ISSN: 2278-0181

# SOFTWARE ENGINEERING AND CASE

Karthik. Kamath, Narasimha. Prabhu, Sunil. Sridhar
Department Of Information Science and Engineering
Bangalore Institute of Technology

***Abstract*- Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly. It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems. For most types of system, the majority of costs are the costs of changing the software after it has gone into use.**

**Computer aided software engineering is the name given to the software used to support software process activities such as requirement engineering, design, program development and testing. CASE tools therefore include design editors, data dictionaries, compilers, debuggers, system building tools. CASE technology provides software process support by automating some process activities and by providing information about the software that is being developed.**

***Index Terms*-- *CASE technology, CASE tools, process activity, software engineering methods, software techniques.***

## I. INTRODUCTION

The development of software applications is rarely carried out by a single programmer. Rather, it demands for participation of teams of people. The interaction and exchange of information among team members with the development of economically software that is reliable and works efficiently on real machines is termed as software engineering. Software engineering is a part of system engineering. Software engineers should adopt a systematic and organised approach to all aspects of software development. The foundation for software engineering is the process layer. Software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework for a set of key process areas that must be established for effective delivery of software engineering technology. The key process areas form the basis for management control of software projects and establish the context in which technical methods are applied, work products are produced, milestones are established, quality is ensured, and change is properly managed. Engineering discipline Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work to organizational and financial constraints so they look for solutions within these constraints.

## II. SOFTWARE LIFE CYCLE MODELS

A software life cycle model is either a descriptive or prescriptive characterization of how software is or should be developed. A descriptive model describes the history of how a particular software system was developed. Descriptive models may be used as the basis for understanding and improving software development processes or for building empirically grounded prescriptive models. A prescriptive model prescribes how a new software system should be developed. Prescriptive models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order. Typically, it is easier and more common to articulate a prescriptive life cycle model for how software systems should be developed. This means that many idiosyncratic details that describe how software Systems is built in practice can be ignored, generalized, or deferred for later consideration. This, of course, should raise concern for the relative validity and robustness of such life cycle models. When developing different kinds of application systems, in different kinds of development settings, using different programming languages, with differentially skilled staff, etc. However, prescriptive models are also used to package the development tasks and techniques for using a given set of software engineering tools or environment during a Development project. Descriptive life cycle models characterize how particular software systems are actually developed in specific settings. As such, they are less common and more difficult to articulate for an obvious reason: one must observe or collect data throughout the life cycle of a software system, a period of elapsed time often measured in years. Also, descriptive models are specific to the systems observed and only generalizable through systematic comparative analysis. Therefore, this suggests the prescriptive software life cycle models will dominate attention until a sufficient base of observational data is available to articulate empirically grounded descriptive life cycle models.

These two characterizations suggest that there are a variety of purposes for articulating software life cycle models. These characterizations serve as a Guideline to organize, plan, staff, budget, and schedule and manage software project work over organizational time, space, and computing environments. Basis for determining what software engineering tools and methodologies will be most appropriate to support different life cycle activities. Basis for conducting empirical studies to determine what affects software productivity, cost and overall quality.

The phases through which a product progresses is:

- Requirements phase
- Specification phase
- Design phase
- Implementation phase
- Integration phase
- Maintenance phase
- Retirement

*SOFTWARE PROCESS MODEL*

*1). Waterfall Model*

Each one follows the other sequentially and doesn't begin until the previous one is finished. The drawback of the waterfall model is the difficulty of accommodating change after the process is underway.

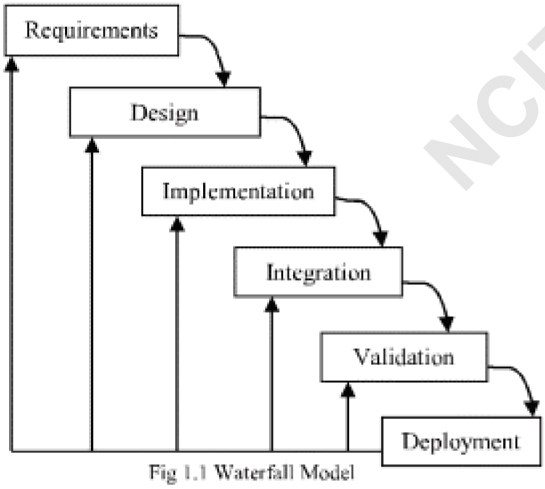


Fig 1.1 Waterfall Model

1) *System requirements:* Establishes the components for building the system, including the hardware requirements, software tools, and other necessary components. Examples include decisions on hardware, such as plug-in boards (number of channels, acquisition speed, and so on), and decisions on external pieces of software, such as databases or libraries.
2) *Software requirements:* Establishes the expectations for software functionality and identifies which system requirements the software affects. Requirements analysis includes determining interaction needed with other applications and databases, performance requirements, user interface requirements, and so on.
3) *Architectural designs:* Determines the software framework of a system to meet the specific requirements. This design defines the major components and the interaction of those components, but it does not define the structure of each component. The external interfaces and tools used in the project can be determined by the designer.
4) *Detailed designs*: Examines the software components defined in the architectural design stage and produces a specification for how each component is implemented.
5) *Coding:* Implements the detailed design specification.
6) *Testing:* Determines whether the software meets the specified requirements and finds any errors present in the code.
7) *Maintenance:* Addresses problems and enhancement requests after the software releases. In some organizations, a change control board maintains the quality of the product by reviewing each change made in the maintenance stage. Consider applying the full waterfall development cycle model when correcting problems or implementing these enhancement requests.

In each stage, documents that explain the objectives and describe the requirements for that phase are created. At the end of each stage, a review to determine whether the project can proceed to the next stage is held. Your prototyping can also be incorporated into any stage from the architectural design and after.

## III. ADVANCED SOFTWARE ENGINEERING

*Component-based software engineering:*

Component-based software engineering (CBSE) emerged in the late 1990s as an approach to software systems development based on reusing software components. Its creation was motivated by designers' frustration that object-oriented development had not led to extensive reuse, as had been originally suggested. Single object classes were too detailed and specific, and often had to be bound with an application at compile time. You had to have detailed knowledge of the classes to use them, and this usually meant that you had to have the component source code. This meant that selling or distributing objects as individual reusable components was practically impossible.

Components are higher-level abstractions than objects and are defined by their interfaces. They are usually larger than individual objects and all implementation details are hidden from other components. CBSE is the process of defining, implementing, and integrating or composing loosely coupled independent components into systems. It has become as an important software development approach because software systems are becoming larger and more complex. Customers are demanding more dependable software that is delivered and deployed more quickly. The only way that we can cope with complexity and deliver better software more quickly is to reuse rather than implement software components.

The essentials of component-based software engineering are:

1) Independent components that are completely specified by their interfaces. There should be a clear separation between the component interface and its implementation.

This means that one implementation of a component can be replaced by another, without changing other parts of the system.

2) Component standards that facilitate the integration of components. These standards are embodied in a component model. They define, at the very minimum, how component interfaces should be specified and how components communicate. Some models go much further and define interfaces that should be implemented by all conformant components. If components conform to standards, then their operation is independent of their programming language. Components written in different languages can be integrated into the same system.
3) Middleware that provides software support for component integration. To make independent, distributed components work together, you need middleware support that handles component communications. Middleware for component support handles low-level issues efficiently and allows you to focus on application-related problems. In addition, middleware for component support may provide support for resource allocation, transaction management, security, and concurrency.
4) A development process that is geared to component-based software engineering.

Component-based development embodies good software engineering practice. It makes sense to design a system using components, even if you have to develop rather than reuse these components.

*THE CBSE PROCESS*


Fig 1.7 Component Identification Process

Fig 1.7 shows the principal sub activities within a sub process, such as initial discovery of user requirements, are carried out in the same way as in other software processes. However, the essential difference between these process and software process based on original software development are:

1) The user requirements are initially developed in outline rather than in detail, and stakeholders are encouraged to be as flexible as possible in defining their requirements. Requirements that are too specific limit the number of components that could meet these requirements. However, unlike incremental development, you need a complete set of requirements so that you can identify as many components as possible for reuse.
2) Requirements are refined and modified early in the process depending on the components available. If the user requirements cannot be satisfied from available components, you should discuss the related requirements that can be supported. Users may be willing to change their minds if this means cheaper or quicker system delivery.
3) There is a further component search and design refinement activity after the system architecture has been designed. Some apparently usable components may turn out to be unsuitable or do not work properly with other chosen components. Although not shown in Figure this implies that further requirements changes may be necessary.
4) Development is a composition process where the discovered components are integrated. This involves integrating the components with the component model infrastructure and, often, developing adaptors that reconcile the interfaces of incompatible components. Of course, additional functionality may also be required over and above that provided by reused components.

## IV. EMERGING TECHNOLOGIES

1.) Service Oriented Software Engineering
2.) Aspect Oriented Software Engineering

*Service oriented software engineering*

Service-oriented architectures (SOAs) are a way of developing distributed systems where the system components are stand-alone services, executing on geographically distributed computers. Figure encapsulates the idea of a SOA. Service providers design and implement services and specify the interface to these services. They also publish information about these services in an accessible registry. Service requestors (sometimes called service clients) who wish to make use of a service discover the specification of that service and locate the service provider. They can then bind their application to that specific service and communicate with it, using standard service protocols.

Fig 1.8 encapsulates the idea of a SOA. Service providers design an implement services and specify the interface to these services. They also publish information about these services in an accessible registry. Service requestors(Sometimes called service clients) who wish to make use of a service discovers the specification of that service and locates the service provider. They can then bind their application to that specific service and communicate with it, using standard service protocols.

From the outset, there has been an active standardization process for SOA, working alongside technical developments. All of the major hardware and software companies are committed to these standards. As a result, SOA have not suffered from the incompatibilities that normally arise with technical innovations, where different suppliers maintain their proprietary version of the technology. Figure 1.8 shows the stack of key standards that have been established to support web services. Because of this early standardization, problems, such as the multiple incompatible component models in CBSE, discussed in Chapter 17, have not arisen in service-oriented system development.

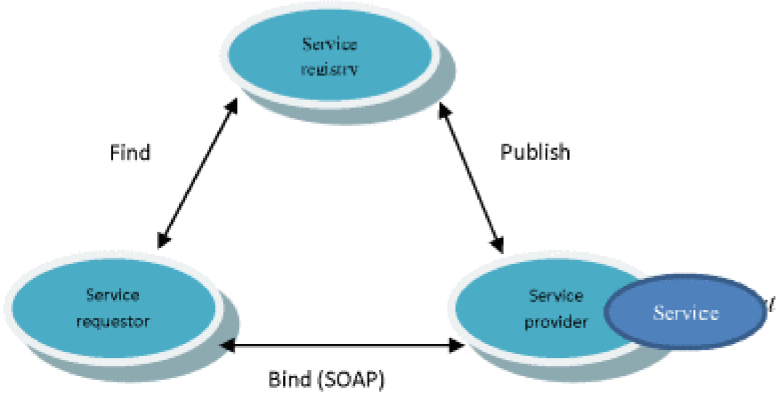


Fig 1.8 Service Oriented Architecture

When you intend to use a web service, you need to know where the service is located (it's URI) and the details of its interface. These are described in a service description expressed in an XML-based language called WSDL. The WSDL specification defines three things about a web service: what the service does, how it communicates, and where to find it:

1) The 'what' part of a WSDL document, called an interface, specifies what operations the service supports, and defines the format of the messages that are sent and received by the service?
2) The 'how' part of a WSDL document, called a binding, maps the abstract interface to a concrete set of protocols. The binding specifies the technical details of how to communicate with a web service.
3) The 'where' part of a WSDL document describes the location of a specific web service implementation (its endpoint).

The WSDL conceptual model (Figure 1.11) shows the elements of a service description. Each of these is expressed in XML and may be provided in separate files. These parts are:

a. An introductory part that usually defines the XML namespaces used and which may include a documentation section providing additional information about the service.
b. An optional description of the types used in the messages exchanged by the service.
c. A description of the service interface; that is, the operations that the service provides for other services or users.
d. A description of the input and output messages processed by the service.

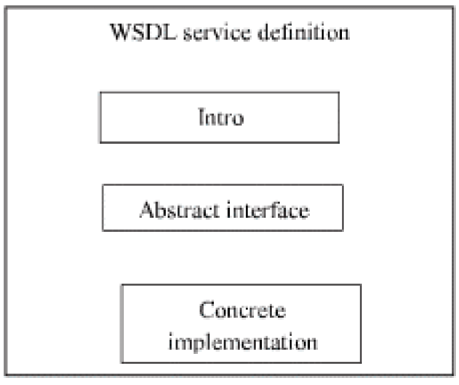


Fig 1.11 Organization of a WSDL specification

*t oriented software engineering*

Aspect-oriented software engineering (AOSE) is an approach to software development that is intended to address

this problem and so make programs easier to

Maintain and reuse. AOSE is based around abstractions called aspects, which implement system functionality that may be required at several different places in a Program. Aspects encapsulate functionality that cross-cuts and coexists with other functionality that is included in a system. They are used alongside other abstractions such as objects and methods. An executable aspect-oriented program is created by automatically combining (weaving) objects, methods, and aspects, according to specifications that are included in the program source code. An important characteristic of aspects is that they include a definition of where they should be included in a program, as well as the code implementing the crosscutting concern. You can specify that the cross-cutting code should be included before or after a specific method call or when an attribute is accessed. Essentially, the aspect is woven into the core program to create a new augmented system. The key benefit of an aspect-oriented approach is that it supports the separation of concerns. Separating concerns into independent elements rather than including different concerns in the same logical abstraction is good software engineering practice. By representing cross-cutting concerns as aspects,\ these concerns can be understood, reused, and modified independently, without regard for where the code is used. For example, user authentication may be represented as an aspect that requests a login name and password. This can be automatically woven into the program wherever authentication is required.

Say you have a requirement that user authentication is required before any change to personal details is made in a database. You can describe this in an aspect by stating that the authentication code should be included before each call to methods that update personal details. Subsequently, you may extend the requirement for authentication to all database updates. This can easily be implemented by modifying the aspect. You simply change the definition of where the authentication code is to be woven into the system. You do not have to search through the system looking for all occurrences of these methods. You are therefore less likely to make mistakes and introduce accidental security vulnerabilities into your program. Research and development in aspect-orientation has primarily focused on aspect oriented programming. Aspect-oriented programming languages such as Aspect J have been developed that extend object-oriented Programming to include aspects. Major companies have used aspect-oriented programming. Major companies have used aspect-oriented programming.

Aspect weavers are extensions to compilers that process the definition of aspects and the object classes and methods that define the system. The weaver then generates a new program with the aspects included at the specified join points. The aspects are integrated so that the cross-cutting concerns are executed at the right places in the final system.

Figure 1.15 illustrates this aspect weaving for the authentication and logging aspects that should be included in the MHC-PMS. There are three different approaches to aspect weaving:

1) Source code pre-processing, where a weaver takes source code input and generates new source code in a language such as Java or C++, which can then be compiled using the standard language compiler. This approach has been adopted for the Aspect-X language with its associated X-Weaver.

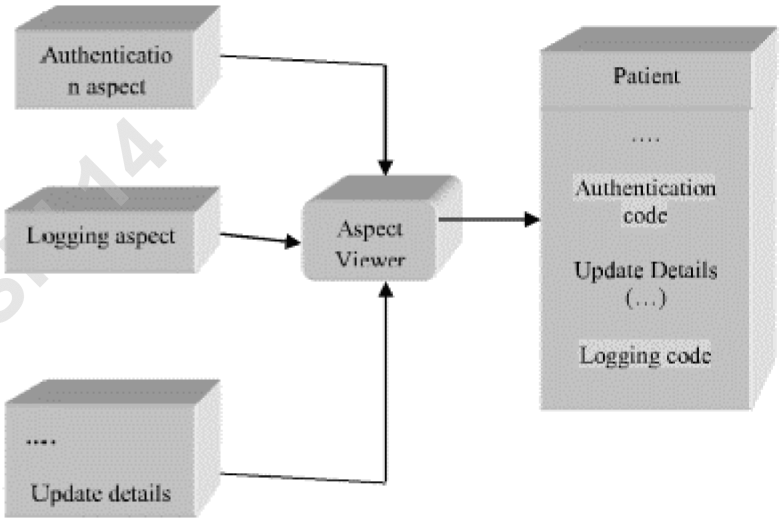


Fig 1.15 Aspect Weaving

2) Link time weaving, where the compiler is modified to include an aspect weaver. An aspect-oriented language such as Aspect-J is processed and standard Java byte code is generated. This can then be executed directly by a Java interpreter or further processed to generate native machine code.
3) Dynamic weaving at execution time. In this case, join points are monitored and when an event that is referenced in a point cut occurs, the corresponding advice is integrated with the executing program.

Computer-aided software engineering (CASE) is the scientific application of a set of tools and methods to a software system which is meant to result in high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems together with automated tools that can be used in the software

development process. The term "computer-aided software engineering" (CASE) can refer to the software used for the automated development of systems software, i.e., computer code. The CASE functions include analysis, design, and programming. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language. CASE software supports the software process activities such as requirement engineering, design, program development and testing. Therefore, CASE tools include design editors, data dictionaries, compilers, debuggers, system building tools, etc. CASE also refers to the methods dedicated to an engineering discipline for the development of information system using automated tools. CASE is mainly used for the development of quality software which will perform effectively. CASE history in a nutshell. Software designers have used diagrammatic representations of their designs since the earliest days of software development. Over time the nature of these design diagrams has changed and so have the tools used to produce them. Much like early word processors replaced typewriters, early CASE tools served as electronic replacements for paper, pencil, and stencil. Many of these early CASE tools became unused "shelf ware" because they did not provide significant value to software designers. Later CASE tools added sophisticated code generation, reverse engineering, and version control features. These features add value via increased automation of some design tasks, for example, converting a design into a source code skeleton. However, current CASE tools fail to address the essential cognitive challenges facing software designers. International Data Corporation (IDC), a market research firm that collects data on all aspects of the computer hardware and software industries, has published a series of 2 reports on OOAD tools.

*CASE tool and its scope*

A CASE (Computer Aided Software Engineering) tool is a generic term used to denote any form of automated support for software engineering. In a more restrictive sense, a CASE tool means any tool used to automate some activity associated with software development. Many CASE tools are available. Some of these CASE tools assist in phase related tasks such as specification, structured analysis, design, coding, testing, etc.; and others to non-phase activities such as project management and configuration management.

*Benefits of CASE:*

Several benefits accrue from the use of a CASE environment or even isolated CASE tools. Some of those benefits are:

- A key benefit arising out of the use of a CASE environment is cost saving through all development phases. Different studies carry out to measure the impact of CASE put the effort reduction between 30% to 40%.
- Use of CASE tools leads to considerable improvements to quality. This is mainly due to the facts that one can effortlessly iterate through the different phases of software development and the chances of human error are considerably reduced.
- CASE tools take out most of the drudgery in a software engineer's work. For example, they need not check meticulously the balancing of the DFDs but can do it effortlessly through the press of a button.
- CASE tools have led to revolutionary cost saving in software maintenance efforts. This arises not only due to the tremendous value of a CASE environment in traceability and consistency checks, but also due to the systematic information capture during the various phases of software development as a result of adhering to a CASE environment.

## III. CONCLUSION

We believe that the software engineering has provided us with some valuable insights into methods of software production and development. Software engineering research has progressed much since the early days. It has provided an effective means for efficient software development with the advance in emergent technologies like service and aspect oriented software engineering, the domain of software development has been expanded to focus on quality of service and customer satisfaction.

## REFERENCES

[1] Walt Scacchi, (February 2001) "Process Models in Software Engineering"
[2] Ian Somerville, "Software Engineering", Addison Wesley, 7th editions, 2004.
[3] Nabil Mohammed Ali Munassar and A. Govardhan "Comparison between five models of software engineering"
[4] Ian Somerville, "Software Engineering", Addison Wesley, 9th editions, 2004.
[5] Jongmoon Baik, (December 2000) "The effects of case tools on software development"