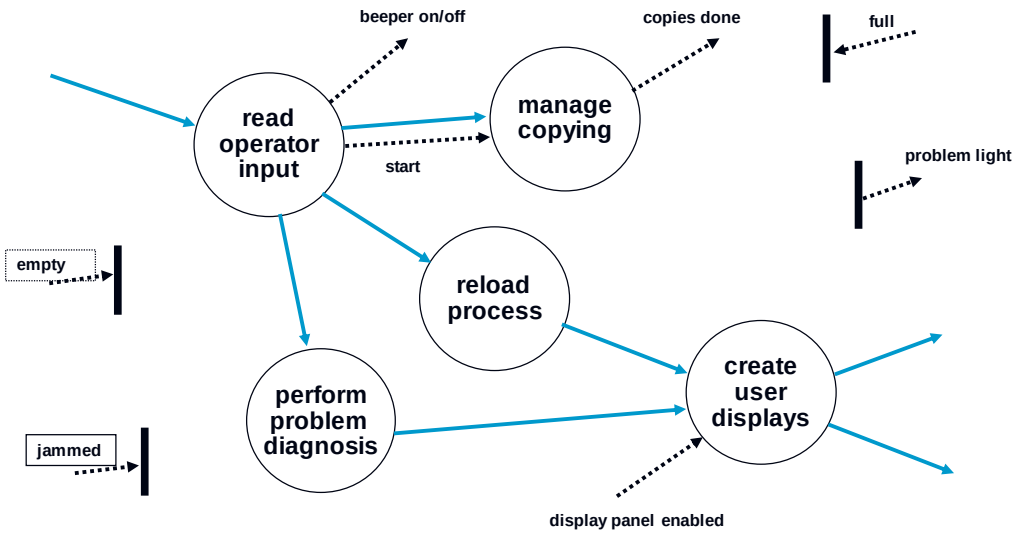
The diagram represents “events” and the processes that manage these events.
An “event” is a Boolean condition that can be ascertained by:

- Listing all sensors that are "read" by the software.
- Listing all interrupt conditions.
- Listing all "switches" that are actuated by an operator.
- Listing all data conditions.
- Recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible CSPEC inputs/outputs.

The Control Model

- The control flow diagram is "superimposed" on the DFD and shows events that control the processes noted in the DFD.
- Control flows—events and control items—are noted by dashed arrows.
- A vertical bar implies an input to or output from a control spec (CSPEC) — a separate specification that describes how control is handled.
- A dashed arrow entering a vertical bar is an input to the CSPEC
- A dashed arrow leaving a process implies a data condition.
- A dashed arrow entering a process implies a control input read directly by the process.
- Control flows do not physically activate/deactivate the processes—this is done via the CSPEC.

Control Flow Diagram



Control Specification (CSPEC)

The CSPEC can be:

- ❑ state diagram (sequential spec)
- ❑ state transition table
- ❑ decision tables
- ❑ activation tables

} combinatorial spec

Guidelines for Building a CSPEC

- List all sensors that are "read" by the software.
- List all interrupt conditions.
- List all "switches" that are actuated by the operator.
- List all data conditions.
- Recalling the noun-verb parse that was applied to the software statement of scope, review all "control items" as possible CSPEC inputs/outputs.
- Describe the behavior of a system by identifying its states; identify how each state is reach and defines the transitions between states.
- Focus on possible omissions ... a very common error in specifying control, e.g., ask: "Is there any other way I can get to this state or exit from it?"

8.7 Class-Based Modeling

This section describes the process of developing an object-oriented analysis (OOA) model. The generic process described begins with guidelines for identifying potential analysis classes, suggestions for defining attributes and operations for those classes, and a discussion of the Class-Responsibility-

Collaborator (CRC) model. The CRC card is used as the basis for developing a network of objects that comprise the object-relationship model.

8.7.1 Identifying Analysis Classes

- Identify analysis classes by examining the problem statement
- Use a "grammatical parse" to isolate potential classes
- Identify the attributes of each class
- Identify operations that manipulate the attributes

Analysis Classes manifest themselves in one of the following ways:

- *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- *Things* (e.g., reports, displays, letters, signals) that are part of the information domain for the problem.
- *Occurrences or events* (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, and team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.

Performing a "grammatical parse" on a processing narrative for a problem helps extracting the nouns. After identifying the nouns, a number of potential classes are proposed in a list. The list will be continued until all nouns in the processing narratives have been considered. Each entry is in the list is a potential object.

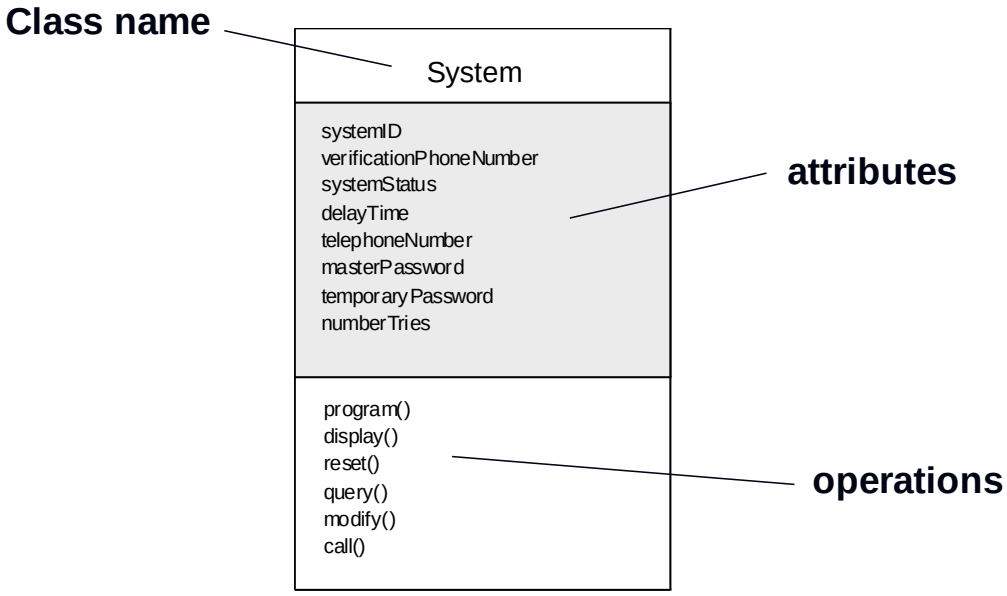
How do I determine whether a potential class should, in fact, become an analysis class?

1. *Retained Information*: The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
2. *Needed Services*: The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
3. *Multiple attributes*: During R.A., the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
4. *Common attributes*: a set of attributes can be defined for the potential class, and these attributes apply to all instances of the class.
5. *Common operations*: a set of operations can be defined for the potential class, and these operations apply to all instances of the class.
6. *Essential Requirements*: External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as classes in the requirement model.

To be considered a legitimate class for inclusion in the requirements model, a potential class should satisfy all of these characteristics.

8.7.2 Specifying Attributes

Attributes are set of data objects that fully define the class within the context of the problem space. To develop a meaningful set of attributes for an analysis class, a software engineer can study a use-case and select those "things" that "reasonably" belong to the class. An important question that should be answered for each class: what data items fully define the class in the context of the problem at hand.



Example above is shown in figure.

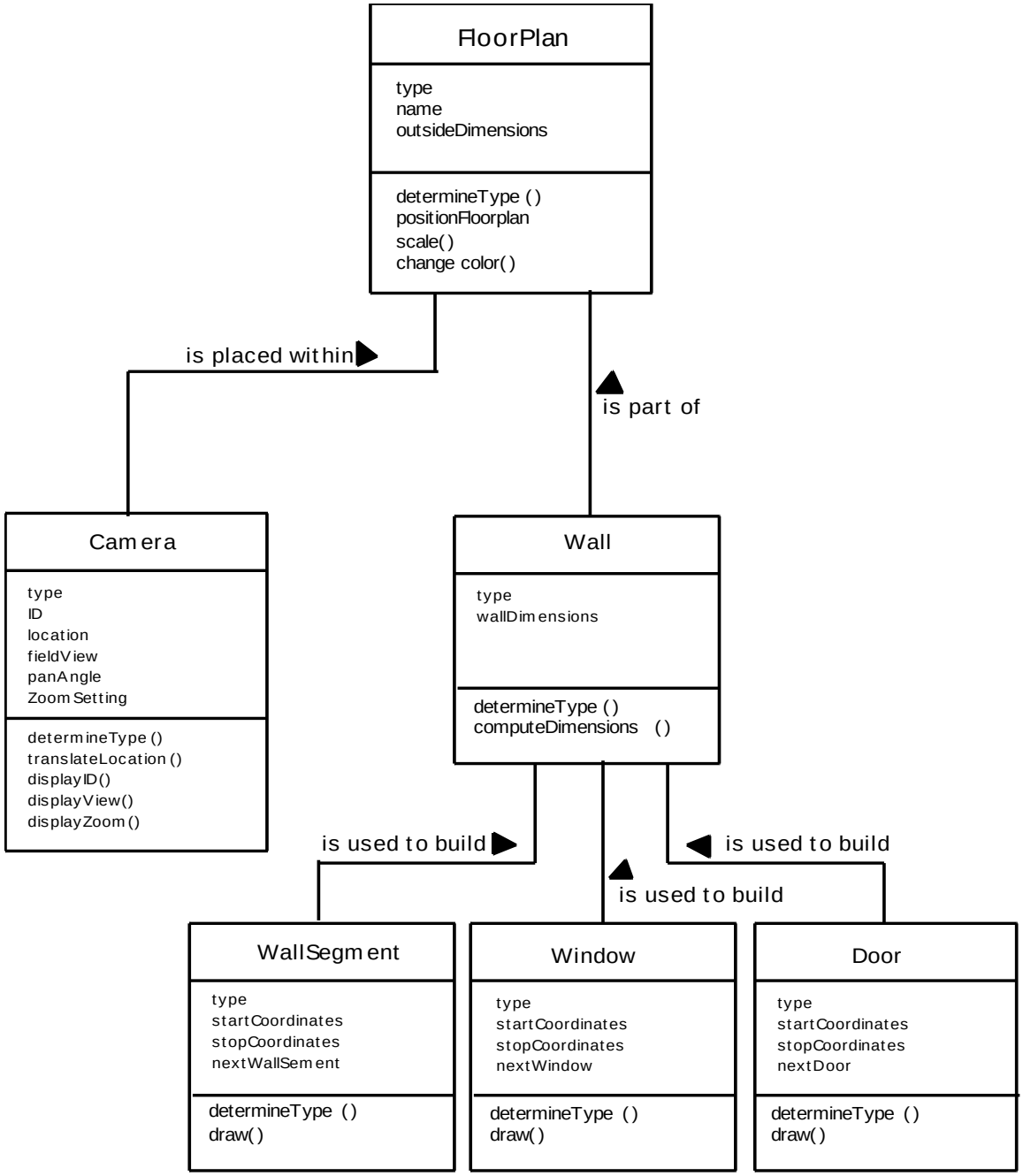
8.7.3 Defining Operations

Operations define the behavior of an object divided into 4 broad categories:

1. Operations that manipulate data (adding, deleting, selecting, reformatting.)
2. Operations that perform a computation.

3. Operations that inquire about the state of an object.
4. Operations that monitor an object for the occurrence of a controlling event.

Class Diagram

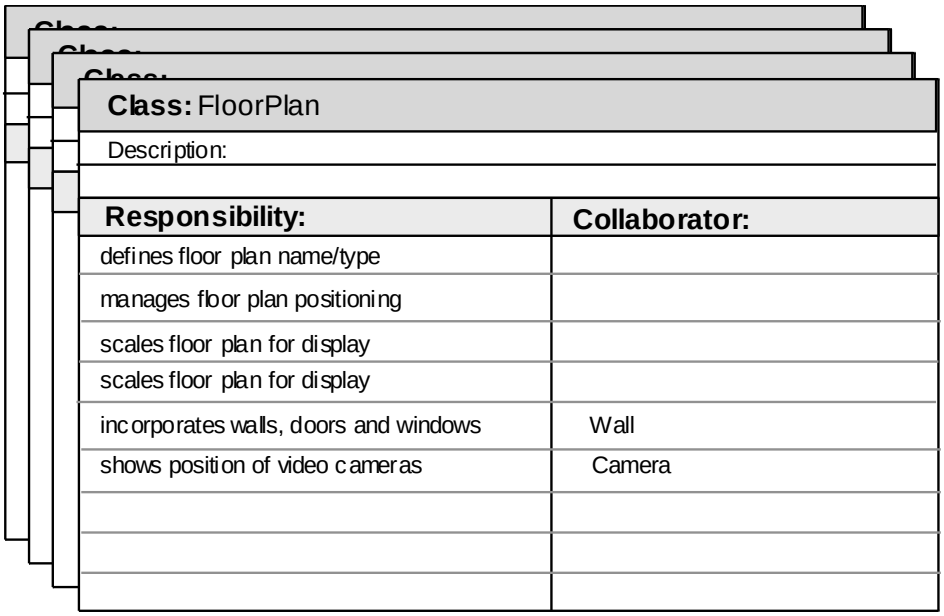


8.7.4 Class Responsibility Collaborator (CRC) Modeling

Class-Responsibility-Collaborator (CRC) Modeling provides a simple means for identifying and organizing the classes that are relevant to system or product requirement.

CRC modeling is described as follows:
"A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right."

Responsibilities are the attributes and operations that are relevant for the class. "Anything the class knows or does."
Collaborators are those classes that are required to provide a class with the information needed to complete a responsibility.
In general, collaboration implies either a request for information or a request for some action.



Classes: The taxonomy of class types can be extended by considering the following categories:

- *Entity classes*, also called *model* or *business* classes, are extracted directly from the statement of the problem (e.g., FloorPlan and Sensor).
- *Boundary classes* are used to create the interface (e.g., interactive screen or printed reports) that the user sees and interacts with as the software is used.
- *Controller classes* manage a "unit of work" [UML03] from start to finish. That is, controller classes can be designed to manage
    - the creation or update of entity objects;
    - the instantiation of boundary objects as they obtain information from entity objects;
    - complex communication between sets of objects;
    - Validation of data communicated between objects or between the user and the application.

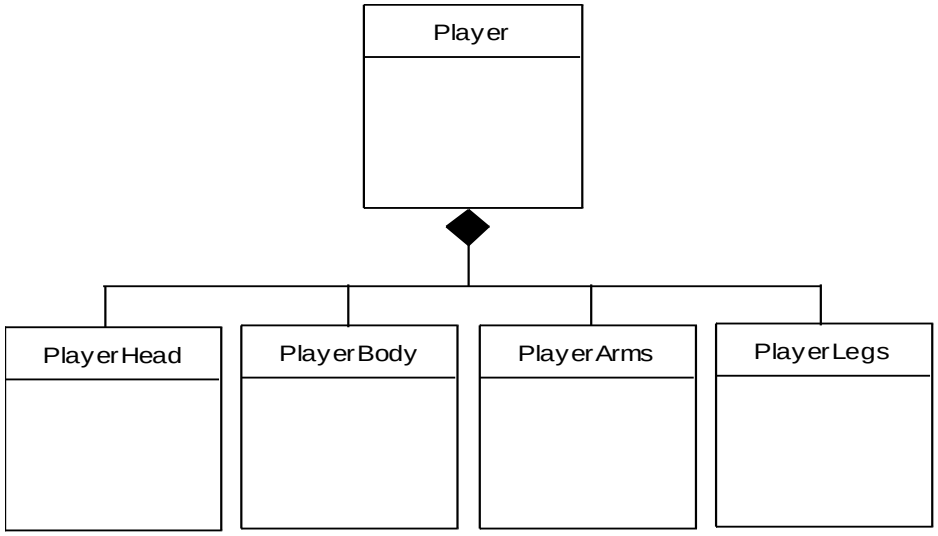
Responsibilities

1. System intelligence should be distributed across classes to best address the needs of the problem
2. Each responsibility should be stated as generally as possible

3. Information and the behavior related to it should reside within the same class
4. Information about one thing should be localized with a single class, not distributed across multiple classes.
5. Responsibilities should be shared among related classes, when appropriate.

Collaborations

- Classes fulfill their responsibilities in one of two ways:
    1. A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
    2. A class can collaborate with other classes.
- Collaborations identify relationships between classes
- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- Three different generic relationships between classes [WIR90]:
    1. the *is-part-of* relationship
    2. the *has-knowledge-of* relationship
    3. the *depends-upon* relationship

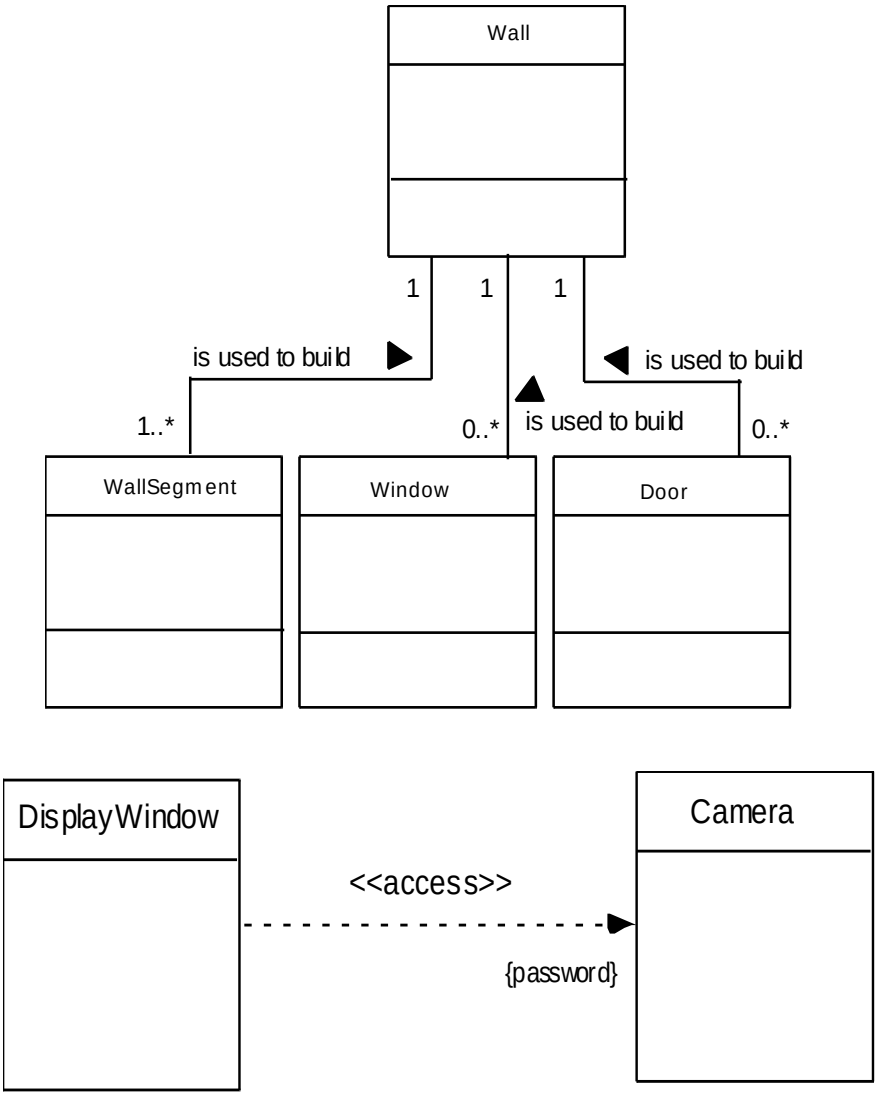


Reviewing the CRC Model

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use-case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
4. When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use-case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

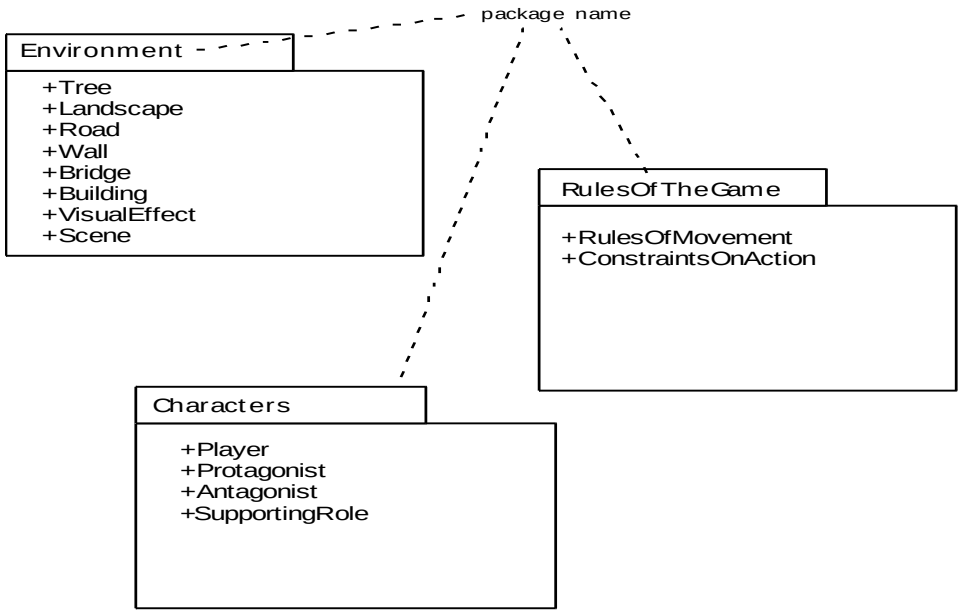
8.7.5 Associations and Dependencies

- Two analysis classes are often related to one another in some fashion
  - In UML these relationships are called *associations*
  - Associations can be refined by indicating *multiplicity* (the term *cardinality* is used in data modeling)
- In many instances, a client-server relationship exists between two analysis classes.
  - In such cases, a client-class depends on the server-class in some way and a *dependency relationship* is established



8.7.6 Analysis Packages

- Various elements of the analysis model (e.g., use-cases, analysis classes) are categorized in a manner that packages them as a grouping
- The plus sign preceding the analysis class name in each package indicates that the classes have public visibility and are therefore accessible from other packages.
- Other symbols can precede an element within a package. A minus sign indicates that an element is hidden from all other packages and a # symbol indicates that an element is accessible only to packages contained within a given package.



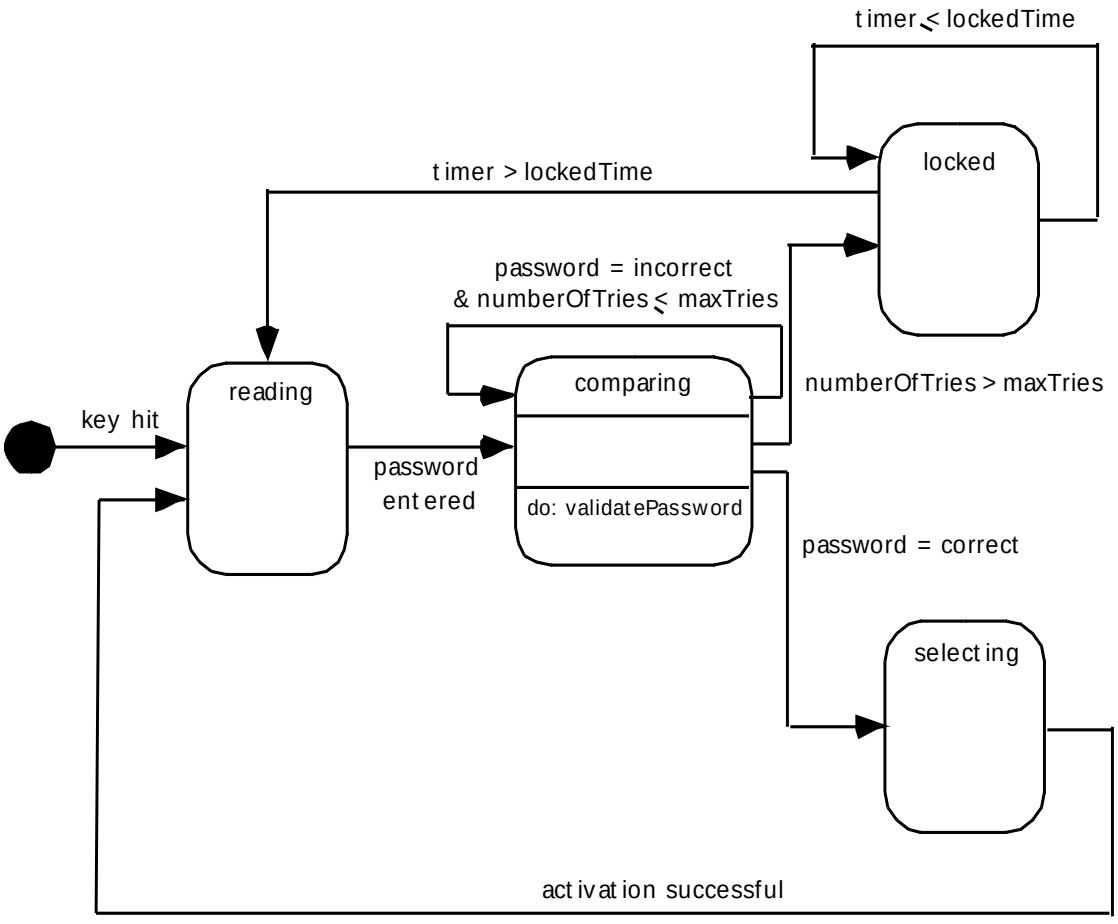
8.8 Creating a Behavioral Model

- The behavioral model indicates how software will respond to external events or stimuli. To create the model, the analyst must perform the following steps:
    1. Evaluate all use-cases to fully understand the sequence of interaction within the system.
    2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
    3. Create a sequence for each use-case.
    4. Build a state diagram for the system.
    5. Review the behavioral model to verify accuracy and consistency.

State Representations

- In the context of behavioral modeling, two different characterizations of states must be considered:
  - the state of each class as the system performs its function and
  - the state of the system as observed from the outside as the system performs its function
- The state of a class takes on both passive and active characteristics [CHA93].
  - A *passive state* is simply the current status of all of an object's attributes.

The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing.



The States of a System

- State—a set of observable circumstances that characterizes the behavior of a system at a given time
- State transition—the movement from one state to another
- Event—an occurrence that causes the system to exhibit some predictable form of behavior

- Action—process that occurs as a consequence of making a transition

Writing the Software Specification

Specification Guidelines

- use a layered format that provides increasing detail as the "layers" deepen
- use consistent graphical notation and apply textual terms consistently (stay away from aliases)
- be sure to define all acronyms
- be sure to include a table of contents; ideally, include an index and/or a glossary
- write in a simple, unambiguous style (see "editing suggestions" on the following pages)
- always put yourself in the reader's position, "Would I be able to understand this if I wasn't intimately familiar with the system?"

Be on the lookout for persuasive connectors, ask why?
keys: *certainly, therefore, clearly, obviously, it follows that ...*

Watch out for vague terms
keys: *some, sometimes, often, usually,ordinarily, most, mostly ...*

When lists are given, but not completed, be sure all items are understood
keys: *etc., and so forth, and so on, such as*

Be sure stated ranges don't contain unstated assumptions
e.g., *Valid codes range from 10 to 100.* Integer? Real? Hex?

Beware of vague verbs such as *handled, rejected, processed, ...*

Beware "passive voice" statements
e.g., *The parameters are initialized.* By what?

Beware "dangling" pronouns
e.g., *The I/O module communicated with the data validation module and its contol flag is set.* Whose control flag?

DESIGN ENGINEERING

- The goal of design engineering is to produce a model or representation that exhibits firmness, commodity, and delight. To accomplish this, a designer must practice diversification and then convergence.

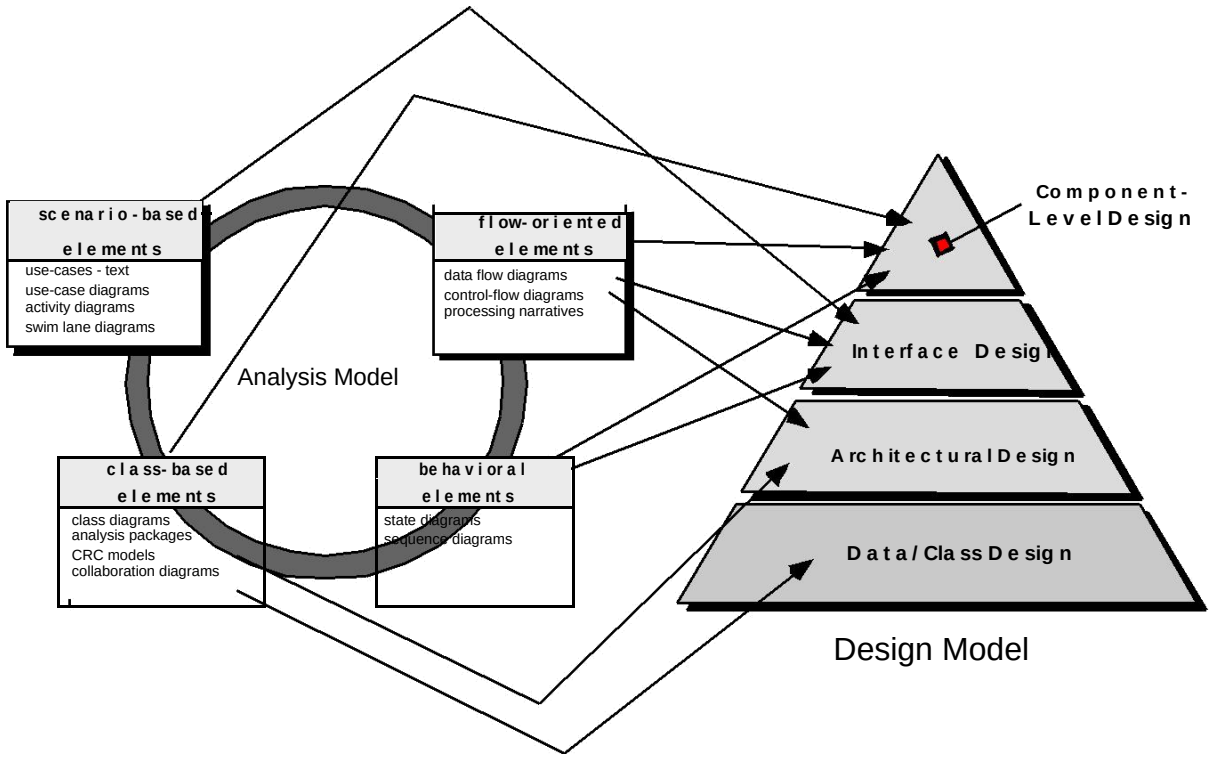
- Diversification and Convergence - the qualities which demand intuition and judgment are based on experience.

    - Principles and heuristics that guide the way the model is evolved.
    - Set of criteria that enables quality to be judge.
    - Process of iteration that ultimately leads to final design representation.

- Design engineering for computer software changes continually as new methods, better analysis, and broader understanding evolve. Even today, most software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines.

From Analysis Model To Design Model:

- Each element of the analysis model provides information that is necessary to create the four design models
    - The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
    - The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
    - The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
    - The component-level design transforms structural elements of the software architecture into a procedural description of software components



DESIGN PROCESS AND DESIGN QUALITY

- *The software design is an iterative process through which requirements are translated into a **blueprint** for constructing the software.*

- Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design. Three characteristics that serve as a guide for the evaluation of a good design.

    - The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.

    - The design must be a readable, understandable guide for those how generate code and for those who test and subsequently support the software.

    - The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains form an implementation perspective.

■ **Quality Guidelines**

In order to evaluate the quality of a design representation, we must establish technical criteria for good design.

- A design should exhibit an architecture that

  (a) has been created using recognizable architectural styles or patterns. (b) is composed of components that exhibit good design characteristics. (c) can be implemented in an evolutionary fashion, thereby facilitating
  implementation and testing.
- A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
- A design should contain distinct representations of data, architecture, interfaces, and components.
- A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
- A design should lead to components that exhibit independent functional characteristics.
- A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
- A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
- A design should be represented using a notation that effectively communicates its meaning.

■ **Quality Attributes**

Hewlett Packard developed set of software quality attributes name FURPS. Functionality, Usability, Reliability, Performance and Supportability. The FURPS quality attributes represent a target for all software design.

- *Functionality* is assessed by evaluating the feature set and capabilities of the program, the generality of the functions that are delivered, and the security of the overall system.
- *Usability* is assessed by considering human factors, overall aesthetics, consistency and documentation.
- *Reliability* is evaluated by measuring the frequency and severity of failure, the accuracy of output results, the mean-time-to-failure (MTTF), the ability to recover from failure and the predictability of the program.
- *Performance* is measuring by processing speed, response time, resource consumption, throughput and efficiency.
- *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability-these three attributes represent amore common term, maintainability-in addition, testability, compatibility, and configurability.

DESIGN CONCEPTS

Fundamental software design concepts provide the necessary frame work for ―getting it right.‖

- **Abstraction -** Many levels of abstraction can be posed.
  - The highest level states the solution in broad terms using the language of the problem environment.
  - The lower level gives more detailed description of the solution.
  - *Procedural Abstraction* refers to a sequence of instruction that have specific and limited functions.
  - *Data Abstraction* is a named collection of data that describes a data object.

- **Architecture** – provides overall structure of the software and the ways in which the structure provides conceptual integrity for a system. The goal of software design is to derive and architectural rendering of a system which serves as a framework from which more detailed design activities can be conducted.
  - The Architectural design is represented by the following models
    - *Structural Model* – Organized collection of program components.
    - *Framework Model* – Increases level of design abstraction by identifying repeatable architectural design frameworks that are encountered in similar types of applications.
    - *Dynamic Model* – Addresses the behavioral aspects of the program architecture.
    - *Process Model* – Focuses on design of business or technical process that system must accommodate.
    - *Functional Model* – Used to represent functional hierarchy of the system.

- **Patterns** – A pattern is a named nugget of insight which conveys the essences of proven solutions to a recurring problem with a certain context amidst competing concerns. The design pattern provides a description that enables a designer to determine
  - Whether the pattern is applicable to the current work?
  - Whether the pattern can be reused?
  - Whether the pattern can serve as a guide for developing a similar but functionally or structurally different pattern?

- **Modularity** – the software is divided into separately named and addressable components called modules that are integrated to satisfy problem requirements.

- **Information Hiding** – modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

Information hiding provides greatest benefits when modifications are required during testing and software maintenance.

- **Functional Independence** – is achieved by developing modules with single minded function and an aversion to excessive interaction with other modules.
  Independence is assessed using two qualitative criteria

  - ***Cohesion*** – is a natural extension of information hiding concept, module performs single task, requiring little interaction with other components in other parts of a program.

  - ***Coupling*** – interconnection among modules and a software structures, depends on interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

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

- **Refactoring** – important design activity suggested for agile methods, refactoring is a recognization technique that simplifies design of a component without changing its function or behavior. ―Refactoring is a process of changing a software system in such away that it does not alter the external behavior of the code yet improves its internal structure.

- **Design Classes** – describes some element of problem domain, focusing on aspects of the problem that are user or customer visible. The software team must define a set of design classes that
  (1) Refine the analysis classes by providing design detail that will enable the classes to be implemented and
  (2) Create a new set of design classes that implement a software infrastructure to support the business solution. Five different types of design classes each representing a different layer of the design architecture is suggested

  - *User Interface classes* define all abstractions that are necessary for Human Computer Interaction (HCI). In many cases, HCI occurs within the context of a metaphor (e.g., a checkbook, an order form, a fax machine) and the design classes for the interface may be visual representations of the elements of the metaphor.

  - *Business domain classes* are often refinements of the analysis classes defined earlier. The classes identify the attributes and services (methods) that are required to implement some element of the business domain.

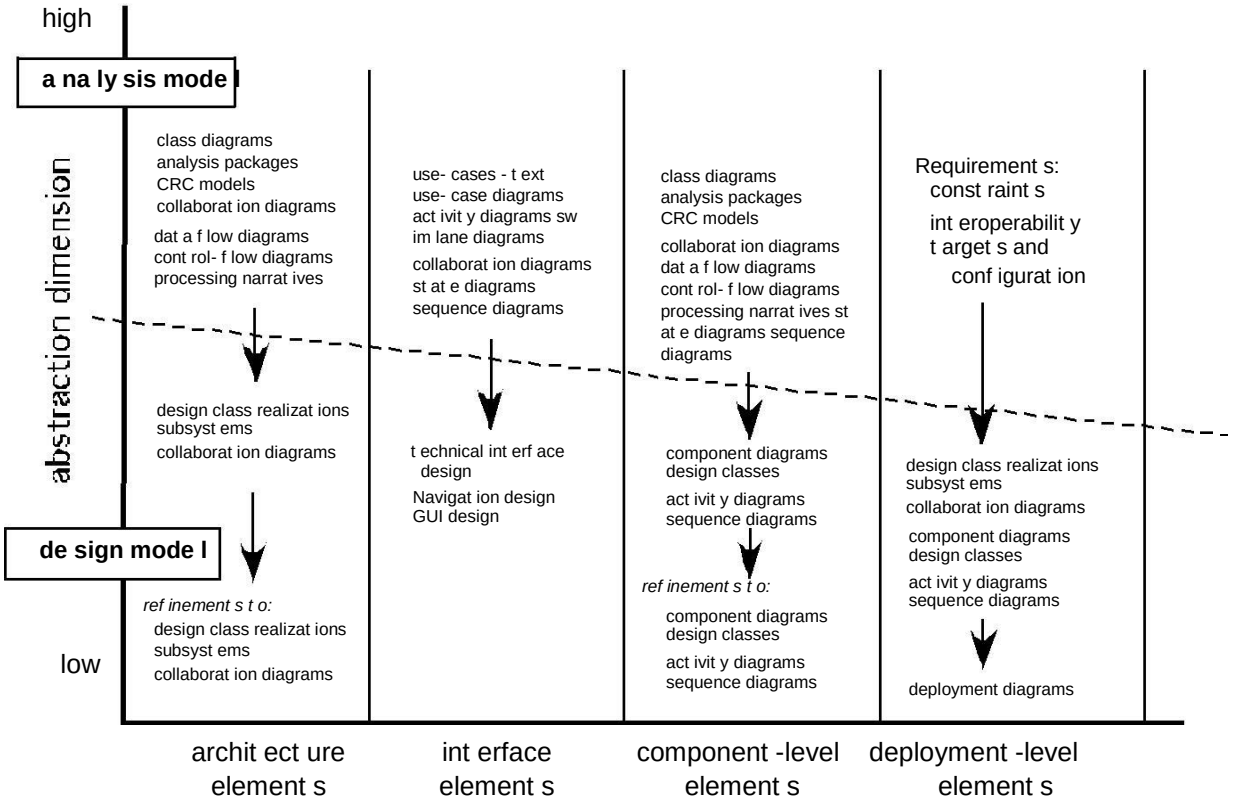
  - *Process classes* implement lower-level business abstractions required to fully manage the business domain classes.

  - *Persistent classes* represent data stores (e.g, a database) that will persist beyond the execution of the software.

- *System classes* implement software management and control functions that enable the system to operate and communicate within its computing environment and with the outside world.

THE DESIGN MODEL

- The design model can be viewed in two in two different dimensions. The process dimension indicates the evolution of the design model as design tasks are executed as part of the software process. The abstraction dimension represents the level of detail as each element of the analysis model is transformed into a design equivalent and then refined iteratively.



■ DATA DESIGN ELEMENTS

- Like other software engineering activities, data design creates a model of data and/or information that is represented at a high level of abstraction.
- At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications.

- At the application level, the translation of a data model into a database is pivotal to achieving the business objectives of a system.

- At the business level, the collection of information stored in disparate databases and reorganized into a ―data warehouse‖ enables data mining or knowledge discovery that can have an impact on the success of the business itself.

■ ARCHITECTURAL DESIGN ELEMENTS

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

- The architectural model is derived from three sources:

  (1) Information about the application domain for the software to be built;
  (2) Specific analysis model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand
  (3) the availability of architectural patterns and styles.

■ INTERFACE DESIGN ELEMENTS

- The interface design for software is the equivalent to a set of detailed drawing for the doors, windows and external utilities of a house. These drawings depict the size and shape of doors and windows, the manner in which they operate, the way in which utilities connections (e.g., water, electrical, gas, and telephone) come into the house and are distributed among the rooms depicted in the floor plan.

- There are three important elements of interface design:
  (1) the user interface (UI)
  (2) external interfaces to other systems, devices, networks, or other producers or consumers of information; and
  (3) internal interfaces between various design components. These interface design elements allow the software to communicate externally and enable internal communication and collaboration among the components that populate the software architecture.

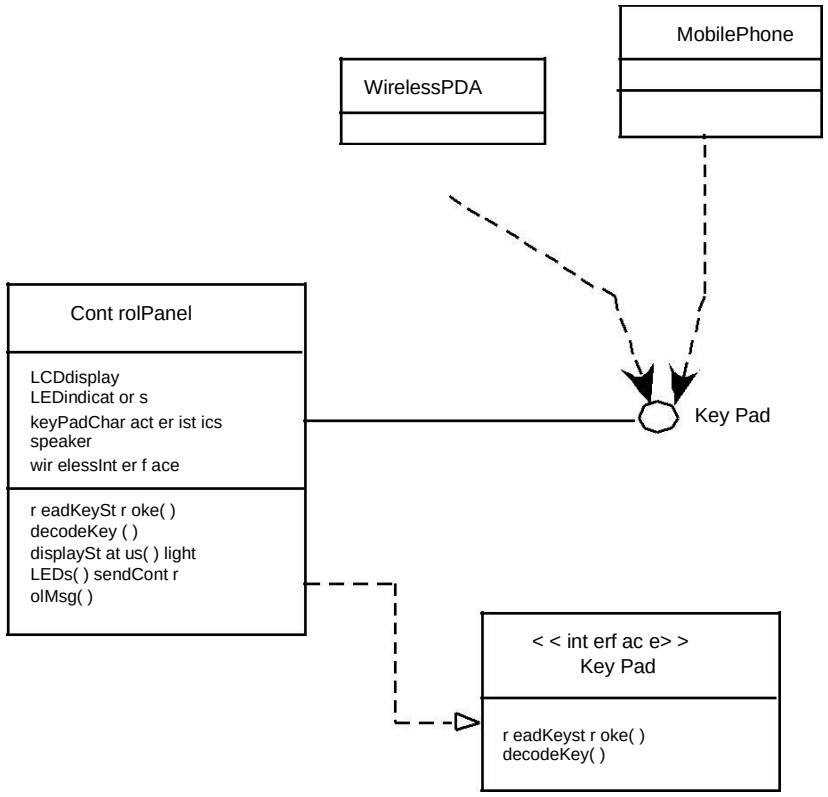
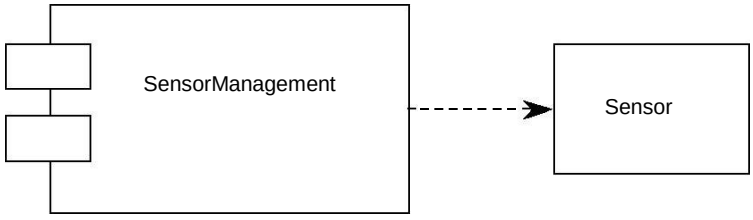


Figure 9 . 6 UML int erf ac e represent at ion f or **Co n t r o lPa n e l**

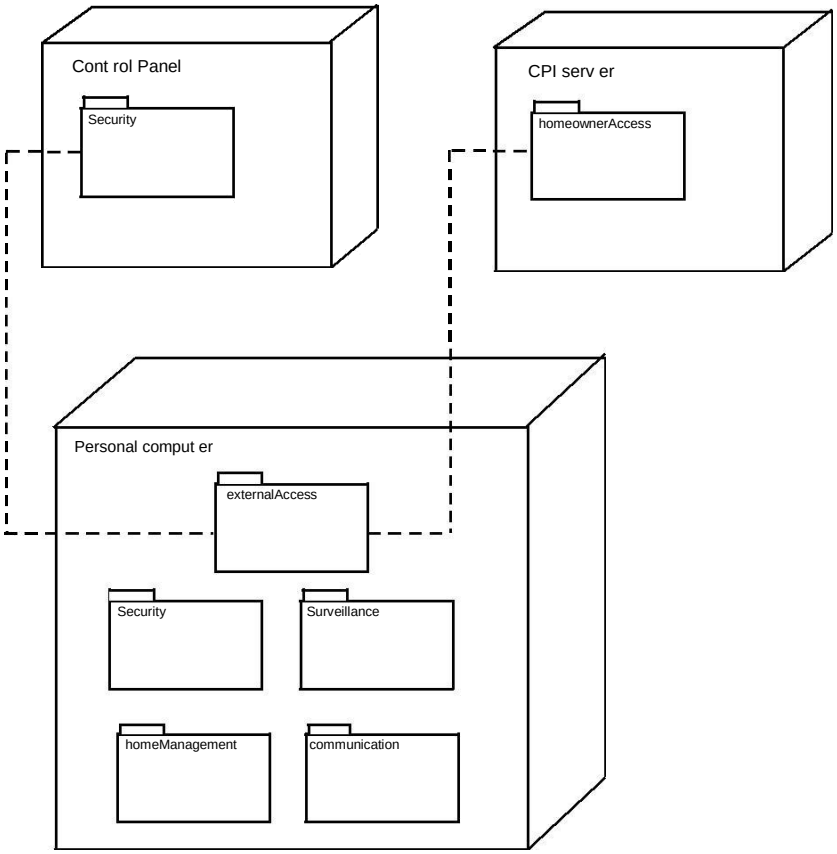
■ COMPONENT-LEVEL DESIGN ELEMENTS

- The component-level design for software is equivalent to a set of detailed drawings for each room in a house. These drawing depict wiring and plumbing within each room, the location of electrical receptacles and switches, faucets, sinks, showers, tubs, drains, cabinets, and closets. They also describe the flooring to be used, the moldings to be applied, and every other detail associated with a room. The component-level design for software fully describes the internal detail of each software component.



■ DEPLOYMENT-LEVEL DESIGN ELEMENTS

- Deployment level design elements indicates how software functionality and subsystems will be allocates within the physical computing environment that will support the software.



Pattern Based Software Design:

- Mature engineering disciplines make use of thousands of design patterns for such things as buildings, highways, electrical circuits, factories, weapon systems, vehicles, and computers
- Design patterns also serve a purpose in software engineering
- Architectural patterns
  - Define the overall structure of software
  - Indicate the relationships among subsystems and software components
  - Define the rules for specifying relationships among software elements
- Design patterns
  - Address a specific element of the design such as an aggregation of components or solve some design problem, relationships among components, or the mechanisms for effecting inter-component communication
  - Consist of creational, structural, and behavioral patterns
- Coding patterns
  - Describe language-specific patterns that implement an algorithmic or data structure element of a component, a specific interface protocol, or a mechanism for communication among components

CREATING AN ARCHITECTURAL DESIGN

- Design is an activity concerned with making major decisions, often of s structural nature.
- It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes.
- Design builds coherent, well-planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels.
    - What is Architectural design?
    - Who does it?
    - Why is it important?
    - What are the steps?
    - What is the work product?
    - How do I ensure that I have done it right?

SOFTWARE ARCHITECTURE

- Effective software architecture and its explicit representation and design have become dominant themes in software engineering.

■ **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.

■ **Why is Architecture Important?**

- 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 model of how the system is structured and how its components work together‖.

DATA DESIGN

The data design action translates data objects defined as part of the analysis model into data structures at the software component level and, when necessary, a database architecture at the application level.

■ **At the architectural level …**

- Design of one or more databases to support the application architecture
- Design of methods for _mining' the content of multiple databases
  - navigate through existing databases in an attempt to extract appropriate business-level information
  - Design of a data warehouse—a large, independent database that has access to the data that are stored in databases that serve the set of applications required by a business

■ **At the component level …**

- refine data objects and develop a set of data abstractions
- implement data object attributes as one or more data structures
- review data structures to ensure that appropriate relationships have been established
- simplify data structures as required
  - The systematic analysis principles applied to function and behavior should also be applied to data.

- All data structures and the operations to be performed on each should be identified.
- A data dictionary should be established and used to define both data and program design.
- Low level data design decisions should be deferred until late in the design process.
- The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.
- A library of useful data structures and the operations that may be applied to them should be developed.
- A software design and programming language should support the specification and realization of abstract data types.

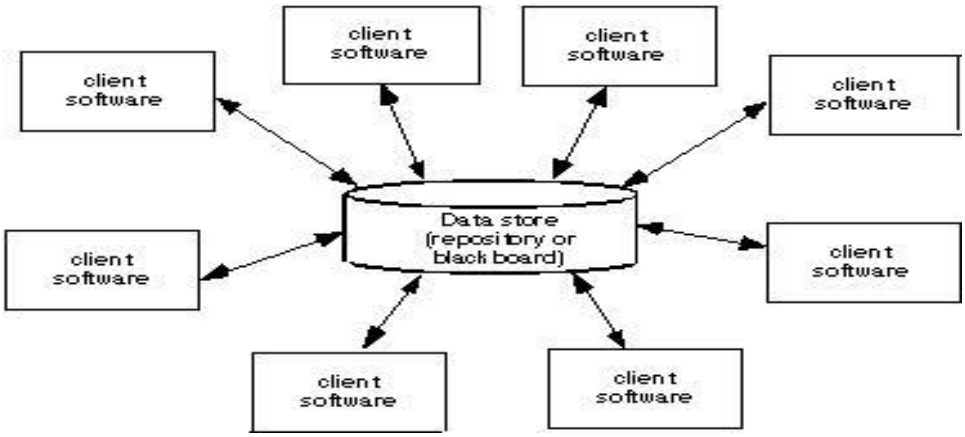
ARCHITECTURAL STYLES

- 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.

- Data-centered architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

■ **Data-Centered Architecture**

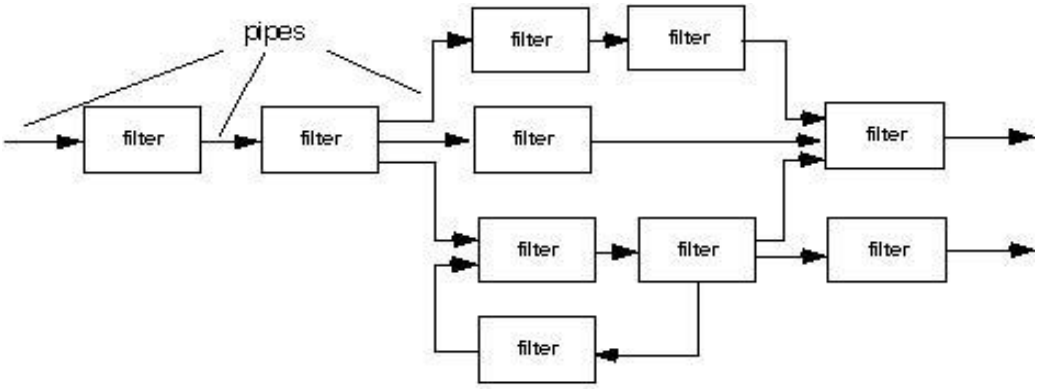
- A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. The following figure illustrates a typical data-centered style.



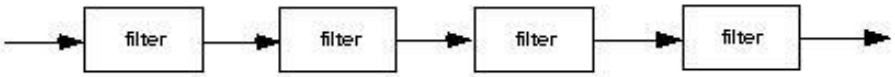
■

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 structure has a set of components, called filters, connected by pipes that transmit data from one component to the next.
- If the data flow degenerates into a single line of transforms, it is termed batch sequential.



(a) pipes and filters



(b) batch sequential

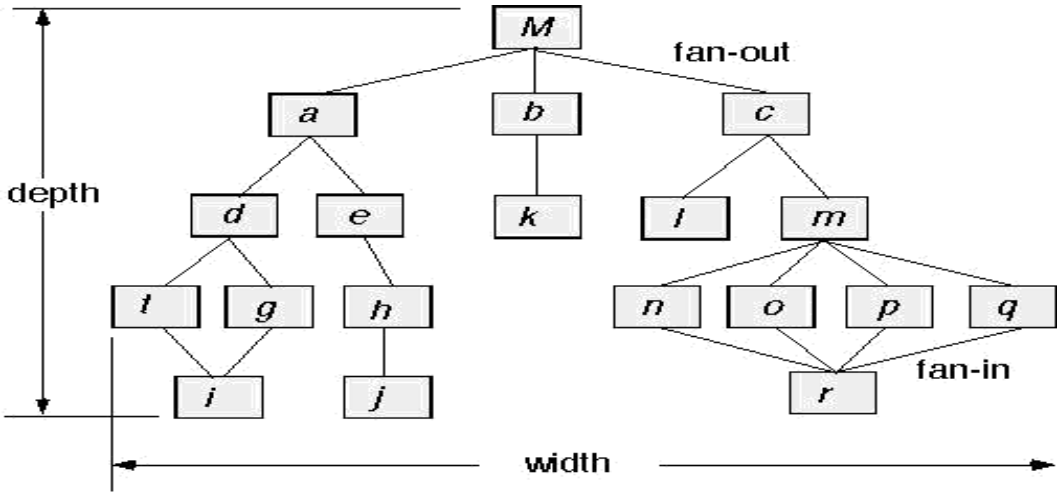
■

Call and Return Architecture

- This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. Two substyles exist within this category:

    - *Main program/subprogram architecture.* This classic program structure decomposes function into a control hierarchy where a ―main‖ program invokes a number of program components, which in turn may invoke still other components. The following figure illustrates architecture of this type.

    - *Remote procedure call architecture.* The components of main program /subprogram architecture are distributed across multiple computers on a network.

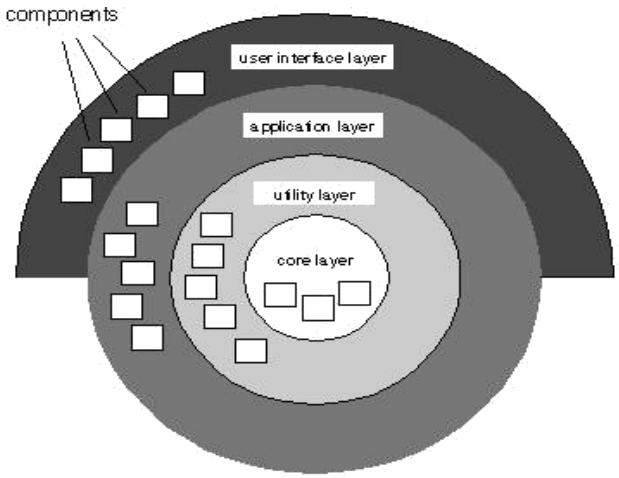


■ **Object-oriented architecture**

- The component of a system encapsulates data and the operations that must be applied to manipulate the data communication and coordination between components is accomplished via message passing.

■ **Layered Architecture**

- The basic structure of a layered architecture is illustrated in the following figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.



ARCHITECTURAL PATTERNS

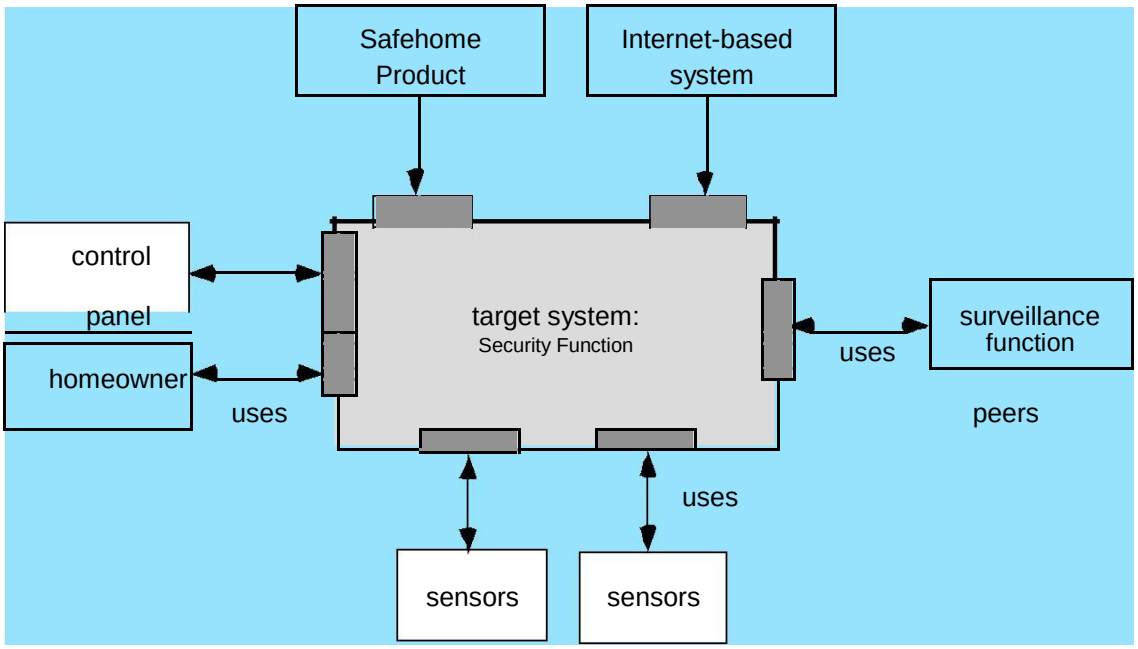
- Concurrency—applications must handle multiple tasks in a manner that simulates parallelism
    - *operating system process management* pattern
    - *task scheduler* pattern
- Persistence—Data persists if it survives past the execution of the process that created it. Two patterns are common:
    - 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

- 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
    - An *archetype* is an abstraction (similar to a class) that represents one element of system behavior
- The designer specifies the structure of the system by defining and refining software components that implement each archetype

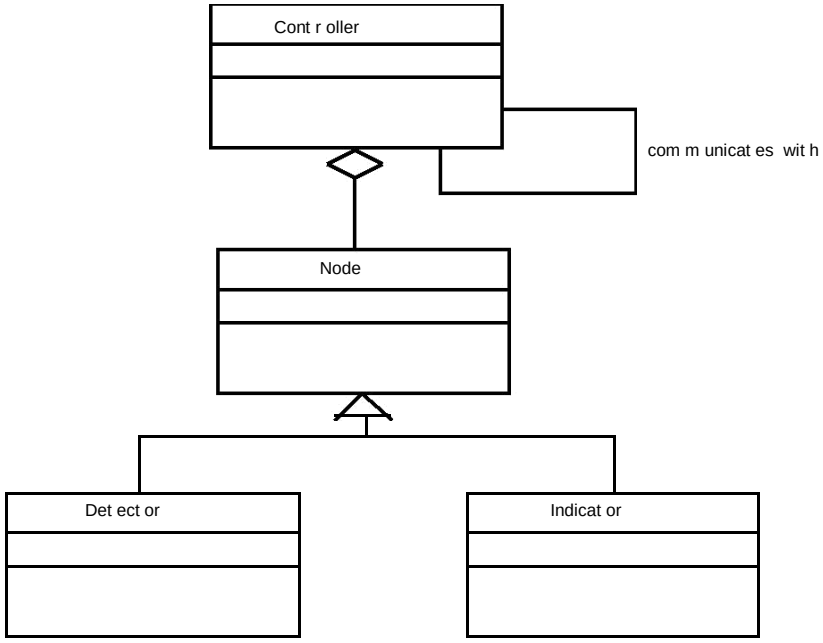
Architectural Context

- A system engineer must model context. A system context diagram accomplishes this requirement by representing the flow of information into and out of the system, the user interface and relevant support processing. 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. The generic structure of the architectural context diagram is illustrated in Figure.



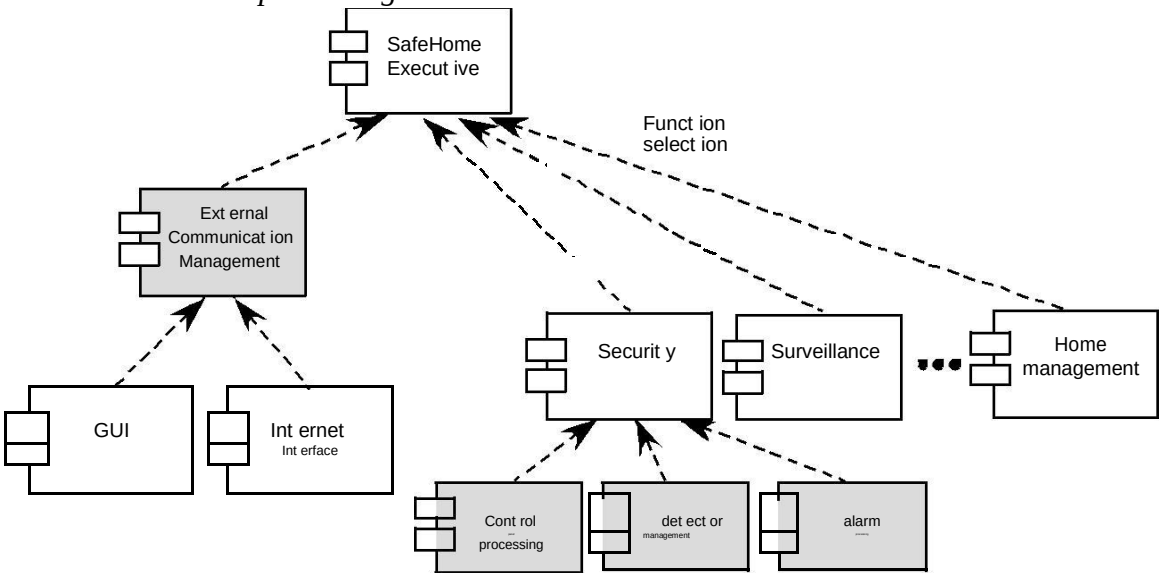
Archetypes

- the *SafeHome* home security function, we might define the following archetypes:
    - Node – Represents a cohesive collection of input and output elements of the home security function. For example a node might be comprised of
      (1) various sensors and
      (2) a variety of alarm (output) indicators.
    - Dectector – An abstraction that encompasses all sensing equipment that feeds information into the target system.
    - Indicator – An abstraction that represents all mechanism (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.



Refining the Architecture into Components

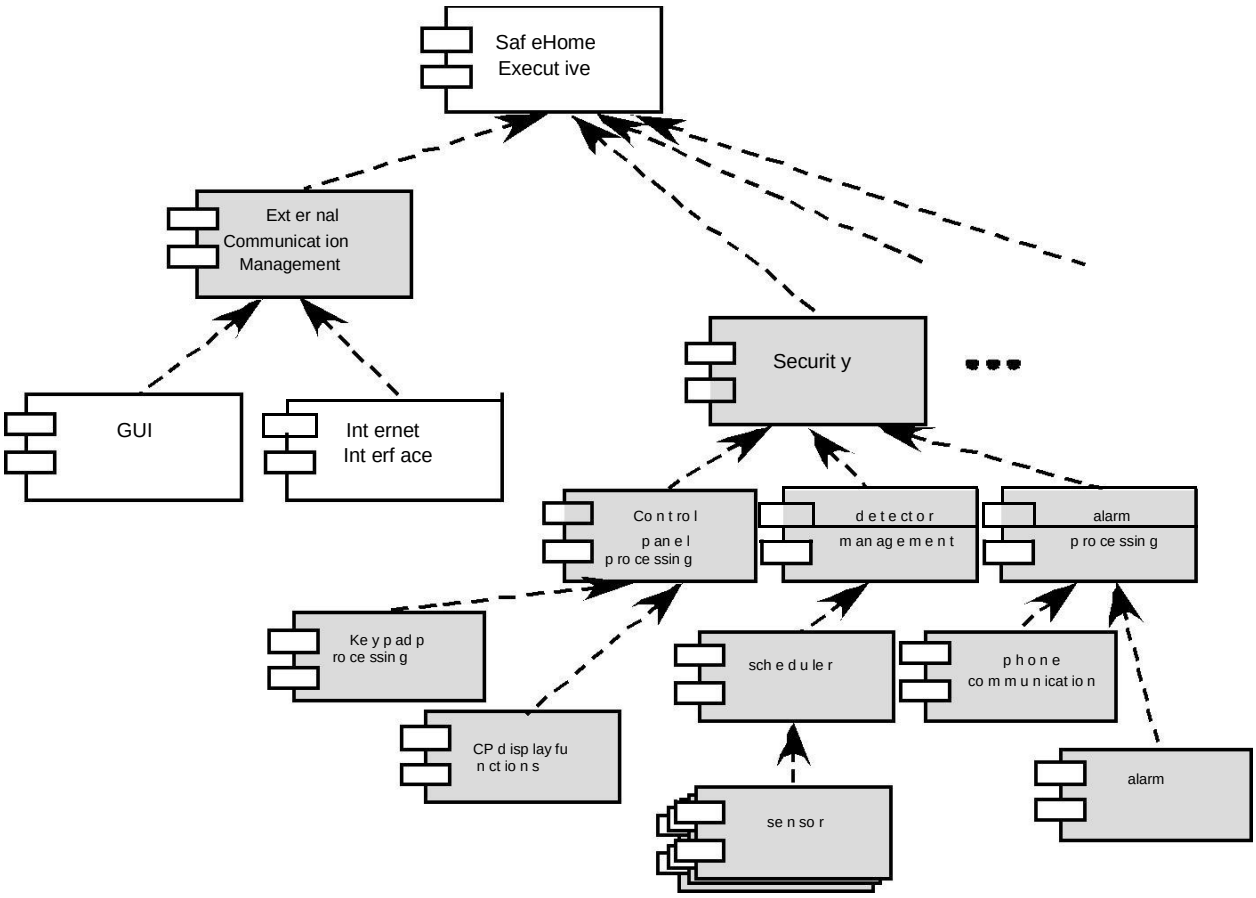
- Continuing the *SafeHome* home security function example, we might define the set of top-level components that address the following functionality.
  - *External communication management* – coordinates communication of the security function with external entities, for example, internet-based systems. External alarm notification.
  - *Control panel processing* – manages all control panel functionality.
  - *Detector management* – coordinates access to all detectors attached to the system.
  - *Alarm processing* – verifies and acts on all alarm conditions.



■

Describing Instantiations of the System

- The instantiation of the architecture means that the architecture is applied to a specific problem with the intent of demonstrating that the structure and components are appropriate.
    - Fig Illustrates an instantiation of the *SafeHome* architecture for the security system. Components shown in figure are refined further to show additional detail.



Modeling Component-Level Design

Introduction:

- Component-level design occurs after the first iteration of the architectural design
- It strives to create a design model from the analysis and architectural models
  - The translation can open the door to subtle errors that are difficult to find and correct later
  - ―Effective programmers should not waste their time debugging – they should not introduce bugs to start with.‖ Edsgar Dijkstra
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code
- The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

Definition:

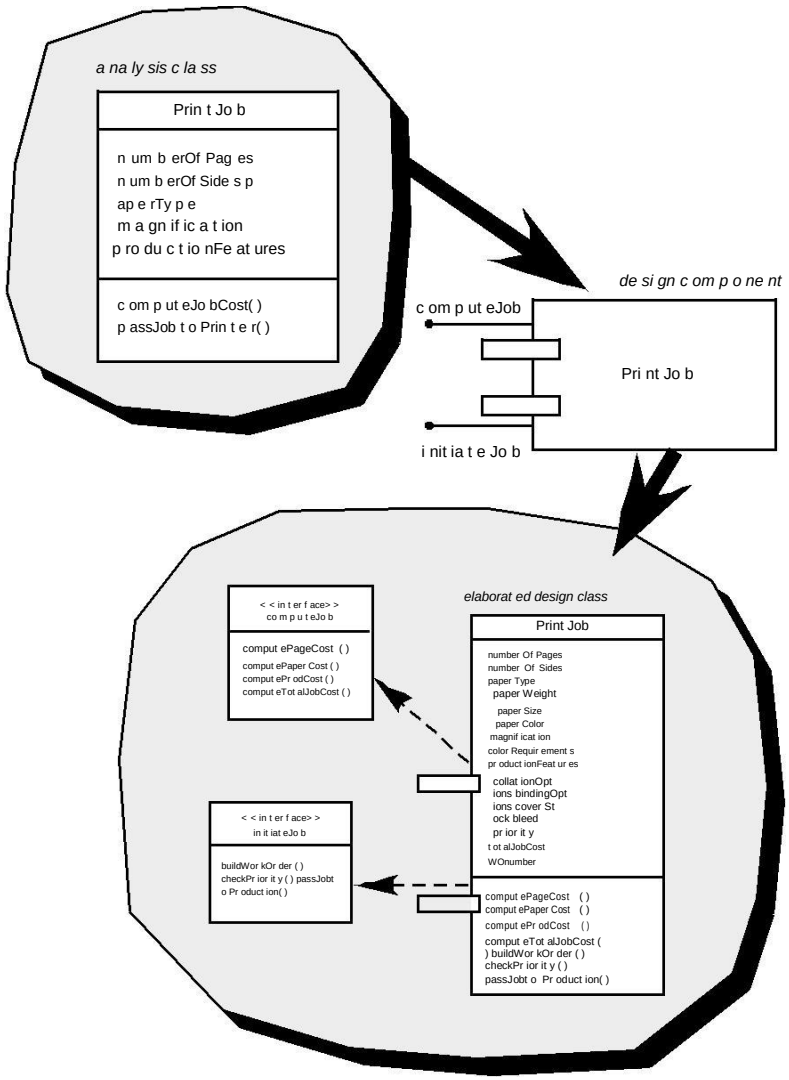
- A software component is a modular building block for computer software
  - It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- A component communicates and collaborates with
  - Other components
  - Entities outside the boundaries of the system
- Three different views of a component
  - An object-oriented view
  - A conventional view
  - A process-related view

Object-oriented view

- A component is viewed as a set of one or more collaborating classes
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
  - This also involves defining the interfaces that enable classes to communicate and collaborate
- This elaboration activity is applied to every component defined as part of the architectural design
- Once this is completed, the following steps are performed
  - Provide further elaboration of each attribute, operation, and interface
  - Specify the data structure appropriate for each attribute
  - Design the algorithmic detail required to implement the processing logic associated with each operation

Design the mechanisms required to implement the interface to include the messaging that occurs between objects.

Component in Object-Oriented View:

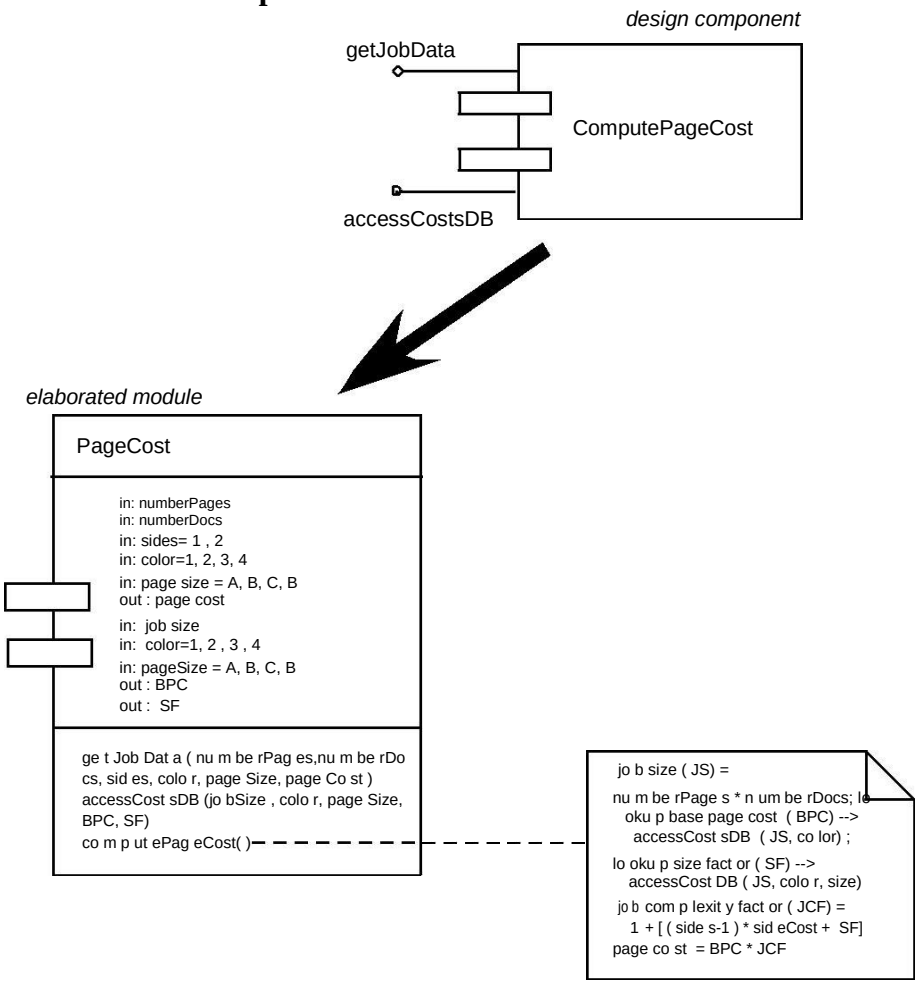


Conventional View:

- A component is viewed as a <u>functional</u> element (i.e., a module) of a program that incorporates
  - The <u>processing logic</u>
  - The internal <u>data structures</u> that are required to implement the processing logic
  - An <u>interface</u> that enables the component to be invoked and data to be passed to it
- A component serves one of the following roles
  - A <u>control</u> component that coordinates the invocation of all other problem domain components
  - A <u>problem domain</u> component that implements a complete or partial function that is required by the customer
  - An <u>infrastructure</u> component that is responsible for functions that support the processing required in the problem domain

– Conventional software components are derived from the data flow diagrams (DFDs) in the analysis model
  - Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy
  - Control components reside near the top
  - Problem domain components and infrastructure components migrate toward the bottom
  - Functional independence is strived for between the transforms
– Once this is completed, the following steps are performed for each transform
  1. Define the interface for the transform (the order, number and types of the parameters)
  2. Define the data structures used internally by the transform
  3. Design the algorithm used by the transform (using a stepwise refinement approach)
–

Conventional Component:



Process-related View:

- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- As the software architecture is formulated, components are selected from the library and used to populate the architecture
- Because the components in the library have been created with reuse in mind, each contains the following:
  - A complete description of their interface
  - The functions they perform
  - The communication and collaboration they require

Designing Class-Based Components

Component-level Design Principles:

- The Open-Closed Principle (OCP).
  - ―A module [component] should be open for extension but closed for modification.
    - Extensible within the functional domain it addresses without the need to make internal (code or logic-level) modifications.
    - There shall be an abstraction between the design class and the functionality that is likely to be extended.
- Example:
  - A Detector class that must check the status of each type of security sensor.
  - Instead of an if-then-else, we put a ―sensor interface‖ as a consistent view of sensors to Detector
  - A new type of sensor can be added with no change to Detector class
- The Liskov Substitution Principle (LSP).

"Subclasses should be substitutable for their base classes.

- Any derived class must honor any contract between the base class and the components that use it
  Contract:
- Precondition that must be true before a component uses a base class
- Post-condition that should be true after the component uses a base class
- Dependency Inversion Principle (DIP).
- ―Depend on abstractions. Do not depend on concretions.‖
  - Abstractions are the place where a design can be extended without great complication
  - Depend on abstractions (like interfaces) rather than concrete components

- The Interface Segregation (isolation) Principle (ISP). ―Many client-specific interfaces are better than one general purpose interface.
    - Specialized interfaces for each major category of clients
    - Interface for a client: only operations useful for that client
- Operations used by many, shall be specified in each interface

Component Packaging Principles

- Release reuse equivalency principle
  - The granularity of reuse is the granularity of release
  - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- Common closure principle
  - Classes that change together belong together
  - Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- Common reuse principle
  - Classes that aren't reused together should not be grouped together
  - Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

Component-Level Design Guidelines

- Components
  - Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
  - Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
  - Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
  - Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
  - Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency

Cohesion:

- Cohesion is the ―single-mindedness‘ of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as high as possible

- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
  - Functional
    - A module performs one and only one computation and then returns a result
  - Layer
    - A higher layer component accesses the services of a lower layer component
  - Communicational
    - All operations that access the same data are defined within one class
  - Sequential
    - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
  - Procedural
    - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
  - Temporal
    - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
  - Utility
    - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

Coupling:

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible
- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
  - Data coupling
    - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
  - Stamp coupling
    - A whole data structure or class instantiation is passed as a parameter to an operation
  - Control coupling
    - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
    - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result

- Common coupling
  - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
- Content coupling
  - One component secretly modifies data that is stored internally in another component

➢ **Other kinds of coupling (unranked)**

- Subroutine call coupling
  - When one operation is invoked it invokes another operation within side of it
- Type use coupling
  - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
  - If/when the type definition changes, every component that declares a variable of that data type must also change
- Inclusion or import coupling
  - Component A imports or includes the contents of component B
- External coupling
  - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

Conducting Component level Design

1) Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
2) Identify all design classes that correspond to the infrastructure domain
   - These classes are usually not present in the analysis or architectural models
   - These classes include GUI components, operating system components, data management components, networking components, etc.
3) Elaborate all design classes that are not acquired as reusable components
   - Specify message details (i.e., structure) when classes or components collaborate
   - Identify appropriate interfaces (e.g., abstract classes) for each component
   - Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
   - Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams
4) Describe persistent data sources (databases and files) and identify the classes required to manage them
5) Develop and elaborate behavioral representations for a class or component
   - This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
6) Elaborate deployment diagrams to provide additional implementation detail

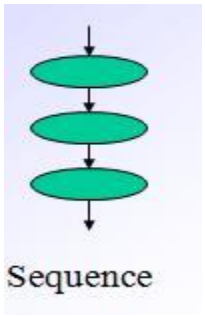
- Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments

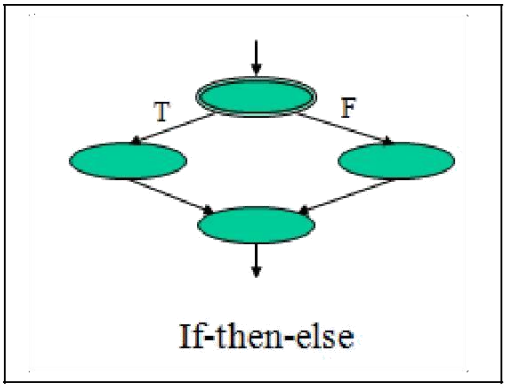
7) Factor every component-level design representation and always consider alternatives
   - Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
   - The final decision can be made by using established design principles and guidelines

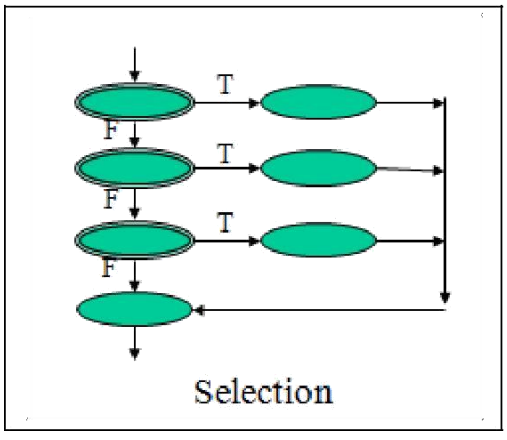
Designing Conventional Components

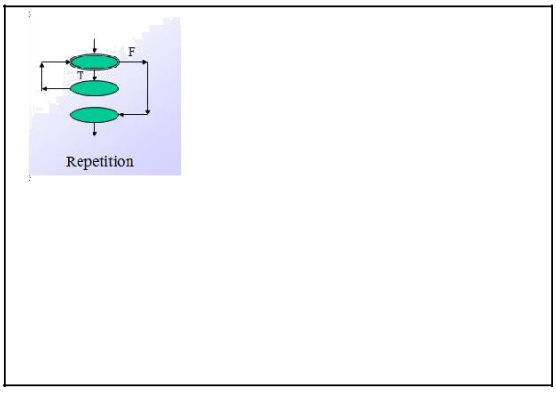
- Conventional design constructs emphasize the maintainability of a functional/procedural program
  - Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Various notations depict the use of these constructs
  - **Graphical design notation**
    - Sequence, if-then-else, selection, repetition
  - **Tabular design notation**
  - **Program design language**
    - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

Graphical design notation:









Tabular Design Notation:

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

	Rule s					
Condit ions	**1**	**2**	**3**	**4**	**5**	**6**
regular cust omer	T	T				
silver cust omer			T	T		
gold cust omer					T	T
special discount	F	T	F	T	F	T
Rule s						
no discount	✓					
apply 8 percent discount			✓	✓		
apply 15 percent discount					✓	✓
apply addit ional x percent discount		✓		✓		✓

PERFORMING USER INTERFACE DESIGN

■ User Interface design create an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

■ **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

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.

- **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 ANALYSIS AND DESIGN

- The overall process for analyzing and designing a user interface begins with the creation of different models of system function.

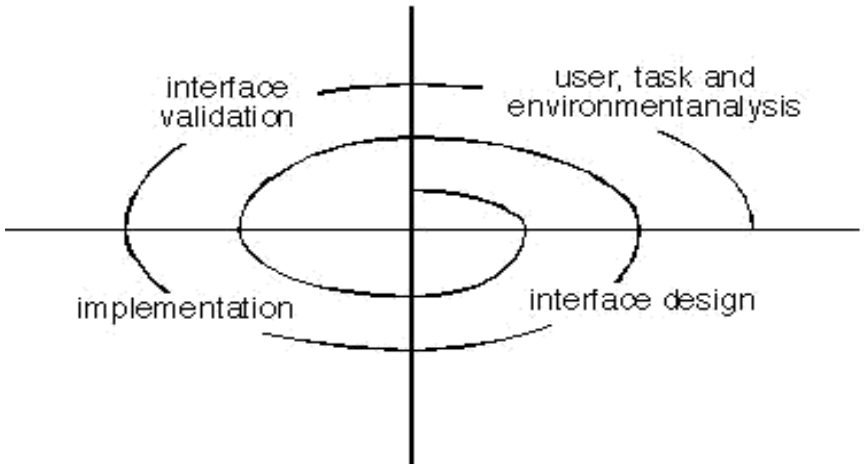
■ **Interface Analysis and Design Models**

- 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**

- The analysis and design process for user interfaces is iterative and can be represented using a spiral model.
- The user interface analysis and design process encompasses four distinct framework activities.
    - User, task and environment analysis and modeling.
    - Interface design.
    - Interface construction (implementation).
    - Interface validation.

- The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are
    - Where will the interface be located physically?
    - Will the user be sitting, standing or performing other tasks unrelated to the interface?
    - Does the interface hardware accommodate space, light or noise constraints?
    - Are there special human factors consideration driven by environmental factors?
- The information gathered as part of the analysis activity is used to create an analysis model for the interface.
- The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.
- Validation focuses on
    (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements.
    (2) the degree to which the interfaced is easy to use and easy to learn
    (3) the users acceptance of the interface as a useful tool in their work.



INTERFACE ANALYSIS

Interface analysis means understanding

- the people (end-users) who will interact with the system through the interface;
- the tasks that end-users must perform to do their work,
- the content that is presented as part of the interface
- the environment in which these tasks will be conducted**.**

User Analysis

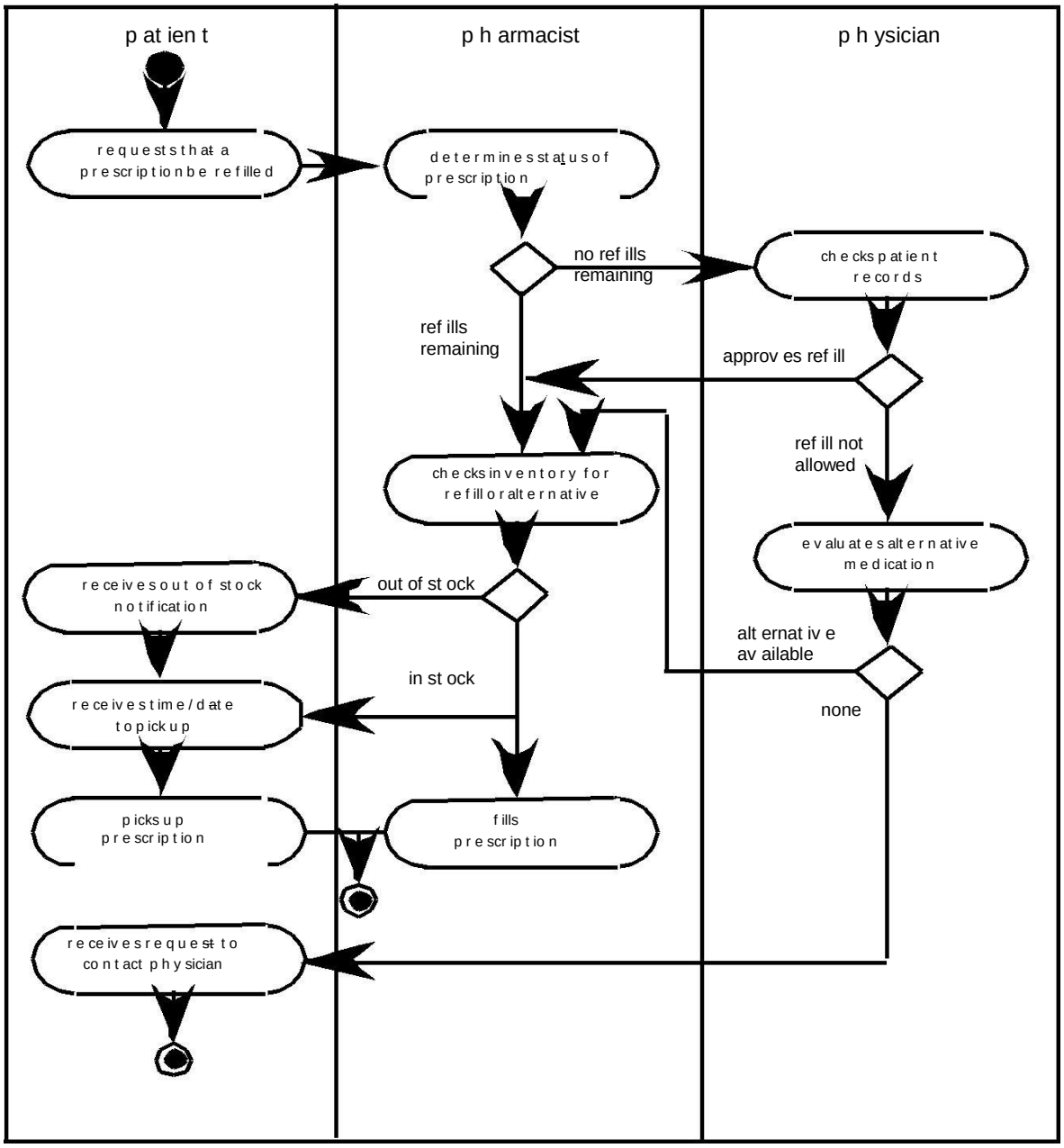
- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- 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?
- 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 the 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
- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved

■ **Swimlane Diagram**



■ **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 to be used to enhance understanding?
- How will error messages and warning be presented to the user?

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.

■ **Interface Design Patterns**

The Design pattern is an abstraction that prescribes a design solution to a specific, well bounded design problem

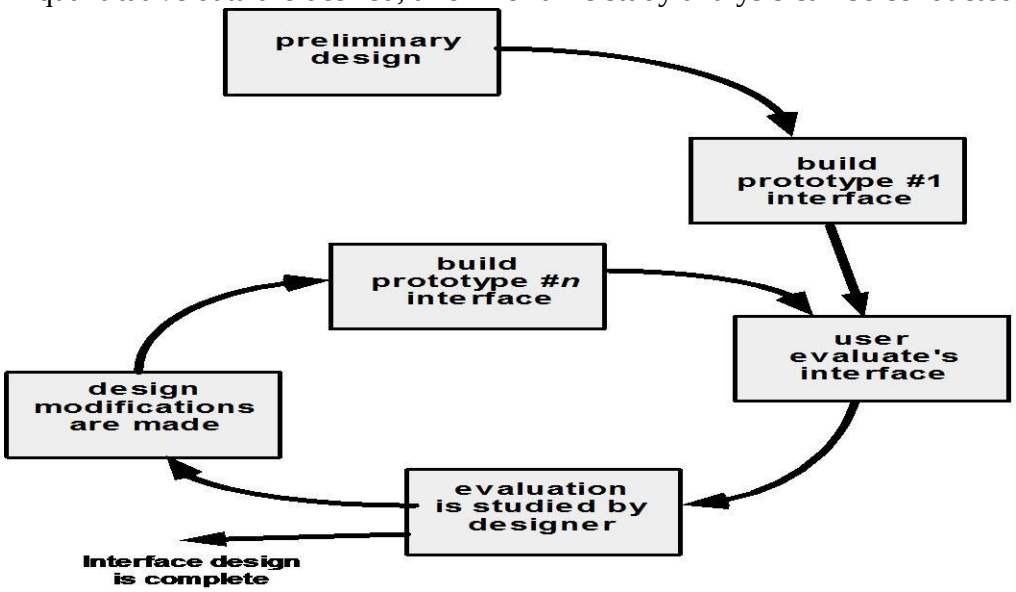
- Patterns are available for
  - The complete UI
  - Page layout
  - Forms and input
  - Tables
  - Direct data manipulation
  - Navigation
  - Searching
  - Page elements
  - e-Commerce

■ **Design Issues**

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

DESIGN EVALUATION

- Design evaluation determines whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal ―test drive,‖ in which a user provide impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end-users.
- After the design model has been completed, a first level prototype is created.
- The design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews.
  - The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
  - The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
  - The number of actions, tasks and system states indicated by the design model imply the memory load on users of the system
  - Interface style, help facilities and error handling protocol provide indication of the complexity of the interface and the degree to which it will be accepted by the user.
- To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be
  (1) simple yes/no response,
  (2) numeric response,
  (3) scaled (subjective) response,
  (4) likert scales (e.g., strongly agree, somewhat agree),
  (5) percentage (subjective) response or
  (6) open-ended.
- If quantitative data are desired, a form of time study analysis can be conducted.



4.1 Software process and project metrics

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

Process and Project Metrics

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

Process Metrics

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

Statistical Process Control

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

Project Metrics

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

Software Measurement

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

Size-Oriented Metrics

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

Function-Oriented Metrics

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

Reconciling LOC and FP Metrics

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

Object-Oriented Metrics

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

Use Case-Oriented Metrics

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

WebApp Project Metrics

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

Software Quality Metrics

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

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

Integrating Metrics with Software Process

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

Arguments for Software Metrics

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

Baselines

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

Metrics for Small Organizations

- Most software organizations have fewer than 20 software engineers.
- Best advice is to choose simple metrics that provide value to the organization and don't require a lot of effort to collect.

- Even small groups can expect a significant return on the investment required to collect metrics, if this activity leads to process improvement.

Establishing a Software Metrics Program

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

4.2 Software Quality Assurance

Overview

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

Quality Concepts

- Variation control is the heart of quality control (software engineers strive to control the process applied, resources expended, and end product quality attributes).
- Quality of design - refers to characteristics designers specify for the end product to be constructed
- Quality of conformance - degree to which design specifications are followed in manufacturing the product
- Quality control - series of inspections, reviews, and tests used to ensure conformance of a work product to its specifications
- Quality assurance - consists of the auditing and reporting procedures used to provide management with data needed to make proactive decisions
- Cost of Quality
  Prevention costs - quality planning, formal technical reviews, test equipment, training
  Appraisal costs - in-process and inter-process inspection, equipment calibration and maintenance, testing
  Failure costs - rework, repair, failure mode analysis

External failure costs - complaint resolution, product return and replacement, help line support, warranty work

Software Quality Assurance

1. Conformance to software requirements is the foundation from which software quality is measured.
2. Specified standards are used to define the development criteria that are used to guide the manner in which software is engineered.
3. Software must conform to implicit requirements (ease of use, maintainability, reliability, etc.) as well as its explicit requirements.

SQA Group Activities

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

Software Reviews

- Purpose is to find errors before they are passed on to another software engineering activity or released to the customer.
- Software engineers (and others) conduct formal technical reviews (FTRs) for software engineers.
- Using formal technical reviews (walkthroughs or inspections) is an effective means for improving software quality.

Software Defects

- Industry studies suggest that design activities introduce 50-65% of all defects or errors during the software process.
- Review techniques have been shown to be up to 75% effective in uncovering design flaws which ultimately reduce the cost of subsequent activities in the software process.
- Defect amplification models can be used to show that the benefits of detecting and removing defects from activities that occur early in the software process.

Formal Technical Reviews

- Involves 3 to 5 people (including reviewers)

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

Formal Technical Review Guidelines

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

Sample Driven Reviews

- Samples of all software engineering work products are reviewed to determine the most error-prone.
- Full FTR resources are focused on the likely error-prone work products based on sampling results
- To be effective the sample driven review process must be driven by quantitative measures of the work products.
    1. Inspect a representative fraction **(a_i)** of the content of each software work product (i) and record the number of faults (f_i) found within (ai).
    2. Develop a gross estimate of the number of faults within work product i by multiplying f_i by $1/a_i$.
    3. Sort work products in descending order according to the gross estimate of the number of faults in each.
    4. Focus on available review resources on those work products with the highest estimated number of faults.

Statistical Quality Assurance

1. Information about software defects is collected and categorized.

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

Six Sigma Software Engineering

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

Software Reliability

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

Software Safety

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

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

ISO 9000 Quality Standards

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

SQA Plan

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

4.3 - Software Testing Strategies

Overview

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

Strategic Approach to Software Testing

- Many software errors are eliminated before testing begins by conducting effective technical reviews
- Testing begins at the component level and works outward toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- The developer of the software conducts testing and may be assisted by independent test groups for large projects.
- Testing and debugging are different activities.
- Debugging must be accommodated in any testing strategy.

Verification and Validation

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

Organizing for Software Testing

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

Software Testing Strategy

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

- System Testing – verifies that all system elements mesh properly and that overall system function and performance has been achieved

Strategic Testing Issues

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

Unit Testing

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

Integration Testing

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

Integration Testing Strategies

- Top-down integration testing
  1. Main control module used as a test driver and stubs are substitutes for components directly subordinate to it.
  2. Subordinate stubs are replaced one at a time with real components (following the depth-first or breadth-first approach).
  3. Tests are conducted as each component is integrated.
  4. On completion of each set of tests and other stub is replaced with a real component.

5. Regression testing may be used to ensure that new errors not introduced.

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

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

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

General Software Test Criteria

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

Object-Oriented Test Strategies

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

Object-Oriented Unit Testing

- smallest testable unit is the encapsulated class or object

- similar to system testing of conventional software
- do not test operations in isolation from one another
- driven by class operations and state behavior, not algorithmic detail and data flow across module interface

Object-Oriented Integration Testing

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

WebApp Testing Strategies

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

Validation Testing

- Focuses on visible user actions and user recognizable outputs from the system
- Validation tests are based on the use-case scenarios, the behavior model, and the event flow diagram created in the analysis model
    - Must ensure that each function or performance characteristic conforms to its specification.
    - Deviations (deficiencies) must be negotiated with the customer to establish a means for resolving the errors.

- Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.

Acceptance Testing

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

System Testing

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

Bug Causes

- The symptom and the cause may be geographically remote (symptom may appear in one part of a program).
- The symptom may disappear (temporarily) when another error is corrected.
- The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
- The symptom may be caused by human error that is not easily traced.
- The symptom may be a result of timing problems, rather than processing problems.
- It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
- The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
- The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Debugging Strategies

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

Bug Removal Considerations

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

4.4 – Testing Conventional Applications

Overview

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

Software Testing Objectives

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

Software Testability Checklist

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

Good Test Attributes

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

Test Case Design Strategies

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

White-Box Testing Questions

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

Basis Path Testing

- White-box technique usually based on the program flow graph
- The cyclomatic complexity of the program computed from its flow graph using the formula V(G) = E – N + 2 or by counting the conditional statements in the program design language (PDL) representation and adding 1

- Determine the basis set of linearly independent paths (the cardinality of this set id the program cyclomatic complexity)
- Prepare test cases that will force the execution of each path in the basis set.

Control Structure Testing

- White-box technique focusing on control structures present in the software
- Condition testing (e.g. branch testing) – focuses on testing each decision statement in a software module, it is important to ensure coverage of all logical combinations of data that may be processed by the module (a truth table may be helpful)
- Data flow testing – selects test paths based according to the locations of variable definitions and uses in the program (e.g. definition use chains)
- Loop testing – focuses on the validity of the program loop constructs (i.e. simple loops, concatenated loops, nested loops, unstructured loops), involves checking to ensure loops start and stop when they are supposed to (unstructured loops should be redesigned whenever possible)

Black-Box Testing Questions

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

Graph-based Testing Methods

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

Equivalence Partitioning

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

Boundary Value Analysis

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

Orthogonal Array Testing

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

Model-Based Testing

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

Specialized Testing

- Graphical User Interface (GUI) – test cases can be developed from behavioral model of user interface, use of automated testing tools is strongly recommended.
- Client/Sever Architectures – operational profiles derived from usage scenarios are tested at three levels (client application "disconnected mode", client and server software without network, complete application)
  - Applications function tests
  - Server tests
  - Database tests
  - Transaction tests
  - Network communications tests
- Documentation and Help
  - Review and inspection to check for editorial errors
  - Black-Box for live tests
    - Graph-based testing to describe program use
    - Equivalence partitioning and boundary value analysis to describe classes of input and interactions
- Real-Time Systems
  1. Task testing – test each task independently
  2. Behavioral testing – using technique similar to equivalence partitioning of external event models created by automated tools
  3. Intertask testing – testing for timing errors (e.g. synchronization and communication errors)
  4. System testing – testing full system, especially the handling of Boolean events (interrupts), test cased based on state model and control specification

Testing Patterns

- Provide software engineers with useful advice as testing activities begin
- Provide a vocabulary for problem-solvers
- Can focus attention on forces behind a problem (when and why a solution applies)

- Encourage iterative thinking (each solution creates a new context in which problems can be solved

4.5 - Product Metrics for Software

Overview

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

Definitions

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

Benefits of Product Metrics

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

Measurement Process Activities

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

- *Feedback* – recommendations derived from interpretation of the metrics is transmitted to the software development team

Metrics Characterization and Validation Principles

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

Measurement Collection and Analysis Principles

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

Goal-Oriented Software Measurement (GQM)

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

Attributes of Effective Software Metrics

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

Requirements Model Metrics

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

- Specificity
- Completeness

Architectural Design Metrics

- Structural complexity (based on module fanout)
- Data complexity (based on module interface inputs and outputs)
- System complexity (sum of structural and data complexity)
- Morphology (number of nodes and arcs in program graph)
- Design structure quality index (DSQI)

OO Design Metrics

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

Class-Oriented Metrics

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

Component-Level Design Metrics

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

Operation-Oriented Metrics

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

Using WebApp Design Metrics

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

WebApp Interface Metrics

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

Aesthetic (graphic layout) metrics

- Word count
- Body text percentage

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

Content Metrics

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

Navigation Metrics

- Page link complexity
- Connectivity
- Connectivity density

Halstead's Software Science (Source Code Metrics)

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

Testing Metrics

- Metrics that predict the likely number of tests required during various testing phases
  - Architectural design metrics
  - Cyclomatic complexity can target modules that are candidates for extensive unit testing
  - Halstead effort

- Metrics that focus on test coverage for a given component
  - Cyclomatic complexity lies at the core of basis path testing

Object-Oriented Testing Metrics

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

Maintenance Metrics

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