4D International Journal of Management and Science
ISSN No: 2250-0669
@4D Crossconnect.com, Inc, 2012
www.4dinternationaljournal.com
Vol. 3, Issue2, July 2013

ANALYZING AND DEVELOPING SOFTWARE DEVELOPMENT MODELS

Mr. Sachin Kumar Suryan[1] and Prof (Dr.) Sanjeev Gupta[2]

ABSTRACT

Software systems come and go through a series of passages that account for their inception, initial development, productive operation, upkeep, and retirement from one generation to another. This paper categorizes and examines a number of methods for describing or modeling how software systems are developed. It begins with background and definitions of traditional software life cycle models that dominate most textbook discussions and current software development practices. This is followed by a more comprehensive review of the alternative models of software evolution that are of current use as the basis for organizing software engineering projects and technologies. The overall framework in which software is conceived, developed, and maintained is known as the Software Development Life Cycle (SDLC). This chapter discusses the various types of SDLCs, along with their advantages and disadvantages. A life cycle model defines the phases, milestones, deliverables, and evaluation criteria of the software development process. These form the basis of the work breakdown structure (WBS), used for project planning and management.

Key words: Software Development Life Cycle, Work Breakdown Structure, Spiral Model.

---

[1] Associate Professor (MCA Dept), FIMT, Kapashera, New Delhi
[2] ATM Global Business School, New Delhi

INTRODUCTION

In the early days of computing, software was developed by many individuals following their own methods. Often, the methods employed some form of "code and fix", where the programmer writes some code and then tests it to see how it performs. The programmer then uses the test results to modify or fix the code and tests again. Programmers were able to get by with this type of development for two reasons. First, no better way had been developed, and second, software was not that complex. As software grew more complicated and organizations relied on computers for more of their operations, including finances and even human lives, this laissez faire approach to programming gave way to more disciplined methods.

Software development is the set of activities and processes that will eventually result in a software product. This may include inventing, improving, selecting among alternative solutions, and then describing computer programs that meet user requirements within the constraints of the environment.

Back ground: A software development methodology or system *development methodology* in software engineering is a framework that is used to structure, plan, and control the process of developing an information system. A software development methodology refers to the framework that is used to structure, plan, and control the process of developing an information system.. Each of the available methodologies is best suited to specific kinds of projects, based on various technical, organizational, project and team considerations.

The framework of a software development methodology consists of:

A software development philosophy, with the approach or approaches of the software development process Multiple tools, models and methods, to assist in the software development process. These frameworks are often bound to some kind of organization, which further develops, supports the use, and promotes the methodology. The methodology is often documented in some kind of formal documentation.

REVIEW OF LITERATURE

Software life cycle model

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 (Curtis, Krasner, Iscoe, 1988). 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.

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.

Software process model

Software process networks can be viewed as representing multiple interconnected task chains (Kling 1982, Garg 1989). Task chains represent a non-linear sequence of actions that structure and transform available computational objects (resources) into intermediate or finished products. Non-linearity implies that the sequence of actions may be non-deterministic, iterative, accommodate multiple/parallel alternatives, as well as partially ordered to account for incremental progress. Task actions in turn can be viewed a non-linear sequences of primitive actions which denote atomic units of computing work, such as a user's selection of a command or menu entry using a mouse or keyboard.

Winograd and others have referred to these units of cooperative work between people and computers as "structured discourses of work" (Winograd 1986), while task chains have become popularized under the name of "workflow" (Bolcer 1998).

Task chains can be employed to characterize either prescriptive or descriptive action sequences. Prescriptive task chains are idealized plans of what actions should be accomplished, and in what order.

Task chains join or split into other task chains resulting in an overall production network or web (Kling 1982). The production web represents the "organizational production system" that transforms raw computational, cognitive, and other organizational resources into assembled, integrated and usable software systems.

However, prescriptive task chains and actions cannot be formally guaranteed to anticipate all possible circumstances or idiosyncratic foul-ups that can emerge in the real world of software

development (Bendifallah 1989, Mi 1990). Thus, any software production web will in some way realize only an approximate or incomplete description of software development.

Articulation work is a kind of unanticipated task that is performed when a planned task chain is inadequate or breaks down. It is work that represents an open-ended non-deterministic sequence of actions taken to restore progress on the disarticulated task chain, or else to shift the flow of productive work onto some other task chain (Bendifallah 1987, Grinter 1996, Mi 1990, Mi 1996, Scacchi and Mi 1997).

TRADITIONAL SOFTWARE LIFE CYCLE MODELS:

Traditional models of software evolution have been with us since the earliest days of software engineering. In this section, we identify four. The classic software life cycle (or "waterfall chart") and stepwise refinement models are widely instantiated in just about all books on modern programming practices and software engineering.

Classic Software Life Cycle:-

The classic software life cycle is often represented as a simple prescriptive waterfall software phase model, where software evolution proceeds through an orderly sequence of transitions from one phase to the next in order (Royce 1970). Such models resemble finite state machine descriptions of software evolution.

The Waterfall Model of Software Development (Royce 1970)

Stepwise Refinement

In this approach, software systems are developed through the progressive refinement and enhancement of high-level system specifications into source code components (Wirth 1971, Mili 1986). However, the choice and order of which steps to choose and which refinements to apply remain unstated. Instead, formalization is expected to emerge within the heuristics and skills that are acquired and applied through increasingly competent practice. This model has been most effective and widely applied in helping to teach individual programmers how to organize their software development work. Many interpretations of the classic software life cycle thus subsume this approach within their design and implementations.

Incremental Development and Release:-

Developing systems through incremental release requires first providing essential operating functions, then providing system users with improved and more capable versions of a system at regular intervals (Basili 1975). This model combines the classic software life cycle with iterative

enhancement at the level of system development organization. It also supports a strategy to periodically distribute software maintenance updates and services to dispersed user communities. This in turn accommodates the provision of standard software maintenance contracts. It is therefore a popular model of software evolution used by many commercial software firms and system vendors. This approach has also been extended through the use of software prototyping tools and techniques (described later), which more directly provide support for incremental development and iterative release for early and ongoing user feedback and evaluation (Graham 1989). It provides an example view of an incremental development, build, and release model for engineering large Ada-based software systems, developed by Royce (1990) at TRW.

Elsewhere, the Clean room software development method at use in IBM and NASA laboratories provides incremental release of software functions and/or subsystems (developed through stepwise refinement) to separate in-house quality assurance teams that apply statistical measures and analyses as the basis for certifying high-quality software systems (Sellby1987,Mills1987)

Alternatives to the Traditional Software Life Cycle Models:

There are at least three alternative sets of models of software development. These models are alternatives to the traditional software life cycle models. These three sets focus of attention to either the products, production processes, or production settings associated with software development. Collectively, these alternative models are finer-grained, often detailed to the point of computational formalization, more often empirically grounded, and in some cases address the role of new automated technologies in facilitating software development. As these models are not in widespread practice, we examine each set of models in the following sections.

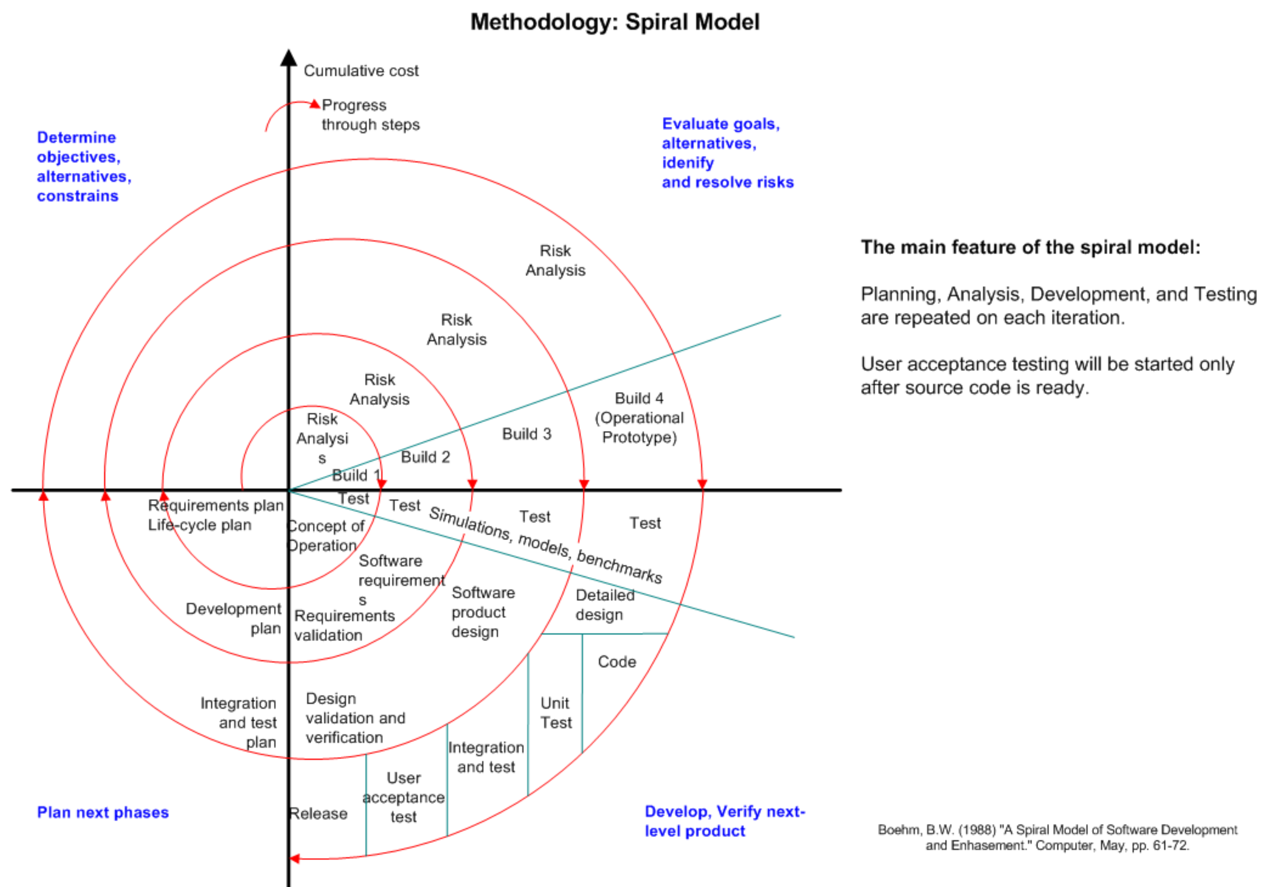
Software Product Development Models:

Software products represent the information-intensive artifacts that are incrementally constructed and iteratively revised through a software development effort. Such efforts can be modeled using software product life cycle models. These product development models represent an evolutionary revision to the traditional software life cycle models (MacCormack 2001). The revisions arose due to the availability of new software development technologies such as software prototyping languages and environments, reusable software, application generators, and documentation support environments. Each of these technologies seeks to enable the creation of executable software implementations either earlier in the software development effort or more rapidly. Therefore in this regard, the models of software development may be implicit in the use of the

technology, rather than explicitly articulated. This is possible because such models become increasingly intuitive to those developers whose favorable experiences with these technologies substantiate their use. Thus, detailed examination of these models is most appropriate when such technologies are available for use or experimentation.

Rapid Iteration, Incremental Evolution, and Evolutionary Delivery

There are a growing number of technological, social and economic trends that are shaping how a new generation of software systems are being developed that exploit the Internet and World Wide Web. These include the synchronize and stabilize techniques popularized by Microsoft and Netscape at the height of the fiercely competitive efforts to dominate the Web browser market of the mid 1990's (Cusumano and Yoffie, 1999, MacCormack 2001). They also include the development of open source software systems that rely on a decentralized community of volunteer software developers to collectively develop and test software systems that are incrementally enhanced, released, experienced, and debugged in an overall iterative and cyclic manner (DiBona 1999, Fogel 1999, Mockus 2000).



There are two classes of non-operational software process models of the great interest. These are the spiral model and the continuous transformation models. There is also a wide selection of other non-operational models, which for brevity we label as miscellaneous models. Each is examined in turn.

The Spiral Model: - The spiral model of software development and evolution represents a risk driven approach to software process analysis and structuring (Boehm 1987, Boehm *et al*, 1998). This approach, developed by Barry Boehm, incorporates elements of specification-driven, prototype-driven process methods, together with the classic software life cycle. It does so by representing iterative development cycles as an expanding spiral, with inner cycles denoting early system analysis and prototyping, and outer cycles denoting the classic software life cycle.

Process Models. Many variations of the non-operational life cycle and process models have been proposed, and appear in the proceedings of the international software process workshops sponsored by the ACM, IEEE, and Software Process Association. These include fully

Interconnected life cycle models that accommodate transitions between any two phases subject to satisfaction of their pre- and post-conditions, as well as compound variations on the traditional life cycle and continuous transformation models.

Emerging Trends and New Directions

In addition to the ongoing interest, debate, and assessment of process-centered or process-driven software engineering environments that rely on process models to configure or control their operation (Ambriola 1999, Garg and Jazayeri 1996), there are a number of promising avenues for further research and development with software process models. These opportunities areas and sample direction for further exploration include:

Software process simulation (Raffo et al, 1999, Raffo and Scacchi 2000) efforts which seek to determine or experimentally evaluate the performance of classic or operational process models using a sample of alternative parameter configurations or empirically derived process data (cf. Cook and Wolf 1998). Simulation of empirically derived models of software evolution or evolutionary processes also offers new avenues for exploration (Chatters, Lehman, *et al.*, 2000, Mockus 2000).

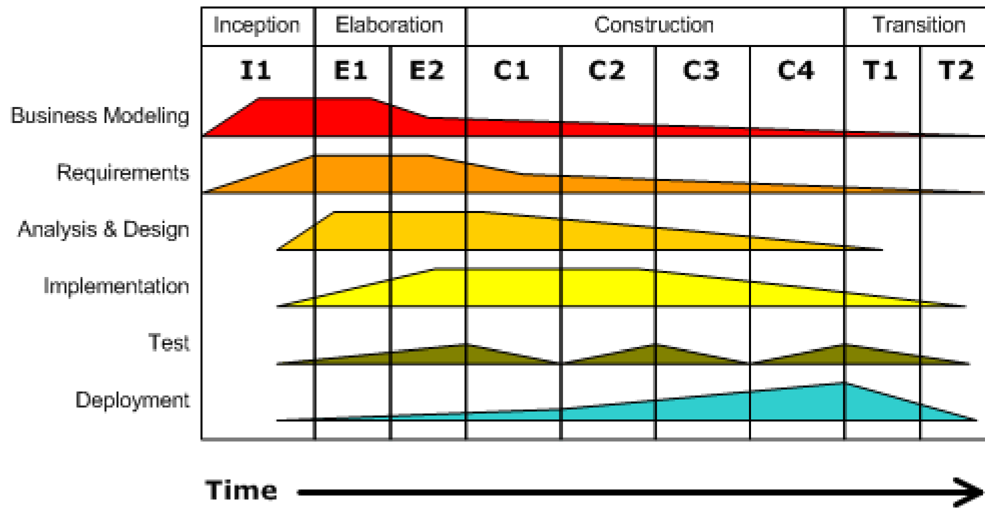
Web-based software process models and process engineering environments (Bolcer 1998, Grundy 1998, Penedo 2000, Scacchi and Noll 1997) that seek to provide software development workspaces and project support capabilities that are tied to adaptive process models.

Software process and business process reengineering (Scacchi and Mi 1997, Scacchi and Noll 1997, Scacchi 2000) which focuses attention to opportunities that emerge when the tools, techniques, and concepts for each disciplined are combined to their relative advantage.

This in turn is giving rise to new techniques for redesigning, situating, and optimizing software process models for specific organizational and system development settings (Scacchi and Noll 1997, Scacchi 2000). Understanding, capturing, and operationalizing process models that characterize the practices and patterns of globally distributed software development associated with open source software (DiBona 1999, Fogel 1999, Mockus 2000), as well as other emerging software development processes, such as extreme programming (Beck 1999) and Web-based virtual software development enterprises or workspaces (Noll and Scacchi 1999,2001, Penedo 2000).

Extreme Programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development,[1][2][3] it advocates frequent "releases" in short development cycles (timeboxing), which is intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.

The Rational Unified Process (RUP) is an iterative software development process framework created by the Rational Software Corporation, a division of IBM since 2003. RUP is not a single concrete prescriptive process, but rather an adaptable process framework, intended to be tailored by the development organizations and software project teams that will select the elements of the process that are appropriate for their needs.



Agile Unified Process (AUP) is a simplified version of the IBM Rational Unified Process (RUP) developed by Scott Ambler. It describes a simple, easy to understand approach to developing business application software using agile techniques and concepts yet still remaining true to the RUP. The AUP applies agile techniques including test driven development (TDD), Agile Modeling, agile change management, and database refactoring to improve productivity.

Analysis

Comparison of Methodologies (Post, & Anderson 2006)							
	SDLC	**RAD**	**Open Source**	**Objects**	**JAD**	**Prototyping**	**End User**
Control	Formal	MIS	Weak	Standards	Joint	User	User
Time Frame	Long	Short	Medium	Any	Medium	Short	Short
Users	Many	Few	Few	Varies	Few	One or Two	One
MIS staff	Many	Few	Hundreds	Split	Few	One or Two	None
Transaction/DSS	Transaction	Both	Both	Both	DSS	DSS	DSS
Interface	Minimal	Minimal	Weak	Windows	Crucial	Crucial	Crucial
Documentation	Vital	Limited	Internal	In	Limited	Weak	None

and training				Objects			
Integrity and security	Vital	Vital	Unknown	In Objects	Limited	Weak	Weak
Reusability	Limited	Some	Maybe	Vital	Limited	Weak	None

Conclusions

Contemporary models of software development must account for software the inter relationships between software products and production processes, as well as for the roles played by tools, people and their workplaces. Modeling these patterns can utilize features of traditional software life cycle models, as well as those of automatable software process models. Nonetheless, we must also recognize that the death of the traditional system life cycle model may be at hand. New models for software development enabled by the Internet, group facilitation and distant coordination within open source software communities, and shifting business imperatives in response to these conditions are giving rise to a new generation of software processes and process models. These new models provide a view of software development and evolution that is incremental, intra active, ongoing, interactive, and sensitive to social and organizational circumstances, while at the same time, increasingly amenable to automated support, facilitation, and collaboration over the distance of space and time.

References:-

1.SELECTING A DEVELOPMENT APPROACH. Revalidated: March 27, 2008. Retrieved 27 Oct 2008.

2.Gerhard Fischer, "The Software Technology of the 21st Century: From Software Reuse to Collaborative Software Design", 2001

3.SaaSSDLC.com — Software as a Service Systems Development Life Cycle Project

4.Morkel Theunissen, et.al.(2003). "Standards and Agile Software Development"

5.House of Representatives (1999). *Systems Development Life-Cycle Policy*. p.13.

6.Blanchard, B. S., & Fabrycky, W. J.(2006) *Systems engineering and analysis* (4th ed.) New Jersey: Prentice Hall. p.31

7.IEEE SDLC Standards

8.http://www.sereferences.com/white-papers.php

9.A Guide to the Project Management Body of Knowledge (PMBOK Guide) – 2000 edition. Project Management Institute, 2000.

10.http://en.wikipedia.org/wiki/Software_engineering

11.Why We Need a Theory for Software Engineering, Ivar Jacobson and Ian Spence, Dr. Dobbs Journal, October 02, 2009 (Retrieved March 26, 2010)