Ayisha Farhath Abbas
Assistant Professor
Department Of Computer Science

UNIT I

SOFTWARE AND ENGINEERING

• A computer software is THE PRODUCTS that software professionals build. It encompasses programs that execute within a computer.

• Software engineers build and support software.

• We develop software and at the same time we're developing it on another software. "Software is both a product and a vehicle that delivers a product".

• Software delivers the most important product of our time: "information".

• Software is logical rather than a physical.

• Software is developed or engineered, while hardware is manufactured.

• Software doesn't wear out, but hardware does wear out.

THE NATURE OF SOFTWARE

7 categories of computer software

1. System software: a collection of programs written to service other programs Operating systems or utility software: (e.g. Windows, Ubuntu, compilers, editors…), characterized by multi-threading

2. Application software: stand alone programs, I see them as any App you can download from the Google Store.

3. Engineering/scientific software: characterized by number crunching algorithms. They are real time apps. (e.g. Matlab, CATIA…)

4. Embedded software: software that's embedded a certain product (cars, telephones, robots, toys...)

5. Product-line software: provides a specific capability for use by many different customers.

6. Web applications: user-server communication software

7. Artificial intelligence software: makes use of non-numerical algorithms to solve complex problems that are not amenable to computation or straight forward analysis.

9. Web Apps are the most common software today, and they are characterized by: • Network intensiveness, Concurrency, more people are using the app at the same time, and the number of user vary from day to day so there's unpredictable load. Performance must be good, and availability has to be almost 24/7. Web Apps are data driven, we use web app to access databases on the other side. Web Apps are content sensitive, aesthetics is important. They also evolve continuously, and web app need to hit the market as quickly as possible. Web Apps need to provide security,

THE UNIQUE NATURE OF WEBAPPS

In the early days of the World Wide Web, websites were just a set of linked hypertext files which presented information using text and limited graphics. The augmentation of HTML by development tools like Java, XML enabled web engineers to provide computing capability along with informational content.

Web-based systems and applications (WebApps) are sophisticated tools that not only present stand-alone information but also integrate databases and business applications. Web-based systems and applications "involve a mixture between print publishing and software development, between marketing and computing, between internal communications and external relations, and between art and technology."

The following are the common attributes for WebApps:

● Network intensiveness: A WebApp resides on a network (Internet or Intranet) and must serve the needs of a diverse community of clients.

● Concurrency: A large number of users may access the WebApp at one time.

● Unpredictable load. The number of users of the WebApp may vary by orders of magnitude from day to day.

● Performance: If a WebApp user must wait too long, he or she may decide to go elsewhere.

● Availability: Although the expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis.

● Data driven: The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user

● Content sensitive: The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp.

● Continuous evolution: Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.

● Immediacy: WebApps often exhibit a time-to-market that can be a matter of a few days or weeks.

● Security: Because WebApps are available via network access sensitive content must be protected and secure modes of data transmission must be provided.

● Aesthetics: An undeniable part of the appeal of a WebApp is its look and feel.

SOFTWARE ENGINEERING

• Software engineering is the application of engineering to the development of software in a systematic method.

• A concerted effort should be made to understand the problem before a software solution is developed, therefore DESIGN becomes a pivotal activity.

• Software should always exhibit high quality.

• Software engineering encompasses a process, methods for managing and engineering software and tools

• Software engineering: “the application of a systematic, disciplined, quantifiable approach to the development, and maintenance of software.

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),

• An action (e.g. architectural design) encompasses a set of tasks,

• A task focuses on a small, but well-defined objective (e.g. conducting a unit test) A process defines who is doing what when and how to reach a certain goal.

SOFTWARE MYTHS

Software myths are erroneous beliefs about software and the process that is used to build it.

We categorize myths from three different perspectives.

Management myths:

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 (sometimes called the “Mongolian horde” concept).

Reality: Software development is not a mechanistic process like manufacturing. As new people are added, people who are 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 out-sources software projects.

Customer myths:

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.

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

Practitioner's myths:

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.

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 GENERIC PROCESS MODEL

SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY
- Quality focus - Bedrock that supports Software Engineering.
- Process - Foundation for software Engineering
- Methods - Provide technical How-to's for building software
- Tools - Provide semi-automatic and automatic support to methods

Fig: Software Engineering-A layered technology

A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created.

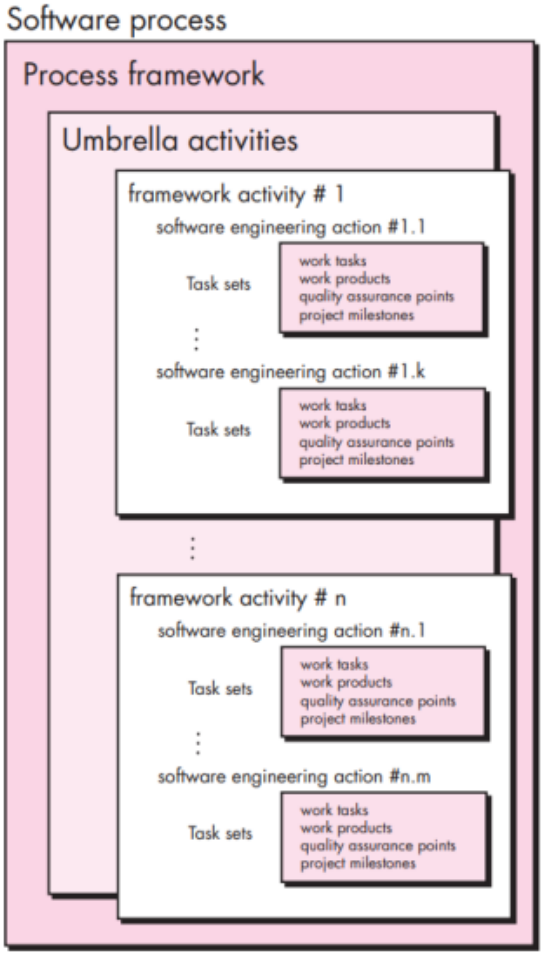
● Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.

● A generic process framework for software engineering defines five framework activities — communication, planning, modeling, construction, and deployment.

● 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.

● One important aspect of the software process called 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.



Defining a Framework Activity:

To properly execute any one of these activities as part of the software process, a key question that the software team needs to ask is:

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?

For a small software project requested by one person (at a remote location) with simple, straightforward requirements, the communication activity might encompass little more than a phone call with the appropriate stakeholder. The work tasks that this action encompasses are:

1. Make contact with stakeholder via telephone.

2. Discuss requirements and take notes

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

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

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.

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. You should choose a task set that best accommodates the needs of the project and the characteristics of your team.

CAPABILITY MATURITY MODEL INTEGRATION (CMMI)

• Developed by SEI (Software Engineering institute)

• Assess the process model followed by an organization and rate the organization with different levels

• A set of software engineering capabilities should be present as organizations reach different levels of process capability and maturity.

CMMI process meta model can be represented in different ways

1. A continuous model

2. A staged model

Continuous model:

-Lets organization select specific improvement that best meet its business objectives and minimize risk-Levels are called capability levels.

-Describes a process in 2 dimensions

-Each process area is assessed against specific goals and practices and is rated according to the following capability levels.

CMMI

• Six levels of CMMI

Level 0: Incomplete - Process is adhoc . Objective and goal of process areas are not known

Level 1: Performed - Goal,o bjective, work tasks, work products and other activities of software process are carried out

Level 2: Managed - Activities are monitored, reviewed, evaluated and controlled

Level 3: Defined - Activities are standardized, integrated and documented

Level 4: Quantitatively managed - Metrics and indicators are available to measure the process and quality

Level 5: Optimized - Continuous process improvement based on quantitative feed back from the user -Use of innovative ideas and techniques, statistical quality control and other methods for process improvement.

Staged model

- This model is used if you have no clue of how to improve the process for quality software.
- It gives a suggestion of what things other organizations have found helpful to work first
- Levels are called maturity levels

PRESCRIPTIVE PROCESS MODELS

Prescriptive process models were originally proposed to bring order to the chaos of software development. Software engineering work and the product that it produces remain on "the edge of chaos."

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.

LINEAR SEQUENTIAL MODEL OR THE WATERFALL MODEL

• Used when requirements are well understood in the beginning
• Also called classic life cycle
• A systematic, sequential approach to Software development
• Begins with customer specification of Requirements and progresses through planning, modeling, construction and deployment. This Model suggests a systematic, sequential approach to SW development that begins at the system level and progresses through analysis, design, code and testing.

Software requirements analysis: The requirements gathering process is intensified and focused specifically on software. To understand the nature of the program(s) to be built, the software engineer ("analyst") must understand the information domain for the software, as well as required function, behavior, performance, and interface. Requirements for both the system and the software are documented and reviewed with the customer.

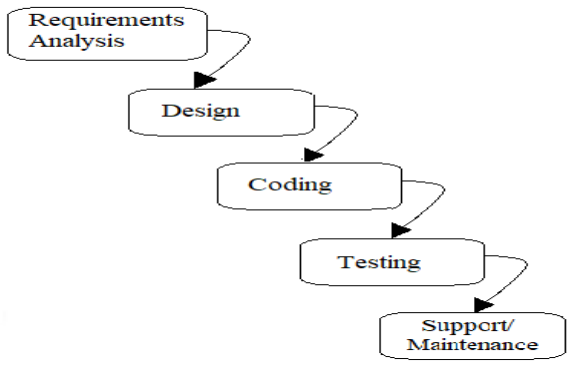
Design: Software design is actually a multistep process that focuses on four distinct attributes of a program: data structure, software architecture, interface representations, and procedural (algorithmic) detail. The design process translates requirements into a representation of the software that can be assessed for quality before coding begins.

Code generation: The design must be translated into a machine-readable form. The code generation step performs this task. If design is performed in a detailed manner, code generation can be accomplished mechanistically.

Testing: Once code has been generated, program testing begins. The testing process focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors and ensure that defined input will produce actual results that agree with required results.

Support: Software will undoubtedly undergo change after it is delivered to the customer. Change will occur because errors have been encountered, because the software must be adapted to accommodate changes in its external environment (e.g., a change required because of a new operating system or peripheral device), or because the customer requires functional or performance enhancements.

Linear Sequential Model or Waterfall Model



Problems in waterfall model

• Real projects rarely follow the sequential flow since they are always iterative

• The model requires requirements to be explicitly spelled out in the beginning, which is often difficult

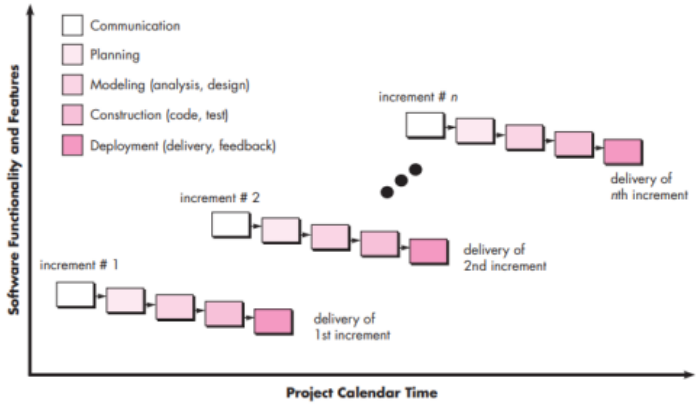
• A working model is not available until late in the project time plan

INCREMENTAL PROCESS MODEL

● The incremental model combines elements of linear and parallel process flows.

● The incremental model applies linear sequences in a staggered fashion as calendar time progresses.

● Each linear sequence produces deliverable “increments” of the software in a manner that is similar to the increments produced by an evolutionary process flow.



The Incremental Model

● 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 (or undergoes detailed evaluation).

● As a result, 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.

THE RAD (Rapid Application Development) MODEL

• An incremental software process model
• Having a short development cycle
• High-speed adoption of the waterfall model using a component based construction approach
• Creates a fully functional system within avery short span time of 60 to 90 days

The RAD approach encompasses the following phases:

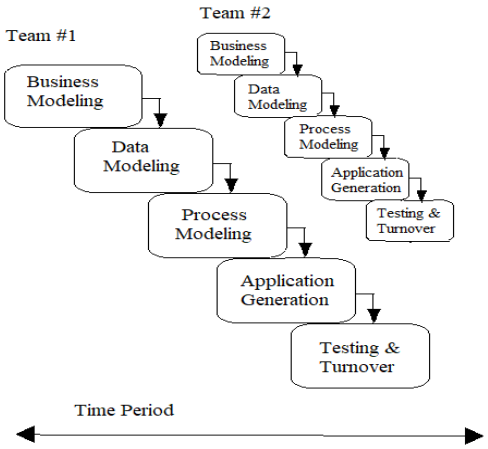
Business modeling: The information flow among business functions is modeled in a way that answers these questions: What information drives the business process? What information is generated? Who generates it? Where does the information go? Who processes it?

Data modeling. The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business. The characteristics (called attributes) of each object are identified and the relationships between these objects defined.

Process modeling. The data objects defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application generation. RAD assumes the use of fourth generation techniques . Rather than creating software using conventional third generation programming languages the RAD process works to reuse existing program components or create reusable components. In all cases, automated tools are used to facilitate construction of the software.

Testing and turnover. Since the RAD process emphasizes reuse, many of the program components have already been tested. This reduces overall testing time. However, new components must be tested and all interfaces must be fully exercised.



Problems in RAD

• Requires a number of RAD teams
• Requires commitment from both developer and customer for rapid-fire completion of activities

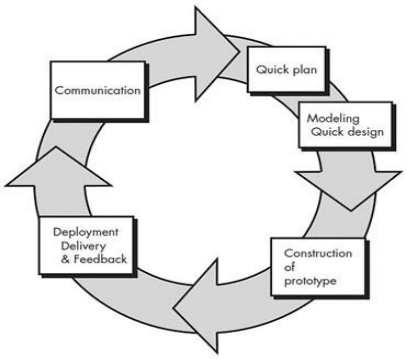
• Requires modularity
• Not suited when technical risks are high

EVOLUTIONARY PROCESS MODELS

Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software. There are two common evolutionary process models.

PROTOTYPING

• Mock up or model (throw away version) of a software product
• Used when customer defines a set of objective but does not identify input, output or processing requirements
• Developer is not sure of:
– efficiency of an algorithm
– adaptability of an operating system
– human/machine interaction



Steps in prototyping
• Begins with requirement gathering
• Identify whatever requirements are known
• Outline areas where further definition is mandatory
• A quick design occur
• Quick design leads to the construction of prototype
• Prototype is evaluated by the customer
• Requirements are refined
• Prototype is turned to satisfy the needs of customer

Limitations of prototyping
• In a rush to get it working, overall software quality or long term maintainability are generally overlooked
• Use of inappropriate OS or PL
• Use of inefficient algorithm

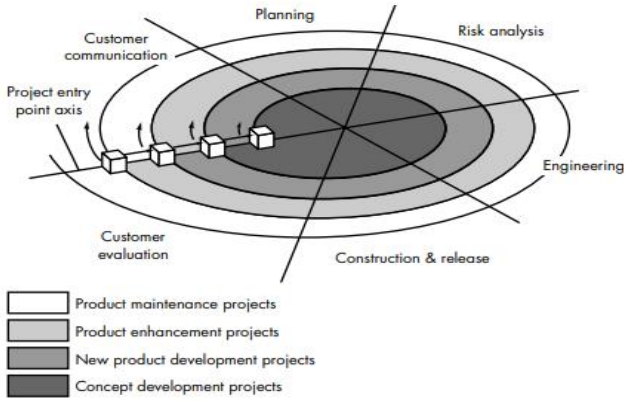
THE SPIRAL MODEL

An evolutionary model which combines the best feature of the classical life cycle and the iterative nature of prototype model. Include new element : Risk element. Starts in middle and continually visits the basic tasks of communication, planning, modeling, construction and deployment.

Software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced. A spiral model is divided into a number of framework activities, also called task regions. Typically, there are between three and six task regions. The spiral model that contains six task regions:

- Customer communication—tasks required to establish effective communication between developer and customer.
- Planning—tasks required to define resources, timelines, and other project related information.
- Risk analysis—tasks required to assess both technical and management risks.
- Engineering—tasks required to build one or more representations of the application.
- Construction and release—tasks required to construct, test, install, and provide user support (e.g., documentation and training).
- Customer evaluation—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

FIGURE 2.8
A typical spiral model



Advantages

The spiral model is a realistic approach to the development of large-scale systems and software. Because software evolves as the process progresses, the developer and customer better understand and react to risks at each evolutionary level. The spiral model uses prototyping as a risk reduction mechanism but, more important, enables the developer to apply the prototyping approach at any stage in the evolution of the product. 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 spiral model demands a direct consideration of technical risks at all stages of the project and, if properly applied, should reduce risks before they become problematic.

Disadvantages

It may be difficult to convince customers (particularly in contract situations) that the evolutionary approach is controllable. It demands considerable risk assessment expertise and relies on this expertise for success. If a major risk is not uncovered and managed, problems will undoubtedly occur. Finally, the model has not been used as widely as the linear sequential or prototyping paradigms. It will take a number of years before efficacy of this important paradigm can be determined with absolute certainty.

CONCURRENT DEVELOPMENT MODEL

The concurrent development model is called as concurrent model.

• The communication activity has completed in the first iteration and exits in the awaiting changes state.

• The modeling activity completed its initial communication and then go to the underdevelopment state.

• If the customer specifies the change in the requirement, then the modeling activity moves from the under development state into the awaiting change state.

• The concurrent process model activities moving from one state to another state.

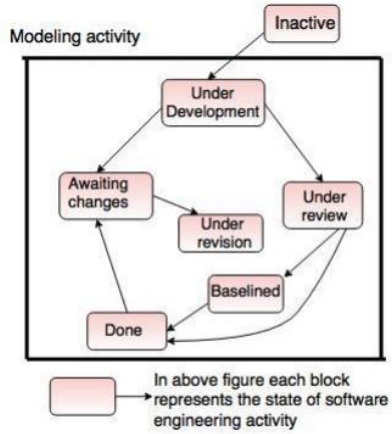


Fig. - One element of the concurrent process model

Advantages of the concurrent development model

• This model is applicable to all types of software development processes.

• It is easy for understanding and use.

• It gives immediate feedback from testing.

• It provides an accurate picture of the current state of a project.

Disadvantages of the concurrent development model

• It needs better communication between the team members. This may not be achieved all the time. • It requires to remember the status of the different activities.

SPECIALIZED PROCESS MODELS

COMPONENT-BASED DEVELOPMENT

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

The engineering activity begins with the identification of candidate classes. This is accomplished by examining the data to be manipulated by the application and the algorithms

that will be applied to accomplish the manipulation. Corresponding data and algorithms are packaged into a class.

Classes created in past software engineering projects are stored in a class library or repository. Once candidate classes are identified, the class library is searched to determine if these classes already exist. If they do, they are extracted from the library and reused. If a candidate class does not reside in the library, it is engineered using object-oriented methods

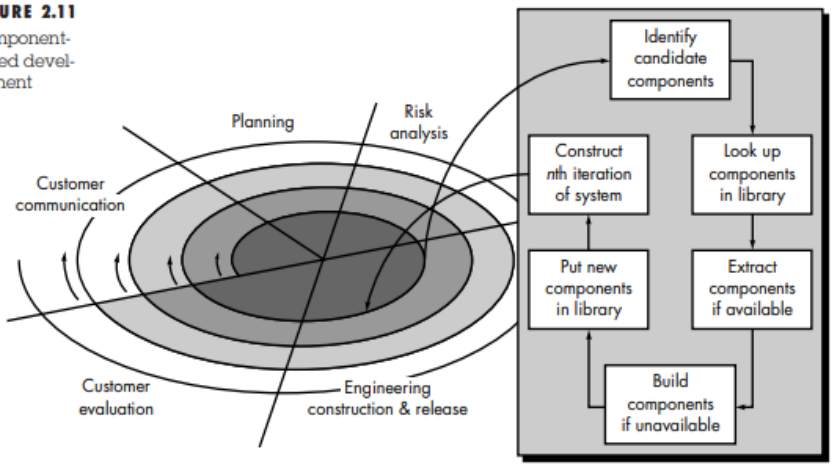


FIGURE 2.11 Component-based development

12 This is a simplified description of class definition. For a more detailed discussion, see Chapter 20.

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

The unified software development process is representative of a number of component-based development models that have been proposed in the industry. Using the Unified Modeling Language (UML), the unified process defines the components that will be used to build the system and the interfaces that will connect the components. Using a combination of iterative and incremental development, the unified process defines the function of the system by applying a scenario-based approach (from the user point of view).

Advantages

- Divides large projects into smaller subprojects
- CBSE is language independent.
- Reduced time to market
- Increased productivity/quality
- Reusability of components
- Easy to replace components

Disadvantages

- Making Components Reusable
- Increased Development time
- Harder to identify requirements
- Decreases usability
- Choosing Middleware
- Incompatible technologies
- Testing is Harder
- Unknown uses of components

- High initial cost
- Training of developers

FORMAL METHODS MODEL

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called cleanroom software engineering, is currently applied by some software development organizations.

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 ad hoc review but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might go undetected. Although it is not destined to become a mainstream approach, the formal methods model offers the promise of defect-free software.

Yet, the following concerns about its applicability in a business environment have been voiced:

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

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

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

ASPECT-ORIENTED SOFTWARE DEVELOPMENT

SD is a programming paradigm that aims to modularize cross-cutting concerns in software development. It enhances modularity by isolating aspects (cross-cutting concerns) into separate modules.

Motivation: Traditional approaches (like object-oriented programming) often result in tangled code where concerns are interwoven, making maintenance and evolution difficult.

Key Concepts

Aspect: A modular unit that encapsulates cross-cutting concerns (e.g., logging, security, transaction management).

Join Point: A specific point in the execution of a program, where an aspect can be applied.

Advice: Code that specifies what actions to take at a particular join point (e.g., before, after, or around).

Pointcut: Specifies where aspects should be applied in the code (e.g., methods of a certain class).

Core Principles of AOSD

Separation of Concerns: AOSD promotes clearer separation between core business logic and cross-cutting concerns, enhancing code modularity and maintainability.

Aspect Weaving: The process of integrating aspects with the main application code at compile-time, load-time, or runtime.

Aspect Interference: Ensuring that aspects do not interfere with each other and with the main functionality of the program.

Benefits of AOSD

Modularity: Enhances modularity by isolating concerns into separate modules (aspects).

Maintainability: Eases maintenance by making it easier to locate and modify code related to specific concerns.

Reusability: Aspects can be reused across different parts of the software system.

Scalability: Facilitates scalability by managing concerns independently, thus allowing easier addition or modification of features.

Tools and Frameworks

AspectJ: A widely-used aspect-oriented extension to Java that provides support for AOSD concepts like aspects, join points, advice, and pointcuts.

Spring AOP: Part of the Spring Framework that provides AOP features to support aspects and allows for cross-cutting concerns to be applied to Spring-managed objects.

Challenges and Considerations

Complexity: AOSD introduces additional complexity, especially in understanding and managing aspects across a large codebase.

Performance Overhead: Aspect weaving at runtime can introduce performance overhead, especially for heavily used aspects.

Tool Support: Availability and maturity of tools and frameworks for AOSD vary, impacting adoption and ease of use.

Application Areas

Logging: Capturing and recording events in the application.

Security: Enforcing security policies across different modules.

Transaction Management: Managing database transactions consistently across the application.

Conclusion

Future Directions: Research and development in AOSD continue to explore improved methodologies, tools, and integration with other software engineering practices.

Recommendations: Consider AOSD for projects where modularity, maintainability, and separation of concerns are critical.