SOFTWARE ENGINEERING

UNIT-1

1. The Nature of Software

Defining Software

Software Application Domains

Legacy Software

2. The Unique Nature of WebApps

3. Software Engineering

4. The Software Process

5. Software Engineering Practice

The Essence of Practice

General Principles

6. Software Myths

7. A Generic Process Model

Defining a Framework Activity

Identifying a Task Set

Process Patterns

8. Process Assessment and Improvement

9. Prescriptive Process Models

9.1The Waterfall Model

9.2Incremental Process Models

9.3Evolutionary Process Models

9.4Concurrent Models

9.5A Final Word on Evolutionary Processes

10. Specialized Process Models

Component-Based Development

The Formal Methods Model

Aspect-Oriented Software Development

11. The Unified Process

A Brief History

Phases of the Unified Process

12. Personal and Team Process Models

Personal Software Process (PSP)

Team Software Process (TSP)

13. Process Technology

14. Product and Process

1. The Nature of Software

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 or more broadly, *by a network of computers* that are accessible by local hardware. Whether 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 that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources***.

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.

Software manages business information to enhance competitiveness;
Software provides a gateway to worldwide information networks (e.g., the Internet).
Software provides the means for acquiring information in all of its forms.
Software role has undergone significant change over the last half-century.
Software industry has become a dominant factor industrialized world.

Engineering Discipline:

- Engineering is a disciplined approach with some organized steps in a managed way to construction, operation, and maintenance of software.
- Engineering of a product goes through a series of stages, i.e., planning, analysis and specification, design, construction, testing, documentation, and deployment.
- The disciplined approach may lead to better results.
- The general stages for engineering the software include feasibility study and preliminary investigation, requirement analysis and specification, design, coding, testing, deployment, operation, and maintenance.

Software Crisis:

- Software crisis, the symptoms of the problem of engineering the software, began to enforce the practitioners to look into more disciplined software engineering approaches for software development.

- The software industry has progressed from the desktop PC to network-based computing to service-oriented computing nowadays.

- The development of programs and software has become complex with increasing requirements of users, technological advancements, and computer awareness among people.

- *Software crisis symptoms*

    - complexity,

    - hardware versus software cost,

    - Lateness and costliness,

    - poor quality,

    - unmanageable nature,

    - immaturity,

    - lack of planning and management practices,

    - Change, maintenance and migration,

    - etc.

What is Software Engineering?

- *The solution to these software crises is to introduce systematic software engineering practices for systematic software development, maintenance, operation, retirement, planning, and management of software.*

- The systematic means the methodological and pragmatic way of development, operation and maintenance of software.

- Systematic development of software helps to understand problems and satisfy the client needs.

- Development means the construction of software through a series of activities, i.e., analysis, design, coding, testing, and deployment.

- Maintenance is required due to the existence of errors and faults, modification of existing features, addition of new features, and technological advancements.

- Operational software must be correct, efficient, understandable, and usable for work at the client site.

- IEEE defines

- *The systematic approach to the development, operation, maintenance, and retirement of software.*

-

- Defining Software

Software is:
(1) **instructions** (computer programs) that when executed provide desired features, function, and performance;
(2) **data structures** that enable the programs to adequately manipulate information, and
(3) **descriptive information** in both hard copy and virtual forms that describes the operation and use of the programs.

Software characteristics

- Software has logical properties rather than physical.
- Software is mobile to change.
- Software is produced in an engineering manner rather than in classical sense.
- Software becomes obsolete but does not wear out or die.
- Software has a certain operating environment, end user, and customer.
- Software development is a labor-intensive task

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

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

Although some similarities exist between software development and hardware manufacturing, the two activities are fundamentally different.

In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different.

Both activities require the construction of a "product," but the approaches are different. Software costs are concentrated in engineering. This means that software projects cannot be managed as if they were manufacturing projects.

2. Software doesn't "wear out."

The above picture depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); ***defects are corrected*** and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time.

As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.

Stated simply, the hardware begins to wear out.
Software doesn't "wear out."
But Software does deteriorate!

The above picture explains the software failure in development process. During development process, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve". Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—***the software is deteriorating due to change.***

3. Although the industry is moving toward component-based construction, most software continues to be custom built.

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems.

The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new.

In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. For example, today's interactive user interfaces are built with reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structures and processing detail required to build the interface are contained within a library of reusable components for interface construction.

Software Application Domains

(Types of Software or Categories of Software)

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

i. **System software**—a collection of programs written to service other programs.

System software processes complex, but determinate information structures.
e.g., compilers, editors, and file management utilities

Systems applications process largely indeterminate data.
(e.g., operating system components, drivers, networking software, telecommunications processors)

ii. **Application software**—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., point-of-sale transaction processing, real-time manufacturing process control.

iii. **Engineering/scientific software**—is a special software to implement Engineering and Scientific applications. Applications range from astronomy to volcanology, from automotive stress analysis to space shuttle orbital dynamics, and from molecular biology to automated manufacturing. However, modern applications within the engineering/scientific area are moving away from conventional numerical algorithms. Computer-aided design, system simulation, and other interactive applications have begun to take on real-time and even system software characteristics.

iv. **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.

e.g., key pad control for a microwave oven.

Provide significant function and control capability

e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems.

v. **Product-line software**—designed to provide a specific capability for use by many different customers.

e.g., inventory control products, word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications.

vi. **Web applications**—called "WebApps," this network-centric software category spans a wide array of applications. WebApps are linked with hypertext files. WebApps are evolving into sophisticated computing environments that not only provide stand-alone features, computing functions, and content to the end user, but also are integrated with corporate databases and business applications.

vii. **Artificial intelligence software**— makes use of nonnumerical algorithms to solve complex problems of straightforward analysis.

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

viii. **Open-world computing**—Software related to wireless networking may soon lead to true pervasive, distributed computing. The challenge for software engineers will be to develop systems and application software that will allow mobile devices, personal computers, and enterprise systems to communicate across vast networks.

ix. **Netsourcing**—the World Wide Web is rapidly becoming a computing engine as well as a content provider. The challenge for software engineers is to architect simple (e.g., personal financial planning) and sophisticated applications that provide a benefit to targeted end-user markets worldwide.

x. **Open source**—a growing trend that results in distribution of source code for systems applications (e.g., operating systems, database, and development environments) so that many people can contribute to its development.

The challenge for software engineers is to build source code that is self-descriptive, but more importantly, to develop techniques that will enable both customers and developers to know what changes have been made and how those changes manifest themselves within the software.

Legacy Software

Older programs —often referred to as *legacy software*—have been the focus of continuous attention and concern since the 1960s. Dayani-Fard and his colleagues [Day99] describe legacy software in the following way:

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

Liu and his colleagues [Liu98] extend this description by noting that "many legacy systems remain supportive to core business functions and are 'indispensable' to the business." Hence, legacy software is characterized by longevity and business criticality.

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.

The only reasonable answer may be: *Do nothing,* at least until the legacy system must undergo some significant change. 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 useful in 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".

2. The Unique Nature of WebApps

Web Apps means Web Applications. WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

Web engineers to provide computing capability along with informational content. *Web-based systems and applications* were born.

WebApps are one of a number of distinct software categories. And yet, it can be argued that WebApps are different.

The following attributes are encountered in the vast majority of WebApps.

i. **Network intensiveness.** A WebApp ***resides on a network and must serve the needs of a different types of clients***. The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication (e.g., a corporate Intranet).

ii. **Concurrency.** A large number of ***users may access the WebApp at one time***. In many cases, the patterns of usage among end users will vary greatly.

iii. **Unpredictable load.** The number of users of the WebApp may vary by orders of magnitude from day to day. One ***hundred users*** may show up on Monday; 10,000 may use the system on Thursday.

iv. **Performance.** WebApp should work effectively in terms of ***processing speed***. If a WebApp user must wait too long (for access, for serverside processing, for client-side formatting and display), he or she may decide to go elsewhere.

v. **Availability.** Although expectation of 100 percent availability is unreasonable, users of popular WebApps often ***demand access on a 24/7/365 basis***. Users in Australia or Asia might demand access during times when traditional domestic software applications in North America might be taken off-line for maintenance.

vi. **Data driven.** The primary function of many WebApps is to use hypermedia to present ***text, graphics, audio, and video*** content to the end user. In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

vii. **Content sensitive.** The quality and aesthetic (beauty) ***nature of content*** remains an important determinant of the quality of a WebApp.

viii. **Continuous evolution. (Updated version)** -Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously. It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

ix. **Immediacy.** Although *immediacy*—the compelling need to get ***software to market quickly***—is a characteristic of many application domains, WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.7

x. **Security.** Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to ***protect sensitive content and provide secure modes of data transmission***, ***strong security measures must be implemented*** throughout the infrastructure that supports a WebApp and within the application itself.

xi. **Aesthetics.** An undeniable part of the appeal of a ***WebApp is its look and feel***. When an application has been designed to market or sell products or ideas, aesthetics may have as much to do with success as technical design.

3. ***Software Engineering***

What is Software Engineering?

software engineering practices for systematic software development, maintenance, operation, retirement, planning, and management of software.

• The systematic means the methodological and pragmatic way of development, operation and maintenance of software.

• Systematic development of software helps to understand problems and satisfy the client needs.

• Development means the construction of software through a series of activities, i.e., analysis, design, coding, testing, and deployment.

• Maintenance is required due to the existence of errors and faults, modification of existing features, addition of new features, and technological advancements.

• Operational software must be correct, efficient, understandable, and usable for work at the client site.

• IEEE defines

The systematic approach to the development, operation, maintenance, and retirement of software.

Software Engineering -

• *It follows that a concerted effort should be made to understand the problem before a software solution is developed.*

. It follows that design becomes a unique activity.

• *It follows that software should exhibit high quality.*

• *It follows that software should be maintainable, software in all of its forms and across all of its application domains should be engineered.*

4. The Software Process

- A *process* is a collection of activities, actions, and tasks that are performed when some work product is to be created.
- An *activity* strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.
- An *action* (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).
- A *task* focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

In the context of software engineering, a process is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to ***deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation*** and those who will use it.

A ***process framework*** establishes the foundation for a complete software engineering process by identifying a small number of ***framework activities*** that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of ***umbrella activities*** that are applicable across the entire software process.

A generic process framework for software engineering encompasses five activities:

i. **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders11 The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
ii. **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a *software project plan*—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
iii. **Modeling.** Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.
iv. **Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
v. **Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and

provides feedback based on the evaluation. These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

For many software projects, framework activities are applied iteratively as a project progresses. That is,

communication,
planning,
modeling,
construction, and
deployment are applied repeatedly through a number of project iterations. Each project iteration produces a *software increment* that provides stakeholders with a subset of overall software features and functionality. As each increment is produced, the software becomes more and more complete.

Software engineering process framework activities are complemented by a number of *umbrella activities.* In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk.

Typical umbrella activities include:

Software project tracking and control—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

Risk management—assesses risks that may affect the outcome of the project or the quality of the product.

Software quality assurance—defines and conducts the activities required to ensure software quality.

Technical reviews—assesses software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

Measurement—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

Software configuration management—manages the effects of change throughout the software process.

Reusability management—defines criteria for work product reuse (including software components) and establishes mechanisms to achieve reusable components.

Work product preparation and production—encompasses the activities required to create work products such as models, documents, logs, forms, and lists. Each of these umbrella activities is discussed in detail later in this book.

The software engineering process is not a rigid prescription that must be followed dogmatically by a software team. Rather, it should be agile and adaptable (to the problem, to the project, to the team, and to the organizational culture). Therefore, a process adopted for one project might be significantly different than a process adopted for another project. Among the differences are

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

• **Degree to which actions and tasks** are defined within each framework activity

• **Degree to which work products are identified and required**

• Manner in which **quality assurance activities** are applied

• Manner in which **project tracking and control activities** are applied
• **Overall degree of detail and rigor** with which the process is described
• **Degree to which the customer and other stakeholders are involved with the project**
• **Level of autonomy given to the software team**
• **Degree to which team organization and roles are prescribed**

5. Software Engineering Practice

Generic software process model composed of a set of activities that establish a framework for software engineering practice. Generic framework activities—**communication, planning, modeling, construction,** and **deployment**—and umbrella activities establish a skeleton architecture for software engineering work.

The Essence of Practice

George Polya outlined the essence of problem solving, and consequently, the essence of software engineering practice:
1. *Understand the problem* (communication and analysis).
2. *Plan a solution* (modeling and software design).
3. *Carry out the plan* (code generation).
4. *Examine the result for accuracy* (testing and quality assurance).
In the context of software engineering, these commonsense steps lead to a series of essential questions

i. **Understand the problem.** It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think. Understanding isn't always that easy. It's worth spending a little time answering a few simple questions:
• ***Who has a stake in the solution to the problem?*** That is, who are the stakeholders?
• ***What are the unknowns?*** What data, functions, and features are required to properly solve the problem?
• ***Can the problem be compartmentalized?*** Is it possible to represent smaller problems that may be easier to understand?
• ***Can the problem be represented graphically?*** Can an analysis model be created?

ii. **Plan the solution.** Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

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

iii. **Carry out the plan.** The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

• ***Does the solution conform to the plan?*** Is source code traceable to the design model?

• ***Is each component part of the solution provably correct?*** Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

iv. **Examine the result.** You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

• ***Is it possible to test each component part of the solution?*** Has a reasonable testing strategy been implemented?

• ***Does the solution produce results that conform to the data, functions, and features that are required?*** Has the software been validated against all stakeholder requirements? It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a commonsense approach to software engineering will never lead you astray.

General Principles

David Hooker has proposed seven principles that focus on software engineering practice as a whole. They are reproduced in the following.

The First Principle: *The Reason It All Exists*

A software system exists for one reason: *to provide value to its users*. All decisions should be made with this in mind. Before specifying a system requirement, before noting a piece of system functionality, before determining the hardware platforms or development processes, ask yourself questions such as: "Does this add real value to the system?" If the answer is "no," don't do it. All other principles support this one.

The Second Principle: *KISS (Keep It Simple, Stupid!)*

Software design is not a random process. There are many factors to consider in any design effort. *All design should be as simple as possible, but no simpler*. This facilitates having a more easily understood and easily maintained system.

Features should be discarded in the name of simplicity. Indeed, the more elegant designs are usually the more simple ones. Simple also does not mean "quick and dirty." In fact, it often takes a lot of thought and work over multiple iterations to simplify. The payoff is software that is more maintainable and less error-prone.

The Third Principle: *Maintain the Vision*

A clear vision is essential to the success of a software project. Without one, a project almost unfailingly ends up being "of two [or more] minds" about itself. Without conceptual integrity, a system threatens to become a patchwork of incompatible designs, held together by the wrong kind of screws. . . . Compromising the architectural vision of a software system weakens and will eventually break even the well-designed systems. Having an empowered architect who can hold the vision and enforce compliance helps ensure a very successful software project.

The Fourth Principle: *What You Produce, Others Will Consume*

Seldom is an industrial-strength software system constructed and used in a vacuum. In some way or other, someone else will use, maintain, document, or otherwise depend on being able to understand your system. So, *always specify, design, and implement knowing someone*

else will have to understand what you are doing. The audience for any product of software development is potentially large.

Specify with an eye to the users. Design, keeping the implementers in mind. Code with concern for those that must maintain and extend the system. Someone may have to debug the code you write, and that makes them a user of your code. Making their job easier adds value to the system.

The Fifth Principle: *Be Open to the Future*

A system with a long lifetime has more value. In today's computing environments, where specifications change on a moment's notice and hardware platforms are obsolete just a few months old, software lifetimes are typically measured in months instead of years. However, true "industrial-strength" software systems must endure far longer. To do this successfully, these systems must be ready to adapt to these and other changes. Systems that do this successfully are those that have been designed this way from the start. *Never design yourself into a corner*.

Always ask "what if," and prepare for all possible answers by creating systems that solve the general problem, not just the specific one.14 This could very possibly lead to the reuse of an entire system. This advice can be dangerous if it is taken to extremes. Designing for the "general problem" sometimes requires performance compromises and can make specific solutions inefficient.

The Sixth Principle: *Plan Ahead for Reuse*

Reuse saves time and effort.15Achieving a high level of reuse is arguably the hardest goal to accomplish in developing a software system. The reuse of code and designs has been proclaimed as a major benefit of using object-oriented technologies. However, the return on this investment is not automatic. To leverage the reuse possibilities that object-oriented [or conventional] programming provides requires forethought and planning. There are many techniques to realize reuse at every level of the system development process *Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated*.

The Seventh principle: *Think!*

This last principle is probably the most overlooked. *Placing clear, complete thought before action almost always produces better results*. When you think about something, you are more likely to do it right. You also gain knowledge about how to do it right again. If you do think about something and still do it wrong, it becomes a valuable experience. A side effect of thinking is learning to recognize when you don't know something, at which point you can research the answer.

When clear thought has gone into a system, value comes out. Applying the first six principles requires intense thought, for which the potential rewards are enormous. If every software engineer and every software team simply followed Hooker's seven principles, many of the difficulties we experience in building complex computer based systems would be eliminated.

6. Software Myths

- 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. For instance, they appear to be reasonable statements of fact (sometimes containing elements of truth), they have an intuitive feel, and they are often announced by experienced practitioners who "know the score."
- 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.
- Software Myths are three types
  1. Managers Myths
  2. Customers Myths (and other non-technical stakeholders)
  3. Practitioners Myths

i. **Management myths.** Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality. Like a drowning person who grasps at a straw, a software manager often grasps at belief in a software myth, if that belief will lessen the pressure (even temporarily).

Myth: *We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?*
Reality: The book of standards may very well exist, but is it used?
Are software practitioners aware of its existence?
Does it reflect modern software engineering practice?
Is it complete?
Is it adaptable?
Is it streamlined to improve time-to-delivery while still maintaining a focus on quality?

In many cases, the answer to all of these questions is "no."

Myth: *If we get behind schedule, we can add more programmers and catch up.*
Reality: Software development is not a mechanistic process like manufacturing. In the words of Mr. Brooks "adding people to a late software project makes it later."

At first, this statement may seem counterintuitive. However, as new people are added, people who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort.

People can be added but only in a planned and well coordinated manner.

Myth: *If I decide to outsource the software project to a third party, I can just relax and let that firm build it.*
Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

ii. **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.

Myth: *A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.*
Reality: Although a comprehensive and stable statement of requirements is not always possible, an ambiguous "statement of objectives" is a recipe for disaster. Unambiguous requirements (usually derived iteratively) are developed only through effective and Continuous communication between customer and developer.

Myth: *Software requirements continually change, but change can be easily accommodated because software is flexible.*
Reality: It is true that software requirements change, but the impact of change varies with the time at which it is introduced. When requirements changes are requested early (before design or code has been started), the cost impact is relatively small. However, as time passes, the cost impact grows rapidly—resources have been committed, a design framework has been established, and change can requires additional resources and major design modification.

iii. **Practitioner's myths.** Myths that are still believed by software practitioners have been fostered by over 50 years of programming culture. During the early days, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: *Once we write the program and get it to work, our job is done.*
Reality: Someone once said that "the sooner you begin 'writing code,' the longer it'll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: *Until I get the program "running" I have no way of assessing its quality.*
Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—*the technical review.* Software reviews are a "quality filter" that have been found to be more effective than testing for finding certain classes of software defects.

Myth: *The only deliverable work product for a successful project is the working program.*
Reality: A working program is only one part of a software configuration that includes many elements. A variety of work products (e.g., models, documents, plans) provide a foundation for successful engineering and, more important, guidance for software support.

Myth: *Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.*
Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework. And reduced rework results in faster delivery times. Many software engineers have adopted an "agile" approach that accommodates change incrementally, thereby controlling its impact and cost.

SOFTWARE ENGINEERING
UNIT-1B

7. A Generic Process Model

Defining a Framework Activity

Identifying a Task Set

Process Patterns

8. Process Assessment and Improvement

9. Prescriptive Process Models

The Waterfall Model

Incremental Process Models

Evolutionary Process Models

Concurrent Models

10. Specialized Process Models

Component-Based Development

The Formal Methods Model

Aspect-Oriented Software Development

11. The Unified Process

A Brief History

Phases of the Unified Process

12. Personal and Team Process Models

Personal Software Process (PSP)

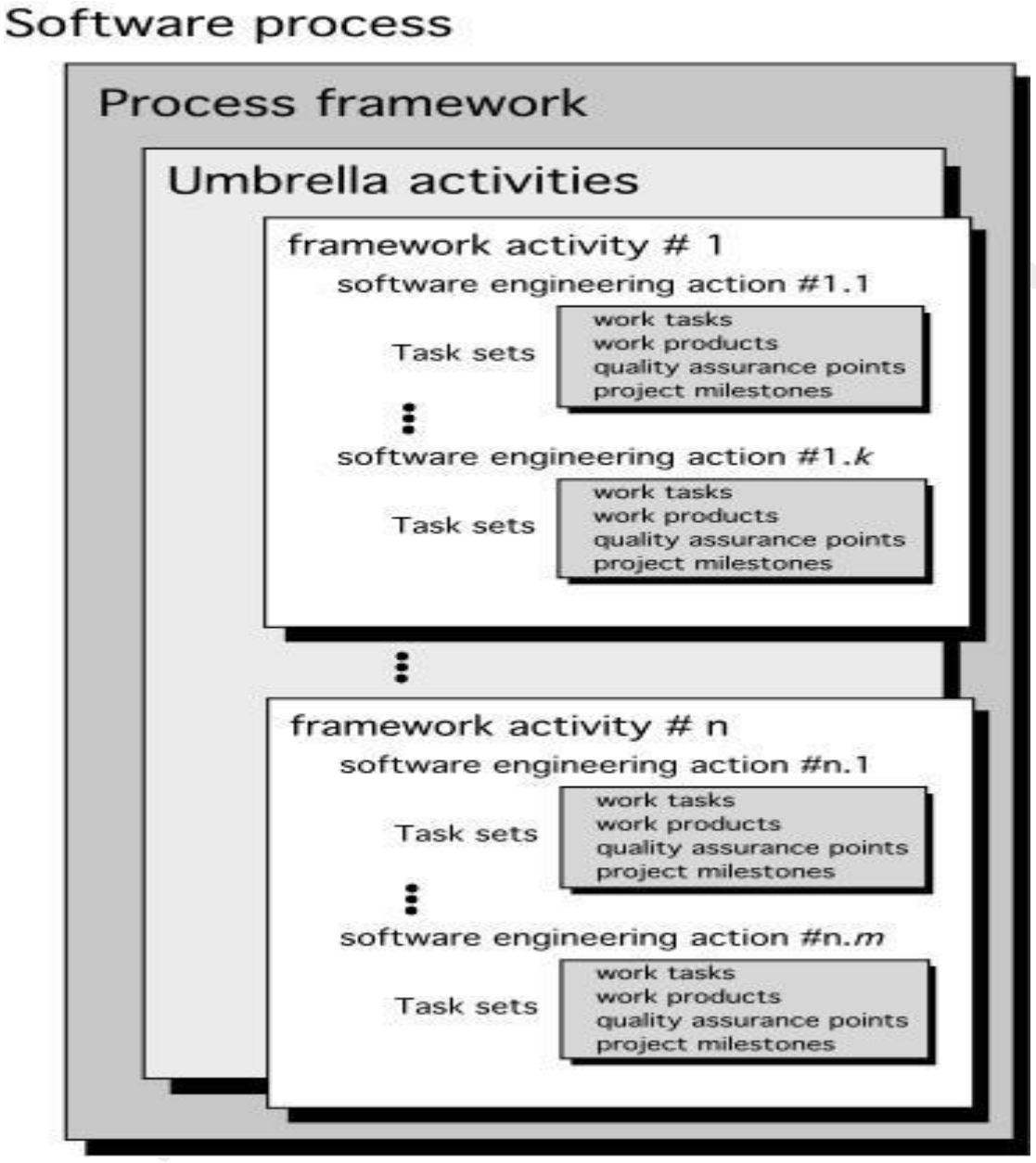
Team Software Process (TSP)

13. Process Technology

14. Product and Process

7. A GENERIC PROCESS MODEL

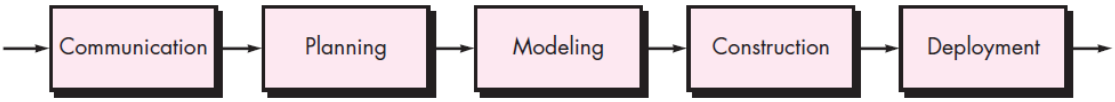
A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.



- The software process is represented schematically in the above figure.
- Referring to the figure, each framework activity is populated by a set of software engineering actions.
- Each software engineering action is defined by a *task set* that identifies
  - the work tasks that are to be completed,
  - the work products that will be produced,
  - the quality assurance points that will be required, and
  - the milestones that will be used to indicate progress.

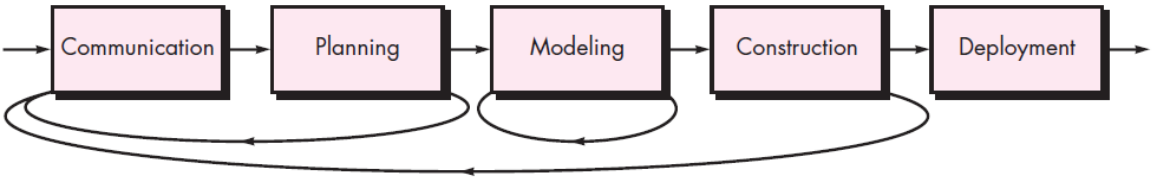
- A generic process framework for software engineering defines five framework activities
  - **communication,**
  - **planning,**
  - **modeling,**
  - **construction,** and
  - **deployment.**
- In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process.

- The *process flow*—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time and is illustrated in the above Figure.
- A linear *process flow* executes each of the five framework activities in sequence, beginning with communication and culminating with deployment which is shown in the following Figure.



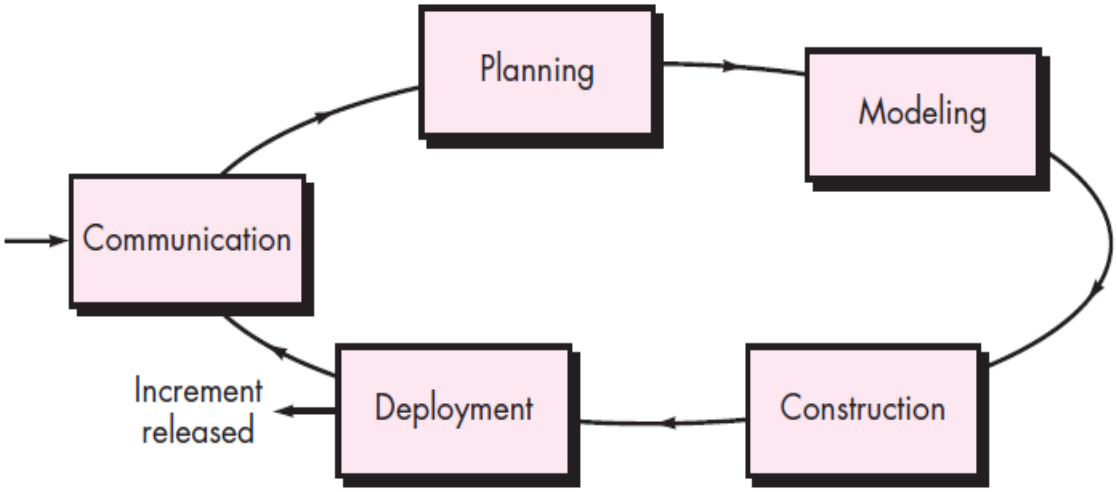
(a) Linear process flow

- An *iterative process flow* repeats one or more of the activities before proceeding to the next (shown in the following Figure).



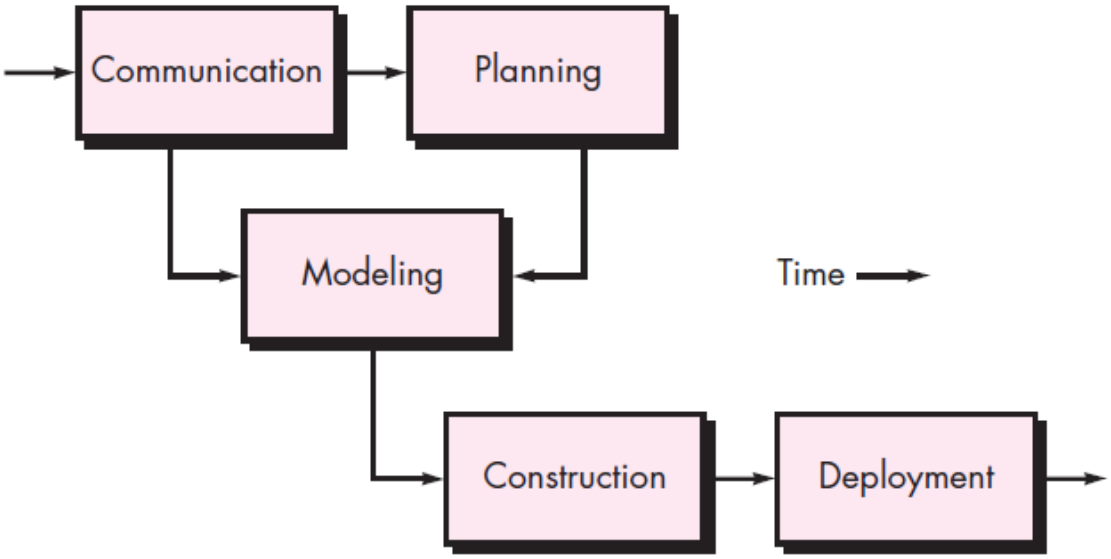
(b) Iterative process flow

- An *evolutionary process flow* executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software (shown in the following Figure).



(c) Evolutionary process flow

- A *parallel process flow* executes one or more activities in parallel with other activities
  e.g., modeling for one aspect of the software might be executed in parallel with construction of another aspect of the software. (shown in the following figure)



(d) Parallel process flow

Defining a Framework Activity

There are five framework activities, they are

- **communication,**
- **planning,**
- **modeling,**
- **construction,** and
- **deployment.**

These five framework activities provide a basic definition of Software Process. These Framework activities provides basic information like *What actions are appropriate for a framework activity, given the nature of the problem to be solved, the characteristics of the people doing the work, and the stakeholders who are sponsoring the project?*

1. Make contact with stakeholder via telephone.

2. Discuss requirements and take notes.

3. Organize notes into a brief written statement of requirements.

4. E-mail to stakeholder for review and approval.

If the project was considerably more complex with many stakeholders, each with a different set of requirements, the communication activity might have six distinct actions: *inception, elicitation, elaboration, negotiation, specification,* and *validation.* Each of these software engineering actions would have many work tasks and a number of distinct work products.

Identifying a Task Set

- Each software engineering action can be represented by a number of different *task sets*—
- Each a collection of software engineering
  - work tasks,
  - related work products,
  - quality assurance points, and
  - project milestones.
- Choose a task set that best accommodates the needs of the project and the characteristics of software team.
- This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.

Process Patterns

- Every software team encounters problems as it moves through the software process.
- It would be useful if proven solutions to these problems were readily available to the team so that the problems could be addressed and resolved quickly.
- A *process pattern*1 describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a template a consistent method for describing problem solutions within the context of the software process.
- By combining patterns, a software team can solve problems and construct a process that best meets the needs of a project.
- Patterns can be defined at any level of abstraction.
- In some cases, a pattern might be used to describe a problem and solution associated with a complete process model (e.g., prototyping).

- In other situations, patterns can be used to describe a problem and solution associated with a framework activity (e.g., **planning**) or an action within a framework activity (e.g., project estimating).
- Ambler has proposed a template for describing a process pattern:

Pattern Name. The pattern is given a meaningful name describing it within the context of the software process (e.g., **TechnicalReviews**).

Forces (Environment). The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type. The pattern type is specified. Ambler suggests three types:

1. ***Stage pattern***—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity).

An example of a stage pattern might be **EstablishingCommunication.** This pattern would incorporate the task pattern **RequirementsGathering** and others.

2. ***Task pattern***—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **RequirementsGathering** is a task pattern).

3. ***Phase pattern***—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be **SpiralModel** or **Prototyping.**

Initial context. Describes the conditions under which the pattern applies. Prior to the initiation of the pattern:

(1) What organizational or team-related activities have already occurred?
(2) What is the entry state for the process?
(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 **Communication** pattern has occurred; and
(3) the project scope, basic business requirements, and project constraints are known.

Problem. The specific problem to be solved by the pattern.

Solution. Describes how to implement the pattern successfully. This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern.

Resulting Context. Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:

(1) What organizational or team-related activities must have occurred?
(2) What is the exit state for the process?

(3) What software engineering information or project information has been developed?

Related Patterns. Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form. For example, the stage pattern **Communication** encompasses the task patterns:
ProjectTeam, CollaborativeGuidelines, ScopeIsolation, RequirementsGathering, ConstraintDescription, and **ScenarioCreation.**

Known Uses and Examples. Indicate the specific instances in which the pattern is applicable. For example, **Communication** is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

Conclusion on Process Patterns

- Process patterns provide an effective mechanism for addressing problems associated with any software process.
- The patterns enable you to develop a hierarchical process description that begins at a high level of abstraction (a phase pattern).
- The description is then refined into a set of stage patterns that describe framework activities
- Once process patterns have been developed, they can be reused for the definition of process variants..

8. PROCESS ASSESSMENT AND IMPROVEMENT

- The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs.
- Process patterns must be coupled with solid software engineering practice
- In addition, the process itself can be assessed 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 and improvement 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 five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.
2. **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**— provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.
3. **SPICE (ISO/IEC15504)**—a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.
4. **ISO 9001:2000 for Software**—a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

9. PRESCRIPTIVE PROCESS MODELS

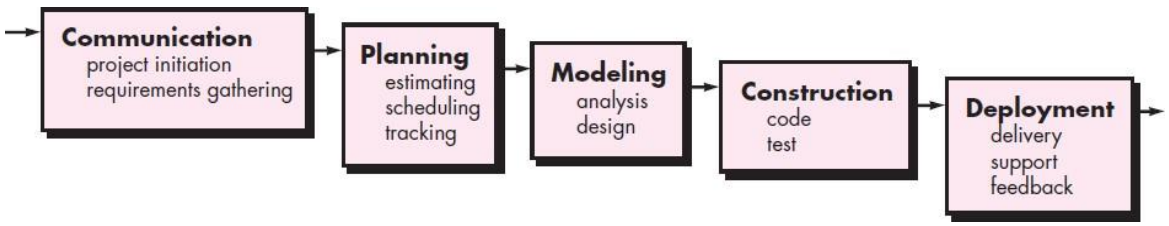
- Prescriptive process models were originally proposed to bring order to the chaos (disorder) of software development. History
- these models have brought a certain amount of useful structure to software engineering work and have provided a reasonably effective road map for software teams.
- The edge of chaos is defined as "a natural state between order and chaos, a grand compromise between structure and surprise".
- The edge of chaos can be visualized as an unstable, partially structured state.
- It is unstable because it is constantly attracted to chaos or to absolute order.
- The prescriptive process approach in which order and project consistency are dominant issues.
- "prescriptive" means prescribe a set of process elements
    - framework activities,
    - software engineering actions,
    - tasks,
    - work products,
    - quality assurance, and
    - change control mechanisms for each project.
- Each process model also prescribes a process flow (also called a *work flow*)—that is, the manner in which the process elements are interrelated to one another.
- All software process models can accommodate the generic framework activities, but each applies a different emphasis to these activities and defines a process flow that invokes each framework activity in a different manner.

The Waterfall Model

The waterfall model, sometimes called the *classic life cycle*, suggests a systematic sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment.

Context: Used when requirements are reasonably well understood.

Advantage: It can serve as a useful process model in situations where requirements are fixed and work is to proceed to complete in a linear manner.



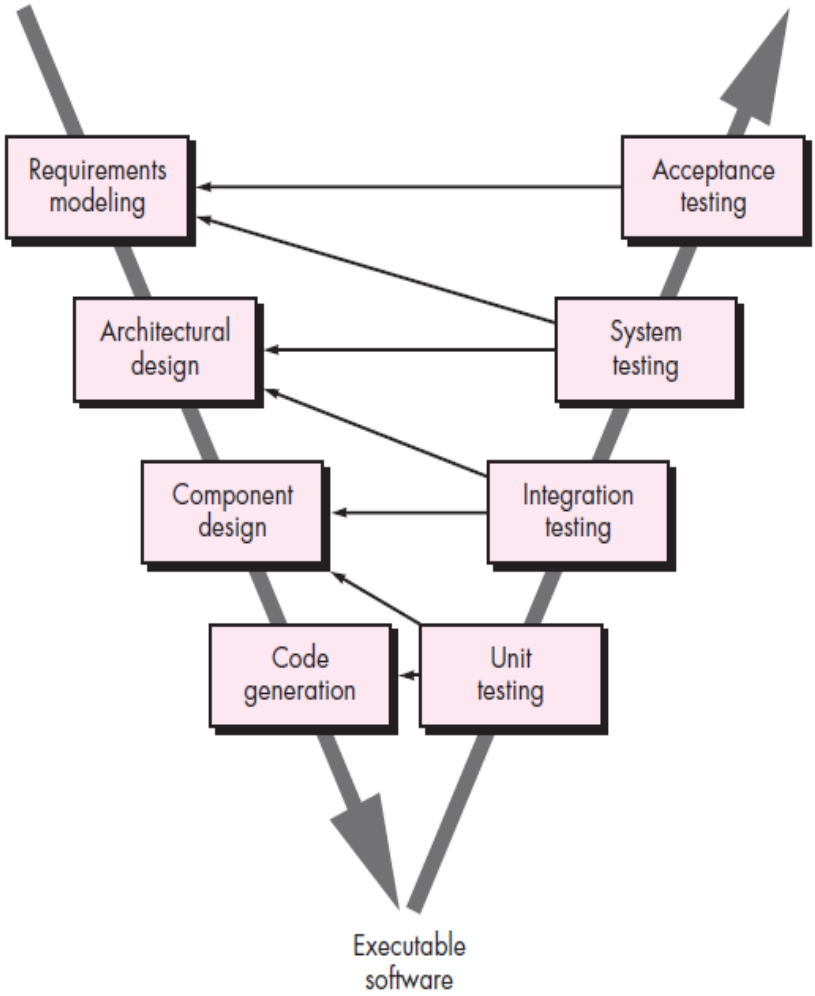
The **problems** that are sometimes encountered when the ***waterfall model*** is applied are:

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

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

iii. The customer must have patience. A working version of the programs will not be available until late in the project time-span. If a major blunder is undetected then it can be disastrous until the program is reviewed.

9.1.2V-model



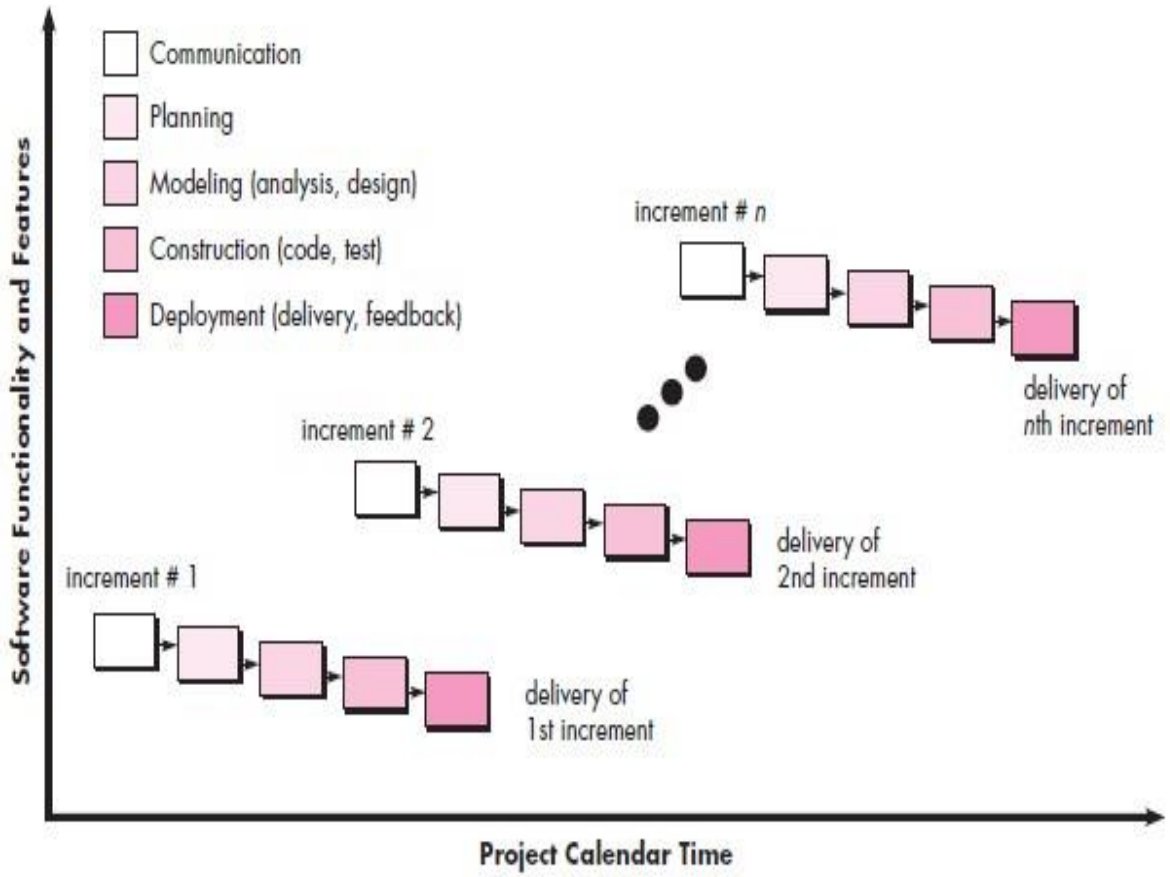
- A variation in the representation of the waterfall model is called the *V-model*.

Represented in the above Figure.

- The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.
- As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.

- Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.
- In reality, there is no fundamental difference between the classic life cycle and the V-model.
- The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

Incremental Process Models



- There are many situations in which initial software requirements are reasonably well defined, but the overall scope of the development effort difficulty to implement linear process.
- Need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases.
- In such cases, you can choose a process model that is designed to produce the software in increments.
- The *incremental* model combines elements of linear and parallel process flows. The above Figure shows the incremental model which applies linear sequences
- Each linear sequence produces deliverable "increments" of the software in a manner that is similar to the increments produced by an evolutionary process flow.
- For example, MS-Word software developed using the incremental paradigm might deliver
    - basic file management, editing, and document production functions in the first increment;

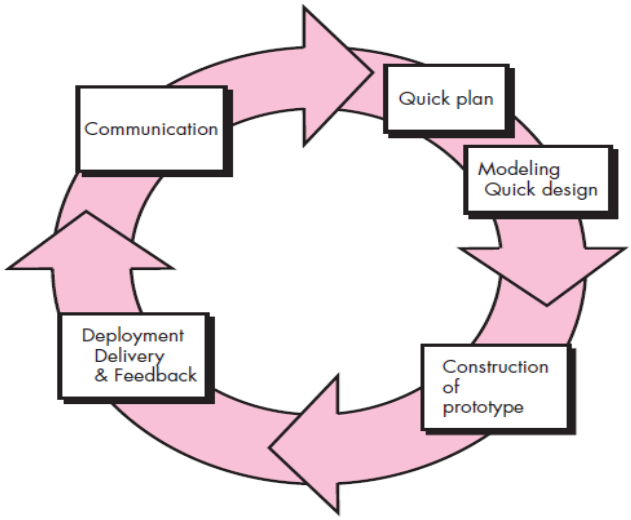
  - more sophisticated editing and document production capabilities in the second increment;
  - spelling and grammar checking in the third increment; and
  - advanced page layout capability in the fourth increment.
  - It should be noted that the process flow for any increment can incorporate the prototyping paradigm.
- When an incremental model is used, the first increment is often a *core product.* That is, basic requirements are addressed but many supplementary features remain undelivered. The core product is used by the customer. As a result of use evaluation, a plan is developed for the next increment.
- The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.
- The incremental process model focuses on the delivery of an operational product with each increment.
- Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.
- Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.
- Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment.

Evolutionary Process Models

Evolutionary process models produce with each iteration produce an increasingly more complete version of the software with every iteration.

Evolutionary models are iterative. They are characterized in a manner that enables software engineers to develop increasingly more complete versions of the software.

Prototyping.



- Prototyping is more commonly used as a technique that can be implemented within the context of anyone of the process model.

- The prototyping paradigm begins with communication. The software engineer and customer meet and define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- Prototyping iteration is planned quickly and modeling occurs. The quick design leads to the construction of a prototype. The prototype is deployed and then evaluated by the customer/user.
- Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.

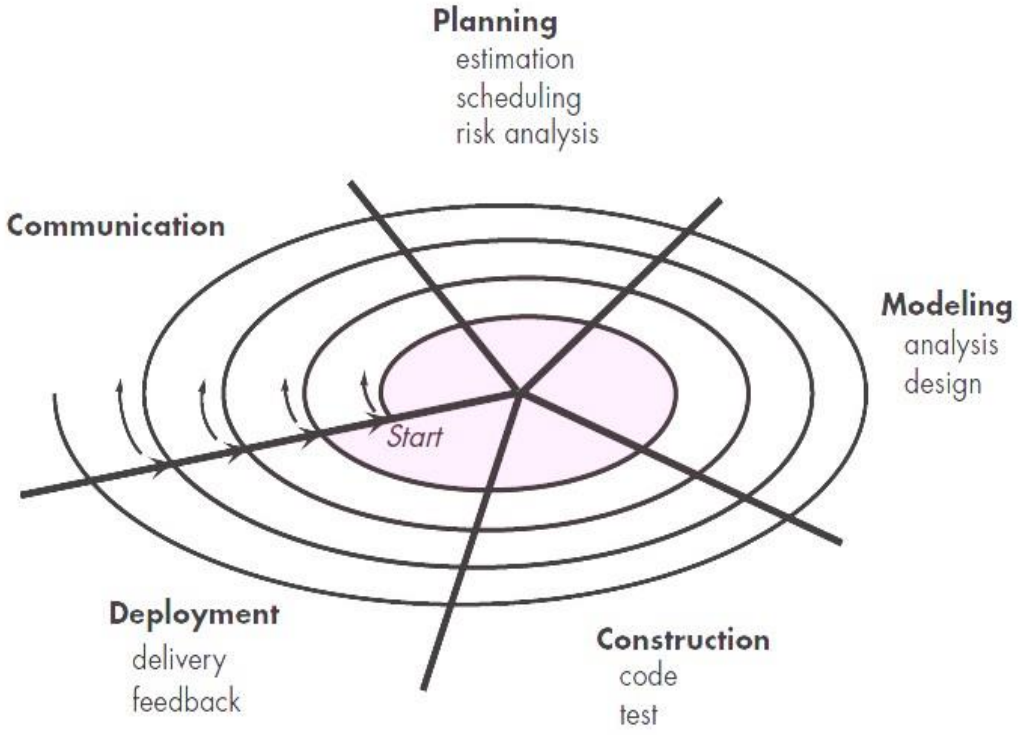
Example:

- If a customer defines a set of general objectives for software, but does not identify detailed input, processing, or output requirements, in such situation *prototyping* paradigm is best approach.
- If a developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system then he can go for this *prototyping* method.

Advantages:

- The prototyping paradigm assists the software engineer and the customer to better understand what is to be built when requirements are fuzzy.
- The prototype serves as a mechanism for identifying software requirements. If a working prototype is built, the developer attempts to make use of existing program fragments or applies tools.
- Prototyping can be **problematic** for the following reasons:
- The customer sees what appears to be a working version of the software, unaware that the prototype is held together "with chewing gum and baling wire", unaware that in the rush to get it working we haven't considered overall software quality or long-term maintainability.
- When informed that the product must be rebuilt so that high-levels of quality can be maintained, the customer cries foul and demands that "a few fixes" be applied to make the prototype a working product. Too often, software development relents.
- The developer often makes implementation compromises in order to get a prototype working quickly. An inappropriate operating system or programming language may be used simply because it is available and known; an inefficient algorithm may be implemented simply to demonstrate capability. After a time, the developer may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

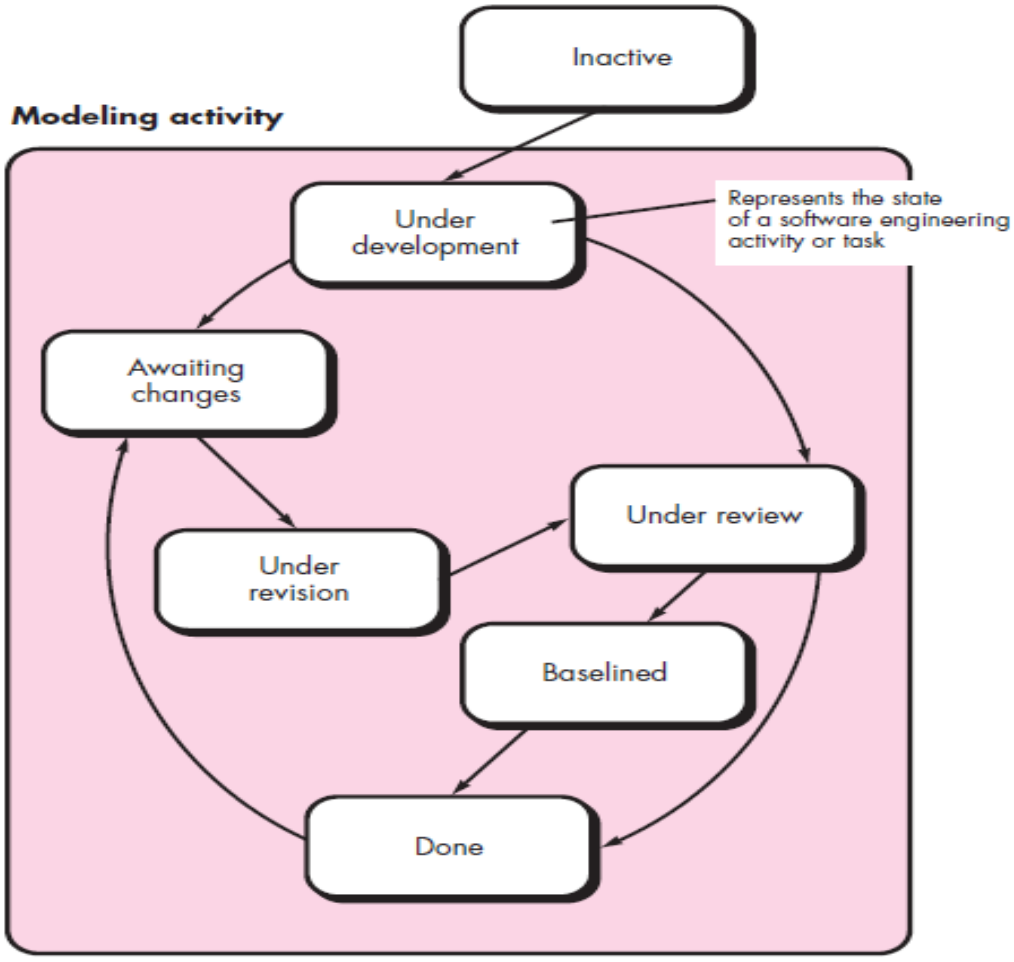
The Spiral Model.



- The Spiral model is proposed by Barry Boehm.
- The *spiral model* is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- It provides the potential for rapid development of increasingly more complete versions of the software.
- Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are
- **Anchor point milestones**- a combination of work products and conditions that are attained along the path of the spiral- are noted for each evolutionary pass.
- The first circuit around the spiral might result in the development of product pecification; 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 the customer after delivery. In addition, the project manager adjusts the planned number of iterations required to complete the software.
- It maintains the systematic stepwise approach suggested by the classic life cycle but incorporates it into an iterative framework that more realistically reflects the real world.
- The first circuit around the spiral might represent a "**concept development project**" which starts at the core of the spiral and continues for multiple iterations until concept development is complete.

- If the concept is to be developed into an actual product, the process proceeds outward on the spiral and a "**new product development project**" commences.
- Later, a circuit around the spiral might be used to represent a "**product enhancement project**." In essence, the spiral, when characterized in this way, remains operative until the software is retired.

Concurrent Models



- 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.
- The above 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 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.
- 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.

10. SPECIALIZED PROCESS MODELS

- Specialized process models take on many of the characteristics of one or more of the traditional models.

Component-Based Development

- Commercial off-the-shelf (COTS) **software components**, developed by vendors who offer them as **products**, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built.
- The *component-based development 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 **constructs applications from prepackaged software components.**
- Modeling and construction activities begin with the identification of **candidate components**.
- These **candidate components** can be designed as either conventional software modules or object-oriented classes or packages of classes.
- Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps:
  1. Available component-based products are researched and evaluated for the application domain in question.
  2. Component integration issues are considered.
  3. A software architecture is designed to accommodate the components.
  4. Components are integrated into the architecture.
  5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits.

The Formal Methods Model

- The *formal methods model* encompasses a set of activities that leads to formal mathematical specification of computer software.
- Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation.
- A variation on this approach, called *clean room software engineering*, is currently applied by some software development organizations.
- 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.
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily—not through the review, but through the application of mathematical analysis.
- When formal methods are used during design, they serve as a basis for program verification and "discover and correct errors" that might otherwise go undetected.
- 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 Software Development

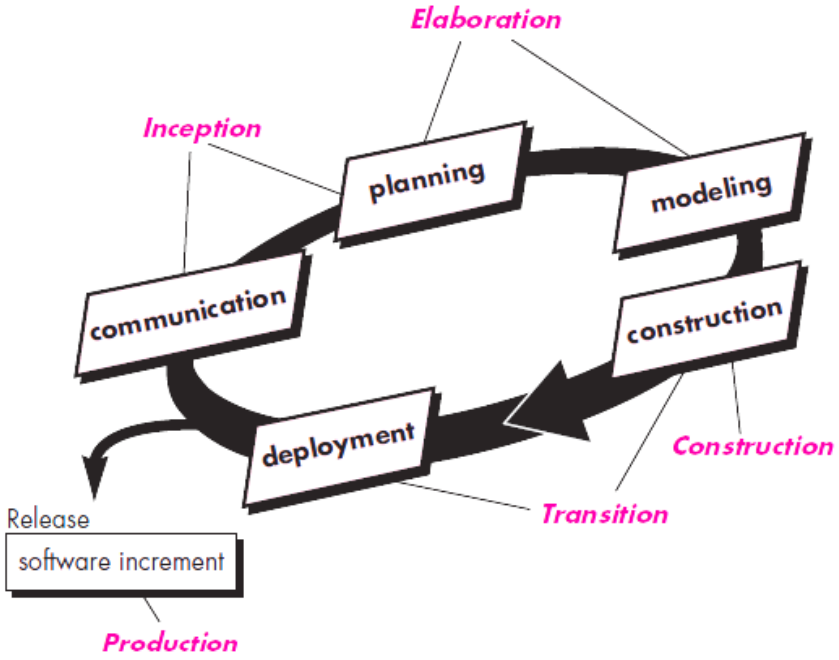
- Localized software characteristics are modeled as components (e.g., objectoriented classes) and then constructed within the context of a system architecture.
- As modern computer-based systems become more sophisticated (and complex), certain *concerns*—customer required properties or areas of technical interest—span the entire architecture.
- Some concerns are high-level properties of a system (e.g., security, fault tolerance). Other concerns affect functions (e.g., the application of business rules), while others are systemic (e.g., task synchronization or memory management).
- 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 an impact across the software architecture.
- ***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*—"mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern".
- ***aspect-oriented component engineering*** **(AOCE):** AOCE uses a concept of horizontal slices through vertically-decomposed software components, called "aspects".
- Components may provide or require one or more "aspect details" relating to a particular aspect, such as a viewing mechanism, extensible affordance and interface kind (user interface aspects);
    - event generation,
    - transport and receiving (distribution aspects);
    - data store/retrieve and indexing (persistency aspects);
    - authentication,
    - encoding and access rights (security aspects);

  - transaction atomicity,
  - concurrency control and logging strategy (transaction aspects); and so on.
- Each aspect detail has a number of properties, relating to functional and/or non-functional characteristics of the aspect detail.
- The evolutionary model is appropriate as aspects are identified and then constructed.
- The parallel nature of concurrent development is essential because aspects are engineered independently.

11. THE UNIFIED PROCESS

- The unified process related to "use case driven, architecture-centric, iterative and incremental" software process.
- The Unified Process is an attempt to draw on the best features and characteristics of traditional software process models.
- 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.
- It suggests a process flow that is iterative and incremental.
- During the early 1990s James Rumbaugh [Rum91], Grady Booch [Boo94], and Ivar Jacobson [Jac92] began working on a "unified method".
- The result was UML—a *unified modeling language* that contains a robust notation for the modeling and development of object-oriented systems.
- By 1997, UML became a de facto industry standard for object-oriented software development.
- UML is used to represent both requirements and design models.
- UML provided the necessary technology to support object-oriented software engineering practice, but it did not provide the process framework.
- Over the next few years, Jacobson, Rumbaugh, and Booch developed the *Unified Process*, a framework for object-oriented software engineering using UML.
- Today, the Unified Process (UP) and UML are widely used on object-oriented projects of all kinds.
- The iterative, incremental model proposed by the UP can and should be adapted to meet specific project needs.

Phases of the Unified Process



- The above figure depicts the different phases in Unified Process.
- The ***inception phase*** of the UP encompasses both customer communication and planning activities.
    - By collaborating with stakeholders, business requirements for the software are identified;
    - a rough architecture for the system is proposed; and
    - a plan for the iterative, incremental nature of the ensuing project is developed.
    - Fundamental business requirements are described.
    - The architecture will be refined.
    - Planning identifies resources, assesses major risks, defines a schedule, and establishes a basis for the phases.
- The **elaboration phase** encompasses the communication and modeling activities of the generic process model.
    - Elaboration refines and expands the preliminary use cases that were developed as part of the inception phase and expands the architectural representation to include five different views of the software—the use case model, the requirements model, the design model, the implementation model, and the deployment model.
    - In some cases, elaboration creates an "executable architectural baseline" that represents a "first cut" executable system.
    - The architectural baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system.
    - In addition, the plan is carefully reviewed.
    - Modifications to the plan are often made at this time.
- The ***construction phase*** of the UP is identical to the construction activity defined for the generic software process. Using the architectural model as input, the construction phase develops or acquires the software components that will make each use case operational for end users.
    - The elaboration phase reflect the final version of the software increment.

  - All necessary and required features and functions for the software increment are then implemented in source code.
  - As components are being implemented, unit tests are designed and executed for each.
  - In addition, integration activities are conducted.
  - Use cases are used to derive a suite of acceptance tests that are executed prior to the initiation of the next UP phase.
- The ***transition phase*** of the UP encompasses the latter stages of the generic construction activity and the first part of the generic deployment (delivery and feedback) activity.
  - Software is given to end users for beta testing and user feedback reports both defects and necessary changes.
  - In addition, the software team creates the necessary support information (e.g., user manuals, troubleshooting guides, installation procedures) that is required for the release.
  - At the conclusion of the transition phase, the software increment becomes a usable software release.
- The ***production phase*** of the UP coincides with the deployment activity of the generic process.
  - During this phase, the ongoing use of the software is monitored, support for the operating environment (infrastructure) is provided, and defect reports and requests for changes are submitted and evaluated.
  - It is likely that at the same time the construction, transition, and production phases are being conducted.
  - Work may have already begun on the next software increment.
  - This means that the five UP phases do not occur in a sequence, but rather with staggered concurrency.

- A software engineering workflow is distributed across all UP phases.
- In the context of UP, a *workflow* is a task set
- That is, a workflow identifies the tasks required to accomplish an important software engineering action and the work products that are produced as a consequence of successfully completing the tasks.
- It should be noted that not every task identified for a UP workflow is conducted for every software project.
- The team adapts the process (actions, tasks, subtasks, and work products) to meet its needs.

12. PERSONAL AND TEAM PROCESS MODELS

- Software process model has been developed at a corporate or organizational level.
- It can be effective only if it is helpful to significant adaptation to meet the needs of the project team that is actually doing software engineering work.
- In an ideal setting, it create a process that best fits your needs, and at the same time, meets the broader needs of the team and the organization.
- Alternatively, the team itself can create its own process, and at the same time meet the narrower needs of individuals and the broader needs of the organization.
- It is possible to create a “personal software process” and/or a “team software process.”
- Both require hard work, training, and coordination, but both are achievable.

Personal Software Process (PSP)

- Every developer uses some process to build computer software.
- The process may be temporary; may change on a daily basis; may not be efficient, effective, or even successful; but a "process" does exist.
- Personal process, an individual must move through four phases, each requiring training and careful instrumentation.
- The *Personal Software Process* (PSP) emphasizes personal measurement of both the work product that is produced and the resultant quality of the work product.
- In addition PSP makes the practitioner responsible for project planning (e.g., estimating and scheduling) and empowers the practitioner to control the quality of all software work products that are developed.
- The PSP model defines five framework activities:

  1. 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.

  2. 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.

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

  4. 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.

  5. 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.
- PSP stresses the need to identify errors early and, just as important, to understand the types of errors that you are likely to make. This is accomplished through a rigorous assessment activity performed on all work products you produce.
- PSP represents a disciplined, metrics-based approach to software engineering that may lead to culture shock for many practitioners. However, when PSP is properly introduced to software engineers [Hum96], the resulting improvement in software engineering productivity and software quality are significant [Fer97].
- However, PSP has not been widely adopted throughout the industry. The reasons, sadly, have more to do with human nature and organizational inertia than they do with the strengths and weaknesses of the PSP approach.
- PSP is intellectually challenging and demands a level of commitment (by practitioners and their managers) that is not always possible to obtain. Training is relatively lengthy, and training costs are high.
- The required level of measurement is culturally difficult for many software people. Can PSP be used as an effective software process at a personal level? The answer is an unequivocal "yes." But even if PSP is not adopted in its entirely, many of the personal process improvement concepts that it introduces are well worth learning.

Team Software Process (TSP)

- *Team Software Process* (TSP) build a "self-directed" project team that organizes itself to produce high-quality software.
- 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 CMM Level 5 behavior normal and expected.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.
- A self-directed team has a consistent understanding of its overall goals and objectives; defines roles and responsibilities for each team member; tracks quantitative project data (about productivity and quality);
- TSP identifies a team process that is appropriate for the project and a strategy for implementing the process;
- TSP defines local standards that are applicable to the team's software engineering work; continually assesses risk and reacts to it; and tracks, manages, and reports project status.
- TSP defines the following framework activities:
    - **project launch,**
    - **high-level design,**
    - **implementation,**
    - **integration and test,** and
    - **postmortem.**
- These activities enable the team to plan, design, and construct software in a disciplined manner while at the same time quantitatively measuring the process and the product.
- The postmortem sets the stage for process improvements.
- TSP makes use of a wide variety of scripts, forms, and standards that serve to guide team members in their work.
- TSP recognizes that the best software teams are self-directed.
- Team members set
    - project objectives,
    - adapt the process to meet their needs,
    - control the project schedule, and
    - analysis of the metrics collected,
    - work continually to improve the team's approach to software engineering.
- Like PSP, TSP is a rigorous approach to software engineering that provides distinct and quantifiable benefits in productivity and quality.
- The team must make a full commitment to the process and must undergo thorough training to ensure that the approach is properly applied.

13. PROCESS TECHNOLOGY

- ***Process technology tools*** have been developed to help software organizations analyze their current process, organize work tasks, control and monitor progress, and manage technical quality.
- ***Process technology tools*** allow a software organization to build an automated model of the process framework, task sets, and umbrella activities.
- The model, normally represented as a network, can then be analyzed to determine typical workflow and examine alternative process structures that might lead to reduced development time or cost.
- Once an acceptable process has been created, other process technology tools can be used to allocate, monitor, and even control all software engineering activities, actions, and tasks defined as part of the process model.
- Each member of a software team can use such tools to develop a checklist of work tasks to be performed, work products to be produced, and quality assurance activities to be conducted.
- The process technology tool can also be used to coordinate the use of other software engineering tools that are appropriate for a particular work task.

PRODUCT AND PROCESS

- If the process is weak, the end product will undoubtedly suffer.
- But an obsessive overreliance on process is also dangerous.
- Product is final developed software
- Process is set of activities, actions and tasks to develop product.
- structured programming languages (product) followed by structured analysis methods (process) followed by data encapsulation (product) followed by the current emphasis on the Software Engineering Institute's Software Development Capability Maturity Model (process).
- These swings are harmful in and of themselves because they confuse the average software practitioner.
- So software analyst must concentrate on Product by streamline the Process.
- All of human activity may be a process, but each of us derives a sense of self-worth from those activities that result (Product) in a representation.
- Thinking of a reusable artifact as only product or only process either obscures the context and ways to use it or obscures the fact that each use results in product that will, in turn, be used as input to some other software development activity.
- The duality of product and process is one important element in keeping creative Speople engaged as software engineering continues to evolve.

LECTURE NOTES

ON

SOFTWARE ENGINEERING

2020 – 2021

II B. Tech I Semester (R19)

Mr. **P. Suresh Babu,** **Assistant Professor**

UNIT- 2

Agile Development: Agility , Agility and the Cost of Change, Agile Process, Extreme Programming, Other Agile Process Models. A Tool Set for the Agile Process, Software Engineering Knowledge, Core Principles, Principles That Guide Each Framework Activity,

Understanding Requirements: Requirements Engineering, Establishing the groundwork, Eliciting Requirements, Developing Use Cases, Building the requirements model, Negotiating Requirements, Validating Requirements.

AGILE DEVELOPMENT

WHAT IS AGILITY?

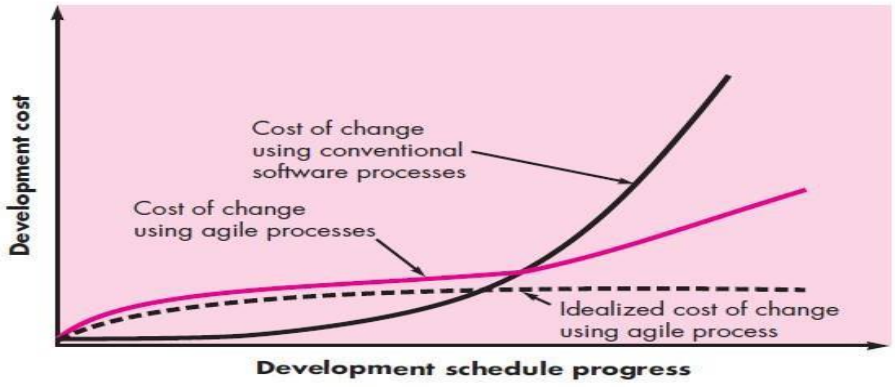
Agile is a software development methodology to build software incrementally using short iterations of 1 to 4 weeks so that the development process is aligned with the changing business needs.

An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

AGILITY AND THE COST OF CHANGE

An agile process reduces the cost of change because software is released in increments and change can be better controlled within an increment.

Agility argue that a well-designed agile process "flattens" the cost of change curve shown in following figure, allowing a software team to accommodate changes late in a software project without dramatic cost and time impact. When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming, the cost of making a change is attenuated. Although debate about the degree to which the cost curve flattens is ongoing, there is evidence to suggest that a significant reduction in the cost of change can be achieved.



AGILE PROCESS

Any agile software process is characterized in a manner that addresses a number of key assumptions about the majority of software projects:

1. It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
2. For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
3. Analysis, design, construction, and testing are not as predictable

Agility Principles

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

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*

- **Competence.** In an agile development 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.

- **Mutual trust and respect.** The agile team must become what DeMarco and Lister call a "jelled" team. 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."

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.

EXTREME PROGRAMMING (XP)

Extreme Programming **(XP),** the most widely used approach to agile software development, emphasizes business results first and takes an incremental, get-something-started approach to building the product, using continual testing and revision.

XP Values

Beck defines a set of **five** *values* that establish a foundation for all work performed as part of XP—**communication, simplicity, feedback, courage, and respect**. Each of these values is used as a driver for specific XP **activities, actions, and tasks**.

In order to achieve effective ***communication*** between software engineers and other stakeholders, XP emphasizes close, yet informal collaboration between customers and developers, the establishment of effective metaphors3 for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.

To achieve ***simplicity***, XP restricts developers to design only for immediate needs, rather than consider future needs. The intent is to create a simple design that can be easily implemented in code). If the design must be improved, it can be *refactored* at a later time.

Feedback is derived from three sources: the implemented software itself, the customer, and other software team members. By designing and implementing an effective testing strategy the software provides the agile team with feedback. XP makes use of the ***unit test*** as its primary testing tactic. As each class is developed, the team develops a unit test to exercise each operation according to its specified functionality.

Beck argues that strict adherence to certain XP practices demands ***courage.*** A better word might be ***discipline***. An agile XP team must have the discipline (courage) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.

By following each of these values, the agile team inculcates ***respect*** among it members, between other stakeholders and team members, and indirectly, for the software itself. As they

achieve successful delivery of software increments, the team develops growing respect for the XP process.

The XP Process

Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities: planning, design, coding, and testing. Following figure illustrates the XP process and notes some of the key ideas and tasks that are associated with each framework activity.

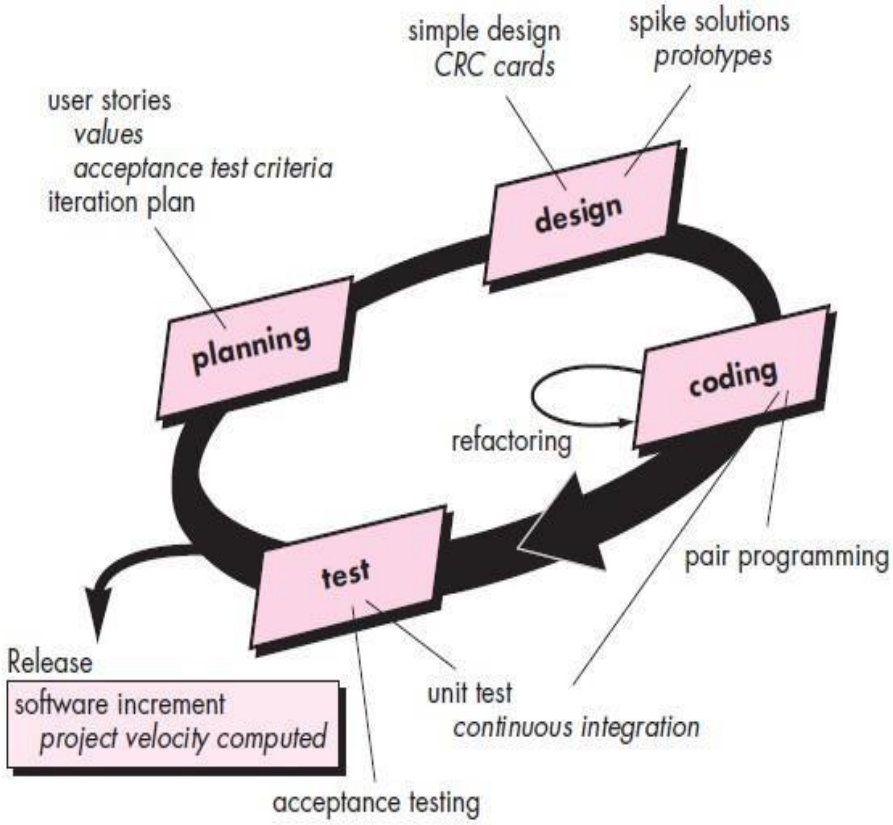


Fig : The Extreme Programming process

Key XP activities are

- **Planning.** The planning activity (also called *the planning game*) begins with *listening*—a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality.
- **Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation. In addition, the design provides

implementation guidance for a story as it is written—nothing less, nothing more. The design of extra functionality If a difficult design problem is encountered as part of the design of a story, XP recommends the immediate creation of an operational prototype of that portion of the design. Called a ***spike solution***, the design prototype is implemented and evaluated. XP encourages ***refactoring***—a construction technique that is also a method for design optimization.

Fowler describes r**efactoring** in the following manner: Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves the internal structure. It is a disciplined way to clean up code [that minimizes the chances of introducing bugs].

- **Coding.** After stories are developed and preliminary design work is done, the team does *not* move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

A key concept during the coding activity is *pair programming*. XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real time problem solving (two heads are often better than one) and real-time quality assurance.

- **Testing.** The creation of unit tests before coding commences is a key element of the XP approach. The unit tests that are created should be implemented using a framework that enables them to be automated. This encourages a regression testing strategy whenever code is modified. As the individual unit tests are organized into a "universal testing suite" integration and validation testing of the system can occur on a daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things go awry. Wells states: "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."

XP ***acceptance tests***, also called ***customer tests***, are specified by the customer and focus on overall system features and functionality that are visible and reviewable by the customer. Acceptance tests are derived from user stories that have been implemented as part of a software release.

Industrial XP

Joshua Kerievsky describes *Industrial Extreme Programming* (IXP) in the following manner: "IXP is an organic evolution of XP. It is imbued with XP's minimalist, customer-centric, test-driven spirit. IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices." IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

- **Readiness assessment.** Prior to the initiation of an IXP project, the organization should conduct a *readiness assessment.* The assessment ascertains whether (1) an appropriate development environment exists to support IXP, (2) the team will be populated by the proper set of stakeholders, (3) the organization has a distinct quality program and supports continuous improvement, (4) the organizational culture will support the new values of an agile team, and (5) the broader project community will be populated appropriately.
- **Project community.** Classic XP suggests that the right people be used to populate the agile team to ensure success. The implication is that people on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team. When XP is to be applied for a significant project in a large organization, the concept of the "team" should morph into that of a *community.* A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders (e.g., legal staff, quality auditors, manufacturing or sales types) who "are often at the periphery of an IXP project yet they may play important roles on the project". In IXP, the community members and their roles should be explicitly defined and mechanisms for communication and coordination between community members should be established.
- **Project chartering.** The IXP team assesses the project itself to determine whether an appropriate business justification for the project exists and whether the project will further the overall goals and objectives of the organization. Chartering also examines the context of the project to determine how it complements, extends, or replaces existing systems or processes.

- **Test-driven management.** An IXP project requires measurable criteria for assessing the state of the project and the progress that has been made to date. Test-driven management establishes a series of measurable "destinations" and then defines mechanisms for determining whether or not these destinations have been reached.
- **Retrospectives.** An IXP team conducts a specialized technical review after a software increment is delivered. Called a *retrospective,* the review examines "issues, events, and lessons-learned" across a software increment and/or the entire software release. The intent is to improve the IXP process.
- **Continuous learning.** Because learning is a vital part of continuous process improvement, members of the XP team are encouraged (and possibly, incented) to learn new methods and techniques that can lead to a higher quality product.

OTHER AGILE PROCESS MODELS

Other agile process models have been proposed and are in use across the industry.

Among the most common are:

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

Adaptive Software Development (ASD)

Adaptive Software Development (ASD) has been proposed by Jim Highsmith as a technique for building complex software and systems. The philosophical underpinnings of ASD focus on human collaboration and team self-organization.

High smith argues that an agile, adaptive development approach based on collaboration is "as much a source of *order* in our complex interactions as discipline and engineering." He defines an ASD "life cycle" that incorporates three phases, speculation, collaboration, and learning.

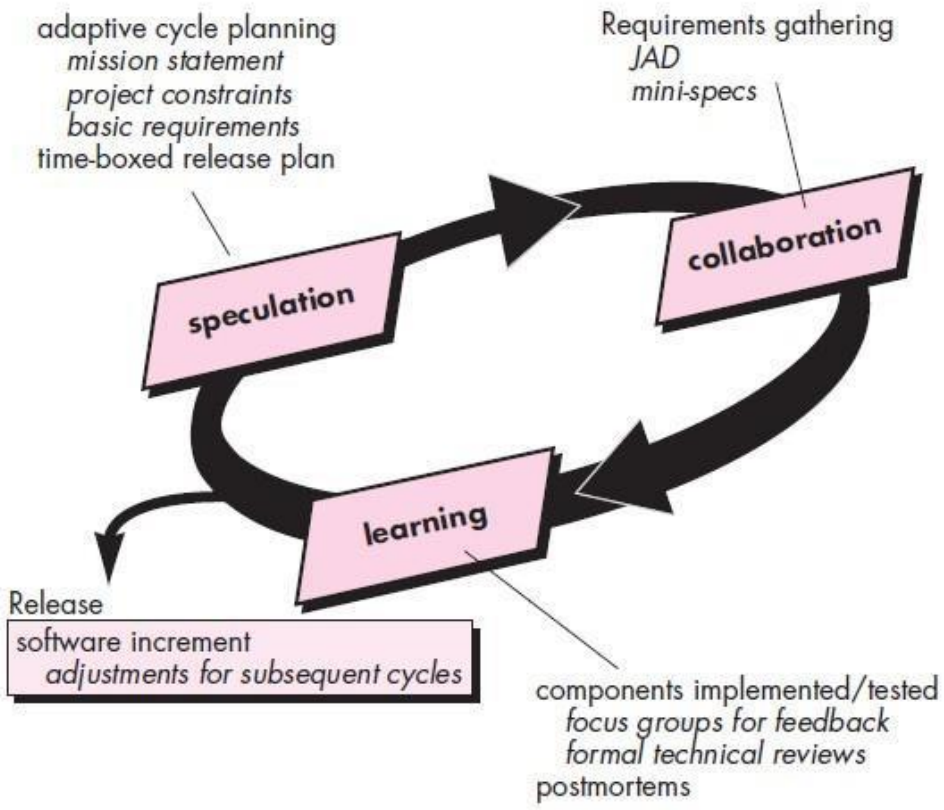


Fig : Adaptive software development

During ***speculation***, the 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 dates or user descriptions), and basic requirements—to define the set of release cycles (software increments) that will be required for the project.

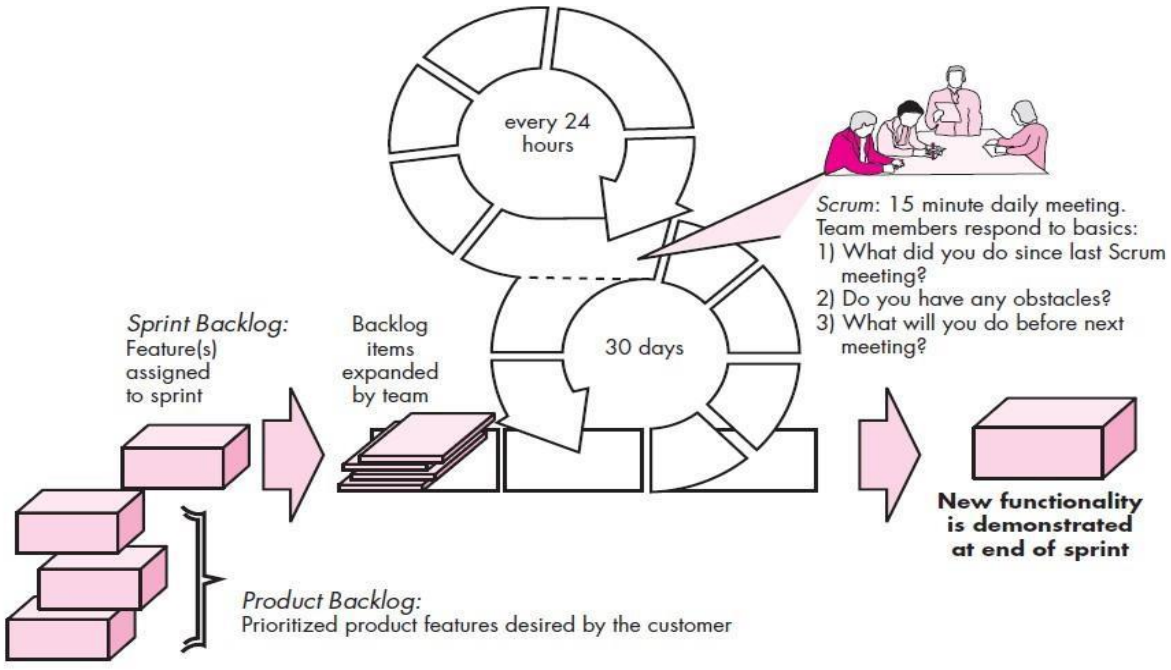
Motivated people use ***collaboration*** in a way that multiplies their talent and creative output beyond their absolute numbers. This approach is a recurring theme in all agile methods. But collaboration is not easy. It encompasses communication and teamwork, but it also emphasizes individualism, because individual creativity plays an important role in collaborative thinking. It is, above all, a matter of trust. People working together must trust one another to (1) criticize without animosity, (2) assist without resentment, (3) work as hard as or harder than they do, (4) have the skill set to contribute to the work at hand, and (5) communicate problems or concerns in a way that leads to effective action.

As members of an ASD team begin to develop the components that are part of an adaptive cycle, the emphasis is on **"learning**" as much as it is on progress toward a completed cycle.

ASD teams learn in **three** ways: **focus groups, technical reviews , and project postmortems**. ASD's overall emphasis on the dynamics of self-organizing teams, interpersonal collaboration, and individual and team learning yield software project teams that have a much higher likelihood of success.

Scrum

Scrum is an agile software development method that was conceived by Jeff Sutherland and his development team in the early 1990s. Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities: requirements, analysis, design, evolution, and delivery. Within each framework activity, work tasks occur within a process pattern called a ***sprint.*** The work conducted within a sprint is adapted to the problem at hand and is defined and often modified in real time by the Scrum team. The overall flow of the Scrum process is illustrated in following figure



Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality. Each of these process patterns defines a set of development actions:

- ***Backlog***—a prioritized list of project requirements or features that provide business value for the customer. Items can be added to the backlog at any time. The product manager assesses the backlog and updates priorities as required.
- ***Sprints***—consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days). Changes (e.g., backlog work items) are not introduced during the sprint. Hence, the sprint allows team members to work in a short-term, but stable environment.
- ***Scrum meetings***—are short (typically 15 minutes) meetings held daily by the Scrum team. Three key questions are asked and answered by all team members
    - What did you do since the last team meeting?
    - What obstacles are you encountering?
    - What do you plan to accomplish by the next team meeting?

A team leader, called a ***Scrum master***, leads the meeting and assesses the responses from each person. The Scrum meeting helps the team to uncover potential problems as early as possible. Also, these daily meetings lead to "**knowledge socialization**"

- ***Demos***—deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer. It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Dynamic Systems Development Method (DSDM)

The *Dynamic Systems Development Method* (DSDM) is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment" The DSDM philosophy is borrowed from a modified version of the **Pareto principle—80 percent of an application can be delivered in 20 percent of the time.** It would take to deliver the complete (100 percent) application. DSDM is an iterative software process in which each iteration follows the 80 percent rule. That is, only enough work is required for each increment to

facilitate movement to the next increment. The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

The *DSDM life cycle* that defines **three** different iterative cycles, preceded by **two** additional life cycle activities:

- ***Feasibility study***—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process
- ***Business study***—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.
- ***Functional model iteration***—produces a set of incremental prototypes that demonstrate functionality for the customer.
- ***Design and build iteration***—revisits prototypes built during ***functional model iteration*** to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users. In some cases, *functional model iteration* and *design and build iteration* occur concurrently.
- ***Implementation***—places the latest software increment into the operational environment. It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place. In either case, DSDM development work continues by returning to the functional model iteration activity.

Crystal

Alistair Cockburn and Jim Highsmith created the *Crystal family of agile methods* in order to achieve a software development approach that puts a premium on "maneuverability" during what Cockburn characterizes as "a resource limited, cooperative game of invention and communication, with a primary goal of delivering useful, working software and a secondary goal of setting up for the next game"

The Crystal family is actually a set of example agile processes that have been proven effective for different types of projects. The intent is to allow agile teams to select the member of the crystal family that is most appropriate for their project and environment.

Feature Driven Development (FDD)

Feature Driven Development (FDD) was originally conceived by Peter Coad and his colleagues as a practical process model for object-oriented software engineering. Stephen Palmer and John Felsing have extended and improvedCoad's work, describing an adaptive, agile process that can be applied to moderately sized and larger software projects.

Like other agile approaches, FDD adopts a philosophy that (1) emphasizes collaboration among people on an FDD team; (2) manages problem and project complexity using feature-based decomposition followed by the integration of software increments, and (3) communication of technical detail using verbal, graphical, and text-based means.

FDD emphasizes software quality assurance activities by encouraging an incremental development strategy, the use of design and code inspections, the application of software quality assurance audits, the collection of metrics, and the use of patterns (for analysis, design, and construction).

In the context of FDD, a *feature* "is a client-valued function that can be implemented in two weeks or less" The emphasis on the definition of features provides the following benefits:

- Because features are small blocks of deliverable functionality, users can describe them more easily; understand how they relate to one another more readily; and better review them for ambiguity, error, or omissions.
- Features can be organized into a hierarchical business-related grouping.
- Since a feature is the FDD deliverable software increment, the team develops operational features every two weeks.
- Because features are small, their design and code representations are easier to inspect effectively.
- Project planning, scheduling, and tracking are driven by the feature hierarchy, rather than an arbitrarily adopted software engineering task set.

Coad and his colleagues suggest the following template for defining a feature:

<action> the **<result>** **<by for of to>** a(n) **<object>**

where an **<object>** is "a person, place, or thing

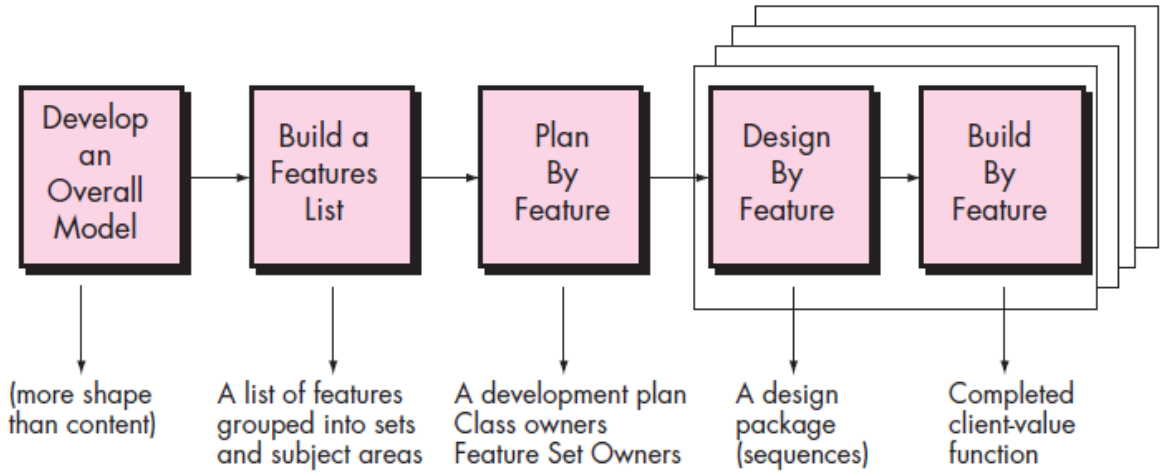


Fig : Feature Driven Development (FDD)

FDD provides greater emphasis on project management guidelines and techniques than many other agile methods. FDD defines **six** milestones during the design and implementation of a feature: "**design walkthrough, design, design inspection, code, code inspection, promote to build**"

Lean Software Development (LSD)

Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering. The lean principles that inspire the LSD process can be summarized as ***eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people,*** **and** ***optimize the whole***. Each of these principles can be adapted to the software process.

Agile Modeling (AM)

Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems. Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. Agile models are more effective than traditional models because they are just barely good, they don't have to be perfect.

Agile modeling adopts all of the values that are consistent with the agile manifesto. The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor. The team must also have the humility

to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Agile Modeling suggests a wide array of "core" and "supplementary" modeling principles, those that make AM unique are :

- **Model with a purpose.** A developer who uses AM should have a specific goal in mind before creating the model. Once the goal for the model is identified, the type of notation to be used and level of detail required will be more obvious.
- **Use multiple models.** There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.
- **Travel light.** As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest. Every work product that is kept must be maintained as changes occur. This represents work that slows the team down. Ambler notes that "Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner
- **Content is more important than representation.** Modeling should impart information to its intended audience. A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.
- **Know the models and the tools you use to create them.** Understand the strengths and weaknesses of each model and the tools that are used to create it.
- **Adapt locally.** The modeling approach should be adapted to the needs of the agile team.

Agile Unified Process (AUP)

The ***Agile Unified Process*** (AUP) adopts a "serial in the large" and "iterative in the small" philosophy for building computer-based systems. By adopting the classic UP phased activities—*inception, elaboration, construction,* and *transition*—AUP provides a serial overlay that enables a team to visualize the overall process flow for a software project. However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible. Each AUP iteration addresses the following activities.

- ***Modeling***. UML representations of the business and problem domains are created.
- ***Implementation.*** Models are translated into source code.
- ***Testing***. Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements.
- ***Deployment.*** Like the generic process activity deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.
- ***Configuration and project management.*** In the context of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.
- ***Environment management.*** Environment management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

A TOOL SET FOR THE AGILE PROCESS

Some proponents of the agile philosophy argue that automated software tools (e.g., design tools) should be viewed as a minor supplement to the team's activities, and not at all pivotal to the success of the team. However, Alistair Cockburn [Coc04] suggests that tools can have a benefit and that "agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage. Some tools are technological, helping distributed teams simulate being physically present. Many tools are physical, allowing people to manipulate them in workshops."

Because acquiring the right people (hiring), team collaboration, stakeholder communication, and indirect management are key elements in virtually all agile process models, Cockburn argues that "tools" that address these issues are critical success factors for agility. For example, a hiring "tool" might be the requirement to have a prospective team member spend a few hours pair programming with an existing member of the team. The "fit" can be assessed immediately.

Collaborative and communication "tools" are generally low tech and incorporate any mechanism ("physical proximity, whiteboards, poster sheets, index cards, and sticky notes" [Coc04]) that provides information and coordination among agile developers.

Active communication is achieved via the team dynamics (e.g., pair programming), while passive communication is achieved by "information radiators" (e.g., a flat panel display that presents the overall status of different components of an increment). Project management tools deemphasize the Gantt chart and replace it with earned value charts or "graphs of tests created versus passed . . . other agile tools are used to optimize the environment in which the agile team works (e.g., more efficient meeting areas), improve the team culture by nurturing social interactions (e.g., collocated teams), physical devices (e.g., electronic whiteboards), and process enhancement (e.g., pair programming or time-boxing)" [Coc04].

SOFTWARE ENGINEERING KNOWLEDGE

In an editorial published in *IEEE Software* a decade ago, Steve McConnell [McC99] made the following comment:

Many software practitioners think of software engineering knowledge almost exclusively as knowledge of specific technologies: Java, Perl, html, C__, Linux, Windows NT, and so on. Knowledge of specific technology details is necessary to perform computer programming. If someone assigns you to write a program in C__, you have to know something about C__ to get your program to work.

You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge—a kind that I think of as "software engineering principles"—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.

McConnell goes on to argue that the body of software engineering knowledge (circa the year 2000) had evolved to a "stable core" that he estimated represented about "75 percent of the knowledge needed to develop a complex system." But what resides within this stable core?

As McConnell indicates, core principles—the elemental ideas that guide software engineers in the work that they do—now provide a foundation from which software engineering models, methods, and tools can be applied and evaluated.

CORE PRINCIPLES

Software engineering is guided by a collection of core principles that help in the application of a meaningful software process and the execution of effective software engineering methods. At the process level, core principles establish a philosophical foundation that guides a software team as it performs framework and umbrella activities, navigates the process flow, and produces a set of software engineering work products. At the level of practice, core principles establish a collection of values and rules that serve as a guide as you analyze a problem, design a solution, implement and test the solution, and ultimately deploy the software in the user community identified a set of general principles that span software engineering process and practice:

(1) provide value to end users,

(2) keep it simple,

(3) maintain the vision (of the product and the project),

(4) recognize that others consume (and must understand) what you produce,

(5) be open to the future,

(6) plan ahead for reuse, and

(7) think! Although these general principles are important, they are characterized at such a high level of abstraction that they are sometimes difficult to translate into day-to-day software engineering practice

PRINCIPLES THAT GUIDE EACH FRAMEWORK ACTIVITY

Communication Principles

Principle 1. Listen. Try to focus on the speaker's words, rather than formulating your response to those words. Ask for clarification if something is unclear, but avoid constant interruptions. Never become contentious in your words or actions (e.g., rolling your eyes or shaking your head) as a person is talking.

Principle 2. Prepare before you communicate. Spend the time to understand the problem before you meet with others. If necessary, do some research to understand business domain jargon. If you have responsibility for conducting a meeting, prepare an agenda in advance of the meeting.

Principle 3. Someone should facilitate the activity. Every communication meeting should have a leader (a facilitator) to keep the conversation

moving in a productive direction, (2) to mediate any conflict that does occur, and (3) to ensure than other principles are followed.

Principle 4. Face-to-face communication is best. But it usually works better when some other representation of the relevant information is present. For example, a participant may create a drawing or a "strawman" document that serves as a focus for discussion.

Principle 5. Take notes and document decisions. Things have a way of falling into the cracks. Someone participating in the communication should serve as a "recorder" and write down all important points and decisions.

Principle 6. Strive for collaboration. Collaboration and consensus occur when the collective knowledge of members of the team is used to describe product or system functions or features. Each small collaboration serves to build trust among team members and creates a common goal for the team.

Principle 7. Stay focused; modularize your discussion. The more people involved in any communication, the more likely that discussion will bounce from one topic to the next. The facilitator should keep the conversation modular, leaving one topic only after it has been resolved

Principle 8. If something is unclear, draw a picture. Verbal communication goes only so far. A sketch or drawing can often provide clarity when words fail to do the job.

Principle 9. (a) Once you agree to something, move on. (b) If you can't agree to something, move on. (c) If a feature or function is unclear and cannot be clarified at the moment, move on. Communication, like any software engineering activity, takes time. Rather than iterating endlessly, the people who participate should recognize that many topics require discussion (see Principle 2) and that "moving on" is sometimes the best way to achieve communication agility.

Principle 10. Negotiation is not a contest or a game. It works best when both parties win. There are many instances in which you and other stakeholders must negotiate functions and features, priorities, and delivery dates. If the team has collaborated well, all parties have a common goal. Still, negotiation will demand compromise from all parties.

Planning Principles

Principle 1. Understand the scope of the project. It's impossible to use a road map if you don't know where you're going. Scope provides the software team with a destination.

Principle 2. Involve stakeholders in the planning activity. Stakeholders define priorities and establish project constraints. To accommodate these realities, software engineers must often negotiate order of delivery, time lines, and other project-related issues.

Principle 3. Recognize that planning is iterative. A project plan is never engraved in stone. As work begins, it is very likely that things will change. As a consequence, the plan must be adjusted to accommodate these changes. In addition, iterative, incremental process models dictate replanning after the delivery of each software increment based on feedback received from users.

Principle 4. Estimate based on what you know. The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. If information is vague or unreliable, estimates will be equally unreliable.

Principle 5. Consider risk as you define the plan. If you have identified risks that have high impact and high probability, contingency planning is necessary. In addition, the project plan (including the schedule) should be adjusted to accommodate the likelihood that one or more of these risks will occur.

Principle 6. Be realistic. People don't work 100 percent of every day. Noise always enters into any human communication. Omissions and ambiguity are facts of life. Change will occur. Even the best software engineers make mistakes. These and other realities should be considered as a project plan is established.

Principle 7. Adjust granularity as you define the plan. Granularity refers to the level of detail that is introduced as a project plan is developed. A "high-granularity" plan provides significant work task detail that is planned over relatively short time increments (so that tracking and control occur frequently). A "low-granularity" plan provides broader work tasks that are

planned over longer time periods. In general, granularity moves from high to low as the project time line moves away from the current date. Over the next few weeks or months, the project can be planned in significant detail. Activities that won't occur for many months do not require high granularity (too much can change).

Principle 8. Define how you intend to ensure quality. The plan should identify how the software team intends to ensure quality. If technical reviews3 are to be conducted, they should be scheduled. If pair programming (Chapter 3) is to be used during construction, it should be explicitly defined within the plan.

Principle 9. Describe how you intend to accommodate change. Even the best planning can be obviated by uncontrolled change. You should identify how changes are to be accommodated as software engineering work proceeds. For example, can the customer request a change at any time? If a change is requested, is the team obliged to implement it immediately? How is the impact and cost of the change assessed?

Principle 10. Track the plan frequently and make adjustments as required. Software projects fall behind schedule one day at a time. Therefore, it makes sense to track progress on a daily basis, looking for problem areas and situations in which scheduled work does not conform to actual work conducted. When slippage is encountered, the plan is adjusted accordingly.

Modeling Principles

Principle 1. The primary goal of the software team is to build software, not create models. Agility means getting software to the customer in the fastest possible time. Models that make this happen are worth creating, but models that slow the process down or provide little new insight should be avoided.

Principle 2. Travel light—don't create more models than you need. Every model that is created must be kept up-to-date as changes occur. More importantly, every new model takes time that might otherwise be spent on construction (coding and testing). Therefore, create only those models that

make it easier and faster to construct the software.

Principle 3. Strive to produce the simplest model that will describe the problem or the software. Don't overbuild the software [Amb02b]. By keeping models simple, the resultant software will also be simple. The result is software that is easier to integrate, easier to test, and easier to maintain (to change). In addition, simple models are easier for members of the software team to understand and critique, resulting in an ongoing form of feedback that optimizes the end result.

Principle 4. Build models in a way that makes them amenable to change. Assume that your models will change, but in making this assumption don't get sloppy. For example, since requirements will change, there is a tendency to give requirements models short shrift. Why? Because you know that they'll change anyway. The problem with this attitude is that without a reasonably complete requirements model, you'll create a design (design model) that will invariably miss important functions and features.

Principle 5. Be able to state an explicit purpose for each model that is created. Every time you create a model, ask yourself why you're doing so. If you can't provide solid justification for the existence of the model, don't spend time on it.

Principle 6. Adapt the models you develop to the system at hand. It may be necessary to adapt model notation or rules to the application; for example, a video game application might require a different modeling technique than real-time, embedded software that controls an automobile engine.

Principle 7. Try to build useful models, but forget about building perfect models. When building requirements and design models, a software engineer reaches a point of diminishing returns. That is, the effort required to make the model absolutely complete and internally consistent is not worth the benefits of these properties. Am I suggesting that modeling should be sloppy or low quality? The answer is "no." But modeling should be conducted with an eye to the next software engineering steps. Iterating endlessly to make a model "perfect" does not serve the need for agility.

Principle 8. Don't become dogmatic about the syntax of the model. If it communicates content successfully, representation is secondary.

Although everyone on a software team should try to use consistent notation during modeling, the most important characteristic of the model is to communicate information that enables the next software engineering task. If a model does this successfully, incorrect syntax can be forgiven.

Principle 9. If your instincts tell you a model isn't right even though it seems okay on paper, you probably have reason to be concerned. If you are an experienced software engineer, trust your instincts. Software work teaches many lessons—some of them on a subconscious level. If something tells you that a design model is doomed to fail (even though you can't prove it explicitly), you have reason to spend additional time examining the model or developing a different one.

Principle 10. Get feedback as soon as you can. Every model should be reviewed by members of the software team. The intent of these reviews is to provide feedback that can be used to correct modeling mistakes, change misinterpretations, and add features or functions that were inadvertently omitted.

Requirements modeling principles.

Principle 1. The information domain of a problem must be represented and understood. The information domain encompasses the data that flow into the system (from end users, other systems, or external devices), the data that flow out of the system (via the user interface, network interfaces, reports, graphics, and other means), and the data stores that collect and organize persistent data objects (i.e., data that are maintained permanently).

Principle 2. The functions that the software performs must be defined. Software functions provide direct benefit to end users and also provide internal support for those features that are user visible. Some functions transform data that flow into the system. In other cases, functions effect some level of control over internal software processing or external system elements. Functions can be described at many different levels of abstraction, ranging from a general statement of purpose to a detailed description of the processing elements that must be invoked.

Principle 3. The behavior of the software (as a consequence of external events) must be represented. The behavior of computer software is driven

by its interaction with the external environment. Input provided by end users, control data provided by an external system, or monitoring data collected over a network all cause the software to behave in a specific way.

Principle 4. The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion. Requirements modeling is the first step in software engineering problem solving. It allows you to better understand the problem and establishes a basis for the solution (design). Complex problems are difficult to solve in their entirety. For this reason, you should use a divide-and-conquer strategy. A large, complex problem is divided into subproblems until each subproblem is relatively easy to understand. This concept is called partitioning or separation of concerns, and it is a key strategy in requirements modeling.

Principle 5. The analysis task should move from essential information toward implementation detail. Requirements modeling begins by describing the problem from the end-user's perspective. The "essence" of the problem is described without any consideration of how a solution will be implemented. For example, a video game requires that the player "instruct" its protagonist on what direction to proceed as she moves into a dangerous maze. That is the essence of the problem. Implementation detail (normally described as part of the design model) indicates how the essence will be implemented. For the video game, voice input might be used. Alternatively,

Design Modeling Principles

Principle 1. Design should be traceable to the requirements model.

The requirements model describes the information domain of the problem, user-visible functions, system behavior, and a set of requirements classes that package business objects with the methods that service them. The design model translates this information into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.

Principle 2. Always consider the architecture of the system to be built.

Software architecture (Chapter 9) is the skeleton of the system to be built. It affects interfaces, data structures, program control flow and behavior, the manner in which testing can be conducted, the maintainability of the resultant system, and much more. For all of these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered.

Principle 3. Design of data is as important as design of processing functions. Data design is an essential element of architectural design. The manner in which data objects are realized within the design cannot be left to chance. A well-structured data design helps to simplify program flow, makes the design and implementation of software components easier, and makes overall processing more efficient.

Principle 4. Interfaces (both internal and external) must be designed with care. The manner in which data flows between the components of a system has much to do with processing efficiency, error propagation, and design simplicity. A well-designed interface makes integration easier and assists the tester in validating component functions.

Principle 5. User interface design should be tuned to the needs of the end user. However, in every case, it should stress ease of use. The user interface is the visible manifestation of the software. No matter how sophisticated its internal functions, no matter how comprehensive its data structures, no matter how well designed its architecture, a poor interface design often leads to the perception that the software is "bad."

Principle 6. Component-level design should be functionally independent. Functional independence is a measure of the "single-mindedness" of a software component. The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function or subfunction.5

Principle 7. Components should be loosely coupled to one another and to the external environment. Coupling is achieved in many ways—via a component interface, by messaging, through global data. As the level of coupling increases, the likelihood of error propagation also increases and the overall maintainability of the software decreases. Therefore, component coupling

should be kept as low as is reasonable.

Principle 8. Design representations (models) should be easily understandable. The purpose of design is to communicate information to practitioners who will generate code, to those who will test the software, and to others who may maintain the software in the future. If the design is difficult to understand, it will not serve as an effective communication medium.

Principle 9. The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity. Like almost all creative activities, design occurs iteratively. The first iterations work to refine the design and correct errors,

Construction Principles

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 principles: Before you write one line of code, be sure you

• Understand of the problem you're trying to solve.

• Understand basic design principles and concepts.

• Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.

• Select a programming environment that provides tools that will make your work easier.

• Create a set of unit tests that will be applied once the component you code is completed.

Programming principles: As you begin writing code, be sure you

• Constrain your algorithms by following structured programming [Boh00] practice.

• Consider the use of pair programming.

• Select data structures that will meet the needs of the design.

• Understand the software architecture and create interfaces that are consistent with it.

• Keep conditional logic as simple as possible.

• Create nested loops in a way that makes them easily testable.
• Select meaningful variable names and follow other local coding standards.

Write code that is self-documenting.
• Create a visual layout (e.g., indentation and blank lines) that aids understanding.
Validation Principles: After you've completed your first coding pass, be sure you
• Conduct a code walkthrough when appropriate.
• Perform unit tests and correct errors you've uncovered.
• Refactor the code.

Testing Principles.

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

Principle 1. All tests should be traceable to customer requirements.
The objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.

Principle 2. Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

Principle 3. The Pareto principle applies to software testing. In this context the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

Principle 4. Testing should begin "in the small" and progress toward testing "in the large." The first tests planned and executed generally focus

on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

Principle 5. Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.

Deployment Principles

Principle 1. Customer expectations for the software must be managed. Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately. This results in feedback that is not productive and ruins team morale. In her book on managing expectations, Naomi Karten [Kar94] states: "The starting point for managing expectations is to become more conscientious about what you communicate and how." She suggests that a software engineer must be careful about sending the customer conflicting messages (e.g., promising more than you can reasonably deliver in the time frame provided or delivering more than you promise for one software increment and then less than promised for the next).

Principle 2. A complete delivery package should be assembled and tested. A CD-ROM or other media (including Web-based downloads) containing all executable software, support data files, support documents, and other relevant information should be assembled and thoroughly beta-tested with actual users. All installation scripts and other operational features should be thoroughly exercised in as many different computing configurations (i.e., hardware, operating systems, peripheral devices, networking arrangements) as possible.

Principle 3. A support regime must be established before the software is delivered. An end user expects responsiveness and accurate information when a question or problem arises. If support is ad hoc, or worse, nonexistent, the customer will become dissatisfied immediately. Support should be planned, support materials should be prepared, and appropriate recordkeeping

mechanisms should be established so that the software team can conduct a categorical assessment of the kinds of support requested.

Principle 4. Appropriate instructional materials must be provided to end users. The software team delivers more than the software itself. Appropriate training aids (if required) should be developed; troubleshooting guidelines should be provided, and when necessary, a "what's different about this software increment" description should be published.

Principle 5. Buggy software should be fixed first, delivered later. Under time pressure, some software organizations deliver low-quality increments with a warning to the customer that bugs "will be fixed in the next release." This is a mistake. There's a saying in the software business: "Customers will forget you delivered a high-quality product a few days late, but they will never forget the problems that a low-quality product caused them. The software reminds them every day."

REQUIREMENTS ENGINEERING

Requirements analysis, also called **requirements engineering**, is the process of determining user expectations for a new or modified product. Requirements engineering is a major software engineering action that begins during the **communication activity and continues into the modeling activity.** It must be adapted to the needs of the process, the project, the product, and the people doing the work. Requirements engineering builds a bridge to design and construction.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system. It encompasses **seven** distinct tasks: **inception, elicitation, elaboration, negotiation, specification, validation, and management**.

Inception : It establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation: In this stage, proper information is extracted to prepare to document the requirements. It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product

fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis.

- **Problems of scope.** The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.
- **Problems of understanding.** The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that are ambiguous or un testable.
- **Problems of volatility.** The requirements change over time.

Elaboration: The information obtained from the customer during inception and elicitation is expanded and refined during elaboration. This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information. Elaboration is driven by the creation and refinement of user scenarios that describe **how** the end user (and other actors) will interact with the system.

Negotiation: To negotiate the requirements of a system to be developed, it is necessary to identify conflicts and to resolve those conflicts. You have to reconcile these conflicts through a process of negotiation. Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority. Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification: The term *specification* means **different things to different people**. A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.

Validation: The work products produced as a consequence of requirements engineering are assessed for quality during a validation step. Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies,

omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

The primary requirements validation mechanism is the **technical review**. The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies, conflicting requirements, or unrealistic requirements.

Requirements management. Requirements for computer-based systems change, and the desire to change requirements persists throughout the life of the system. Requirements management is a set of activities that help the project team identify, control, and track requirements and changes to requirements at any time as the project proceeds. Many of these activities are identical to the software configuration management (SCM) techniques.

ESTABLISHING THE GROUNDWORK

Identifying Stakeholders

A ***stakeholder*** is anyone who has a direct interest in or benefits from the system that is to be developed. At inception, you should create a list of people who will contribute input as requirements are elicited..

Recognizing Multiple Viewpoints

Because many different stakeholders exist, the requirements of the system will be explored from many different points of view. The information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.

Working toward Collaboration

The job of a requirements engineer is to identify areas of commonality and areas of conflict or inconsistency. It is, of course, the latter category that presents a challenge. Collaboration does not necessarily mean that requirements are defined by committee. In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

Asking the First Questions

Questions asked at the inception of the project should be "**context free**" . The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:

- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

These questions help to identify all stakeholders who will have interest in the software to be built. In addition, the questions identify the measurable benefit of a successful implementation and possible alternatives to custom software development.

The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:

- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

The final set of questions focuses on the effectiveness of the communication activity itself. Gause and Weinberg call these "**meta-questions**" and propose the following list:

- Are you the right person to answer these questions? Are your answers
- "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

These questions will help to "**break the ice**" and initiate the communication that is essential to successful elicitation. But a question-and-answer meeting format is not an approach that has been overwhelmingly successful.

ELICITING REQUIREMENTS

Requirements elicitation (also called *requirements gathering*) combines elements of problem solving, elaboration, negotiation, and specification

Collaborative Requirements Gathering

Many different approaches to collaborative requirements gathering have been proposed. Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:

- Meetings are conducted and attended by both software engineers and other 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" (can be a customer, a developer, or an outsider) controls the meeting.
- A "definition mechanism" (can be work sheets, flip charts, or wall stickers or
- an electronic bulletin board, chat room, or virtual forum) is used.

The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements in an atmosphere that is conducive to the accomplishment of the goal.

During inception basic questions and answers establish the scope of the problem and the overall perception of a solution. Out of these initial meetings, the developer and customers write a **one- or two-page "product request**."

A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate. The product request is distributed to all attendees before the meeting date.

While reviewing the product request in the days before the meeting, each attendee is asked to make a list of objects that are part of the environment that surrounds the system, other objects that are to be produced by the system, and objects that are used by the system to perform its functions. In addition, each attendee is asked to make another list of services that manipulate or interact with the objects. Finally, lists of constraints (e.g., cost, size, business rules) and performance criteria (e.g., speed, accuracy) are also developed. The attendees are informed that the lists are not expected to be exhaustive but are expected to reflect each person's perception of the system.

The lists of objects can be pinned to the walls of the room using large sheets of paper, stuck to the walls using adhesive-backed sheets, or written on a wall board. After individual lists are presented in one topic area, the group creates a combined list by eliminating redundant entries, adding any new ideas that come up during the discussion, but not deleting anything.

Quality Function Deployment

Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software. QFD "**concentrates on maximizing customer satisfaction from the software engineering process**". To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.

QFD identifies **three** types of requirements :

- **Normal requirements.** The objectives and goals that are stated for a product or system during meetings with the customer. If these requirements are present, the customer is satisfied. Examples of normal requirements might be requested types of graphical displays, specific system functions, and defined levels of performance.
- **Expected requirements.** These requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them. Their absence will be a cause for significant dissatisfaction.
- **Exciting requirements.** These features go beyond the customer's expectations and prove to be very satisfying when present.

Although QFD concepts can be applied across the entire software process, QFD uses customer interviews and observation, surveys, and examination of historical data as raw data for the requirements gathering activity. These data are then translated into a table of requirements— called the ***customer voice table***—that is reviewed with the customer and other stakeholders.

Usage Scenarios

As requirements are gathered, an overall vision of system functions and features begins to materialize. However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users. To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called ***use cases***, provide a description of how the system will be used.

Elicitation Work Products

The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built. For most systems, the work products include

- 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 and the domain constraints that apply to each.
- A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
- Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in requirements elicitation.

DEVELOPING USE CASES

Use cases are defined from an actor's point of view. An actor is a role that people (users) or devices play as they interact with the software.

The first step in writing a use case is to define the set of "actors" that will be involved in the story. *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. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself. Every actor has one or more goals when using the system. It is important to note that an actor and an end user are not necessarily the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role in the context of the use case. Different people may play the role of each actor.

Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify **primary actors** during the first iteration and **secondary actors** as more is learned about the system.

Primary actors interact to achieve required system function and derive the intended benefit from the system. *Secondary actors* support the system so that primary actors can do their work. Once actors have been identified, use cases can be developed.

Jacobson suggests a number of questions that should be answered by a use case:

- Who is the primary actor, the secondary actor(s)?
- What are the actor's goals?
- What preconditions should exist before the story begins?
- What main tasks or functions are performed by the actor?
- What exceptions might be considered as the story 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?

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

BUILDING THE REQUIREMENTS MODEL

The intent of the analysis model is to provide a description of the required informational, functional, and behavioral domains for a computer-based system. The model changes dynamically as you learn more about the system to be built, and other stakeholders understand more about what they really require..

Elements of the Requirements Model

The specific elements of the requirements model are dictated by the analysis modeling method that is to be used. However, a set of generic elements is common to most requirements models.

- **Scenario-based elements.** The system is described from the user's point of view using a scenario-based approach.
- **Class-based elements.** Each usage scenario implies a set of objects that are manipulated as an actor interacts with the system. These objects are categorized into classes a collection of things that have similar attributes and common behaviors.

- **Behavioral elements.** The behavior of a computer-based system can have a profound effect on the design that is chosen and the implementation approach that is applied. Therefore, the requirements model must provide modeling elements that depict behavior.
- **Flow-oriented elements.** Information is transformed as it flows through a computer- based system. The system accepts input in a variety of forms, applies functions to transform it, and produces output in a variety of forms.

Analysis Patterns

Analysis patterns suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.

Geyer-Schulz and Hahsler suggest two benefits that can be associated with the use of analysis patterns:

First, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem by providing reusable analysis models with examples as well as a description of advantages and limitations.

Second, analysis patterns facilitate the transformation of the analysis model into a design model by suggesting design patterns and reliable solutions for common problems.

Analysis patterns are integrated into the analysis model by reference to the pattern name.

NEGOTIATING REQUIREMENTS

The intent of negotiation is to develop a project plan that meets stakeholder needs while at the same time reflecting the real-world constraints (e.g., time, people, budget) that have been placed on the software team. The best negotiations strive for a "**win-win**" result. That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you win by working to realistic and achievable budgets and deadlines.

Boehm defines a set of negotiation activities at the beginning of each software process iteration. Rather than a single customer communication activity, the following activities are defined:

1. Identification of the system or subsystem's key stakeholders.
2. Determination of the stakeholders' "win conditions."
3. Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned.

Successful completion of these initial steps achieves a win-win result, which becomes the key criterion for proceeding to subsequent software engineering activities.

VALIDATING REQUIREMENTS

As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity. The requirements represented by the model are prioritized by the stakeholders and grouped within requirements packages that will be implemented as software increments.

A review of the requirements model addresses the following questions:

- Is each requirement consistent with the overall objectives 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?
- 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?
- Have requirements patterns been used to simplify the requirements model?
- Have all patterns been properly validated? Are all patterns consistent with customer requirements?

These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

suresh.mentor@gmail.com

LECTURE NOTES

ON

SOFTWARE ENGINEERING

2020 – 2021

II B. Tech I Semester (R19)

Mr. **P. Suresh Babu,** **Assistant Professor**

suresh.mentor@gmail.com

UNIT- III

Requirements Analysis, Scenario-Based Modeling, UML Models That Supplement the Use Case, Data Modeling Concepts, Class-Based Modeling, Requirements Modeling Strategies, Flow-Oriented Modeling, creating a Behavioral Model, Patterns for Requirements Modelling, Requirements Modeling for WebApps

suresh.mentor@gmail.com

REQUIREMENTS MODELING: SCENARIOS, INFORMATION, AND ANALYSIS CLASSES

REQUIREMENTS ANALYSIS

Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet. Requirements analysis allows you to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

The requirements modeling action results in one or more of the following types of models:

- ***Scenario-based models*** of requirements from the point of view of various system "**actors**"
- ***Data models*** that depict the information domain for the problem
- ***Class-oriented models*** that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements
- ***Flow-oriented models*** that represent the functional elements of the system and how they transform data as it moves through the system
- ***Behavioral models*** that depict how the software behaves as a consequence of external "events"

These models provide a software designer with information that can be translated to architectural, interface, and component-level designs. Finally, the requirements model provides the developer and the customer with the means to assess quality once software is built.

Throughout requirements modeling, primary focus is on ***what,*** **not** ***how***. What user interaction occurs in a particular circumstance, what objects does the system manipulate, what functions must the system perform, what behaviors does the system exhibit, what interfaces are defined, and what constraints apply?

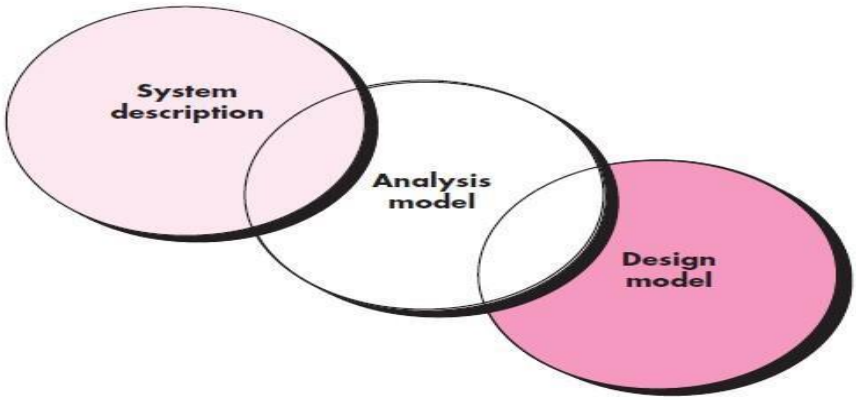


Fig : The requirements model as a bridge between the system description and the design model

The requirements 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 system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements and a software design that describes the software's application architecture, user interface, and component-level structure.

Analysis Rules of Thumb

Arlow and Neustadt suggest a number of worthwhile rules of thumb that should be followed when creating the analysis model:

- ***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 requirements 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 nonfunctional models until design***. That is, 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***. It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, effort should be made to reduce it.
- ***Be certain that the requirements model provides value to all stakeholders***. Each constituency has its own use for the model
- ***Keep the model as simple as it can be***. Don't create additional diagrams when they add no new information. Don't use complex notational forms, when a simple list will do.

Domain Analysis

Domain analysis doesn't look at a specific application, but rather at the domain in which the application resides.

The "specific application domain" can range from avionics to banking, from multimedia video games to software embedded within medical devices. The goal of domain analysis is straightforward: to identify common problem solving elements that are applicable to all applications within the domain, to find or create those analysis classes and/or analysis patterns that are broadly applicable so that they may be reused.

Requirements Modeling Approaches

One view of requirements modeling, called ***structured analysis***, considers data and the processes that transform the data as separate entities. Data objects are modeled in a way that defines their *attributes and relationships*.

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 effect customer requirements. UML and the Unified Process are predominantly object oriented.

Each element of the requirements model is represented in following figure presents the problem from a different point of view.

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Class-based elements model the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined.

suresh.mentor@gmail.com

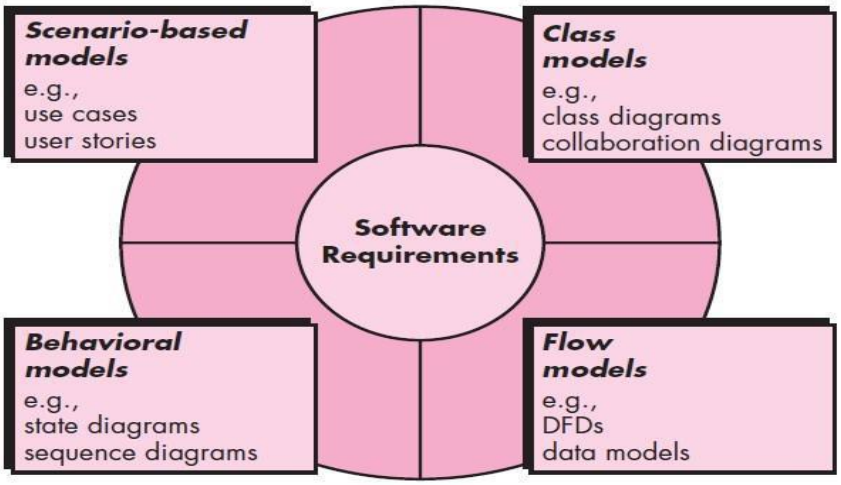


Fig : Elements of the analysis model

Behavioral elements depict how external events change the state of the system or the classes that reside within it. Finally,

Flow-oriented elements represent the system as an information transform, depicting how data objects are transformed as they flow through various system functions.

SCENARIO-BASED MODELING

Scenario-based elements depict how the user interacts with the system and the specific sequence of activities that occur as the software is used.

Creating a Preliminary Use Case

Alistair Cockburn characterizes a use case as a "contract for behavior", the "contract" defines the way in which an actor uses a computer-based system to accomplish some goal. In essence, a use case captures the interactions that occur between producers and consumers of information and the system itself.

A use case describes a specific usage scenario in straightforward language from the point of view of a defined actor. These are the questions that must be answered if use cases are to provide value as a requirements modeling tool. (1) what to write about, (2) how much to write about it, (3) how detailed to make your description, and (4) how to organize the description?

To begin developing a set of use cases, list the functions or activities performed by a specific actor.

Refining a Preliminary Use Case

Each step in the primary scenario is evaluated by asking the following questions:

- *Can the actor take some other action at this point?*
- *Is it possible that the actor will encounter some error condition at this point?* If so, what might it be?
- *Is it possible that the actor will encounter some other behavior at this point (e.g.,behavior that is invoked by some event outside the actor's control)?* If so, what might it be?

Cockburn recommends using a "brainstorming" session to derive a reasonably complete set of exceptions for each use case. In addition to the **three** generic questions suggested earlier in this section, the following issues should also be explored:

- *Are there cases in which some "validation function" occurs during this use case?* This implies that validation function is invoked and a potential error condition might occur.
- *Are there cases in which a supporting function (or actor) will fail to respond appropriately?* For example, a user action awaits a response but the function that is to respond times out.
- *Can poor system performance result in unexpected or improper user actions?* For example, a Web-based interface responds too slowly, resulting in a user making multiple selects on a processing button. These selects queue inappropriately and ultimately generate an error condition.

Writing a Formal Use Case

The typical outline for formal use cases can be in following manner

- The ***goal in context*** identifies the overall scope of the use case.
- The ***precondition*** describes what is known to be true before the use case is initiated.
- The ***trigger*** identifies the event or condition that "gets the use case started"
- The ***scenario*** lists the specific actions that are required by the actor and the appropriate system responses.
- ***Exceptions*** identify the situations uncovered as the preliminary use case is refined Additional headings may or may not be included and are reasonably self-explanatory.

Every modeling notation has limitations, and the use case is no exception. A use case focuses on functional and behavioral requirements and is generally inappropriate for nonfunctional requirements

However, scenario-based modeling is appropriate for a significant majority of all situationsthat you will encounter as a software engineer.

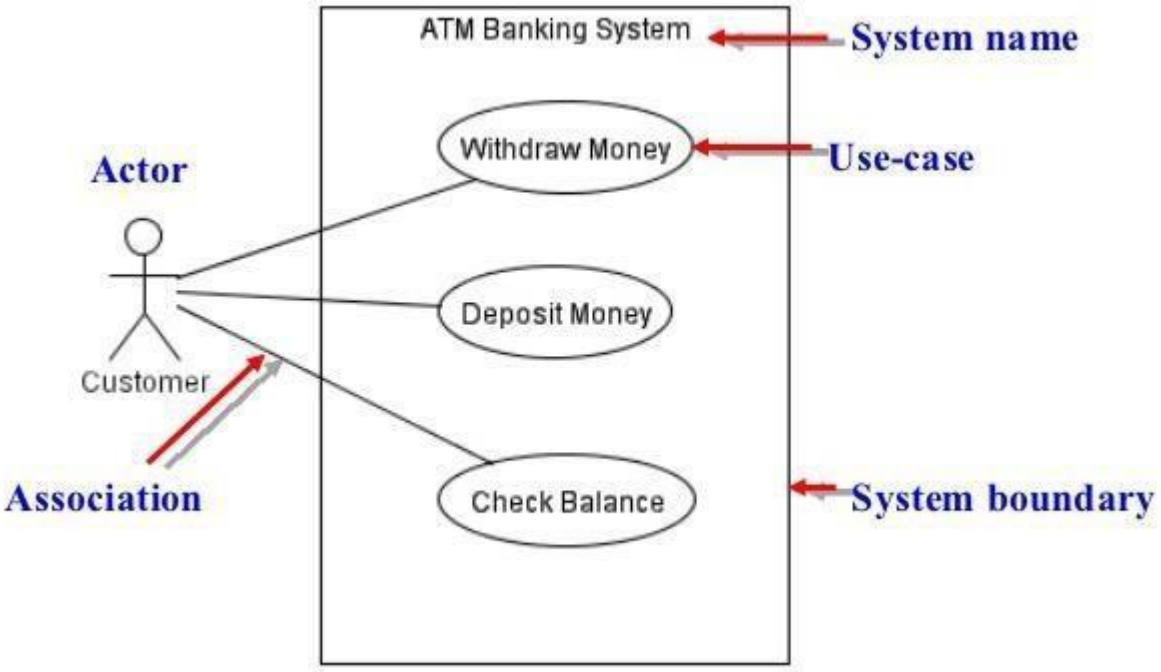


Fig : Simple Use Case Diagram

UML MODELS THAT SUPPLEMENT THE USE CASE

Developing an Activity Diagram

The UML activity diagram supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario. Similar to the flowchart, an activity diagram uses rounded rectangles to imply a specific system function, arrows to represent flow through the system, decision diamonds to depict a branching decision (each arrow emanating from the diamond is labeled), and solid horizontal lines to indicate that parallel activities are occurring. i.e A UML activity diagram represents the actions and decisions that occur as some function is performed.

suresh.mentor@gmail.com

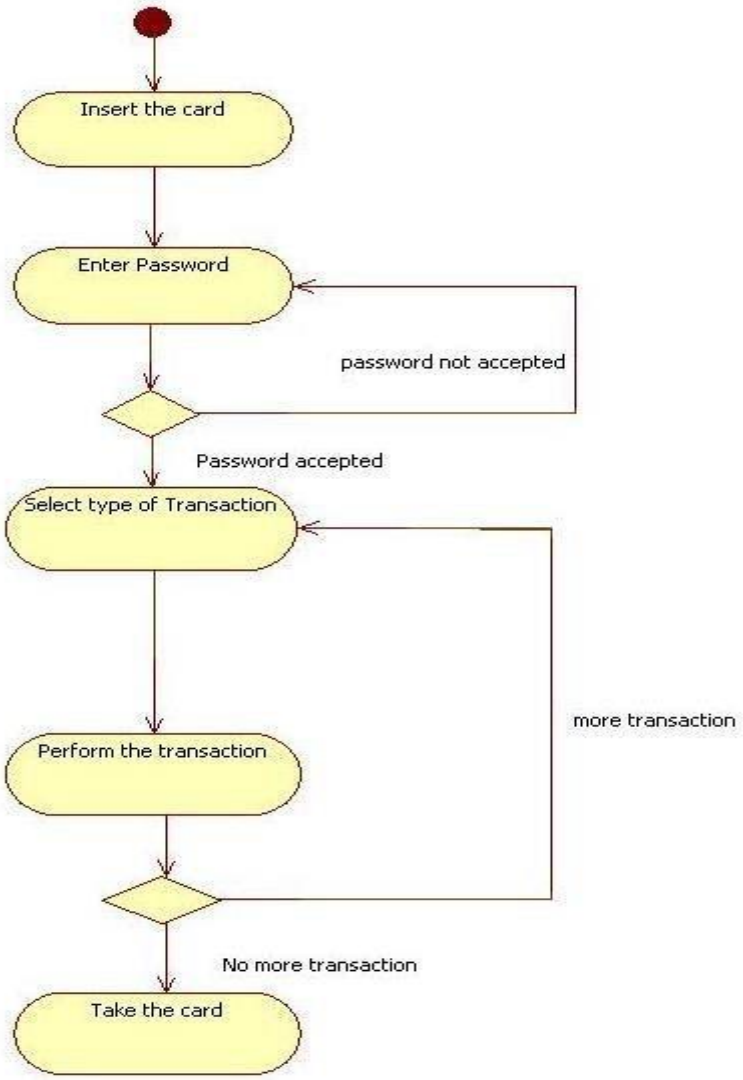


Fig : Activity Diagram for ATM

Swimlane Diagrams

The UML *swimlane diagram* is a useful variation of the activity diagram and allows you to represent the flow of activities described by the use 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.

The following figure represents *swimlane diagram for ATM*

suresh.mentor@gmail.com

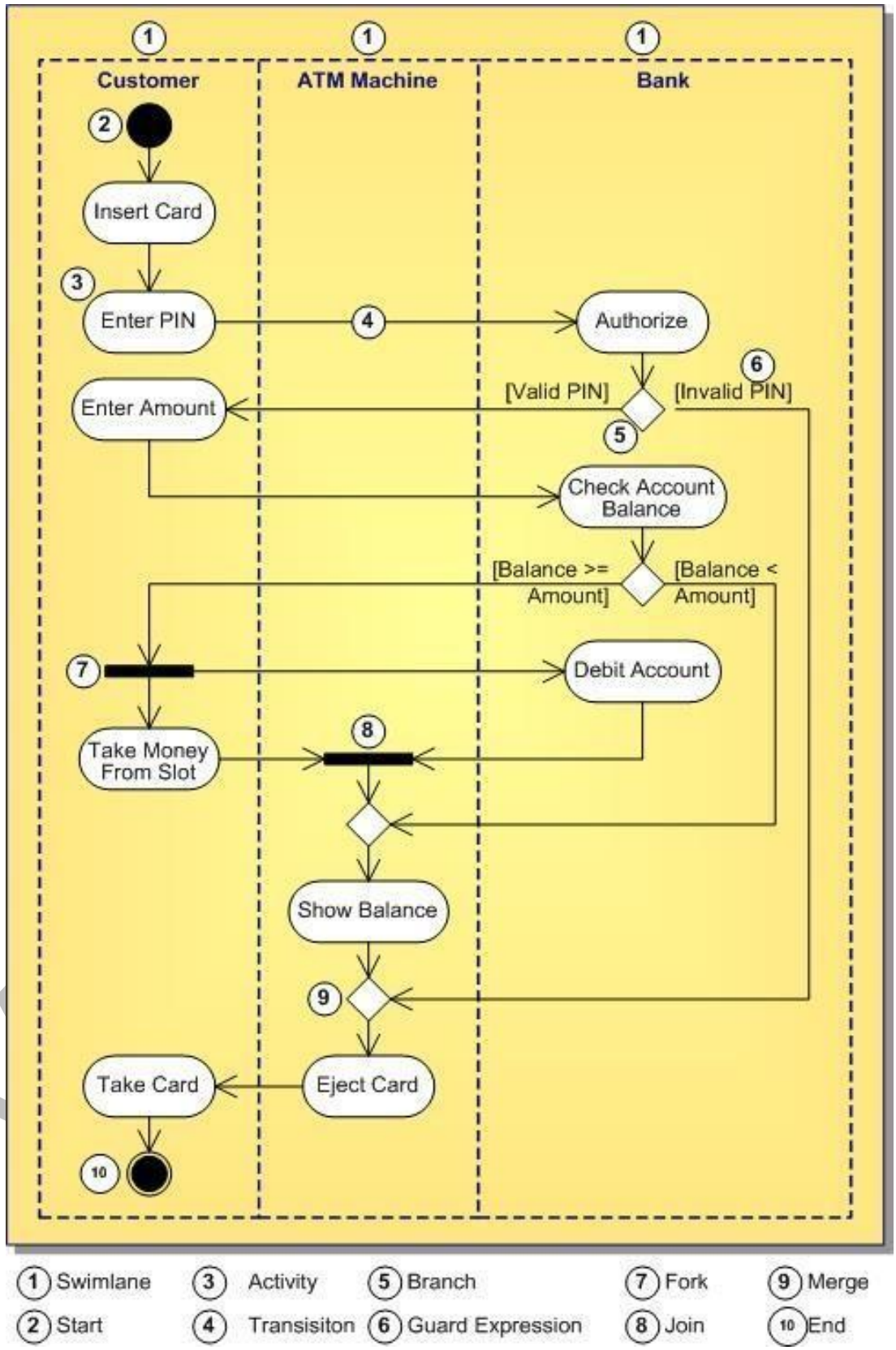


Fig : ***swimlane diagram for ATM***

suresh.mentor@gmail.com

DATA MODELING CONCEPTS

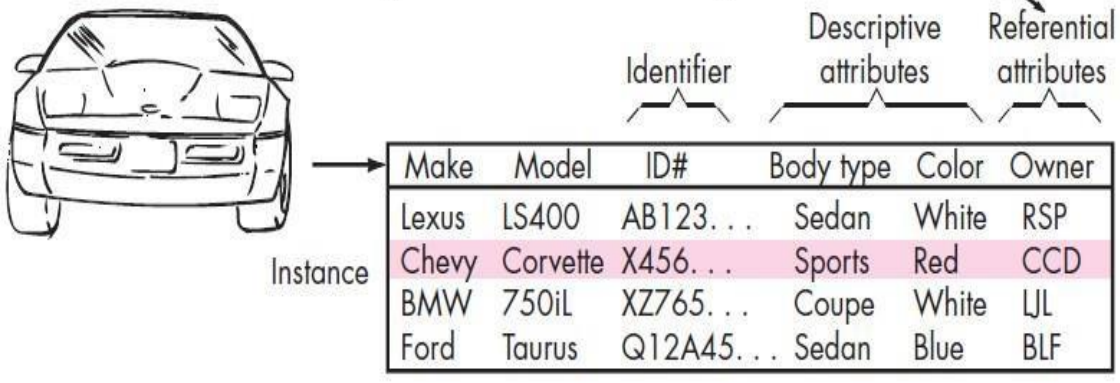
Data modeling is the process of documenting a complex software system design as an easily understood diagram, using text and symbols to represent the way data needs to flow. The diagram can be used as a blueprint for the construction of new software or for re-engineering a legacy application. The most widely used data Model by the Software engineers is **E*ntity-Relationship Diagram* (ERD)**, it addresses the issues and represents all data objects that are entered, stored, transformed, and produced within an application.

Data Objects

A *data object* is a representation of composite information that must be understood by software. A data object can be an **external entity** (e.g., anything that produces or consumes information), **a thing** (e.g., a report or a display), **an occurrence** (e.g., a telephone call) or **event** (e.g., an alarm), **a role** (e.g., salesperson), **an organizational unit** (e.g., accounting department), **a place** (e.g., a warehouse), **or a structure** (e.g., a file).

For example, a **person** or a **car** can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The description of the data object incorporates the data object and all of its attributes.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data. Therefore, the data object can be represented as a table as shown in following table. The headings in the table reflect attributes of the object.



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

Fig : Tabular representation of data objects

Data Attributes

Data attributes define the properties of a data object and take on one of **three** different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table.

Relationships

Data objects are connected to one another in different ways. Consider the two data objects, **person** and **car.** These objects can be represented using the following simple notation and relationships are 1) A person *owns* a car, 2) A person *is insured to drive* a car

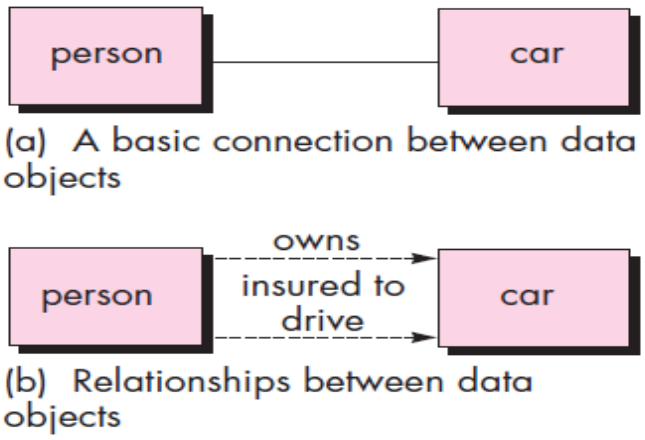


Fig : Relationships between data objects

CLASS-BASED MODELING

Class-based modeling represents the objects that the system will manipulate, the operations that will be applied to the objects to effect the manipulation, relationships between the objects, and the collaborations that occur between the classes that are defined. The elements of a class-based model include classes and objects, attributes, operations, class responsibility-collaborator (CRC) models, collaboration diagrams, and packages.

Identifying Analysis Classes

We can begin to identify classes by examining the usage scenarios developed as part of the requirements model and performing a "**grammatical parse**" on the use cases developed for the system to be built.

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, 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.

Coad and Yourdon suggest **six** selection characteristics that should be used as you consider each potential class for inclusion in the **analysis model**:

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 requirement analysis, 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 requirements model.

.2 Specifying Attributes

Attributes describe a class that has been selected for inclusion in the requirements model. In essence, it is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.

To develop a meaningful set of attributes for an analysis class, you should study each use case and select those "things" that reasonably "belong" to the class.

Defining Operations

Operations define the behavior of an object. Although many different types of operations exist, they can generally be divided into four broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, (3) operations that inquire about the state of an object, and (4) operations that monitor an object for the occurrence of a controlling event.

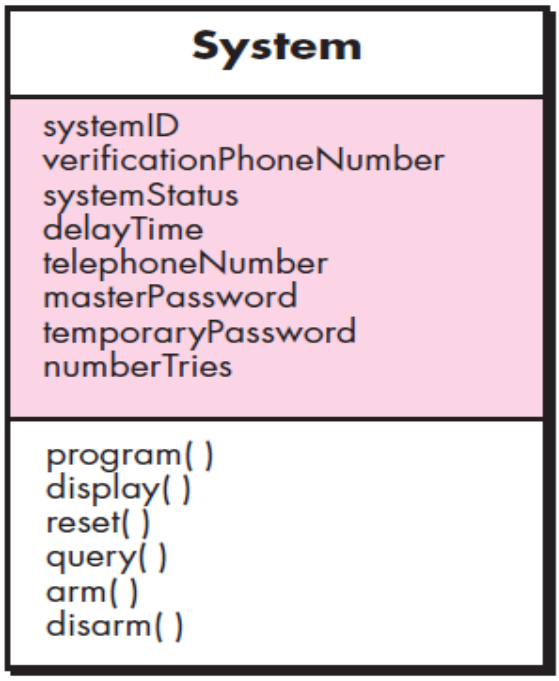


Fig : Class diagram for the system class

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 requirements.

Ambler describes CRC modeling in the following way :

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**.

The CRC model may make use of actual or virtual index cards. The intent is to develop an organized representation of classes. ***Responsibilities*** are the attributes and operations that are relevant for the class. i.e., a responsibility is "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, a *collaboration* implies either a request for information or a request for some action. A simple CRC index card is illustrated in following figure.

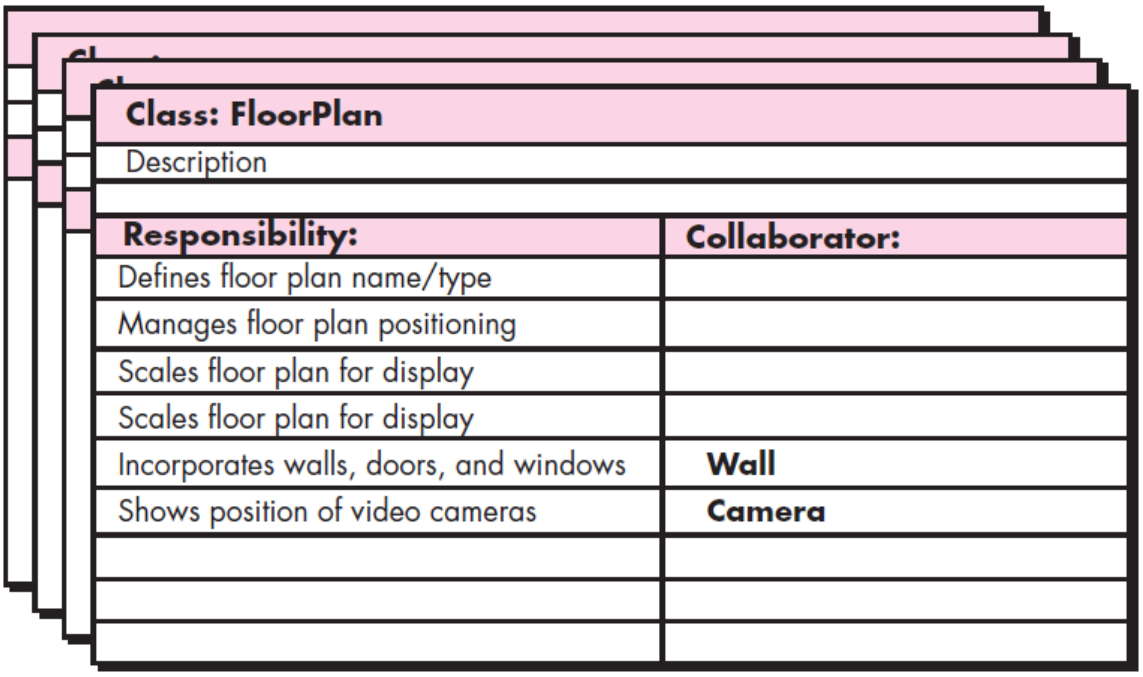


Fig : A CRC model index card

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. These classes typically represent things that are to be stored in a database and persist throughout the duration of the application.

- ***Boundary classes*** are used to create the interface that the user sees and interacts with as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.
- ***Controller classes*** manage a "unit of work" from start to finish. That is, controller classes can be designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, (4) validation of data communicated between objects or between the user and the application. In general, controller classes are not considered until the design activity has begun.

Responsibilities : Wirfs-Brock and her colleagues suggest five guidelines for allocating responsibilities to classes:

1. **System intelligence should be distributed across classes to best address the needs of the problem.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do.
2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities should reside high in the class hierarchy
3. **Information and the behavior related to it should reside within the same class.** This achieves the object-oriented principle called *encapsulation*. Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time.

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.

When a complete CRC model has been developed, stakeholders can review the model using the following approach :

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 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.

Associations and Dependencies

An ***association*** defines a relationship between classes. An association may be further defined by indicating ***multiplicity***. **Multiplicity** defines how many of one class are related to how many of another class.

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. Dependencies are defined by a **stereotype**. A ***stereotype*** is an "**extensibility mechanism**" within UML that allows you to define a special modeling element whose semantics are custom defined. In UML. Stereotypes are represented in double angle brackets (e.g., <<**stereotype**>>).

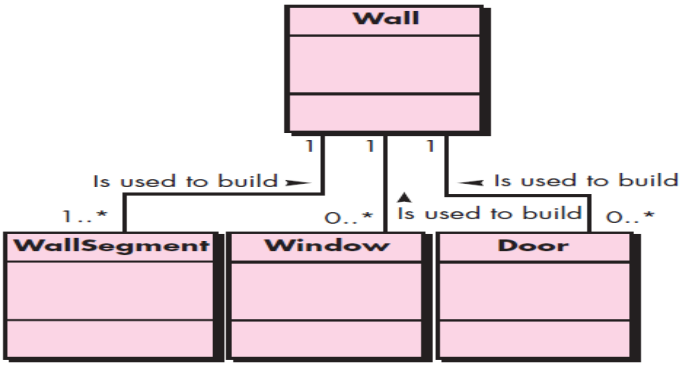


Fig : Multiplicity

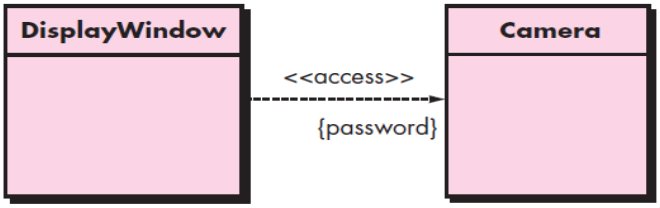


Fig : Dependencies

Analysis Packages

An important part of analysis modeling is categorization. That is, various elements of the analysis model (e.g., use cases, analysis classes) are categorized in a manner that packages them as a grouping—called an *analysis package*—that is given a representative name.

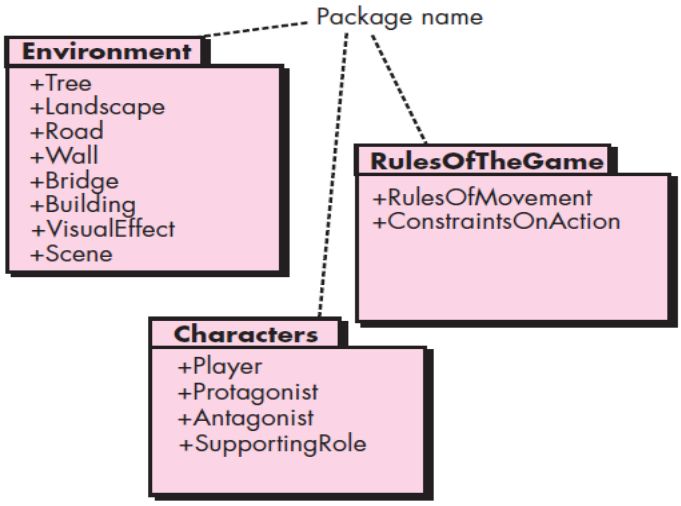


Fig : Packages

suresh.mentor@gmail.com

Requirements Modeling (Flow, Behavior, Patterns and WEBAPPS)

REQUIREMENTS MODELING STRATEGIES

One view of requirements modeling, called ***structured analysis***,. Data objects are modeled in a way that defines their attributes and relationships. Processes that manipulate data objects are modeled in a manner that shows how they transform data as data objects flow through the system.

A second approach to analysis modeled, called ***object-oriented analysis***, focuses on the definition of classes and the manner in which they collaborate with one another to effect customer requirements.

FLOW-ORIENTED MODELING

Flow-oriented modeling is perceived as an outdated technique by some software engineers, it continues to be one of the most widely used requirements analysis notations in use today. The ***data flow diagram*** **(DFD)** is the representation of Flow-oriented modeling. **The purpose of data flow diagrams is to provide a semantic bridge between users and systems developers."**

The DFD takes an input-process-output view of a system. That is, data objects flow into the software, are transformed by processing elements, and resultant data objects flow out of the software. Data objects are represented by labeled **arrows**, and transformations are represented by **circles (also called bubbles).** The DFD is presented in a hierarchical fashion. That is, the first data flow model (sometimes called a level **0 DFD or *context diagram***) represents the system as a whole. Subsequent data flow diagrams refine the context diagram, providing increasing detail with each subsequent level.

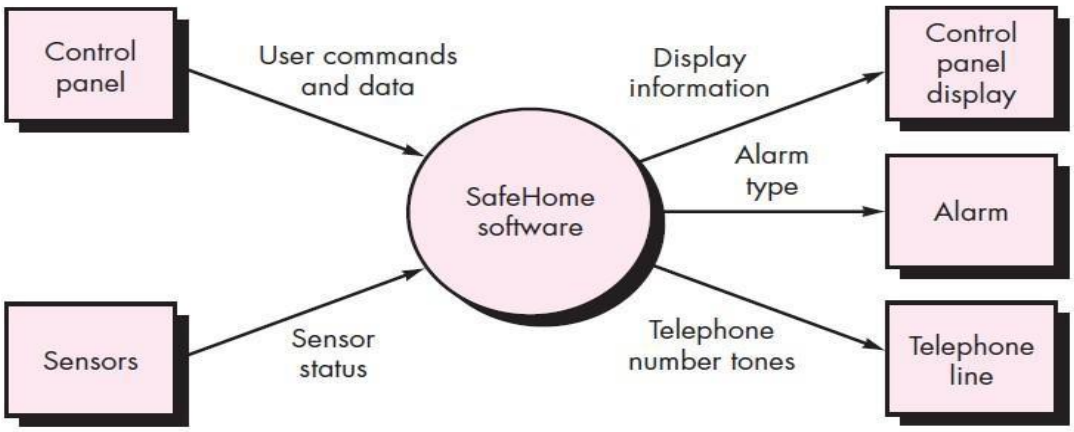


Fig : Context-level DFD for the Safe Home security function

Creating a Data Flow Model

The data flow diagram enables you to develop models of the information domain and functional domain. As the DFD is refined into greater levels of detail, you perform an implicit functional decomposition of the system. At the same time, the DFD refinement results in a corresponding refinement of data as it moves through the processes that embody the application.

A few simple guidelines can aid immeasurably during the derivation of a data flow diagram:

(1) The level 0 data flow diagram should depict the software/system as a single bubble;

(2) Primary input and output should be carefully noted;

(3) Refinement should begin by isolating candidate processes, data objects, and data stores to be represented at the next level;

(4) All arrows and bubbles should be labeled with meaningful names;

(5) *Information flow continuity* must be maintained from level to level,2 and

(6) One bubble at a time should be refined. There is a natural tendency to overcomplicate the data flow diagram.

A **level 0** DFD for the security function is shown in above figure. The primary ***external entities*** **(boxes)** produce information for use by the system and consume information generated by the system. The labeled arrows represent data objects or data object hierarchies.

The **level 0** DFD must now be expanded into a **level 1** data flow model. you should apply a "grammatical parse" to the use case narrative that describes the context-level bubble. That is, isolate all nouns (and noun phrases) and verbs (and verb phrases). *The grammatical parse is not*

foolproof, but it can provide you with an excellent jump start, if you're struggling to define data objects and the transforms that operate on them.

The processes represented at **DFD level 1** can be further refined into **lower levels**. The refinement of DFDs continues until each bubble performs a simple function. That is, until the process represented by the bubble performs a function that would be easily implemented as a program component. a concept, C***ohesion*** can be used to assess the processing focus of a given function. i.e refine DFDs until each bubble is "**single-minded**."

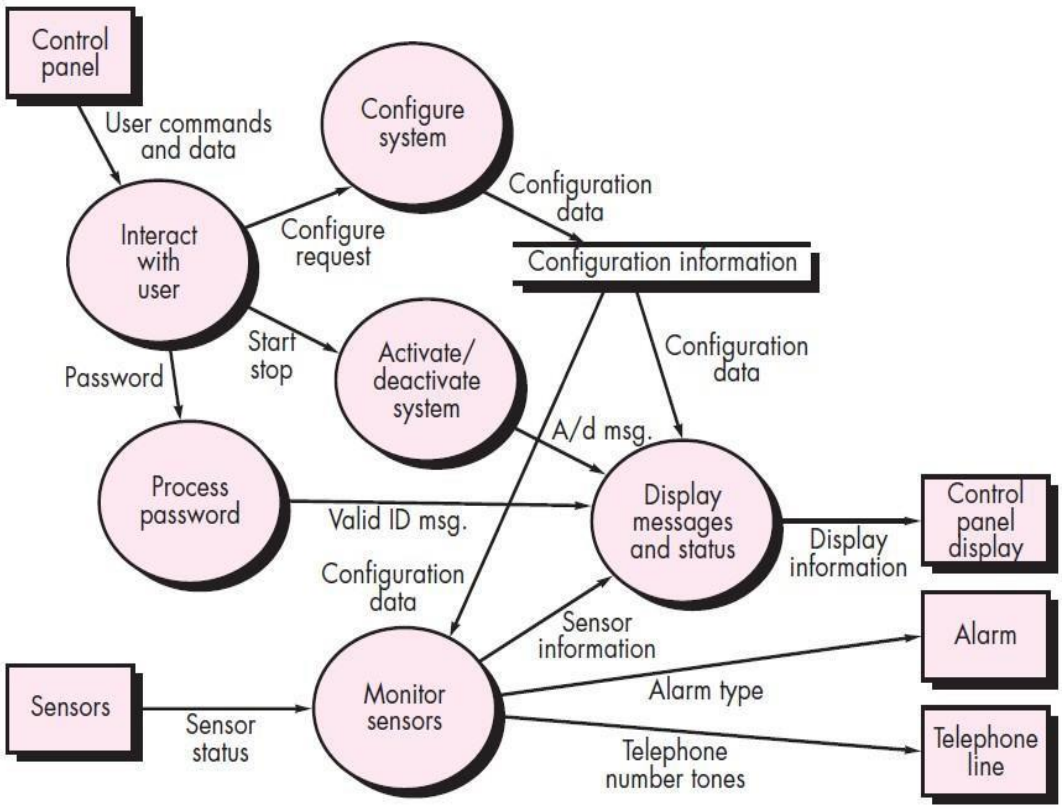


Fig: Level 1 DFD for SafeHome security function

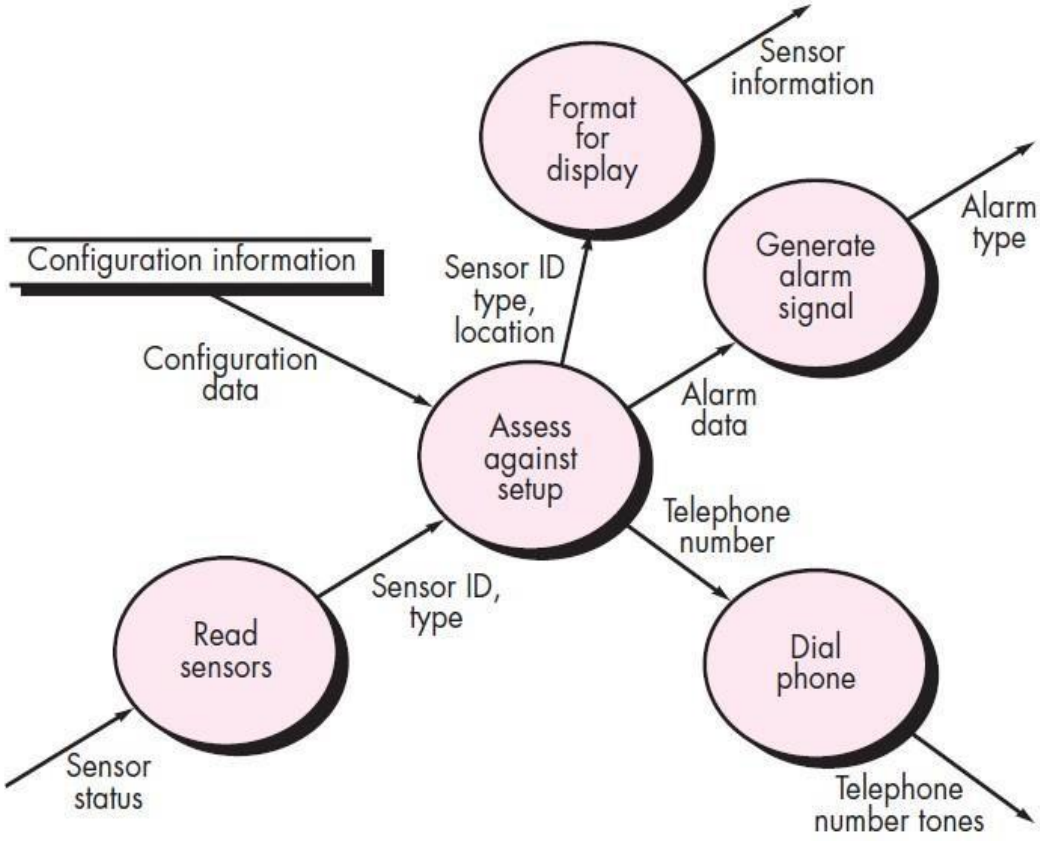


Fig : Level 2 DFD that refines the monitor sensors process

2.14.2. reating a Control Flow Model

The data model and the data flow diagram are all that is necessary to obtain meaningful insight into software requirements. The following guidelines are suggested for creating a Control Flow Model

- List all sensors that are "read" by the software.
- List all interrupt conditions.
- List all "switches" that are actuated by an operator.
- List all data conditions.
- Recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible control specification inputs/outputs.
- Describe the behavior of a system by identifying its states, identify how each state is reached, and define the transitions between states.
- Focus on possible omissions—a very common error in specifying control;

suresh.mentor@gmail.com

The Control Specification

A *control specification* (CSPEC) represents the behavior of the system in **two** different ways. The CSPEC contains a state diagram that is a sequential specification of behavior. It can also contain a program activation table—a combinatorial specification of behavior. The following figure depicts a preliminary state diagram for the level 1 control flow model for *SafeHome.* The diagram indicates how the system responds to events as it traverses the four states defined at this level. By reviewing the state diagram, we can determine the behavior of the system and, more important, ascertain whether there are "holes" in the specified behavior.

The CSPEC describes the behavior of the system, but it gives us no information about the inner working of the processes that are activated as a result of this behavior.

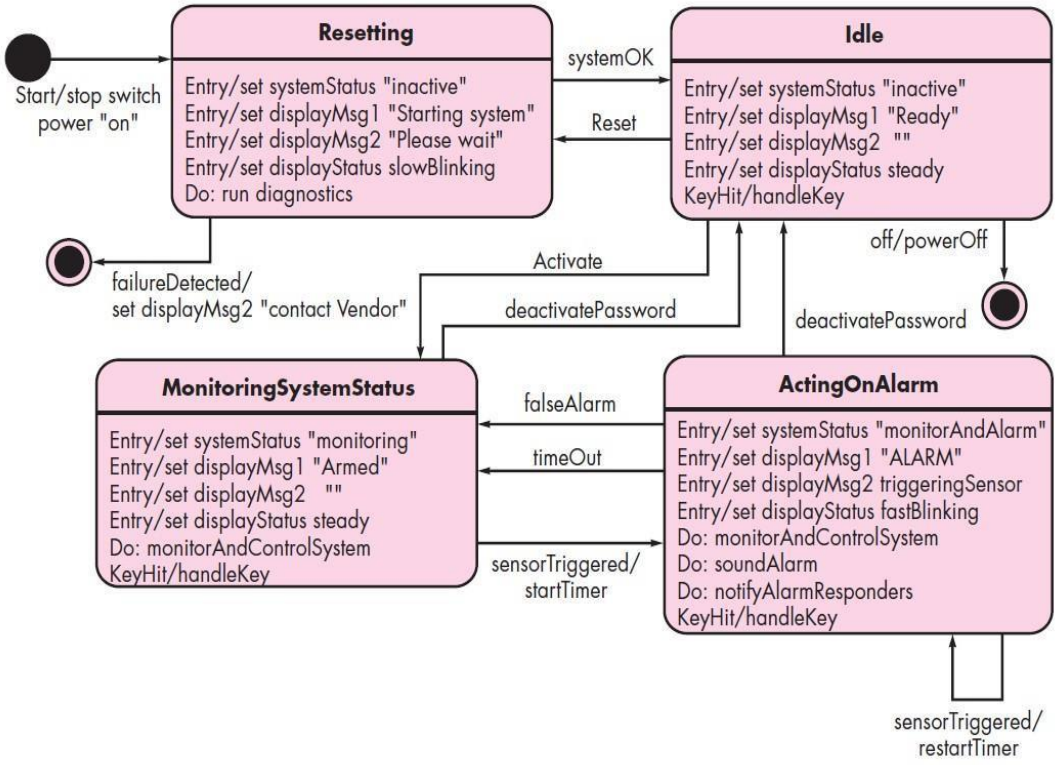


Fig : State diagram for SafeHome security function

The Process Specification

The *process specification* (PSPEC) is used to describe all flow model processes that appear at the final level of refinement. The content of the process specification can include narrative text, a program design language (PDL) description of the process algorithm,

mathematical equations, tables, or UML activity diagrams. By providing a PSPEC to accompany each bubble in the flow model, you can create a "mini-spec" that serves as a guide for design of the software component that will implement the bubble.

CREATING A BEHAVIORAL MODEL

The *behavioral model* indicates how software will respond to external events or stimuli. To create the model, you should perform the following steps:

1. Evaluate all use cases to fully understand the sequence of interaction within the system.

2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.

3. Create a sequence for each use case.

4. Build a state diagram for the system.

5. Review the behavioral model to verify accuracy and consistency.

Identifying Events with the Use Case

The use case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever the system and an actor exchange information. A use case is examined for points of information exchange. To illustrate, we reconsider the use case for a portion of the *SafeHome* security function. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

The underlined portions of the use case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted, and any conditions or constraints should be listed. Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events .

State Representations

In the context of behavioral modeling, two different characterizations of states must be considered: (1) the state of each class as the system performs its function and (2) the state of the system as observed from the outside as the system performs its Function **Two** different behavioral representations are discussed in the paragraphs that follow. The **first** indicates how

an individual class changes state based on external events and the **second** shows the behavior of the software as a function of time.

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

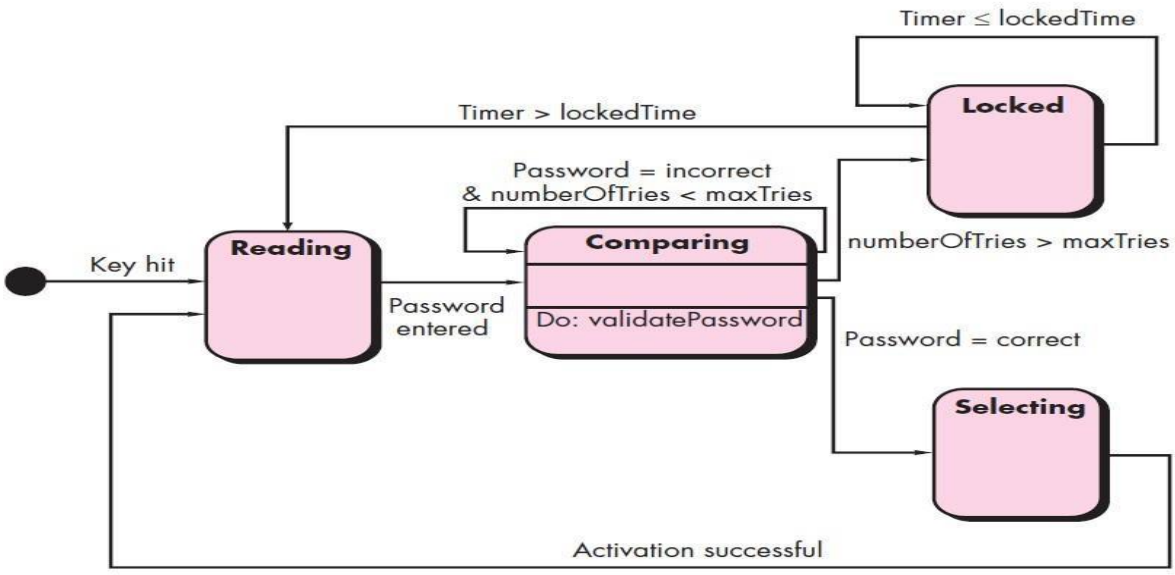


Fig : State diagram for the Control Panel class

Sequence diagrams. The second type of behavioral representation, called a *sequence diagram* in UML, indicates how events cause transitions from object to object. Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time. In essence, the sequence diagram is a shorthand version of the use case. It represents key classes and the events that cause behavior to flow from class to class.

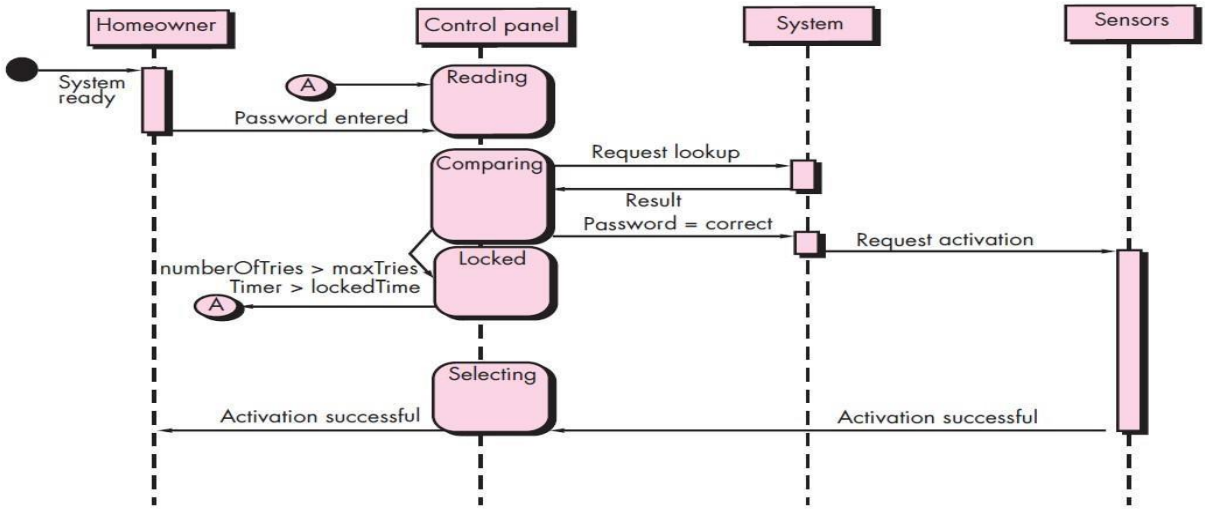


Fig : Sequence diagram (partial) for the *SafeHome* security function

PATTERNS FOR REQUIREMENTS MODELING

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

The pattern can be reused when performing requirements modeling for an application within a domain. Analysis patterns are stored in a repository so that members of the software team can use search facilities to find and reuse them. Once an appropriate pattern is selected, it is integrated into the requirements model by reference to the pattern name.

Discovering Analysis Patterns

The requirements model is comprised of a wide variety of elements: **scenario-based (use cases), data-oriented (the data model), class-based, flow-oriented, and behavioral**. Each of these elements examines the problem from a different perspective, and each provides an opportunity to discover patterns that may occur throughout an application domain, or by analogy, across different application domains.

The most basic element in the description of a requirements model is the **use case**. Use cases may serve as the basis for discovering one or more analysis patterns.

A ***semantic analysis pattern*** **(**SAP) "is a pattern that describes a small set of coherent use cases that together describe a basic generic application"

suresh.mentor@gmail.com

REQUIREMENTS MODELING FOR WEBAPPS

Requirements analysis does take time, but solving the wrong problem takes even more time.

How Much Analysis Is Enough?

The degree to which requirements modeling for WebApps is emphasized depends on the following factors:

- Size and complexity of WebApp increment.
- Number of stakeholders
- Size of the WebApp team.
- Degree to which members of the WebApp team have worked together
- Degree to which the organization's success is directly dependent on the success of the design of a specific part of the WebApp.

It only demands an analysis of those requirements that affect only that part of the WebApp.

Requirements Modeling Input

The requirements model provides a detailed indication of the true structure of the problem and provides insight into the shape of the solution. Requirements analysis refines this understanding by providing additional interpretation. As the problem structure is delineated as part of the requirements model.

Requirements Modeling Output

Requirements analysis provides a disciplined mechanism for representing and evaluating WebApp content and function, the modes of interaction that users will encounter, and the environment and infrastructure in which the WebApp resides. Each of these characteristics can be represented as a set of models that allow the WebApp requirements to be analyzed in a structured manner. While the specific models depend largely upon the nature of the WebApp, there are **five** main classes of models:

- **Content model**—identifies the full spectrum of content to be provided by the WebApp. Content includes text, graphics and images, video, and audio data.
- **Interaction model**—describes the manner in which users interact with the WebApp.

• **Functional model**—defines the operations that will be applied to WebApp content and describes other processing functions that are independent of content but necessary to the end user.

• **Navigation model**—defines the overall navigation strategy for the WebApp.

• **Configuration model**—describes the environment and infrastructure in which the WebApp resides.

4. Content. Model for WebApps

The content model contains structural elements that provide an important view of content requirements for a WebApp. These structural elements encompass content objects and all analysis classes, user-visible entities that are created or manipulated as a user interacts with the Content can be developed prior to the implementation of the WebApp, while the WebApp is being built, or long after the WebApp is operational.

A *content object* might be a textual description of a product, an article describing a news event, an action photograph taken at a sporting event, a user's response on a discussion forum, an animated representation of a corporate logo, a short video of a speech, or an audio overlay for a collection of presentation slides. The content objects might be stored as separate files, embedded directly into Web pages, or obtained dynamically from a database. Content objects can be determined directly from use cases by examining the scenario description for direct and indirect references to content. The content model must be capable of describing the content object **Component.**

Interaction Model for WebApps

Interaction model that can be composed of one or more of the following elements: (1) use cases, (2) sequence diagrams, (3) state diagrams,16 and/or (4) user interface prototypes.

Functional Model for WebApps

The *functional model* addresses two processing elements of the WebApp, each representing a different level of procedural abstraction: (1) user-observable functionality that is delivered by the WebApp to end users, and (2) the operations contained within analysis classes that implement behaviors associated with the class.

User-observable functionality encompasses any processing functions that are initiated directly by the user.

suresh.mentor@gmail.com

Configuration Models for WebApps

The configuration model is nothing more than a list of server-side and client-side attributes. However, for more complex WebApps, a variety of configuration complexities may have an impact on analysis and design. The UML deployment diagram can be used in situations in which complex configuration architectures must be considered.

Navigation Modeling

Navigation modeling considers how each user category will navigate from one WebApp element (e.g., content object) to another. The mechanics of navigation are defined as part of design. At this stage, you should focus on overall navigation requirements. The following questions should be considered:

- Should certain elements be easier to reach than others? What is the priority for presentation?
- Should certain elements be emphasized to force users to navigate in their direction?
- How should navigation errors be handled?
- Should navigation to related groups of elements be given priority over navigation to a specific element?
- Should navigation be accomplished via links, via search-based access, or by some other means?
- Should certain elements be presented to users based on the context of previous navigation actions?
- Should a navigation log be maintained for users?
- Should a full navigation map or menu be available at every point in a user's interaction?
- Should navigation design be driven by the most commonly expected user behaviors or by the perceived importance of the defined WebApp elements?
- Can a user "store" his previous navigation through the WebApp to expedite future usage?
- For which user category should optimal navigation be designed?
- How should links external to the WebApp be handled? Overlaying the existing browser window? As a new browser window? As a separate frame?

These and many other questions should be asked and answered as part of navigation analysis.

UNIT – 4

Unit III:

Design Concepts: Design with Context of Software Engineering, The Design Process, Design Concepts, The Design Model.

Architectural Design: Software Architecture, Architecture Genres, Architecture Styles, Architectural Design, Assessing Alternative Architectural Designs, Architectural Mapping Using Data Flow.

Component-Level Design: Component, Designing Class-Based Components, Conducting Component-level Design, Component Level Design for WebApps, Designing Traditional Components, Component-Based Development.

What is it? Design is what almost every engineer wants to do. It is the place where creativity rules—where stakeholder requirements, business needs, and technical considerations all come together in the formulation of a product or system. Design creates a representation or model of the software, but unlike the requirements model, the design model provides detail about software architecture, data structures, interfaces, and components that are necessary to implement the system.

Who does it? Software engineers conduct each of the design tasks. Why is it important? Design allows you to model the system or product that is to be built. This model can be assessed for quality and improved before code is generated, tests are conducted, and end users become involved in large numbers. Design is the place where software quality is established.

What are the steps? Design depicts the software in a number of different ways. First, the architecture of the system or product must be represented. Then, the interfaces that connect the software to end users, to other systems and devices, and to its own constituent components are modeled. Finally, the software components that are used to construct the system are designed. Each of these views represents a different design action, but all must conform to a set of basic design concepts that guide software design work.

What is the work product? A design model that encompasses architectural, interface, component level, and deployment representations is the primary work product that is produced during software design.

How do I ensure that I've done it right? The design model is assessed by the software team in an effort to determine whether it contains errors, inconsistencies, or omissions; whether better alternatives exist; and whether the model can be implemented within the constraints, schedule, and cost that have been established.

DESIGN WITHIN THE CONTEXT OF SOFTWARE ENGINEERING

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Software design is the last software engineering action within the modeling activity and sets the stage for construction (code generation and testing).

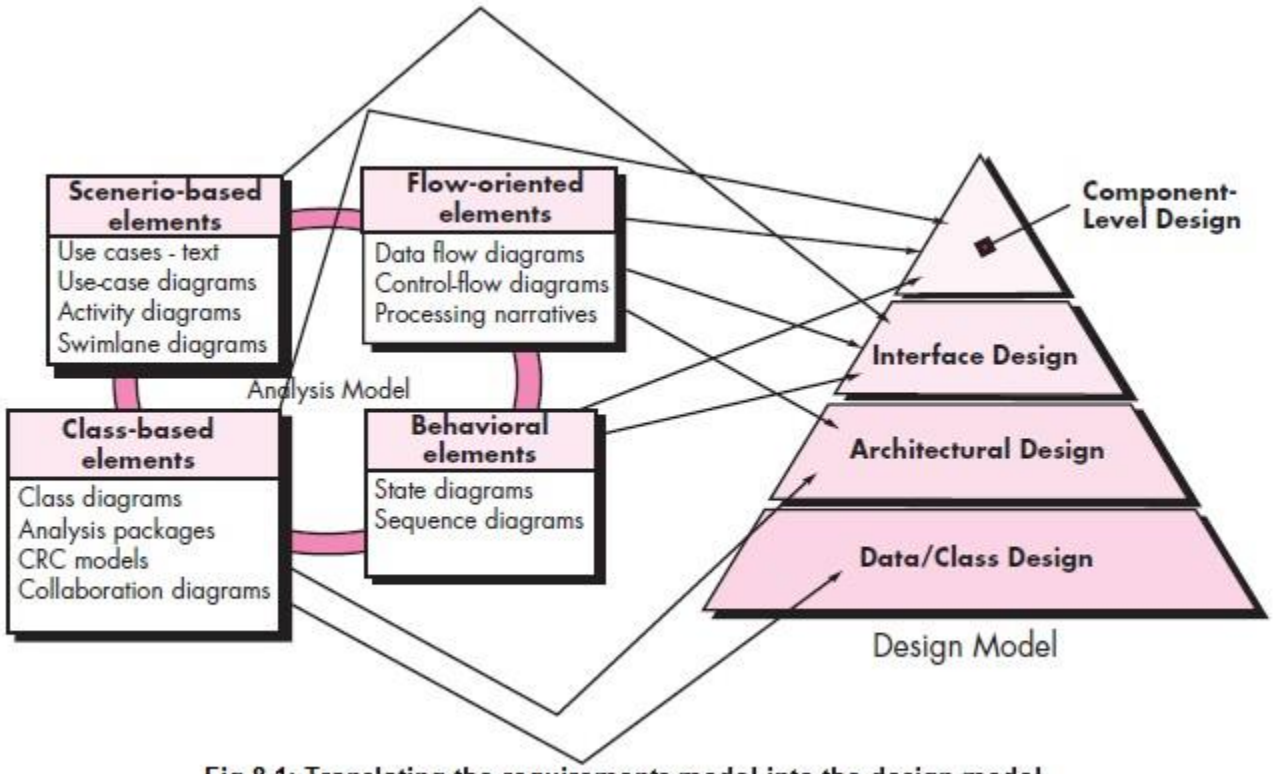


Fig 8.1: Translating the requirements model into the design model

Each of the elements of the requirements model provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 8.1. The requirements model, manifested by *scenario-based, class-based, flow-oriented, and behavioral elements*, feed the design task.

The *data/class design* transforms class models into design class realizations and the requisite data structures required to implement the software. The objects and relationships defined in the CRC diagram and the detailed data content depicted by class attributes and other notation provide the basis for the data design action. Part of class design may occur in conjunction with the design of software architecture.

The *architectural design* defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the

requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

The *interface design* describes how the software communicates with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

The *component-level* design transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

During design you make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained.

Design is the place where quality is fostered in software engineering. Design provides you with representations of software that can be assessed for quality. Design is the only way that you can accurately translate stakeholder's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support activities that follow. Without design, you risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

Software Quality Guidelines and Attributes: Throughout the design process, the quality of the evolving design is assessed with a series of technical reviews. The 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 requirements model, and it must accommodate all of the implicit requirements desired by stakeholders.

• The design must be a readable, understandable guide for those who 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 from an implementation perspective.

Quality Guidelines. In order to evaluate the quality of a design representation, the software team must establish technical criteria for good design. Guide lines are as follows

1. A design should exhibit an architecture that (1) has been created using recognizable architectural styles or patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.
2. A design should be modular; that is, the software should be logically partitioned into elements or subsystems.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.
8. A design should be represented using a notation that effectively communicates its meaning.

Quality Attributes. Hewlett-Packard developed a set of software quality attributes that has been given the acronym 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 measured by considering processing speed, response time, resource consumption, throughput, and efficiency.
• *Supportability* combines the ability to extend the program (extensibility), adaptability, serviceability—these three attributes represent a more common term, maintainability—and in addition, testability, compatibility, configurability, the ease with which a system can be installed, and the ease with which problems can be localized.

The Evolution of Software Design: The evolution of software design is a continuing process. Early design work concentrated on criteria for the development of modular programs

and methods for refining software structures in a topdown manner. Procedural aspects of design definition evolved into a philosophy called structured programming. Later work proposed methods for the translation of data flow or data structure into a design definition. Newer design approaches proposed an object-oriented approach to design derivation. More recent emphasis in software design has been on software architecture and the design patterns that can be used to implement software architectures and lower levels of design abstractions. Growing emphasis on aspect-oriented methods, model-driven development, and test-driven development emphasize techniques for achieving more effective modularity and architectural structure in the designs that are created.

A number of design methods, growing out of the work just noted, are being applied throughout the industry. These methods have a number of common characteristics: (1) a mechanism for the translation of the requirements model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

DESIGN CONCEPTS

Design concepts has evolved over the history of software engineering. Each concept provides the software designer with a foundation from which more sophisticated design methods can be applied. A brief overview of important software design concepts that span both traditional and object-oriented software development is given below.

Abstraction: When you consider a modular solution to any problem, many levels of abstraction can be posed. At the *highest level of abstraction*, a solution is stated in *broad terms* using the language of the problem environment. At *lower levels* of *abstraction*, a *more detailed description* of the solution is provided. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. *A data abstraction* is a named collection of data that describes a data object.

Architecture: Software architecture alludes to "*the overall structure of the software and the ways in which that structure provides conceptual integrity for a system*".
In its simplest form, *architecture* is the structure or organization of program components (modules), the manner in which these components interact, and the structure of data that are used by the components.
One goal of software design is to derive an architectural rendering of a system. A set of architectural patterns enables a software engineer to solve common design problems.

Shaw and Garlan describe a set of properties as part of an architectural design:
Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.
Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.
Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns: A pattern is a named nugget of insight which conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns. Stated
A design pattern describes a design structure that solves a particular design problem within a specific context and amid "forces" that may have an impact on the manner in which the pattern is applied and used.
The intent of each design pattern is to provide a description that enables a designer to determine
(1) whether the pattern is applicable to the current work
(2) whether the pattern can be reused (hence, saving design time)
(3) whether the pattern can serve as a guide for developing a similar, but functionally or structurally different pattern.

Separation of Concerns: Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is *subdivided into pieces* that can each be solved and/or optimized independently.

For two problems, p1 and p2, if the perceived complexity of p1 is greater than the perceived complexity of p2, it follows that the effort required to solve p1 is greater than the effort required to solve p2. As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem. It also follows that the perceived complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately. This leads to *a divide-and-conquer strategy*—it's easier to solve a complex problem when you break it into manageable pieces. This has important implications with regard to software modularity.

Modularity: Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements. It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". The

number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software.
if you subdivide software indefinitely the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 8.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases.

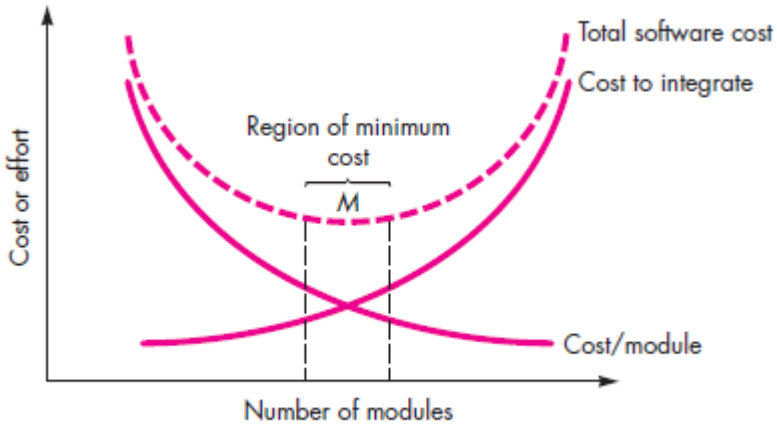


Fig 8.2: Modularity and software cost

Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M, of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

The curves shown in Figure 8.2 do provide useful qualitative guidance when modularity is considered. You should modularize, but care should be taken to stay in the vicinity of M. *Undermodularity* or *overmodularity* should be avoided.

You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.

Information Hiding: The principle of information hiding suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (algorithms and data) contained within a module is inaccessible to other modules that have no need for such information. Hiding implies that effective modularity can be achieved by defining a set of independent modules that

communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module. The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later during software maintenance. Because most data and procedural detail are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

Functional Independence: The concept of functional independence is a direct outgrowth of separation of concerns, modularity, and the concepts of abstraction and information hiding. Functional independence is achieved by developing modules with "singleminded" function and an "aversion" to excessive interaction with other modules. Stated another way, *you should design software so that each module addresses a specific subset of requirements and has a simple interface when viewed from other parts of the program structure*. Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

Independence is assessed using two qualitative criteria: ***cohesion*** and ***coupling.***
Cohesion is an indication of the relative functional strength of a module. Coupling is an indication of the relative interdependence among modules.

A **cohesive module** performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing. Although you should always strive for high cohesion (i.e., single-mindedness), it is often necessary and advisable to have a software component perform multiple functions.

Coupling is an indication of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface. In software design, you should strive for the lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect", caused when errors occur at one location and propagate throughout a system.

Refinement: Stepwise refinement is a *top-down design strategy.* . A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a *macroscopic statement* of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

Refinement is actually a process of elaboration begins with a statement of function (or description of information) that is defined at a high level of abstraction. You then elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs. Refinement helps you to reveal low-level details as design progresses.

Aspects: As requirements analysis occurs, a set of "concerns" is uncovered. These concerns "include requirements, use cases, features, data structures, quality-of-service issues, variants, intellectual property boundaries, collaborations, patterns and contracts". Ideally, a requirements model can be organized in a way that allows you to isolate each concern (requirement) so that it can be considered independently. In practice, however, some of these concerns span the entire system and cannot be easily compartmentalized.

As design begins, requirements are refined into a modular design representation. Consider two requirements, A and B. Requirement A crosscuts requirement B "if a software decomposition [refinement] has been chosen in which B cannot be satisfied without taking A into account".

Refactoring: An important design activity for many agile methods is *refactoring* a *reorganization technique* that simplifies the design (or code) of a component without changing its function or behavior. "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."
When software is refactored, the existing design is examined for redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failure that can be corrected to yield a better design. The result will be software that is easier to integrate, easier to test, and easier to maintain.

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

Design Classes: As the design model evolves, you will define a set of *design classes* that refine the analysis classes by providing design detail that will enable the classes to be implemented, and implement a software infrastructure that supports the business solution.
Five different types of design classes, each representing a different layer of the design architecture, can be developed.

- *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. 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.

They define four characteristics of a well-formed design class:
Complete and sufficient. A design class should be the complete encapsulation of all attributes and methods that can reasonably be expected (based on a knowledgeable interpretation of the class name) to exist for the class.
Primitiveness. Methods associated with a design class should be focused on accomplishing one service for the class. Once the service has been implemented with a method, the class should not provide another way to accomplish the same thing.
High cohesion. A cohesive design class has a small, focused set of responsibilities and single-mindedly applies attributes and methods to implement those responsibilities.
Low coupling. Within the design model, it is necessary for design classes to collaborate with one another. However, collaboration should be kept to an acceptable minimum. If a design model is highly coupled, the system is difficult to implement, to test, and to maintain over time. In general, design classes within a subsystem should have only limited knowledge of other classes. This restriction, called the *Law of Demeter*, suggests that a method should only send messages to methods in neighboring classes.

THE DESIGN MODEL

The design model can be viewed in two different dimensions as illustrated in Figure 8.4. 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. Referring to Figure 8.4, the dashed line indicates the boundary between *the analysis and design models*. The analysis model slowly blends into the design and a clear distinction is less obvious.

The elements of the design model use UML diagrams, that were used in the analysis model. The difference is that these *diagrams are refined and elaborated* as part of design; more implementation-specific detail is provided, and architectural structure and style, components that reside within the architecture, and interfaces between the components and with the outside world are all emphasized.

You should note, however, that model elements indicated along the horizontal axis are not always developed in a sequential fashion. The *deployment model* is usually delayed until the design has been fully developed.

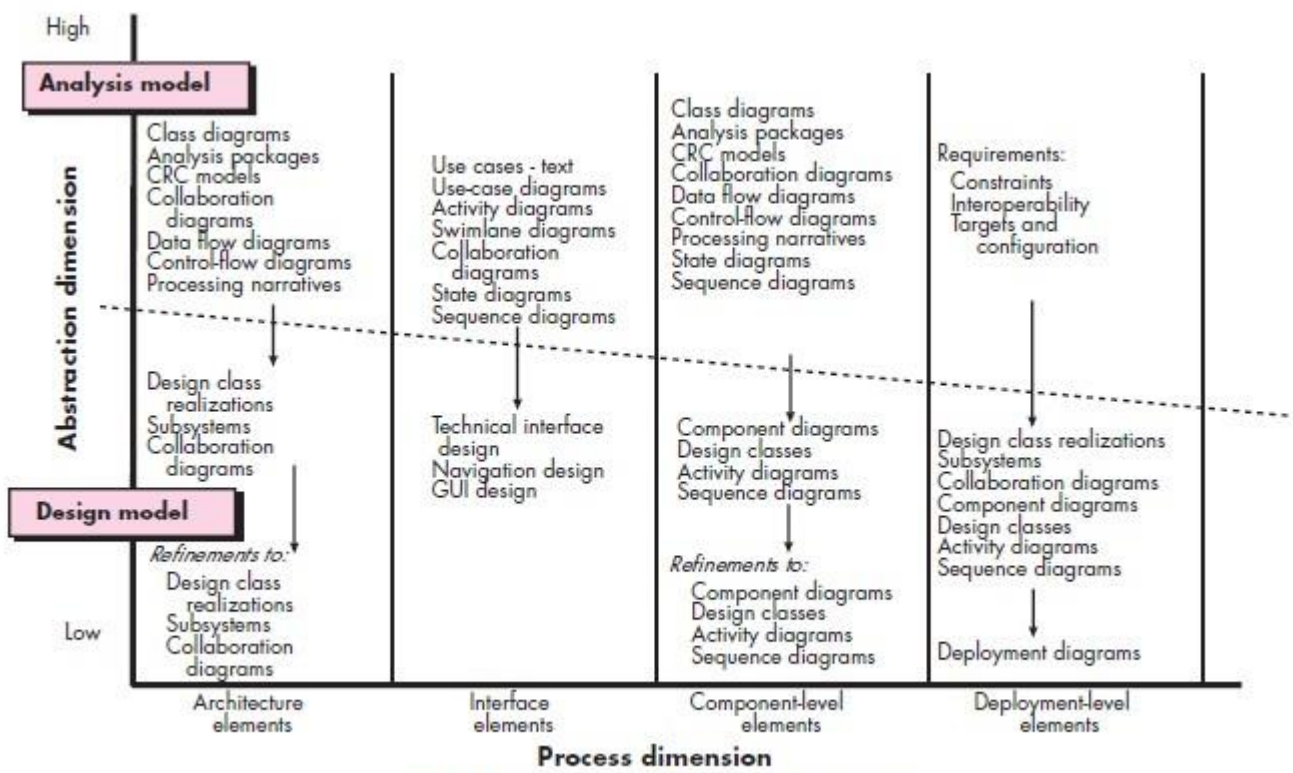


Fig 8.4: Dimensions of the design model

Data Design Elements: Like other software engineering activities, data design (sometimes referred to as data architecting) creates a model of data and/or information that is represented at a high level of abstraction. This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system.

The structure of data has always been an important part of software design. 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 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 requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand; and
(3) the availability of architectural *styles* and *patterns*.
The architectural design element is usually depicted as a set of interconnected subsystems, Each subsystem may have it's own architecture

Interface Design Elements: The interface design for software is analogous to a set of detailed drawings for the doors, windows, and external utilities of a house. The interface design elements for software depict information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
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.

UI design (increasingly called usability design) is a major software engineering action. Usability design incorporates aesthetic elements (e.g., layout, color, graphics, interaction mechanisms), ergonomic elements (e.g., information layout and placement, metaphors, UI navigation), and technical elements (e.g., UI patterns, reusable components).

The design of *external interfaces* requires definitive information about the entity to which information is sent or received. The design of *internal interfaces* is closely aligned with component-level design.

Component-Level Design Elements: The component-level design for software is the equivalent to a set of detailed drawings (and specifications) for each room in a house. The component-level design for software fully describes the internal detail of each software component. To accomplish this, the component-level design defines data structures for all local data objects and algorithmic detail for all processing that occurs within a component and an interface that allows access to all component operations (behaviors). Within the context of object-oriented software engineering, a component is represented in UML diagrammatic form as shown in Figure 8.6.
A UML activity diagram can be used to represent processing logic. Detailed procedural flow for a component can be represented using either pseudocode or some other diagrammatic form (e.g.,

flowchart or box diagram). Algorithmic structure follows the rules established for structured programming (i.e., a set of constrained procedural constructs). Data structures, selected based on the nature of the data objects to be processed, are usually modeled using pseudocode or the programming language to be used for implementation.

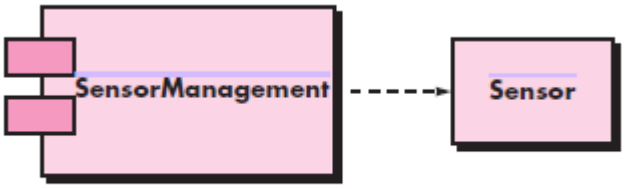


Fig 8.6: A UML component diagram

Deployment-Level Design Elements: Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.

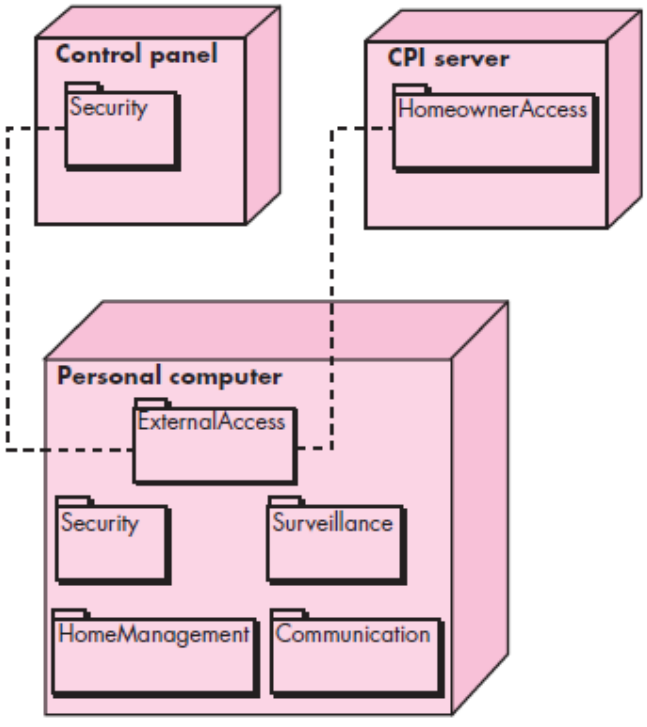


Fig 8.7: A UML deployment diagram

During design, a UML deployment diagram is developed and then refined as shown in Figure 8.7. The diagram shown is in descriptor form. This means that the deployment diagram shows the computing environment but does not explicitly indicate configuration details.

SOFTWARE ARCHITECTURE

(Architecture → Architecture Style → Architecture Pattern → Design)

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.
Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style from the requirements derived during software requirements analysis.
Why is it important? It provides you with the big picture and ensures that you've got it right.
What are the steps? Architectural design begins with *data design* and then proceeds to the derivation of one or more representations of the architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.
What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.
How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

What Is Architecture? : Software architecture must model the structure of a system and the manner in which data and procedural components collaborate with one another.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.
The architecture is not the operational software. Rather, it is a representation that enables you to
(1) analyze the effectiveness of the design in meeting its stated requirements
(2) architectural alternatives at a stage when making design changes is still relatively easy
(3) reduce the risks associated with the construction of the software.
This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a

program module or an object-oriented class, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary for an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

There is a distinct difference between the terms *architecture* and *design*. A design is an instance of an architecture similar to an object being an instance of a class. For example, consider the client-server architecture. I can design a network-centric software system in many different ways from this architecture using either the Java platform (Java EE) or Microsoft platform (.NET framework). So, there is one architecture, but many designs can be created based on that architecture. *Architectural design focuses on the representation of the structure of software components, their properties, and interactions.*

Why Is Architecture Important?: Three key reasons that software architecture is 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".

Architectural Descriptions: Different stakeholders will see an architecture from different viewpoints that are driven by different sets of concerns. This implies that an architectural description is actually a set of work products that reflect different views of the system.

Tyree and *Akerman* note this when they write: "Developers want clear, decisive guidance on how to proceed with design. Customers want a clear understanding on the environmental changes that must occur and assurances that the architecture will meet their business needs. Other architects want a clear, salient understanding of the architecture's key aspects." Each of these "wants" is reflected in a different view represented using a different viewpoint.

The IEEE standard defines an architectural description (AD) as "*a collection of products to document an architecture*." The description itself is represented using multiple views, where each view is "a representation of a whole system from the perspective of a related set of [stakeholder] concerns."

Architectural Decisions: Each view developed as part of an architectural description addresses a specific stakeholder concern. *To develop each view the system architect considers a variety of alternatives and ultimately decides on the specific architectural features that best meet the concern*. Therefore, architectural decisions themselves can be considered to be one view of the architecture.

ARCHITECTURAL GENRES

Although the underlying principles of architectural design apply to all types of architecture, *the architectural genre will often dictate the specific architectural approach to the structure that must be built*. In the context of architectural design, genre implies a specific category within the overall software domain. Within each category, you encounter a number of subcategories. For example, within the genre of buildings, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.

Grady Booch suggests the following architectural genres for software-based systems:

- **Artificial intelligence**—Systems that simulate or augment human cognition, locomotion, or other organic processes.
- **Commercial and nonprofit**—Systems that are fundamental to the operation of a business enterprise.
- **Communications**—Systems that provide the infrastructure for transferring and managing data, for connecting users of that data, or for presenting data at the edge of an infrastructure.
- **Content authoring**—Systems that are used to create or manipulate textual or multimedia artifacts.
- **Devices**—Systems that interact with the physical world to provide some point service for an individual.
- **Entertainment and sports**—Systems that manage public events or that provide a large group entertainment experience.
- **Financial**—Systems that provide the infrastructure for transferring and managing money and other securities.
- **Games**—Systems that provide an entertainment experience for individuals or groups.
- **Government**—Systems that support the conduct and operations of a local, state, federal, global, or other political entity.
- **Industrial**—Systems that simulate or control physical processes.
- **Legal**—Systems that support the legal industry.
- **Medical**—Systems that diagnose or heal or that contribute to medical research.
- **Military**—Systems for consultation, communications, command, control, and intelligence (C4I) as well as offensive and defensive weapons.
- **Operating systems**—Systems that sit just above hardware to provide basic software services.
- **Platforms**—Systems that sit just above operating systems to provide advanced services.

• **Scientific**—Systems that are used for scientific research and applications.
• **Tools**—Systems that are used to develop other systems.
• **Transportation**—Systems that control water, ground, air, or space vehicles.
• **Utilities**—Systems that interact with other software to provide some point service.

SAI (Software Architecture for Immersipresence) is a new software architecture model for designing, analyzing and implementing applications performing distributed, asynchronous parallel processing of generic data streams. The goal of SAI is to provide a universal framework for the distributed implementation of algorithms and their easy integration into complex systems.

ARCHITECTURAL STYLES

Architectural 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.

An architectural style is a transformation that is imposed on the design of an entire system. The intent is to establish a structure for all components of the system.

An architectural pattern, like an architectural style, imposes a transformation on the design of an architecture. However, a pattern differs from a style in a number of fundamental ways:
(1) the scope of a pattern is less broad, focusing on one aspect of the architecture rather than the architecture in its entirety;
(2) a pattern imposes a rule on the architecture, describing how the software will handle some aspect of its functionality at the infrastructure level (e.g., concurrency)
(3) architectural patterns tend to address specific behavioral issues within the context of the architecture (e.g., how real-time applications handle synchronization or interrupts).
Patterns can be used in conjunction with an architectural style to shape the overall structure of a system.

A Brief Taxonomy of Architectural Styles: Although millions of computer-based systems have been created over the past 60 years, the vast majority can be categorized into one of a relatively small number of architectural styles:
Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 9.1 illustrates a typical data-centered style. Client

software accesses a central repository. In some cases the data repository is passive. That is, client software accesses the data independent of any changes to the data or the actions of other client software. Data-centered architectures promote integrability. That is, existing components can be changed and new client components added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

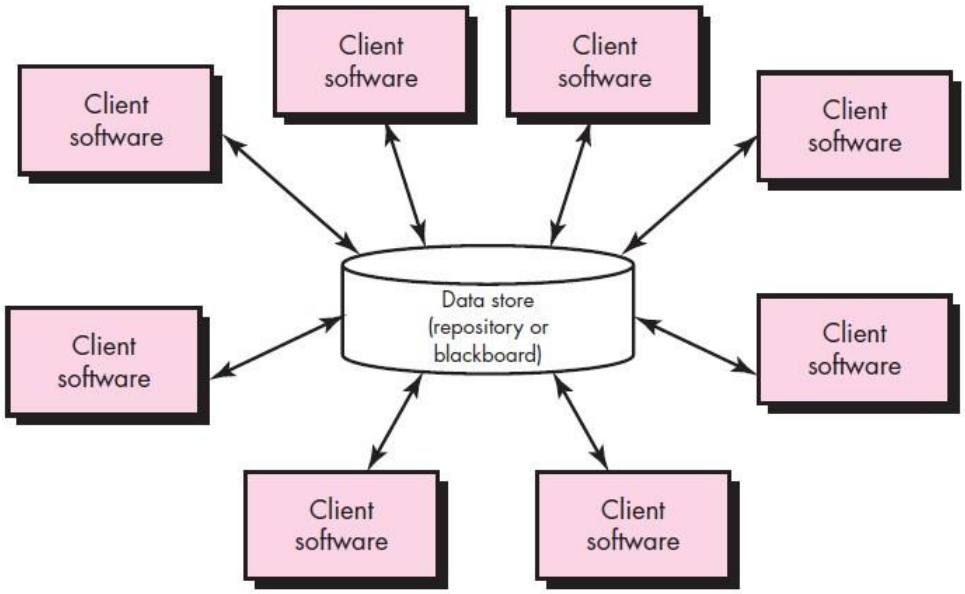
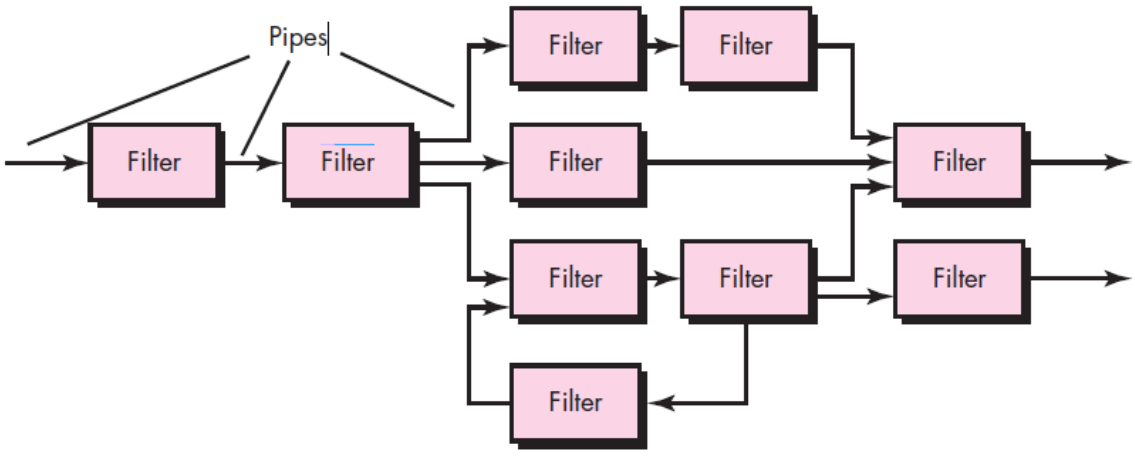


Fig 9.1: Data-centered architecture

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A pipe-and-filter pattern (Figure 9.2) has a set of components, called filters, connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the workings of its neighboring filters.



Pipes and filters

Fig 9.2: Data-flow architecture

Call and return architectures. This architectural style enables you to achieve a program structure that is relatively easy to modify and scale. A number of substyles exist within this category:

• *Main program/subprogram architectures*. This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components that in turn may invoke still other components. Figure 9.3 illustrates an architecture of this type.

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

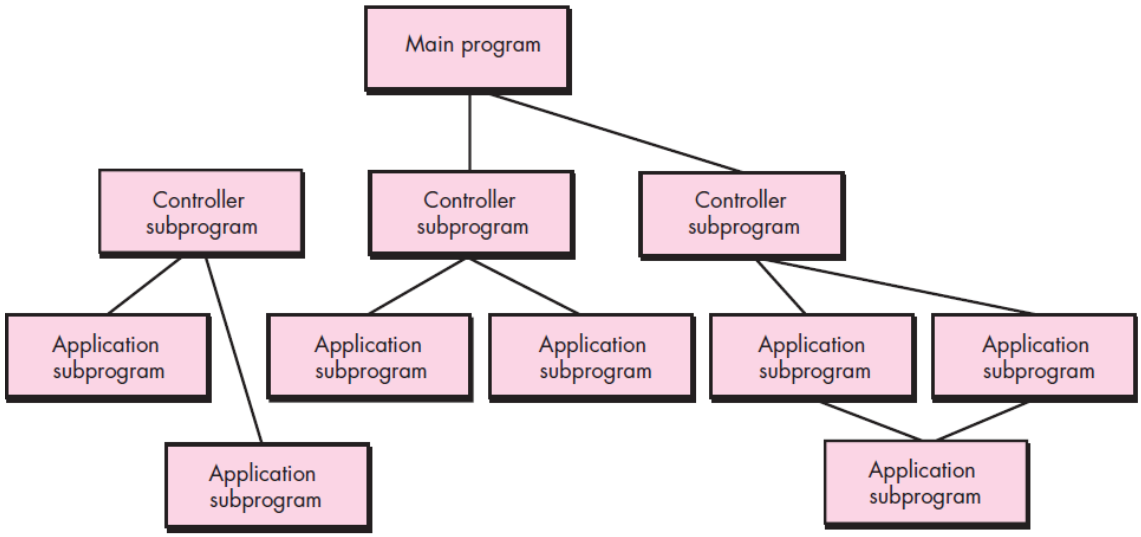


Fig 9.3: Main program / subprogram architecture

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

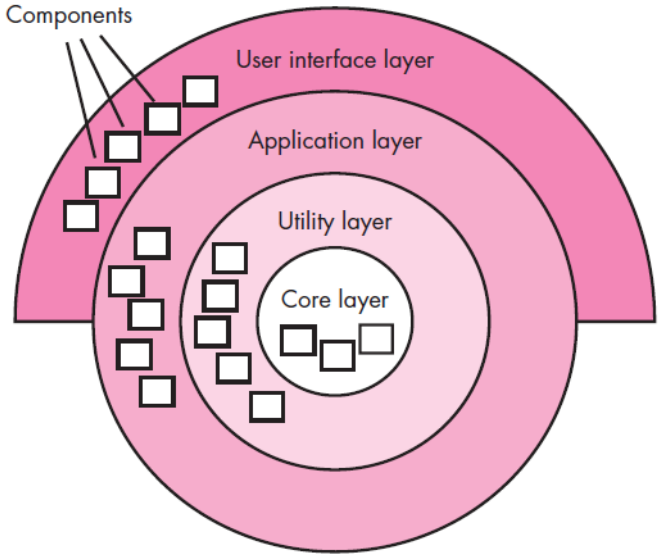


Fig 9.4: Layered architecture

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 9.4. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions. These architectural styles are only a small subset of those available. Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural style and/or combination of patterns that best fits those characteristics and constraints can be chosen. For example, a layered style (appropriate for most systems) can be combined with a data-centered architecture in many database applications.

Architectural Patterns: Architectural patterns address an application-specific problem within a specific context and under a set of limitations and constraints. The pattern proposes an architectural solution that can serve as the basis for architectural design.

For example, the overall architectural style for an application might be call-and-return or object-oriented. But within that style, you will encounter a set of common problems that might best be addressed with specific architectural patterns.

Organization and Refinement: Because the design process often leaves you with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions provide insight into an architectural style:

Control. How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form that the control takes)? Is control synchronized or do components operate asynchronously?
Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components passive or active (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?
These questions provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture.

ARCHITECTURAL DESIGN

The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction. This information can generally be acquired from the requirements model and all other information gathered during requirements engineering. Once context is modeled and all external software interfaces have been described, you can identify a set of architectural archetypes. *An archetype is an abstraction (similar to a class) that represents one element of system behavior*. The set of archetypes provides a collection of abstractions that must be modeled architecturally if the system is to be constructed, but the archetypes themselves do not provide enough implementation detail.

Therefore, the designer specifies the structure of the system by defining and refining software components that implement each archetype. This process continues iteratively until a complete architectural structure has been derived.

Representing the System in Context: *Architectural context represents how the software interacts with entities external to its boundaries*. 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 9.5. Referring to the figure, systems that interoperate with the target system are represented as
• *Superordinate systems*—those systems that use the target system as part of some higher-level processing scheme.
• *Subordinate systems*—those systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.

• *Peer-level systems*—those systems that interact on a peer-to-peer basis (i.e.,information is either produced or consumed by the peers and the target system.

• *Actors—entities* (people, devices) that interact with the target system by producing or consuming information that is necessary for requisite processing. Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

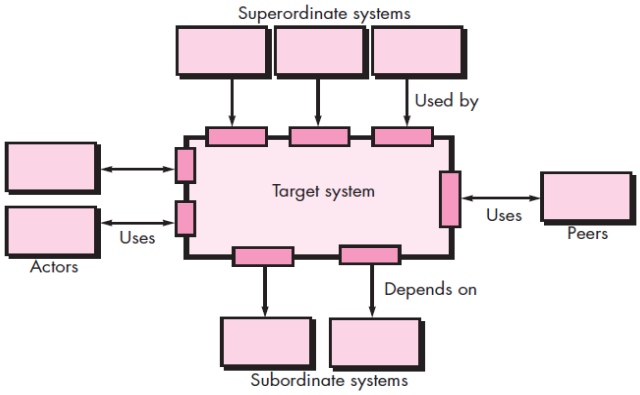


Fig 9.5: Architectural context diagram

Defining Archetypes: *Archetypes are the abstract building blocks of an architectural design*. An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system. In general, a relatively small set of archetypes is required to design even relatively complex systems. The target system architecture is composed of these archetypes, which represent stable elements of the architecture but may be instantiated many different ways based on the behavior of the system.

The following are the archetypes for safeHome: Node, Detector, Indicator., Controller.

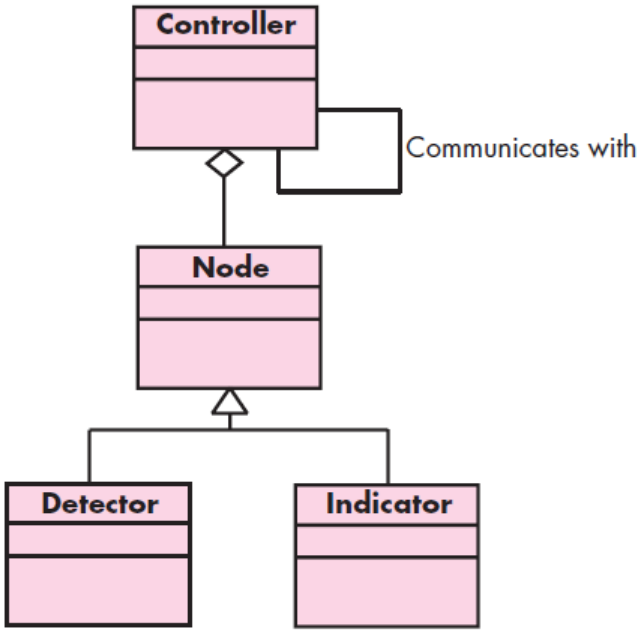


Fig 9.7: UML relationships for safehome function Archetypes

Refining the Architecture into Components: As the software architecture is refined into components, the structure of the system begins to emerge. The architecture must accommodate many infrastructure components that enable application components but have no business connection to the application domain. For example, memory management components, communication components, database components, and task management components are often integrated into the software architecture.

As an example for SafeHome home security, the set of top-level components that address the following functionality:

• *External communication management*—coordinates communication of the security function with external entities such as other Internet-based systems and 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.

Each of these top-level components would have to be elaborated iteratively and then positioned within the overall SafeHome architecture.

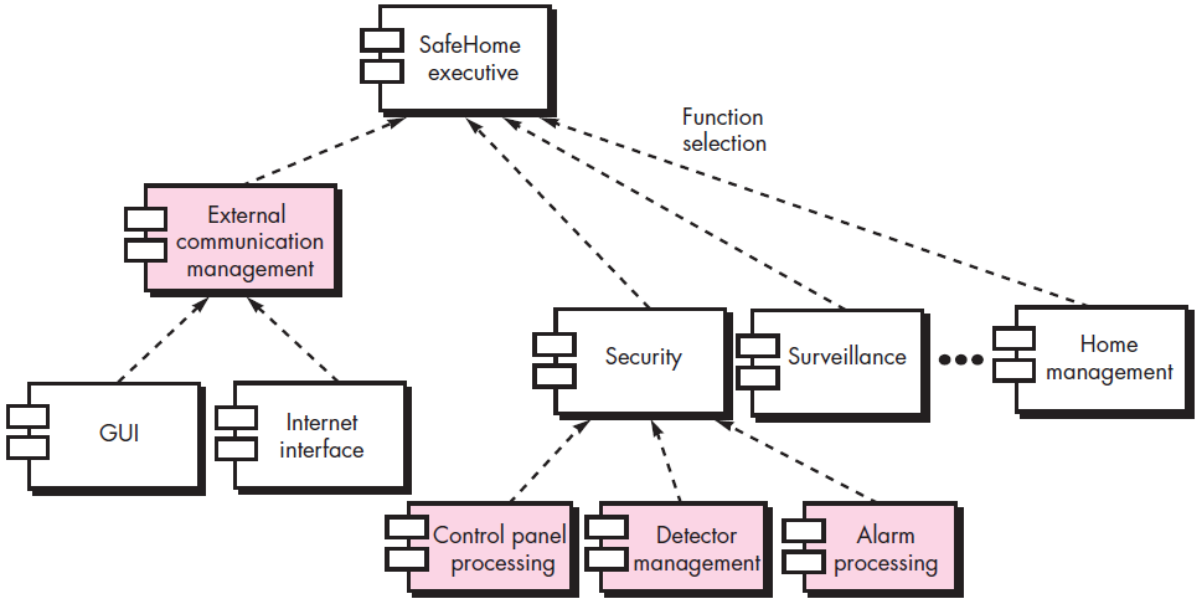


Fig 9.8: Overall architectural structure for *SafeHome* with top-level components

Describing Instantiations of the System: The architectural design that has been modeled to this point is still relatively *high level*. The context of the system has been represented, archetypes that indicate the important abstractions within the problem domain have been defined, the overall structure of the system is apparent, and the major software components have been identified. However, *further refinement* is still necessary.

ASSESSING ALTERNATIVE ARCHITECTURAL DESIGNS

Design results in a number of architectural alternatives that are each assessed to determine which is the most appropriate for the problem to be solved. Two different approaches for the assessment of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

3.8.1 An Architecture Trade-Off Analysis Method: The Software Engineering Institute (SEI) has developed an architecture trade-off analysis method (ATAM) that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

1. *Collect scenarios*. A set of use cases is developed to represent the system from the user's point of view.
2. *Elicit requirements*, constraints, and environment description. This information is determined as part of requirements engineering and is used to be certain that all stakeholder concerns have been addressed.
3. *Describe the architectural styles/patterns* that have been chosen to address the scenarios and requirements. The architectural style(s) should be described using one of the following architectural views:

• Module view for analysis of work assignments with components and the degree to which information hiding has been achieved.
• Process view for analysis of system performance.
• Data flow view for analysis of the degree to which the architecture meets functional requirements.

4. *Evaluate quality attributes* by considering each attribute in isolation. The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. *Identify the sensitivity of quality attributes* to various architectural attributes for a specific architectural style. This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed sensitivity points.
6. *Critique candidate architectures* using the sensitivity analysis conducted in step 5.

The SEI describes this approach in the following manner.
Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.

3.8.2 Architectural Complexity: A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system.
The three types of dependencies:
Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components u and v, if u and v refer to the same global data, then there exists a shared dependence relationship between u and v.
Flow dependencies represent dependence relationships between producers and consumers of resources. For example, for two components u and v, if u must complete before control flows into v (prerequisite), or if u communicates with v by parameters, then there exists a flow dependence relationship between u and v.
Constrained dependencies represent constraints on the relative flow of control among a set of activities. For example, for two components u and v, u and v cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between u and v.

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

3.9 ARCHITECTURAL MAPPING USING DATA FLOW

There is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in using the techniques.

To illustrate one approach to architectural mapping, consider the call and return architecture—an extremely common structure for many types of systems. The call and return architecture can reside within other more sophisticated architectures. For example, the architecture of one or more components of a client-server architecture might be call and return. A mapping technique, called structured design, is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture. The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six step process: (1) the type of information flow is established, (2) flow boundaries are

indicated, (3) the DFD is mapped into the program structure, (4) control hierarchy is defined, (5) the resultant structure is refined using design measures and heuristics, and (6) the architectural description is refined and elaborated.

3.9.1 Transform Mapping: *Transform mapping* is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style.

Step 1. Review the fundamental system model. The fundamental system model or context diagram depicts the security function as a single transformation, representing the external producers and consumers of data that flow into and out of the function. Figure 9.10 depicts a level 0 context model, and Figure 9.11 shows refined data flow for the security function.

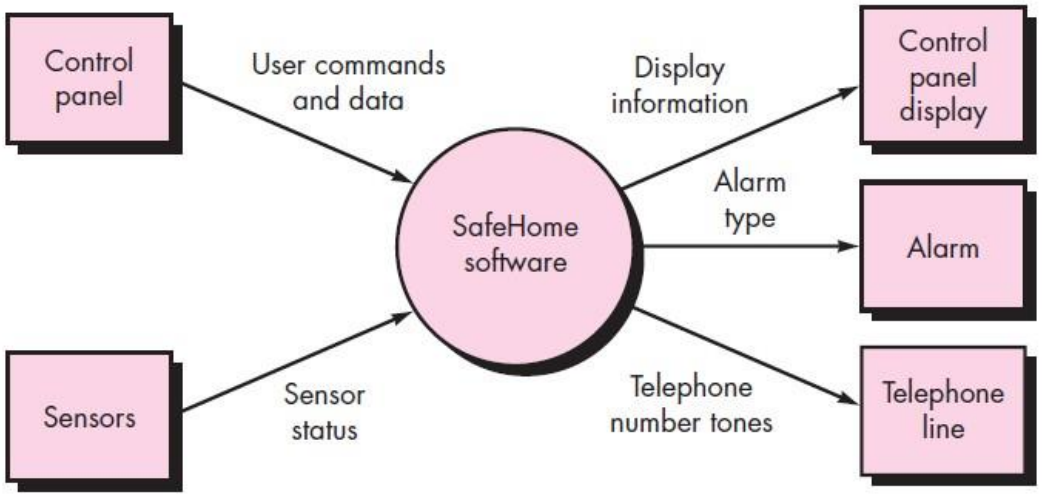


Fig 9.10 Context-level DFD for safeHome security function

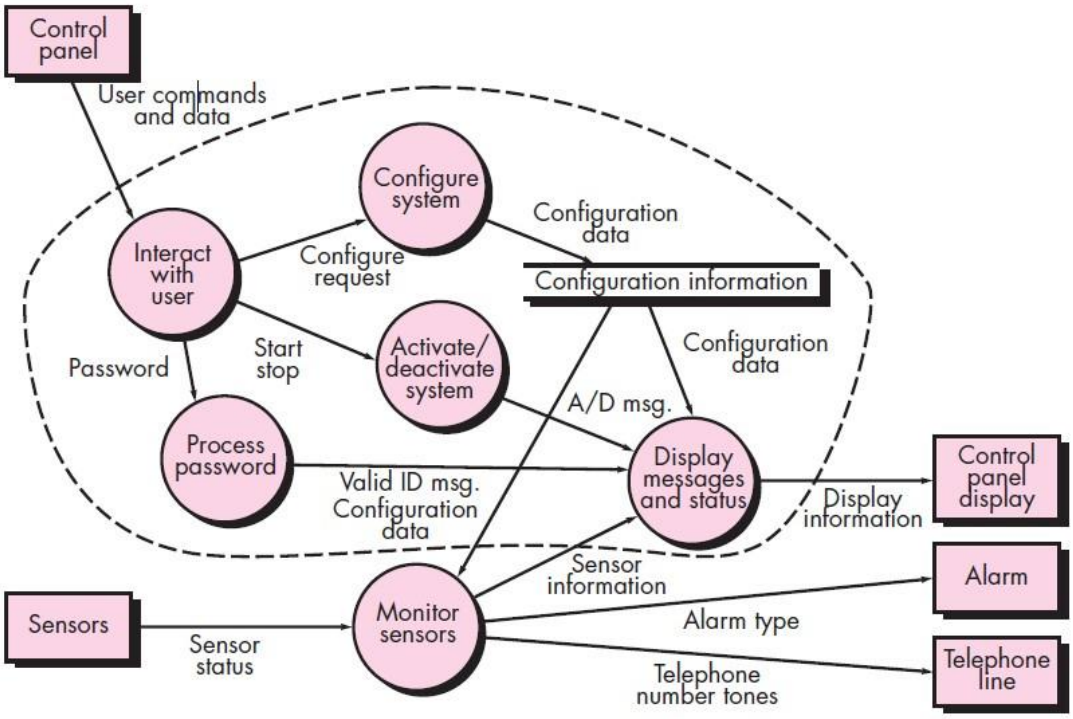


Fig 9.11: Level 1 DFD for safeHome security function

Step 2. Review and refine data flow diagrams for the software. Information obtained from the requirements model is refined to produce greater detail. For example, the level 2 DFD for monitor sensors (Figure 9.12) is examined, and a level 3 data flow diagram is derived as shown in Figure 9.13. The data flow diagram exhibits relatively high cohesion.

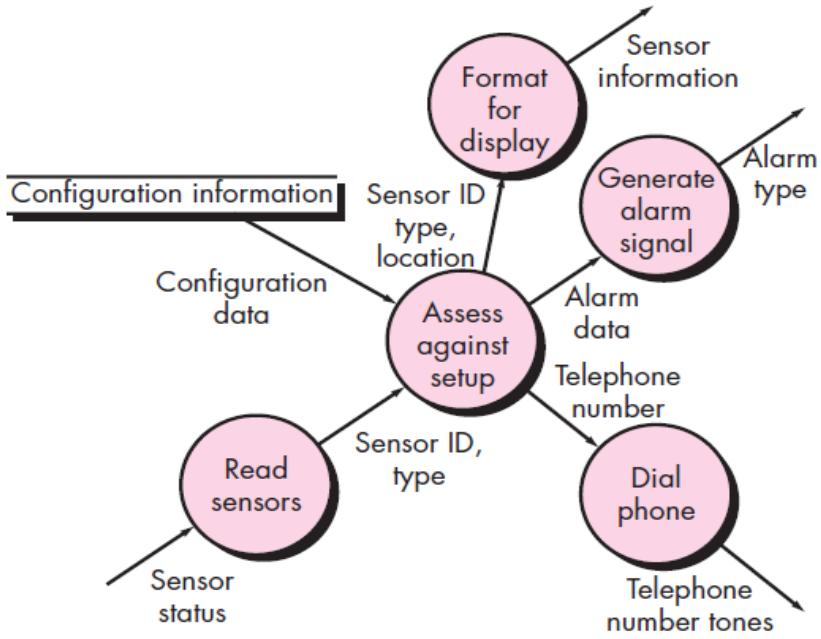


Fig 9.12: Level 2 DFD that refines the monitor sensors transform

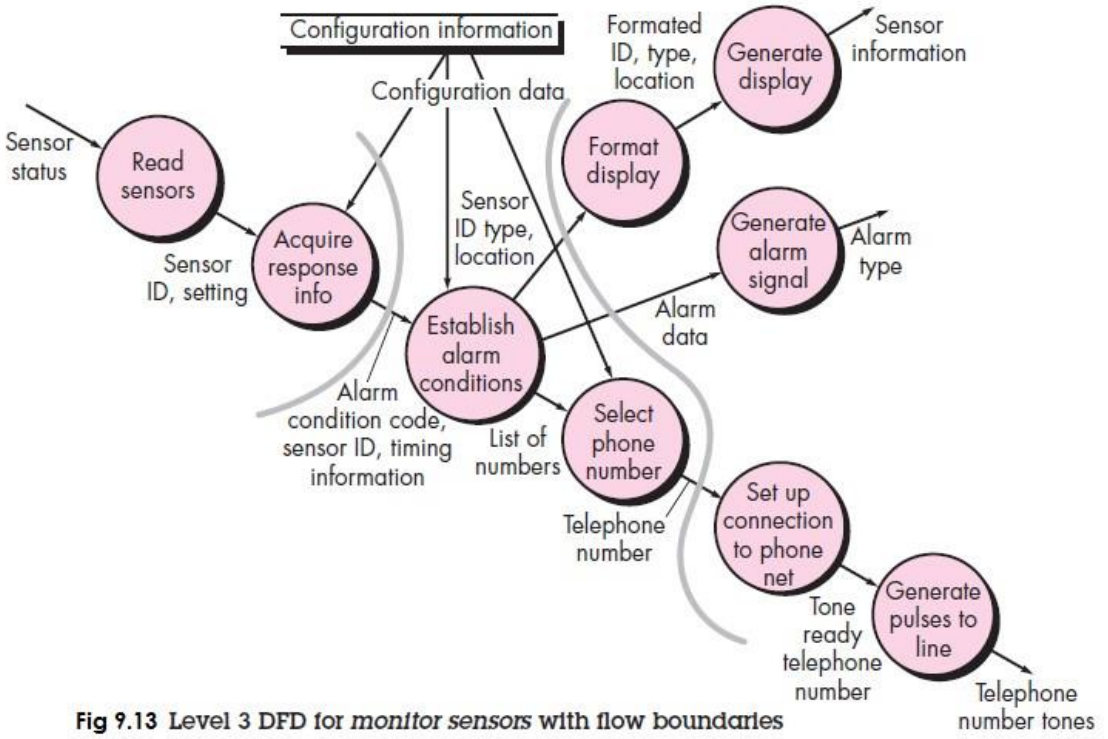


Fig 9.13 Level 3 DFD for *monitor sensors* with flow boundaries

Step 3. Determine whether the DFD has transform or transaction flow characteristics. Evaluating the DFD (Figure 9.13), we see data entering the software along one incoming path and exiting along three outgoing paths. Therefore, an overall transform characteristic will be assumed for information flow.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. Incoming data flows along a path in which information is converted from external to internal form; outgoing flow converts internalized data to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 9.13. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary. The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

Step 5. Perform "first-level factoring." The program architecture derived using this mapping results in a top-down distribution of control. Factoring leads to a program structure in which top-level components perform decision making and low level components perform most input, computation, and output work. Middle-level components perform some control and do moderate amounts of work.

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture.

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. A first-iteration architecture can always be refined by applying concepts of functional independence. Components are exploded or imploded to produce sensible factoring, separation of concerns, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

3.9.2 Refining the Architectural Design: Refinement of software architecture during early stages of design is to be encouraged. Alternative architectural styles may be derived, refined, and evaluated for the "best" approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of components that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

3.10 Component-Level Design

What is it? Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each software component.

Who does it? A software engineer performs component-level design.

Why is it important? You have to be able to determine whether the software will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with other design representations (i.e., the data, architectural, and interface designs). It provides a means for assessing whether data structures, interfaces, and algorithms will work.

What are the steps? Design representations of data, architecture, and interfaces form the foundation for component-level design. The class definition or processing narrative for each component is translated into a detailed design that makes use of diagrammatic or text-based forms that specify internal data structures, local interface detail, and processing logic. Design notation encompasses UML diagrams and supplementary forms. Procedural design is specified using a set of structured programming constructs. It is often possible to acquire existing reusable software components rather than building new ones.

What is the work product? The design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

How do I ensure that I've done it right? A design review is conducted. The design is examined to determine whether data structures, interfaces, processing sequences, and logical conditions are correct and will produce the appropriate data or control transformation allocated to the component during earlier design steps.

3.10 WHAT IS A COMPONENT?: A component is a modular building block for computer software. More formally, component is *"a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."*

Components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

3.10.1 An Object-Oriented View: In the context of object-oriented software engineering, a component contains a set of collaborating classes. Each class within a component has been fully elaborated to include all *attributes* and *operations* that are relevant to its implementation. As part of the design elaboration, all *interfaces* that enable the classes to communicate and collaborate with other design classes must also be defined.

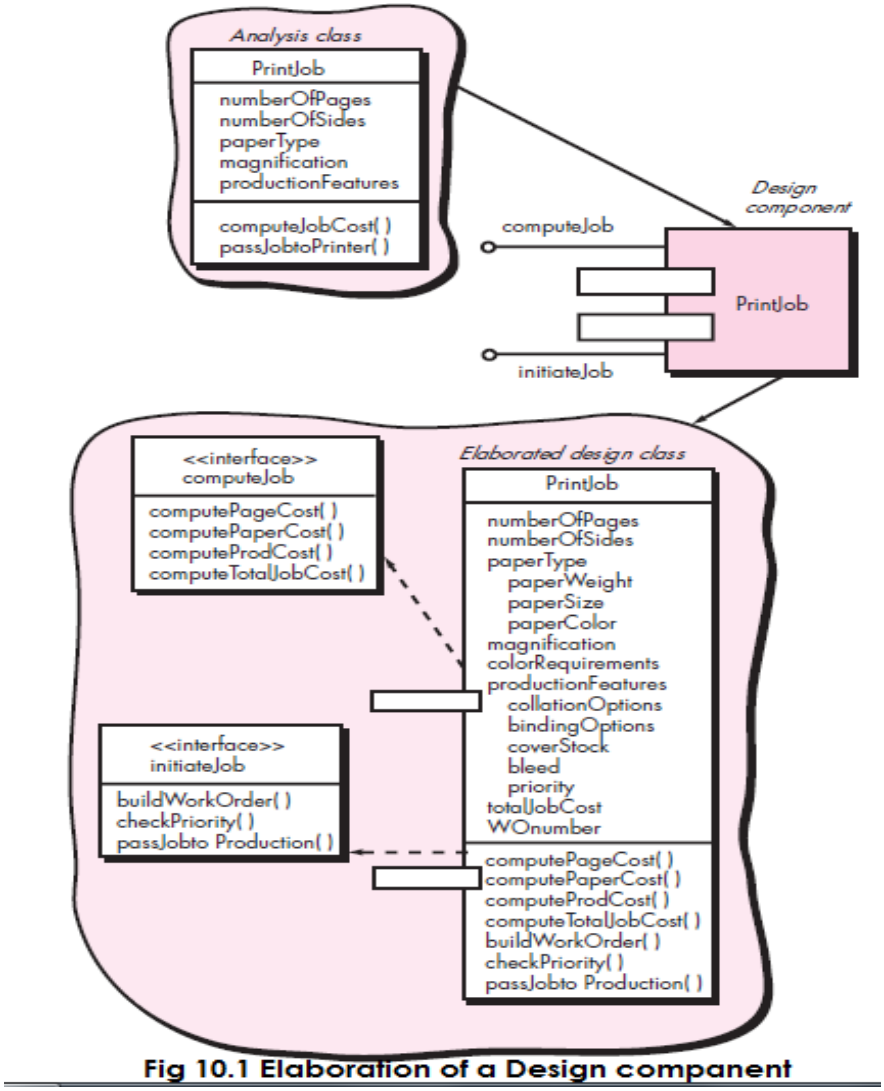


Fig 10.1 Elaboration of a Design component

3.10.2 The Traditional View: In the context of traditional software engineering, a component is a functional element of a program that *incorporates processing logic, the internal data structures* that are required to implement the processing logic, and *an interface* that enables the component to be invoked and data to be passed to it. A traditional component, also called a *module*, resides within the software architecture and serves one of three important roles:
(1) a control component that coordinates the invocation of all other problem domain components,
(2) a problem domain component that implements a complete or partial function that is required by the customer, or
(3) an infrastructure component that is responsible for functions that support the processing required in the problem domain.

Like object-oriented components, traditional software components are derived from the analysis model. To achieve effective modularity, design concepts like functional independence are applied as components are elaborated.

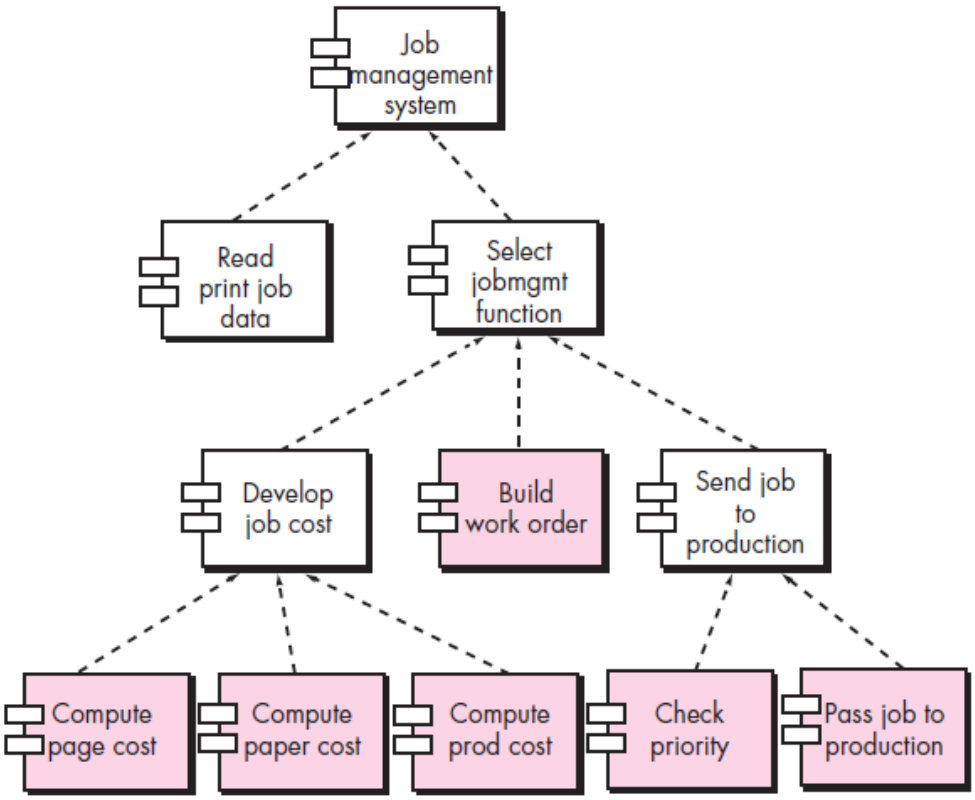


Fig 10.2: Structure chart for a traditional system

During component-level design, each module in Figure 10.2 is elaborated. The module interface is defined explicitly. That is, each data or control object that flows across the interface is represented. The data structures that are used internal to the module are defined. The algorithm that allows the module to accomplish its intended function is designed using the stepwise

refinement approach. The behavior of the module is sometimes represented using a state diagram. Figure 10.3 represents the component-level design using a modified UML notation.

3.10.3 A Process-Related View: The object-oriented and traditional views of component-level design assume that the component is being designed from scratch. That is, you have to create a new component based on specifications derived from the requirements model.

Over the past two decades, the software engineering community has emphasized the need to build systems that make use of existing software components or design patterns. In essence, a catalog of proven design or code-level components is made available to you as design work proceeds. As the software architecture is developed, you choose components or design patterns from the catalog and use them to populate the architecture.

Because these components have been created with reusability in mind, a complete description of their interface, the function(s) they perform, and the communication and collaboration they require are all available.

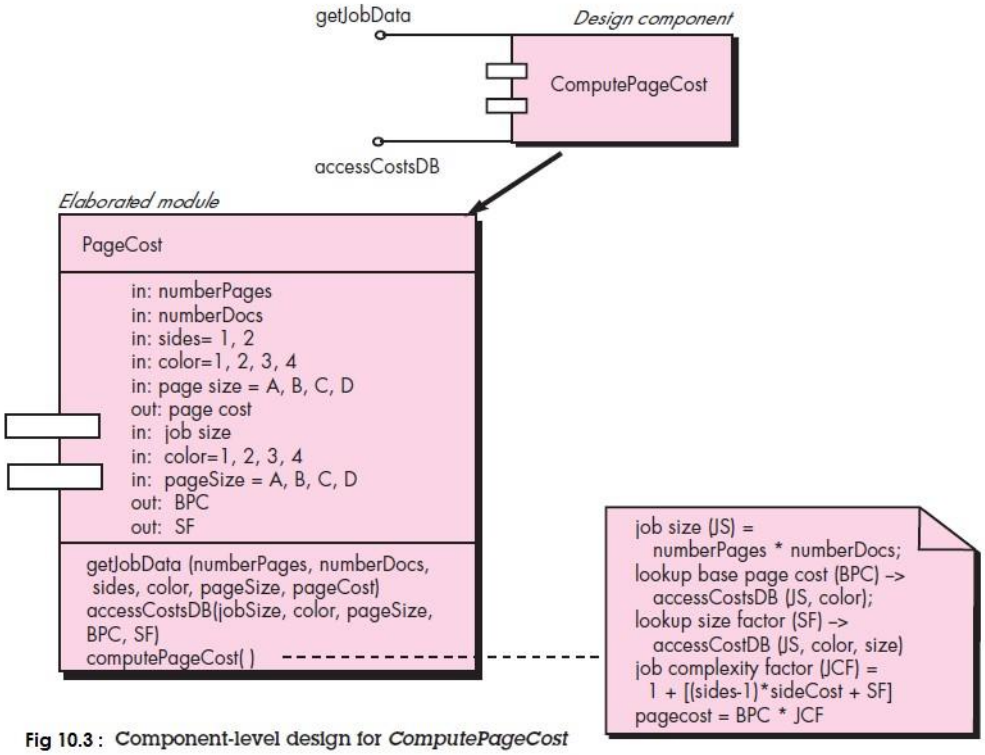


Fig 10.3 : Component-level design for *ComputePageCost*

3.11 DESIGNING CLASS-BASED COMPONENTS

When an object-oriented software engineering approach is chosen, component-level design focuses on the elaboration of problem domain specific classes and the definition and refinement of infrastructure classes contained in the requirements model. The detailed description of the attributes, operations, and interfaces used by these classes is the design detail required as a precursor to the construction activity.

3.11.1 Basic Design Principles: Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied.
The Open-Closed Principle (OCP). "A module [component] should be *open for extension but closed for modification*". This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended without the need to make internal (code or logic-level) modifications to the component itself. To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself.

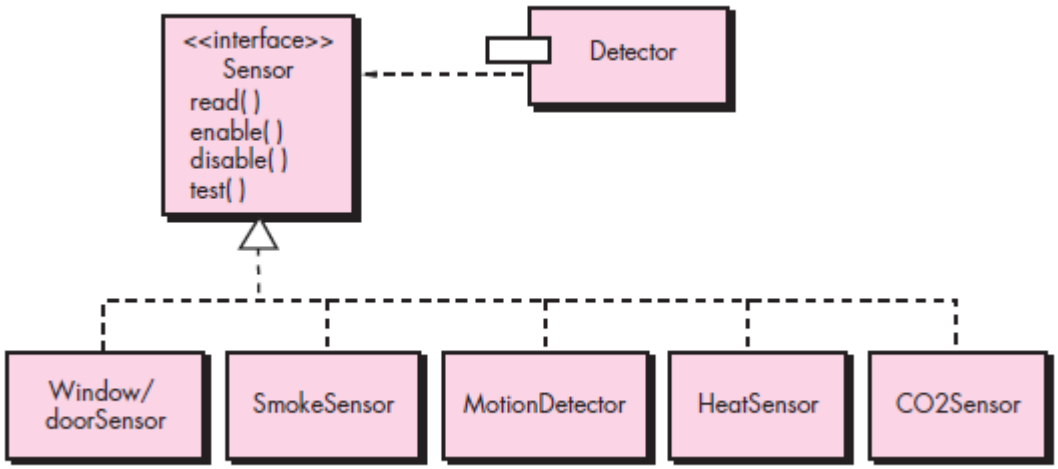


Fig 10.4: Following the OCP

The Liskov Substitution Principle (LSP). "Subclasses should be substitutable for their base classes". This design principle, originally proposed by Barbara Liskov, suggests that a component that uses a base class should continue to function properly if a class derived from the base class is passed to the component instead. LSP demands that any class derived from a base class must honor any implied contract between the base class and the components that use it.

Dependency Inversion Principle (DIP). "Depend on abstractions. Do not depend on concretions". As we have seen in the discussion of the OCP, abstractions are the place where a design can be extended without great complication. The more a component depends on other

concrete components (rather than on abstractions such as an interface), the more difficult it will be to extend.

The Interface Segregation Principle (ISP). "Many client-specific interfaces are better than one general purpose interface". There are many instances in which multiple client components use the operations provided by a server class. *ISP suggests that you should create a specialized interface to serve each major category of clients.* Only those operations that are relevant to a particular category of clients should be specified in the interface for that client. If multiple clients require the same operations, it should be specified in each of the specialized interfaces.

Martin suggests additional *packaging* principles that are applicable to component-level design:
The Release Reuse Equivalency Principle (REP). "The granule of reuse is the granule of release". When classes or components are designed for reuse, there is an implicit contract that is established between the developer of the reusable entity and the people who will use it.
Rather than addressing each class individually, it is often advisable to group reusable classes into packages that can be managed and controlled as newer versions evolve.
The Common Closure Principle (CCP). "Classes that change together belong together." Classes should be packaged cohesively. That is, when classes are packaged as part of a design, they should address *the same functional or behavioral area*. When some characteristic of that area must change, it is likely that only those classes within the package will require modification. This leads to more effective change control and release management.
The Common Reuse Principle (CRP). "Classes that aren't reused together should not be grouped together" If classes are not grouped cohesively, it is possible that a class with no relationship to other classes within a package is changed. This will precipitate unnecessary integration and testing. For this reason, only classes that are reused together should be included within a package.

3.11.2 Component-Level Design Guidelines: Ambler suggests the following guidelines for components, their interfaces, and the dependencies and inheritance characteristics
Components. Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model. Architectural component names should be drawn from the problem domain and should have meaning to all stakeholders who view the architectural model.
Interfaces. Interfaces provide important information about communication and collaboration. However, unfettered representation of interfaces tends to complicate component diagrams. Ambler recommends that (1) lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach, when diagrams grow complex; (2) for consistency, interfaces should flow from the left-hand side of the component box; (3) only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available.

Dependencies and Inheritance. For improved readability, it is a good to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes). In addition, component interdependencies should be represented via interfaces, rather than by representation of a component-to-component dependency.

3.11.3 Cohesion: Cohesion is the "single-mindedness" of a component. Within the context of component-level design for object-oriented systems, cohesion 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. Lethbridge and Laganiére define a number of different types of cohesion.

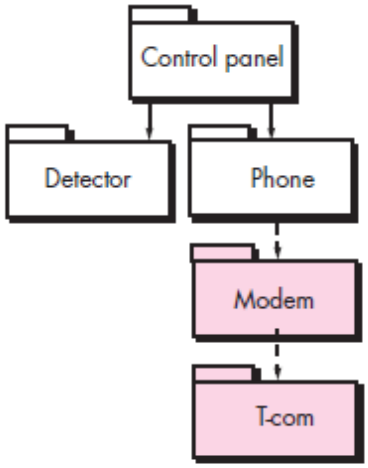


Fig 10.5: Layer cohesion

Functional. Exhibited primarily by operations, this level of cohesion occurs when a component performs a targeted computation and then returns a result.
Layer. Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers. It might be possible to define a set of layered packages as shown in Figure 10.5. The shaded packages contain infrastructure components.
Communicational. All operations that access the same data are defined within one class. In general, such classes focus solely on the data in question, accessing and storing it. Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.

3.11.4 Coupling: Communication and collaboration are essential elements of any object-oriented system. *Coupling is a qualitative measure of the degree to which classes are connected toone another.* As classes (and components) become more interdependent, coupling increases. An important objective in component-level design is to keep coupling as low as is possible.
Lethbridge and Laganiére define the following coupling categories:
Content coupling. Occurs when one component "surreptitiously modifies data that is internal to another component". This violates information hiding—a basic design concept.

Common coupling. Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.

Control coupling. Occurs when operation A() invokes operation B() and passes a control flag to B. The control flag then "directs" logical flow within B. The problem with this form of coupling is that an unrelated change in B can result in the necessity to change the meaning of the control flag that A passes. If this is overlooked, an error will result.

Stamp coupling. Occurs when ClassB is declared as a type for an argument of an operation of ClassA. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex.

Data coupling. Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.

Routine call coupling. Occurs when one operation invokes another. This level of coupling is common and is often quite necessary. However, it does increase the connectedness of a system.

Type use coupling. Occurs when component A uses a data type defined in component B (e.g., this occurs whenever "a class declares an instance variable or a local variable as having another class for its type". If the type definition changes, every component that uses the definition must also change.

Inclusion or import coupling. Occurs when component A imports or includes a package or the content of component B.

External coupling. Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Software must communicate internally and externally. Therefore, coupling is a fact of life. However, the designer should work to reduce coupling whenever possible and understand the ramifications of high coupling when it cannot be avoided.

3.11.5 CONDUCTING COMPONENT-LEVEL DESIGN: The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain. Using the requirements and architectural model, each analysis class and architectural component is elaborated.

Step 2. Identify all design classes that correspond to the infrastructure domain. These classes are not described in the requirements model and are often missing from the architecture model, but they must be described at this point. As we have noted earlier, classes and components in this category include GUI components, operating system components, and object and data management components.

Step 3. Elaborate all design classes that are not acquired as reusable components. Elaboration requires that all interfaces, attributes, and operations necessary to implement the class be described in detail. Design heuristics (e.g., component cohesion and coupling) must be considered as this task is conducted.

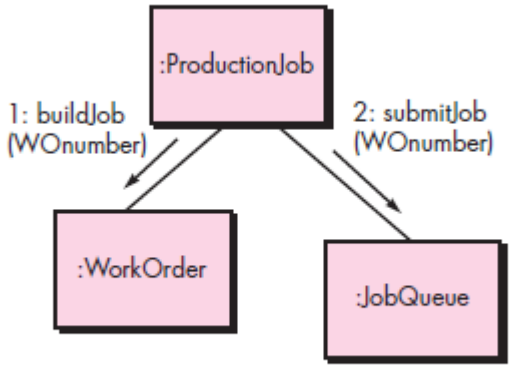


Fig 10.6: Collaboration diagram with messaging

Step 3a. Specify message details when classes or components collaborate. The requirements model makes use of a collaboration diagram to show how analysis classes collaborate with one another. As component-level design proceeds, it is sometimes useful to show the details of these collaborations by specifying the structure of messages that are passed between objects within a system. Although this design activity is optional, it can be used as a precursor to the specification of interfaces that *show how components within the system communicate and collaborate*. Figure 10.6 illustrates a simple collaboration diagram for the printing system.

Step 3b. Identify appropriate interfaces for each component. Within the context of component-level design, a UML interface is "a group of externally visible (i.e., public) operations. The interface contains no internal structure, it has no attributes, no associations. *Interface is the equivalent of an abstract class that provides a controlled connection between design classes*.

Step 3c. Elaborate attributes and define data types and data structures required to implement them. In general, data structures and types used to define attributes are defined within the context of the programming language that is to be used for implementation. UML defines an attribute's data type using the following syntax:

name : *type-expression* _ *initial-value* {*property string*}

where *name* is the attribute name, *type expression* is the data type, initial value is the value that the attribute takes when an object is created, and *property-string* defines a property or characteristic of the attribute.

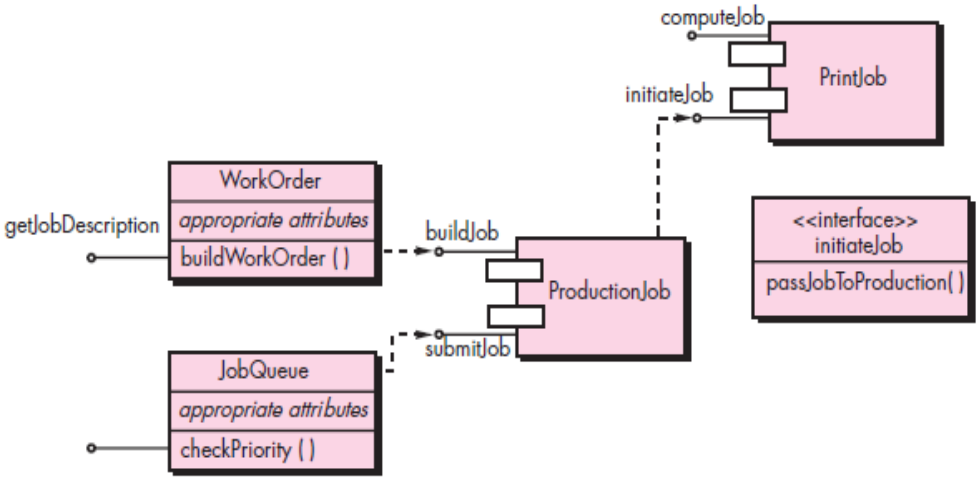


Fig 10.7: Refactoring interfaces and class definitions for PrintJob

Step 3d. Describe processing flow within each operation in detail. This may be accomplished using a programming language-based pseudocode or with a UML activity diagram. Each software component is elaborated through a number of iterations that apply the stepwise refinement concept.

The first iteration defines each operation as part of the design class. In every case, the operation should be characterized in a way that ensures high cohesion; that is, the operation should perform a single targeted function or subfunction. The next iteration does little more than expand the operation name. Figure 10.8 depicts a UML activity diagram for *computePaperCost().*

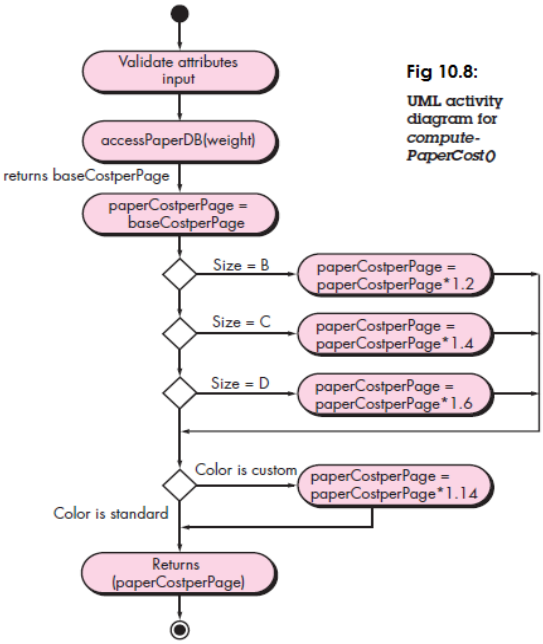


Fig 10.8: UML activity diagram for *compute-PaperCost()*

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them. Databases and files normally transcend the design description of an

individual component. In most cases, these persistent data stores are initially specified as part of architectural design. However, as design elaboration proceeds, it is often useful to provide additional detail about the structure and organization of these persistent data sources.

Step 5. Develop and elaborate behavioral representations for a class or component. UML state diagrams were used as part of the requirements model to represent the externally observable behavior of the system and the more localized behavior of individual analysis classes. During component-level design, it is sometimes necessary to model the behavior of a design class. The dynamic behavior of an object is affected by events that are external to it and the current state of the object as illustrated in Figure 10.9.

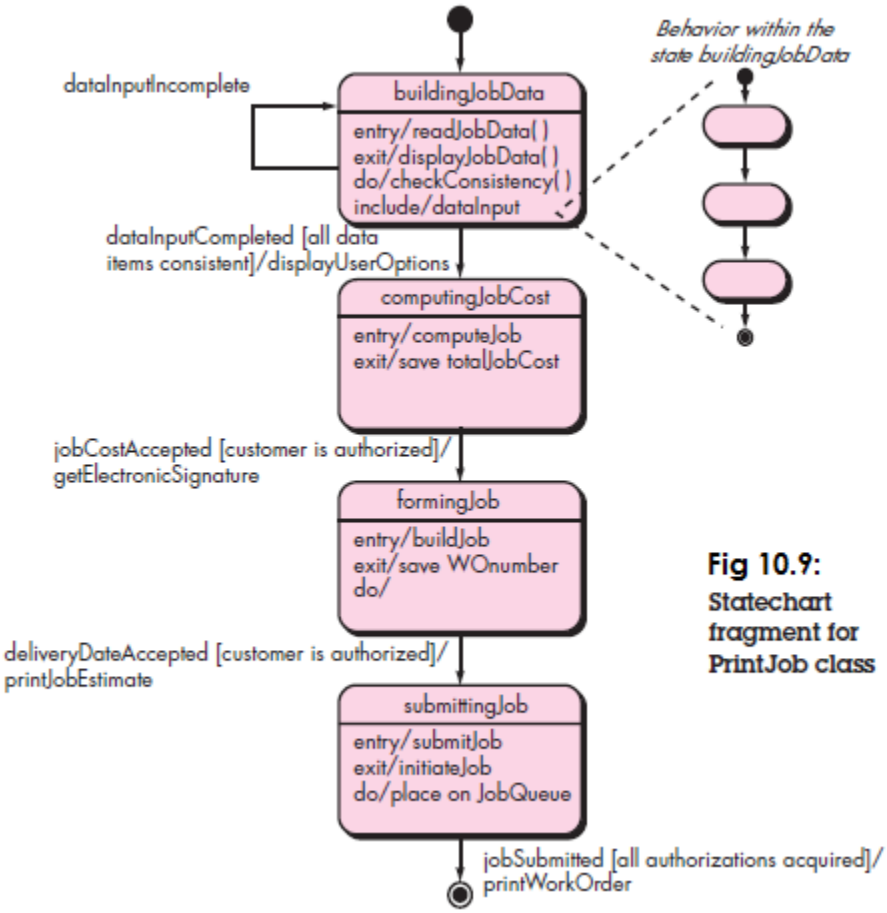


Fig 10.9: Statechart fragment for PrintJob class

Step 6. Elaborate deployment diagrams to provide additional implementation detail. Deployment diagrams are used as part of architectural design and are represented in descriptor form. In this form, major system functions are represented within the context of the computing environment that will house them.

Step 7. Refactor every component-level design representation and always consider alternatives. The design is an iterative process. The first component-level model you create will not be as complete, consistent, or accurate as the n^{th} iteration you apply to the model. It is essential to refactor as design work is conducted. Develop alternatives and consider each carefully, using the design principles and concepts.

3.12 COMPONENT-LEVEL DESIGN FOR WEBAPPS

WebApp component is (1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end user or (2) a cohesive package of content and functionality that provides the end user with some required capability. Therefore, component-level design for WebApps often incorporates elements of content design and functional design.

3.12.1 Content Design at the Component Level: Content design at the component level focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end user.

The formality of *content design* at the component level should be tuned to the characteristics of the WebApp to be built. In many cases, content objects need not be organized as components and can be manipulated individually. However, as the size and complexity grows, it may be necessary to organize content in a way that allows easier reference and design manipulation. In addition, if *content is highly dynamic*, it becomes important to establish a clear structural model that incorporates content components.

3.12.2 Functional Design at the Component Level: Modern Web applications deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion, (2) provide computation or data processing capability that is appropriate for the WebApp's business domain, (3) provide sophisticated database query and access, or (4) establish data interfaces with external corporate systems.
To achieve these (and many other) capabilities, you will design and construct WebApp functional components that are similar in form to software components for conventional software. During architectural design, WebApp content and functionality are combined to create a functional architecture. A functional architecture is a representation of the functional domain of the WebApp and describes the key functional components in the WebApp and how these components interact with each other.

3.13 DESIGNING TRADITIONAL COMPONENTS

The foundations of component-level design for traditional software components were formed in the early 1960s and were solidified with the work of Edsger Dijkstra and his colleagues. The constructs emphasized "*maintenance of functional domain.*" That is, each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are *sequence*, *condition*, and *repetition*. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected

processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to structured programming—an important component-level design technique. The structured constructs were proposed to limit the procedural design of software to a small number of predictable logical structures.

Complexity metrics indicate that the use of the structured constructs reduces program complexity and thereby enhances *readability*, *testability*, and *maintainability*.

The structured constructs are logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

3.13.1 Graphical Design Notation: "A picture is worth a thousand words". There is no question that graphical tools, such as the UML activity diagram or the flowchart, provide useful pictorial patterns that readily depict procedural detail.

The *activity diagram* allows you to represent sequence, condition, and repetition—all elements of structured programming—and is a descendent of an earlier pictorial design representation called a flowchart. A *box* is used to indicate a *processing* step. A *diamond* represents a *logical condition*, and *arrows* show the *flow of control*. Figure 10.10 illustrates three structured constructs.

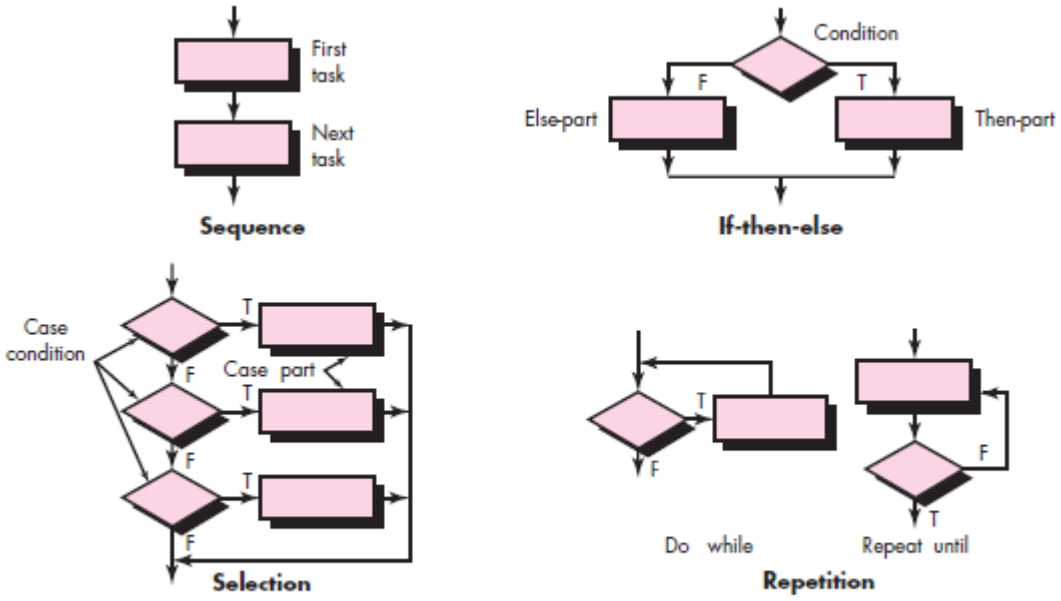


Fig 10.10: Flowchart constructs

The *sequence* is represented as two processing boxes connected by a line (arrow) of control. Condition, also called *if-then-else*, is depicted as a decision diamond that, if true, causes then-part processing to occur, and if false, invokes else-part processing. *Repetition* is represented using two slightly different forms. The *do while* tests a condition and executes a loop task

repetitively as long as the condition holds true. A *repeat until* executes the loop task first and then tests a condition and repeats the task until the condition fails. The *selection* (or select-case) construct shown in the figure is actually an extension of the if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.

3.13.2 Tabular Design Notation: Decision tables provide a notation that translates actions and conditions (described in a processing narrative or a use case) into a tabular form. The table is difficult to misinterpret and may even be used as a machine-readable input to a table-driven algorithm. Decision table organization is illustrated in Figure 10.11. Referring to the figure, the table is divided into four sections. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing rule. The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure (or component).
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 actions occur for a set of conditions.

	Rules					
Conditions	1	2	3	4	5	6
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional *x* percent discount		✓		✓		✓

Fig 10.11: Decision table nomenclature

3.13.3 Program Design Language: Program design language (PDL), also called structured English or pseudocode, incorporates the logical structure of a programming language with the free-form expressive ability of a natural language (e.g., English). Narrative text (e.g., English) is embedded within a programming language-like syntax. Automated tools can be used to enhance the application of PDL.

A basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and input-output (I/O) constructs. It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, interprocess synchronization, and many other features. The application design for which PDL is to be used should dictate the final form for the design language. The format and semantics for some of these PDL constructs are presented in the example that follows.

```
do for all sensors
    invoke checkSensor procedure returning signalValue
    if signalValue > bound [alarmType]
        then phoneMessage = message [alarmType]
            set alarmBell to "on" for alarmTimeSeconds
            set system status = "alarmCondition"
            parbegin
                invoke alarm procedure with "on", alarmTimeSeconds;
                invoke phone procedure set to alarmType, phoneNumber
            endpar
        else skip
    endif
enddofor
end alarmManagement
```

3.14 COMPONENT-BASED DEVELOPMENT

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software "components."

3.14.1 Domain Engineering: The intent of domain engineering is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems. Domain engineering includes three major activities—analysis, construction, and dissemination.

The overall approach to domain analysis is often characterized within the context of object-oriented software engineering. The steps in the process are defined as:

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.

4. Analyze each application in the sample and define analysis classes.
5. Develop a requirements model for the classes.

It is important to note that domain analysis is applicable to any software engineering paradigm and may be applied for conventional as well as object-oriented development.

3.14.2 Component Qualification, Adaptation, and Composition: Domain engineering provides the library of reusable components that are required for component-based software engineering. Some of these reusable components are developed in-house, others can be extracted from existing applications, and still others may be acquired from third parties. Unfortunately, the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application. It is for this reason that a sequence of component-based development actions is applied when a component is proposed for use.

Component Qualification. Component qualification ensures that a candidate component will perform the function required, will properly "fit" into the architectural style specified for the system, and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application.

An interface description provides useful information about the operation and use of a software component, but it does not provide all of the information required to determine if a proposed component can, in fact, be reused effectively in a new application. Among the many factors considered during component qualification are:

- Application programming interface (API).
- Development and integration tools required by the component.
- Run-time requirements, including resource usage (e.g., memory or storage), timing or speed, and network protocol.
- Service requirements, including operating system interfaces and support from other components.
- Security features, including access controls and authentication protocol.
- Embedded design assumptions, including the use of specific numerical or nonnumerical algorithms.
- Exception handling

Component Adaptation. In an ideal setting, domain engineering creates a library of components that can be easily integrated into an application architecture. The implication of "easy integration" is that (1) consistent methods of resource management have been implemented for all components in the library, (2) common activities such as data management exist for all components, and (3) interfaces within the architecture and with the external environment have been implemented in a consistent manner.

Conflicts may occur in one or more of the areas in selection of components. To avoid these conflicts, an adaptation technique called component wrapping is sometimes used. When a software team has full access to the internal design and code for a component *white-box* wrapping is applied. Like its counterpart in software testing white-box wrapping examines the internal processing details of the component and makes code-level modifications to remove any conflict. *Gray-box* wrapping is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. *Black-box* wrapping requires the introduction of pre- and postprocessing at the component interface to remove or mask conflicts.

Component Composition. The component composition task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure and coordination must be established to bind the components into an operational system.

OMG/CORBA. The Object Management Group has published a common object request broker architecture (OMG/CORBA). An object request broker (ORB) provides a variety of services that enable reusable components (objects) to communicate with other components, regardless of their location within a system.

Microsoft COM and .NET. Microsoft has developed a component object model (COM) that provides a specification for using components produced by various vendors within a single application running under the Windows operating system. From the point of view of the application, "the focus is not on how implemented, only on the fact that the object has an interface that it registers with the system, and that it uses the component system to communicate with other COM objects". The Microsoft .NET framework encompasses COM and provides a reusable class library that covers a wide array of application domains.

Sun JavaBeans Components. The JavaBeans component system is a portable, platform-independent CBSE infrastructure developed using the Java programming language. The JavaBeans component system encompasses a set of tools, called the Bean Development Kit (BDK), that allows developers to (1) analyze how existing Beans (components) work, (2) customize their behavior and appearance, (3) establish mechanisms for coordination and communication, (4) develop custom Beans for use in a specific application, and (5) test and evaluate Bean behavior.

3.14.3 Analysis and Design for Reuse: Design concepts such as abstraction, hiding, functional independence, refinement, and structured programming, along with object-oriented methods, testing, software quality assurance (SQA), and correctness verification methods all contribute to the creation of software components that are reusable.

The requirements model is analyzed to determine those elements that point to existing reusable components. Elements of the requirements model are compared to WebRef descriptions of reusable components in a process that is sometimes referred to as "specification matching". If specification matching points to an existing component that fits the needs of the current

application, you can extract the component from a reuse library (repository) and use it in the design of a new system. If components cannot be found (i.e., there is no match), a new component is created i.e design for reuse (DFR) should be considered.

Standard data. The application domain should be investigated and standard global data structures (e.g., file structures or a complete database) should be identified. All design components can then be characterized to make use of these standard data structures.

Standard interface protocols. Three levels of interface protocol should be established: the nature of intramodular interfaces, the design of external technical (nonhuman) interfaces, and the human-computer interface.

Program templates. An architectural style is chosen and can serve as a template for the architectural design of a new software. Once standard data, interfaces, and program templates have been established, you have a framework in which to create the design. New components that conform to this framework have a higher probability for subsequent reuse.

3.14.4 Classifying and Retrieving Components: Consider a large component repository. Tens of thousands of reusable software components reside in it.

A reusable software component can be described in many ways, but an ideal description encompasses the 3C model—*concept, content, and context*. The *concept* of a software component is *"a description of what the component does".* The interface to the component is fully described and the semantics—represented within the context of pre- and post conditions—is identified. The *content* of a component describes how the concept is realized. The *context* places a reusable software component within its domain of applicability.

A reuse environment exhibits the following characteristics:

- A component database capable of storing software components and the classification information necessary to retrieve them.
- A library management system that provides access to the database.
- A software component retrieval system (e.g., an object request broker) that enables a client application to retrieve components and services from the library server.
- CBSE tools that support the integration of reused components into a new design or implementation.

Each of these functions interact with or is embodied within the confines of a reuse library.

The *reuse library* is one element of a larger software repository and provides facilities for the storage of software components and a wide variety of reusable work products (e.g., specifications, designs, patterns, frameworks, code fragments, test cases, user guides).

If an initial query results in a voluminous list of candidate components, the query is refined to narrow the list. Concept and content information are then extracted (after candidate components are found) to assist you in selecting the proper component.

UNIT – 5

Unit IV:

User Interface Design: The Golden Rules, User Interface Analysis and Design, Interface Analysis, Interface Design Steps, WebApp Interface Design, Design Evaluation.

WebApp Design: WebApp Design Quality, Design Goal, A Desigin Pyramid for WebApps, WebApp Interface Design, Aestheic Design, Content Design, Architecture Design, Navigation Design, Component-Level Design, Object-Oriented Hypermedia Design Method(OOHMD).

4.1 USER INTERFACE DESIGN

What is it? User interface design creates 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.

Who does it? A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

Why is it important? If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user's perception of the software.

What are the steps? User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

What is the work product? User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

How do I ensure that I've done it right? An interface prototype is "test driven" by the users, and feedback from the test drive is used for the next iterative modification of the prototype.

THE GOLDEN RULES

The three golden rules on interface design are
1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

4.1.1 Place the User in Control: During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface. User wanted to control the computer, not have the computer control her. Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. The result may be an interface that is easy to build, but frustrating to use. Mandel defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is "inside" the machine (e.g., a user should never be required to type operating system commands from within application software).

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object is an implementation of direct manipulation.

4.1.2 Reduce the User's Memory Load: The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory.

Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results.

Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember.

The visual layout of the interface should be based on a real-world metaphor. For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

4.1.3 Make the Interface Consistent: The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to design rules that are maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented.

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard the user expects this in every application he encounters. A change will cause confusion.

4.2 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. Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

4.2.1 Interface Analysis and Design Models: Four different models come into play when a user interface is to be analyzed and designed. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, gender, physical abilities, education, cultural or ethnic background, motivation, goals and personality . In addition, users can be categorized as:

Novices. No syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general.

Knowledgeable, intermittent users. Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

Knowledgeable, ***frequent users***. Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The user's ***mental model*** (system perception) is the image of the system that end users carry in their heads. For example, if the user of a particular word processor were asked to describe its operation, the system perception would guide the response.

The implementation model combines the outward manifestation of the computer based system (look and feel interface), coupled with all supporting information (books, manuals, videotapes, help) that describes interface syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: "*Know the user, know the tasks*."

4.2.2 The Process: The analysis and design process for user interfaces is iterative and can be represented using a spiral model. Referring to Figure 11.1, the four distinct framework activities: (1) interface analysis and modeling, (2) interface design, (3) interface construction, and (4) interface validation. The spiral shown in Figure 11.1 implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design.

In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed. Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work to understand the system perception for each class of users.

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated.

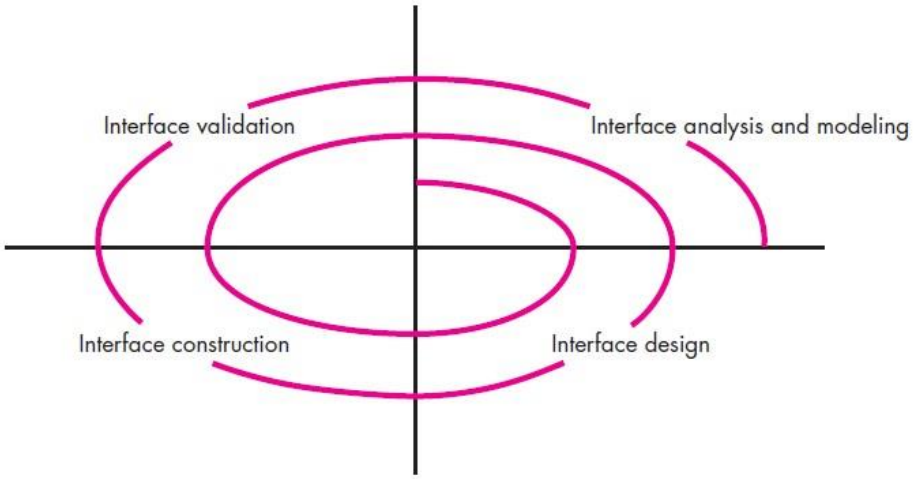


Fig 11.1: The user interface design process

Finally, 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 considerations driven by environmental factors?

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.

Interface construction normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

Interface 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 interface is easy to use and easy to learn, and (3) the users' acceptance of the interface as a useful tool in their work.

Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

4.3 INTERFACE ANALYSIS

A key tenet of all software engineering process models is: understand the problem before you attempt to design a solution. In the case of user interface design, understanding the problem means understanding (1) the people (end users) who will interact with the system through the interface, (2) the tasks that end users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. We examin these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

4.3.1 User Analysis: The only way that you can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:

User Interviews. The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. This can be accomplished in one-on-one meetings or through focus groups.

Sales input. Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

Marketing input. Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

Support input. Support staff talks with users on a daily basis. They are the most likely source of information on what works and what doesn't, what users like and what they dislike, what features generate questions and what features are easy to use.

The following set of questions will help you to better understand the users of a system:

- Are users trained professionals, technicians, 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 that sits behind the interface?

Once these questions are answered, you'll know who the end users are, what is likely to motivate and please them, how they can be grouped into different user classes or profiles, what their mental models of the system are, and how the user interface must be characterized to meet their needs.

4.3.2 Task Analysis and Modelling: The goal of task analysis is to answer 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?

To answer these questions, you must use techniques that are are applied to the user interface.

Use cases. When used as part of task analysis, the use case is developed to show how an end user performs some specific work-related task. The use case provides a basic description of one important work task for the computer-aided design system. From it, you can extract tasks, objects, and the overall flow of the interaction.

Task elaboration. The stepwise elaboration is (also called functional decomposition or stepwise refinement) a mechanism for refining the processing tasks that are required for software to accomplish some desired function.

Task analysis can be applied in two ways. An interactive, computer-based system is often used to replace a manual or semi manual activity. To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface.

Alternatively, you can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception. Regardless of the overall approach to task analysis, you must first define and classify tasks. Each of the major tasks can be elaborated into subtasks.

Object elaboration. Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations. The user interface analysis model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details of each operation are defined.

Workflow analysis. When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply workflow analysis. This technique allows you to understand how a work process is

completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram). See Figure 11.2
Regardless, the flow of events (shown in the figure) enables you to recognize a number of key interface characteristics:

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different than the one defined for pharmacists or physicians.
2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
3. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., Fills prescription could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

Hierarchical representation. A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the following user task and subtask hierarchy.

User task: Requests that a prescription be refilled
• Provide identifying information.
• Specify name.
• Specify userid.
• Specify PIN and password.
• Specify prescription number.
• Specify date refill is required.
To complete the task, three subtasks are defined. One of these subtasks, provide identifying information, is further elaborated in three additional sub-subtasks.

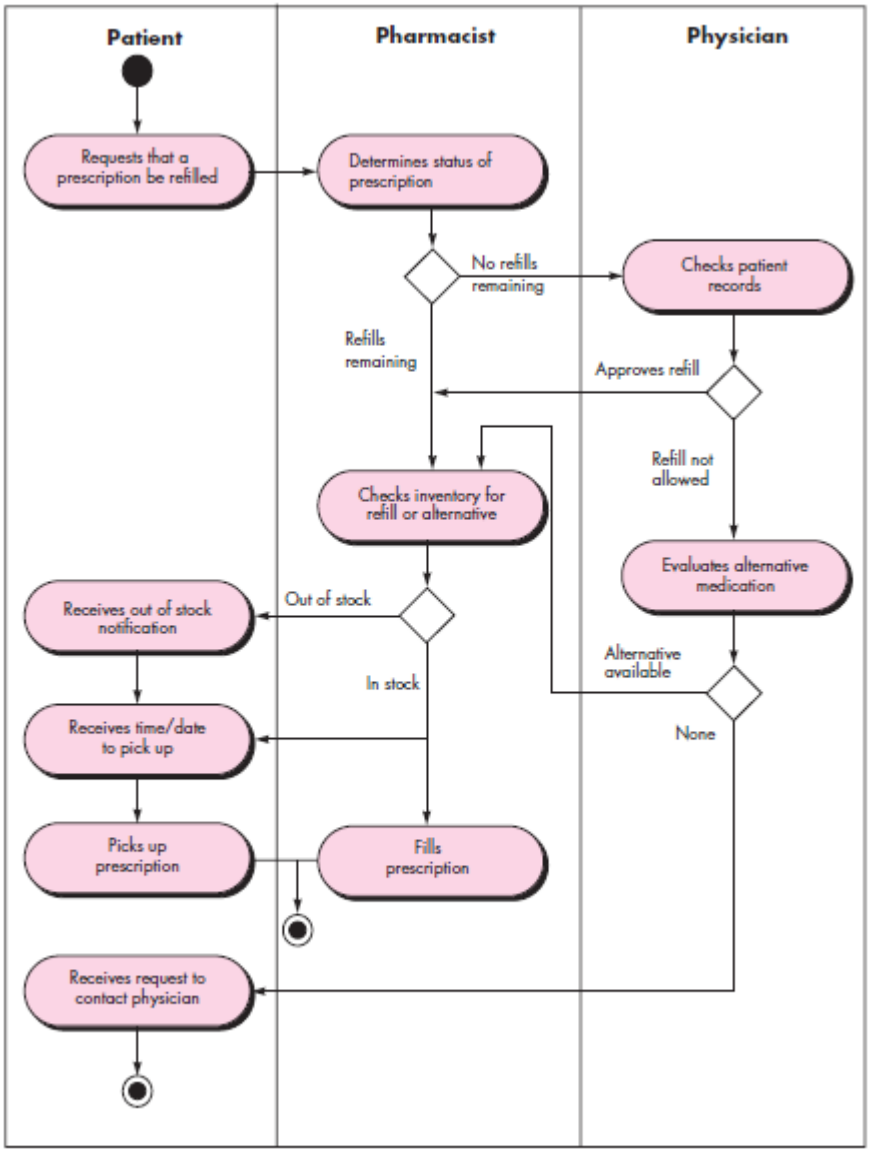


Fig 11.2: Swimlane diagram for prescription refill function

4.3.3 Analysis of Display Content: The user tasks lead to the presentation of a variety of different types of content. For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files).

These data objects may be

(1) generated by components (unrelated to the interface) in other parts of an application, (2) acquired from data stored in a database that is accessible from the application, or (3) transmitted from systems external to the application in question.

During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

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

4.3.4 Analysis of the Work Environment: Hackos and Redish stated the importance of work environment analysis as:
People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use.

4.4 INTERFACE DESIGN STEPS

Once interface analysis has been completed, all tasks (or objects and actions) required by the end user have been identified in detail and the interface design activity commences. Interface design is an iterative process. Although many different user interface design models (e.g., have been proposed, all suggest some combination of the following steps:
1. Using information developed during interface analysis define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

4.4.1 Applying Interface Design Steps: The definition of interface objects and the actions that are applied to them is an important step in interface design. To accomplish this, user scenarios are parsed. That is, a use case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A source object (e.g., a report icon) is dragged and dropped onto a target object (e.g., a printer icon).
When you are satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted.

4.4.2 User Interface Design Patterns: Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As an example of a commonly encountered interface design problem, consider a situation in which a user must enter one or more calendar dates, sometimes months in advance. A vast array of interface design patterns has been proposed over the past decade.

4.4.3 Design Issues: As the design of a user interface evolves, four common design issues almost always surface: *system response time, user help facilities, error information handling, and command labeling*
Response time. System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with desired output or action.
System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress are inevitable. Variability refers to the deviation from average response time.
Help facilities. Almost every user of an interactive, computer-based system requires help now and then. In most cases, however, modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface. A number of design issues must be addressed when a help facility is considered:
• Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.
• How will the user request help? Options include a help menu, a special function key, or a HELP command.
• How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.
• How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.
• How will help information be structured? Options include a “flat” structure inwhich all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext.

Error handling. Error messages and warnings are “bad news” delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration.

In general, every error message or warning produced by an interactive system should have the following characteristics:

• The message should describe the problem in jargon that the user can understand.

• The message should provide constructive advice for recovering from the error.

• The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

• The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or themessage might flash momentarily or be displayed in a color that is easily recognizable as the “error color.”

• The message should be “nonjudgmental.” That is, the wording should never place blame on the user.

Menu and command labeling. The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

• Will every menu option have a corresponding command?

• What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.

• How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?

• Can commands be customized or abbreviated by the user?

• Are menu labels self-explanatory within the context of the interface?

• Are submenus consistent with the function implied by a master menu item?

Application accessibility. As computing applications become *ubiquitous*, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. Accessibility for users (and software engineers) who may be *physically challenged* is an imperative for ethical, legal, and business reasons. A variety of accessibility guidelines - many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others provide specific guidelines for “assistive technology” that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

Internationalization. Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create “globalized” software. That is, user interfacesshould be designed to accommodate a generic core of

functionality that can be deliveredto all who use the software. Localization features enable the interface to becustomized for a specific market.
A variety of internationalization guidelines are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., differentalphabets may create specialized labeling and spacing requirements). The *Unicode standard* has been developed to address the daunting challenge of managingdozens of natural languages with hundreds of characters and symbols.

4.5 WEBAPP INTERFACE DESIGN

WebApp interface need to answer three primary questions for the end user:
Where am I? The interface should (1) provide an indication of the WebApp that has been accessed and (2) inform the user of her location in the content hierarchy.
What can I do now? The interface should always help the user understand his current options—what functions are available, what links are live, what content is relevant?
Where have I been, where am I going? The interface must facilitate navigation. Hence, it must provide a "map" of where the user has been and what paths may be taken to move elsewhere within the WebApp. An effective WebApp interface must provide answers for each of these questions as the end user navigates through content and functionality.

4.5.1 Interface Design Principles and Guidelines: The user interface of a WebApp is its "first impression." Because of the sheer volume of competing WebApps in virtually every subject area, the interface must "grab" a potential user immediately.

Effective interfaces do not concern the user with the inner workings of the system. Effective applications and services perform a maximum of work, while requiring a minimum of information from users. The designer of the WebApp should anticipate that the user might request a download of the driver and should provide navigation facilities that allow this to happen without requiring the user to search for this capability.
Communication. The interface should communicate the status of any activity initiated by the user. Communication can be obvious (e.g., a text message) or subtle (e.g., an image of a sheet of paper moving through a printer to indicate that printing is under way). The interface should also communicate user status (e.g., the user's identification) and her location within the WebApp content hierarchy.
Consistency. The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) should be consistent throughout the WebApp
Controlled autonomy. The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application. For example, navigation to secure portions of the WebApp should be controlled by userID and password.

Efficiency. The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the developer who designs and builds it or the client server environment that executes it.

Flexibility. The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the WebApp in a somewhat random fashion.

Focus. The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.

Fitt's law. "The time to acquire a target is a function of the distance to and size of the target"

If a sequence of selections or standardized inputs (with many different options within the sequence) is defined by a user task, the first selection (e.g., mouse pick) should be physically close to the next selection.

Human interface objects. A vast library of reusable human interface objects has been developed for WebApps. Use them. Any interface object that can be "seen, heard, touched or otherwise perceived" by an end user can be acquired from any one of a number of object libraries.

Latency reduction. Rather than making the user wait for some internal operation to complete (e.g., downloading a complex graphical image), the WebApp should use multitasking in a way that lets the user proceed with work as if the operation has been completed. In addition to reducing latency, delays must be acknowledged so that the user understands what is happening. This includes (1) providing audio feedback when a selection does not result in an immediate action by the WebApp, (2) displaying an animated clock or progress bar to indicate that processing is under way, and (3) providingsome entertainment (e.g., an animation or text presentation) while lengthy processing occurs.

Learnability. A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited. In general the interface should emphasize a simple, intuitive design that organizes content and functionality into categories that are obvious to the user.

Metaphors. An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user. A metaphor should call on images and concepts from the user's experience, but it does not need to be an exact reproduction of a real-world experience.

Maintain work product integrity. A work product must be automatically saved so that it will not be lost if an error occurs To avoid data loss, a WebApp should be designed to *autosave* all user-specified data. The interface should support this function and provide the user with an easy mechanism for *recovering* "lost" information.

Readability. All information presented through the interface should be readable by young and old. The interface designer should emphasize readable type styles, font sizes, and color background choices that enhance contrast.

Track state. When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off. In general, cookies can be

designed to store state information. However, cookies are a controversial technology, and other design solutions may be more palatable for some users.

Visible navigation. A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought to them"

• Reading speed on a computer monitor is approximately 25 percent slower than reading speed for hardcopy. Therefore, do not force the user to read voluminous amounts of text, particularly when the text explains the operation of the WebApp or assists in navigation.

• Avoid "under construction" signs—an unnecessary link is sure to disappoint.

• Users prefer not to scroll. Important information should be placed within the dimensions of a typical browser window.

• Navigation menus and head bars should be designed consistently and should be available on all pages that are available to the user. The design should not rely on browser functions to assist in navigation.

• Aesthetics should never supersede functionality. For example, a simple button might be a better navigation option than an aesthetically pleasing, but vague image or icon whose intent is unclear.

• Navigation options should be obvious, even to the casual user. The user should not have to search the screen to determine how to link to other content or services.

A well-designed interface improves the user's perception of the content or services provided by the site. It need not necessarily be flashy, but it should always be well structured and ergonomically sound.

4.5.2 Interface Design Workflow for WebApps: Information contained within the requirements model forms the basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are then used to prototype and ultimately implement the interface design model. The following tasks represent a rudimentary workflow for WebApp interface design:

1. Review information contained in the requirements model and refine as required.

2. Develop a rough sketch of the WebApp interface layout. An interface prototype (including the layout) may have been developed as part of the requirements modeling activity. If the layout already exists, it should be reviewed and refined as required. If the interface layout has not been developed, you should work with stakeholders to develop it at this time.

3. Map user objectives into specific interface actions. For the vast majority of WebApps, the user will have a relatively small set of primary objectives. These should be mapped into specific interface actions as shown in Figure 11.4. In essence, you must answer the following question: "How does the interface enable the user to accomplish each objective?"

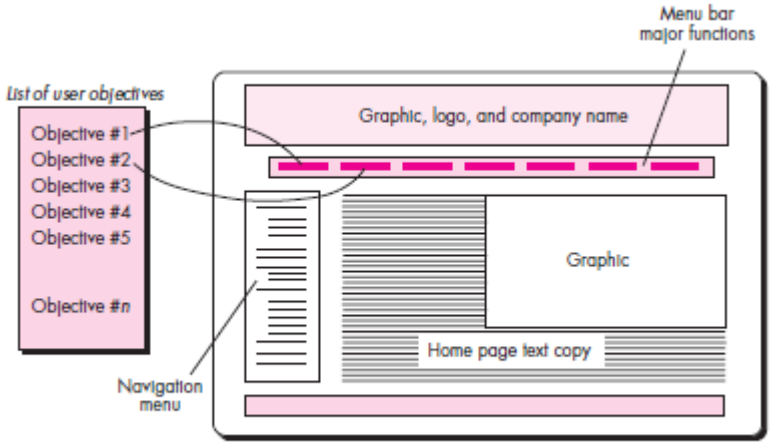


Fig 11.4: Mapping user objectives into interface actions

4. Define a set of user tasks that are associated with each action. Each interface action (e.g., "buy a product") is associated with a set of user tasks. These tasks have been identified during requirements modeling. During design, they must be mapped into specific interactions that encompass navigation issues, content objects, and WebApp functions.

5. Storyboard screen images for each interface action. As each action is considered, a sequence of storyboard images (screen images) should be created to depict how the interface responds to user interaction. Content objects should be identified (even if they have not yet been designed and developed), WebApp functionality should be shown, and navigation links should be indicated.

6. Refine interface layout and storyboards using input from aesthetic design. In most cases, you'll be responsible for rough layout and storyboarding, but the aesthetic look and feel for a major commercial site is often developed by artistic, rather than technical, professionals. Aesthetic design is integrated with the work performed by the interface designer.

7. Identify user interface objects that are required to implement the interface. This task may require a search through an existing object library to find those reusable objects (classes) that are appropriate for the WebApp interface. In addition, any custom classes are specified at this time.

8. Develop a procedural representation of the user's interaction with the interface. This optional task uses UML sequence diagrams and/or activity diagrams to depict the flow of activities (and decisions) that occur as the user interacts with the WebApp.

9. Develop a behavioral representation of the interface. This optional task makes use of UML state diagrams to represent state transitions and the events that cause them. Control mechanisms (i.e., the objects and actions available to the user to alter a WebApp state) are defined.

10. Describe the interface layout for each state. Using design information developed in Tasks 2 and 5, associate a specific layout or screen image with each WebApp state described in Task

11. Refine and review the interface design model. Review of the interface should focus on usability. It is important to note that the final task set you choose should be adapted to the special requirements of the application that is to be built.

4.6 DESIGN EVALUATION

The user interface evaluation cycle takes the form shown in Figure 11.5. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides you with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), you can extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary.

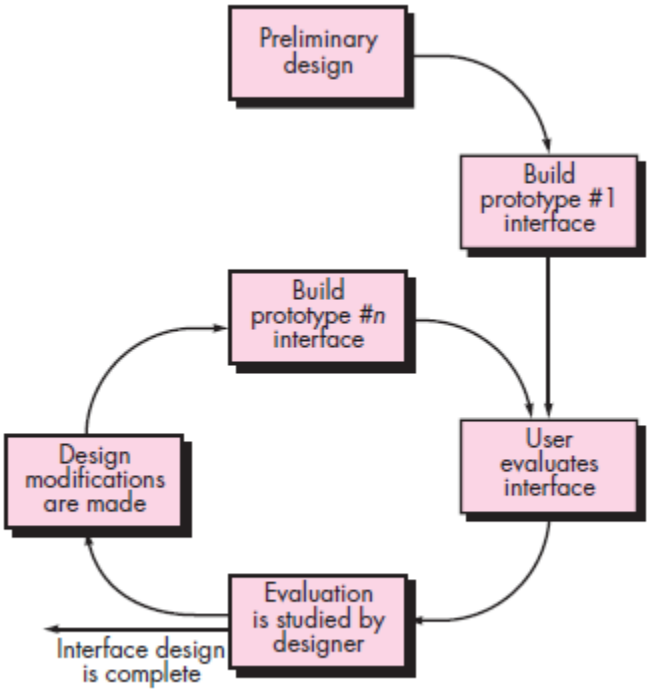


Fig 11.5: The interface design evaluation cycle

The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built? If you identify and correct potential problems early, the number of loops through the evaluation cycle will be reduced and development time will shorten. If a design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews:

1. The length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. 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.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, you can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. 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.

4.7 WEBAPP DESIGN

What is it? Design for WebApps encompasses technical and nontechnical activities that include: establishing the look and feel of the WebApp, creating the aesthetic layout of the user interface, defining the overall architectural structure, developing the content and functionality that reside within the architecture, and planning the navigation that occurs within the WebApp.
Who does it? Web engineers, graphic designers, content developers, and other stakeholders all participate in the creation of a WebApp design model.
Why is it important? Design allows you to create a model that can be assessed for quality and improved before content and code are generated, tests are conducted, and end users become involved in large numbers. Design is the place where WebApp quality is established.
What are the steps? WebApp design encompasses six major steps that are driven by information obtained during requirements modeling. Content design uses the content model (developed during analysis) as the basis for establishing the design of content objects. Aesthetic design (also called graphic design) establishes the look and feel that the end user sees. Architectural design focuses on the overall hypermedia structure of all content objects and functions. Interface design establishes the layout and interaction mechanisms that define the user interface. Navigation design defines how the end user navigates through the hypermedia structure, and component design represents the detailed internal structure of functional elements of the WebApp.
What is the work product? A design model that encompasses content, aesthetics, architecture, interface, navigation, and component-level design issues is the primary work product that is produced during WebApp design
How do I ensure that I've done it right? Each element of the design model is reviewed in an effort to uncover errors, inconsistencies, or omissions. In addition, alternative solutions are considered, and the degree to which the current design model will lead to an effective implementation is also assessed.

4.7.1 Web App Design Quality: Design is the engineering activity that leads to a high-quality product. In fact, the user's perception of "goodness" might be more important than any technical discussion of WebApp quality . Olsina and his colleagues have prepared a "quality requirement tree" that identifies a set of technical attributes—usability, functionality, reliability, efficiency, and maintainability—that lead to high-quality WebApps. Figure 13.1 summarizes their work.

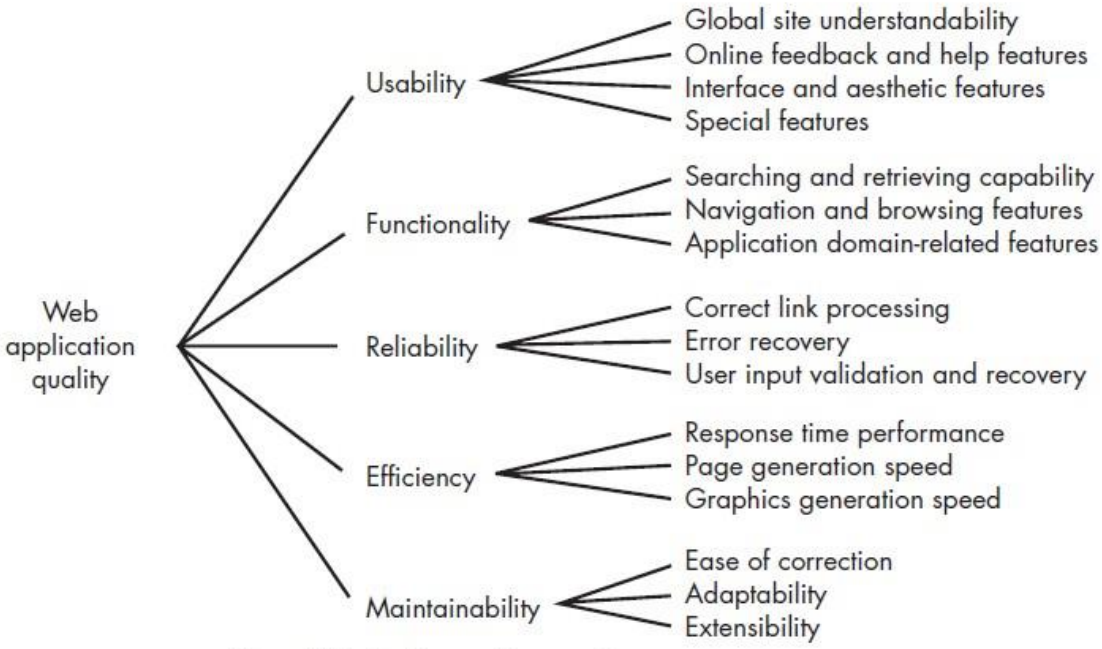


Figure 13.1: Quality requirements Tree

Security. WebApps have become heavily integrated with critical corporate and government databases. E-commerce applications extract and then store sensitive customer information. For these and many other reasons, WebApp security is paramount in many situations. The key measure of security is the ability of the WebApp and its server environment to rebuff unauthorized access and/or thwart an outright malevolent attack.

Availability. Even the best WebApp will not meet users' needs if it is unavailable. In a technical sense, availability is the measure of the percentage of time that a WebApp is available for use. The typical end user expects WebApps to be available 24/7/365. Anything less is deemed unacceptable.3 But "up-time" is not the only indicator of availability. Offutt suggests that "using features available on only one browser or one platform" makes the WebApp unavailable to those with a different browser/platform configuration. The user will invariably go elsewhere.

Scalability. Can the WebApp and its server environment be scaled to handle 100, 1000, 10,000, or 100,000 users? Will the WebApp and the systems with which it is interfaced handle significant variation in volume or will responsiveness drop dramatically? It is not enough to build a WebApp that is successful. It is equally important to build a WebApp that can accommodate the burden of success and become even more successful.

Time-to-market. Although time-to-market is not a true quality attribute in the technical sense, it is a measure of quality from a business point of view. The first WebApp to address a specific market segment often captures a disproportionate number of end users.

4.7.2 Design Goals: *Jean Kaiser* suggests a set of design goals that are applicable to virtually every WebApp regardless of application domain, size, or complexity:

Simplicity. Better the web design should be moderation and simple. Content should be informative but succinct and should use a delivery mode (e.g., text, graphics, video, audio) that is appropriate to the information that is being delivered. Aesthetics should be pleasing, but not overwhelming (e.g., too many colors tend to distract the user rather than enhancing the interaction). Functions should be easy to use and easier to understand.
Consistency. Content should be constructed consistently. . Graphic design (aesthetics) should present a consistent look across all parts of the WebApp. Architectural design should establish templates that lead to a consistent hypermedia structure. Interface design should define consistent modes of interaction, navigation, and content display. Navigation mechanisms should be used consistently across all WebApp elements
Identity. The aesthetic, interface, and navigational design of a WebApp must be consistent with the application domain for which it is to be built. You should work to establish an identity for the WebApp through the design.
Robustness Based on the identity that has been established, a WebApp often makes an implicit "promise" to a user. The user expects robust content and functions that are relevant to the user's needs. If these elements are missing or insufficient, it is likely that the WebApp will fail.
Navigability The navigation should be simple and consistent. It should also be designed in a manner that is intuitive and predictable. That is, the user should understand how to move about the WebApp without having to search for navigation links or instructions.
Visual Appeal. Beauty (visual appeal) is undoubtedly in the eye of the beholder, but many design characteristics do contribute to visual appeal.
Compatibility. A WebApp will be used in a variety of environments and must be designed to be compatible with each.

4.7.3 A Design Pyramid For Webapps
What is WebApp design? The creation of an effective design will typically require a diverse set of skills. Sometimes, for small projects, a single developer may need to be multi-skilled. For larger projects, it may be advisable and/or feasible to draw on the expertise of specialists: Web engineers, graphic designers, content developers, programmers, database specialists, information architects, network engineers, security experts, and testers. Drawing on these diverse skills allows the creation of a model that can be assessed for quality and improved before content and code are generated, tests are conducted, and end-users become involved in large numbers. If analysis is where WebApp quality is established, then design is where the

quality is truly embedded. The appropriate mix of design skills will vary depending upon the nature of the WebApp. Figure 13.2 depicts a design pyramid for WebApps. Each level of the pyramid represents a design action.

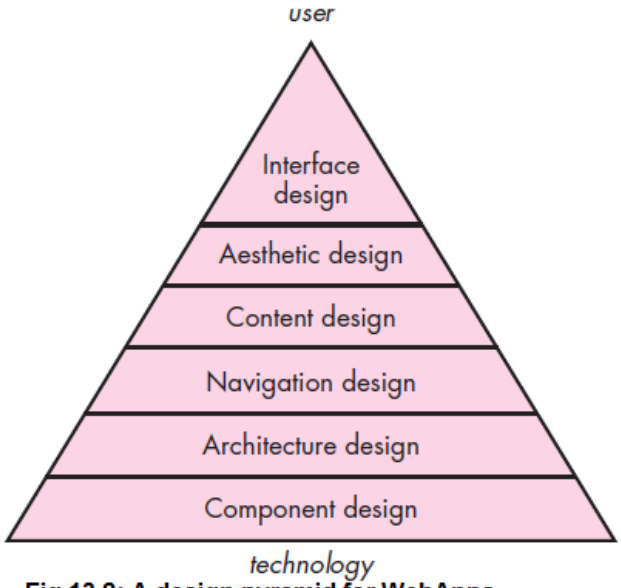


Fig 13.2: A design pyramid for WebApps

4.7.4 Webapp Interface Design: One of the challenges of interface design for WebApps is the indeterminate nature of the user's entry point. The objectives of a WebApp interface are to: (1) establish a consistent window into the content and functionality provided by the interface, (2) guide the user through a series of interactions with the WebApp, and (3) organize the navigation options and content available to the user. To achieve a consistent interface, you should first use aesthetic design to establish a coherent "look." This encompasses many characteristics, but must emphasize the layout and form of navigation mechanisms. To guide user interaction, you may draw on an appropriate metaphor5 that enables the user to gain an intuitive understanding of the interface. To implement navigation options, you can select from one of a number of interaction mechanisms:

• **Navigation menus**—keyword menus (organized vertically or horizontally) that list key content and/or functionality. These menus may be implemented so that the user can choose from a hierarchy of subtopics that is displayed when the primary menu option is selected.

• *Graphic icons*—button, switches, and similar graphical images that enable the user to select some property or specify a decision.

• *Graphic images*—some graphical representation that is selectable by the user and implements a link to a content object or WebApp functionality.

4.7.5 AESTHETIC DESIGN

Aesthetic design, also called *graphic design*, is an artistic endeavor that complements the technical aspects of WebApp design. Without it, a WebApp may be functional, but unappealing.

4.7.5.1 Layout Issues: Like all aesthetic issues, there are no absolute rules when screen layout is designed. However, a number of general layout guidelines are worth considering:
Don't be afraid of white space. It is inadvisable to pack every square inch of a Web page with information. The resulting clutter makes it difficult for the user to identify needed information or features and create visual chaos that is not pleasing to the eye.
Emphasize content. After all, that's the reason the user is there. Nielsen suggests that the typical Web page should be 80 percent content with the remaining real estate dedicated to navigation and other features.
Organize layout elements from top-left to bottom-right. The vast majority of users will scan a Web page in much the same way as they scan the page of a book—top-left to bottom-right. If layout elements have specific priorities, high-priority elements should be placed in the upper-left portion of the page real estate.
Group navigation, content, and function geographically within the page. Humans look for patterns in virtually all things. If there are no discernable patterns within a Web page, user frustration is likely to increase (due to unnecessary searching for needed information).
Don't extend your real estate with the scrolling bar. Although scrolling is often necessary, most studies indicate that users would prefer not to scroll. It is better to reduce page content or to present necessary content on multiple pages.
Consider resolution and browser window size when designing layout. Rather than defining fixed sizes within a layout, the design should specify all layout items as a percentage of available space.

4.7.5.2 Graphic Design Issues: Graphic design considers every aspect of the look and feel of a WebApp. The graphic design process begins with layout and proceeds into a consideration of global color schemes; text types, sizes, and styles; the use of supplementary media (e.g., audio, video, animation); and all other aesthetic elements of an application.

4.8 CONTENT DESIGN

Content design focuses on two different design tasks. First, a design representation for *content objects* and the mechanisms required to establish their relationship to one another is developed. Second, the *information* within a specific content object is created.

4,8.1 Content Objects:. A content object has attributes that include content-specific information (normally defined during WebApp requirements modeling) and implementation-specific attributes that are specified as part of design.

4.8.2 Content Design Issues: Once all content objects are modeled, the information that each object is to deliver must be authored and then formatted to best meet the customer's needs. Content authoring is the job of specialists in the relevant area who design the content object by providing an outline of information to be delivered and an indication of the types of generic content objects (e.g., descriptive text, graphic images, photographs) that will be used to deliver the information. Aesthetic design may also be applied to represent the proper look and feel for the content.

As content objects are designed, they are "chunked" to form WebApp pages. The number of content objects incorporated into a single page is a function of user needs, constraints imposed by download speed of the Internet connection, and restrictions imposed by the amount of scrolling that the user will tolerate.

4.9 ARCHITECTURE DESIGN

Content architecture focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation. WebApp architecture addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.

In most cases, architecture design is conducted in parallel with interface design, aesthetic design, and content design. Because the WebApp architecture may have a strong influence on navigation, the decisions made during this design action will influence work conducted during navigation design.

4.9.1 Content Architecture: The design of content architecture focuses on the definition of the overall hypermedia structure of the WebApp. Although custom architectures are sometimes created, you always have the option of choosing from four different content structures

Linear structures (Figure 13.4) are encountered when a predictable sequence of interactions (with some variation or diversion) is common.
Grid structures (Figure 13.5) are an architectural option that you can apply when WebApp content can be organized categorically in two (or more) dimensions.
Hierarchical structures (Figure 13.6) are undoubtedly the most common WebApp architecture.
A networked or "pure web" structure (Figure 13.7) is similar in many ways to the architecture that evolves for object-oriented systems.
The architectural structures can be combined to form *composite structures*.

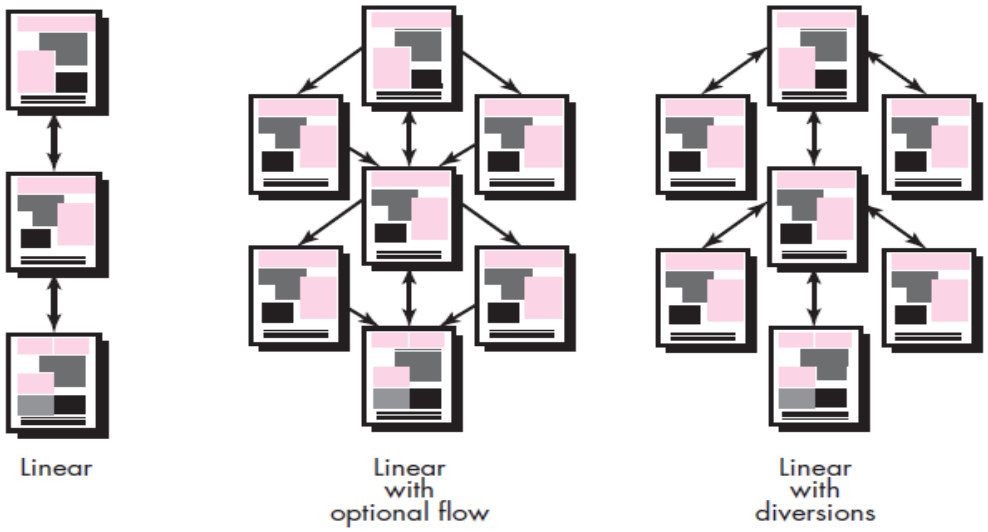


Fig 13.4: Linear Structures

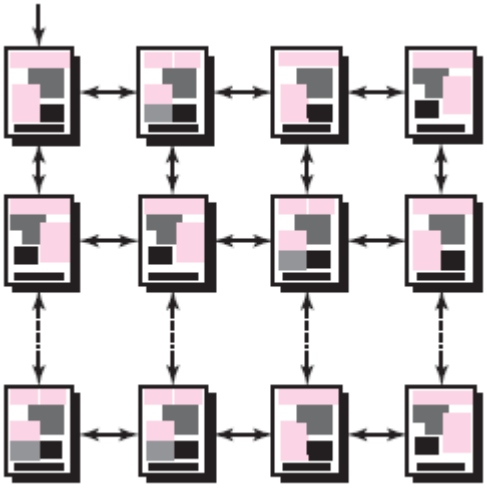


Fig 13.5: Grid Structure

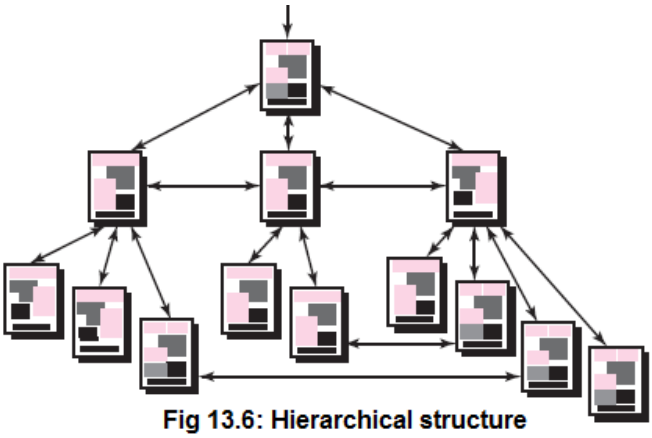


Fig 13.6: Hierarchical structure

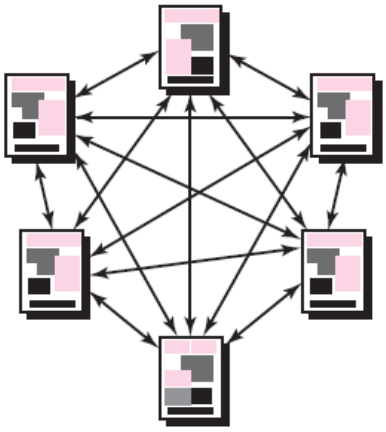


Fig 13.7: Network structure

4.9.2 WebApp Architecture:. Jacyntho and his colleagues describe the basic characteristics of this infrastructure in the following manner:

Applications should be built using layers in which different concerns are taken into account; in particular, application data should be separated from the page's contents (navigation nodes) and these contents, in turn, should be clearly separated from the interface look-and-feel (pages).

The three-layer design architecture that decouples interface from navigation and from application behavior. They argue that keeping interface, application, and navigation separate simplifies implementation and enhances reuse. The Model-View-Controller (MVC) architecture is one of a number of suggested WebApp infrastructure models that decouple the user interface

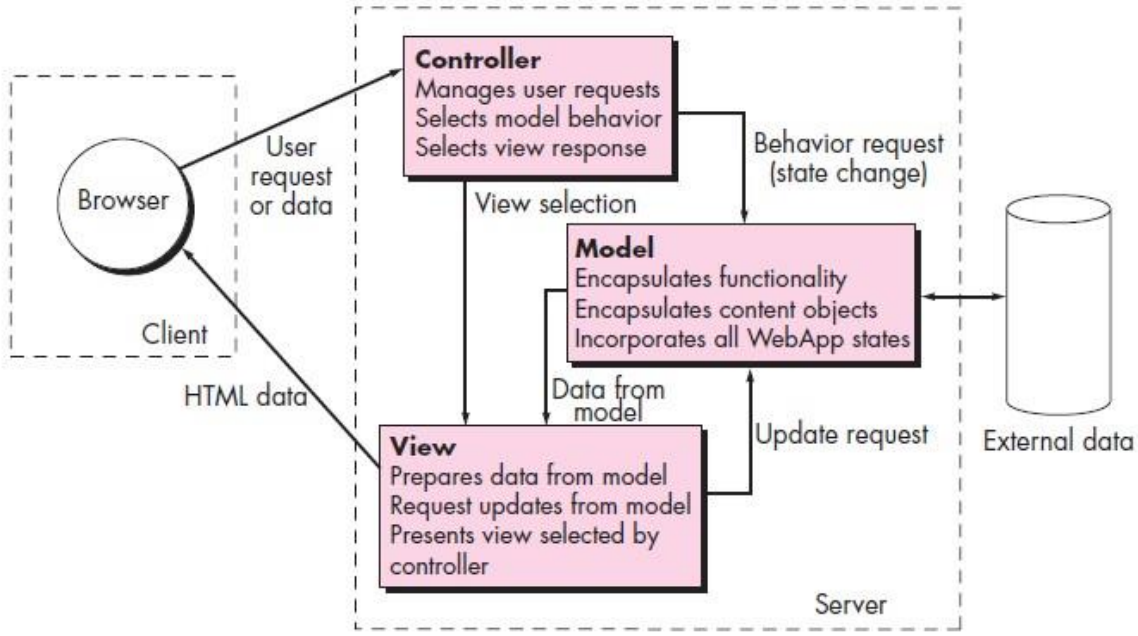


Fig 13.8: The MVC architecture

from the WebApp functionality and informational content. The *model* (sometimes referred to as the "model object") contains all application-specific content and processing logic, including all content objects, access to external data/information sources, and all processing functionality that is application specific. The *view* contains all interface specific functions and enables the presentation of content and processing logic, including all content objects, access to external data/information sources, and all processing functionality required by the end user. The *controller* manages access to the model and the view and coordinates the flow of data between them. In a WebApp, "the view is updated by the controller with data from the model based on user input". A schematic representation of the MVC architecture is shown in Figure 13.8.

4.10 NAVIGATION DESIGN

Once the WebApp architecture has been established and the components (pages, scripts, applets, and other processing functions) of the architecture have been identified, you must define navigation pathways that enable users to access WebApp content and functions. To accomplish this, you should (1) identify the semantics of navigation for different users of the site, and (2) define the mechanics (syntax) of achieving the navigation.

4.10.1 Navigation Semantics: Like many WebApp design actions, navigation design begins with a consideration of the user hierarchy and related use cases developed for each category of user (actor). Each actor may use the WebApp somewhat differently and therefore have different navigation requirements. A series of navigation semantic units (NSUs)—"a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements. The overall navigation structure for a WebApp may be organized as a hierarchy of NSUs.

4.10.2 Navigation Syntax: As design proceeds, your next task is to define the mechanics of navigation. A number of options are available as you develop an approach for implementing each NSU:

• *Individual navigation link*—includes text-based links, icons, buttons and switches, and graphical metaphors. You must choose navigation links that are appropriate for the content and consistent with the heuristics that lead to high-quality interface design.

• *Horizontal navigation bar*—lists major content or functional categories in a bar containing appropriate links. In general, between four and seven categories are listed.

• *Vertical navigation column*—(1) lists major content or functional categories, or (2) lists virtually all major content objects within the WebApp. If you choose the second option, such navigation columns can "expand" to present content objects as part of a hierarchy (i.e., selecting an entry in the original column causes an expansion that lists a second layer of related content objects).

• *Tabs*—a metaphor that is nothing more than a variation of the navigation bar or column, representing content or functional categories as tab sheets that are selected when a link is required.

• *Site maps*—provide an all-inclusive table of contents for navigation to all content objects and functionality contained within the WebApp. In addition to choosing the mechanics of navigation, you should also establish appropriate navigation conventions and aids.

4.11 COMPONENT-LEVEL DESIGN

Modern WebApps deliver increasingly sophisticated processing functions that (1) perform localized processing to generate content and navigation capability in a dynamic fashion, (2) provide computation or data processing capability that are appropriate for the WebApp's business domain, (3) provide sophisticated database query and access, and (4) establish data

interfaces with external corporate systems. To achieve these (and many other) capabilities, you must design and construct program components that are identical in form to software components for traditional software.

4.12 OBJECT-ORIENTED HYPERMEDIA DESIGN METHOD (OOHDM)

One of the most widely discussed WebApp design methods— OOHDM. *Daniel Schwabe* and his colleagues originally proposed the Object-Oriented Hypermedia Design Method (OOHDM), which is composed of four different design activities: *conceptual design, navigational design, abstract interface design, and implementation*. A summary of these design activities is shown in Figure 13.10

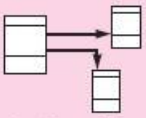
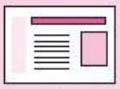
	 Conceptual design	 Navigational design	 Abstract interface design	 Implementation
Work products	Classes, subsystems, relationships, attributes	Nodes links, access structures, navigational contexts, navigational transformations	Abstract interface objects, responses to external events, transformations	Executable WebApp
Design mechanisms	Classification, composition, aggregation, generalization specialization	Mapping between conceptual and navigation objects	Mapping between navigation and perceptible objects	Resource provided by target environment
Design concerns	Modeling semantics of the application domain	Takes into account user profile and task. Emphasis on cognitive aspects.	Modeling perceptible objects, implementing chosen metaphors. Describe interface for navigational objects.	Correctness; application performance; completeness

Fig 13.10: Summary of the OOHDM method.

4.12.1 Conceptual Design for OOHDM: OOHDM conceptual design creates a representation of the subsystems, classes, and relationships that define the application domain for the WebApp. UML may be used to create appropriate class diagrams, aggregations, and composite class representations, collaboration diagrams, and other information that describes the application domain. The class diagrams, aggregations, and related information developed as part of WebApp analysis are reused during conceptual design to represent relationships between classes.

4.12.2 Navigational Design for OOHDM: Navigational design identifies a set of "objects" that are derived from the classes defined in conceptual design. A series of "navigational classes" or "nodes" are defined to encapsulate these objects. UML may be used to create appropriate use

cases, state charts, and sequence diagrams—all representations that assist you in better understanding navigational requirements. In addition, design patterns for navigational design may be used as the design is developed. OOHDM uses a predefined set of navigation classes—nodes, links, anchors, and access structures.

Access structures are more elaborate and include mechanisms such as a WebApp index, a site map, or a guided tour.

4.12.3 Abstract Interface Design and Implementation: The abstract interface design action specifies the interface objects that the user sees as WebApp interaction occurs. A formal model of interface objects, called an abstract data view (ADV), is used to represent the relationship between interface objects and navigation objects, and the behavioral characteristics of interface objects.

The ADV model defines a “static layout” that represents the interface metaphor and includes a representation of navigation objects within the interface and the specification of the interface objects (e.g., menus, buttons, icons) that assist in navigation and interaction. In addition, the ADV model contains a behavioral component that indicates how external events“trigger navigation and which interface transformations occur when the user interacts with the application”.

The OOHDM implementation activity represents a design iteration that is specific to the environment in which the WebApp will operate.