International Journal of Research in IT and Management (IJRIM)
Available online at: http://euroasiapub.org
Vol. 10 Issue 4, April- 2020
ISSN(o): 2231-4334 | ISSN(p): 2349-6517 | Impact Factor: 8.106

DESIGN AND IMPLEMENT THE DIFFERENT ROLES OF SOFTWARE ENGINEERING PROCESS MODELS TO OBTAIN A GENERALIZED MODEL

Parveen Gorya, Assistant Professor, Department of Computer Science, Government College, Narnaul, District Mahendergarh, Haryana.

Manoj Kumar, Assistant Professor, Department of Computer Science, Government College, Narnaul, District Mahendergarh, Haryana.

ABSTRACT

There is a lifecycle for software systems that includes genesis, development, productive operation, maintenance, and eventual retirement. This categorizes and explores a variety of ways for describing or modeling how software systems are built. It starts with history and definitions of standard software life cycle models that dominate most textbook discussions and contemporary software development techniques. This is followed by a more complete discussion of the different models of software evolution that are of current usage as the foundation for managing software engineering projects and technologies. Background Explicit models of software evolution stretch back to the first initiatives constructing major software systems in the Overall, the apparent objective of these early software life cycle.

Keywords: *software process modelling, requirements engineering, organization modelling.*

INTRODUCTION

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 article 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.

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 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 is possible since most such models are intuitive or well-reasoned. This means that many idiosyncratic details that describe how a software system 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, on the other hand, 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**.**

1. Guideline to organize, plan, staff, budget, schedule and manage software project work over organizational time, space, and computing environments.
2. Prescriptive outline for what documents to produce for delivery to client.
3. Basis for determining what software engineering tools and methodologies will be most appropriate to support different life cycle activities.
4. Framework for analyzing or estimating patterns of resource allocation and consumption during the software life cycle (Boehm 1981)
5. Basis for conducting empirical studies to determine what affects software productivity, cost, and overall quality.

Software Process Model

In contrast to software life cycle models, software process models often represent a networked sequence of activities, objects, transformations, and events that embody strategies for accomplishing software evolution. Such models can be used to develop more precise and formalized descriptions of software life cycle activities. Their power emerges from their utilization of a sufficiently rich notation, syntax, or semantics, often suitable for computational processing. Software process networks can be viewed as representing multiple interconnected task chains Task chains represent a non-linear sequence of actions that structure and transform available computational 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 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" while task chains have become popularized under the name of "workflow.

OBJECTIVES

1. The Study Software Evolution That Are of Current Use as The Basis for Organizing Software Engineering
2. The Study the Different Roles of Software Engineering.

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. The incremental release model is closely related to industrial practices where it most often occurs. Military models have also reified certain forms of the classic life cycle model into required practice for government contractors. Each of these four models uses coarse-grain or macroscopic characterizations when describing software evolution. The progressive steps of software evolution are often described as stages, such as requirements specification, preliminary design, and implementation; these usually have little or no further characterization other than a list of attributes that the product of such a stage should possess. Further, these models are independent of any organizational development setting, choice of programming language, software application domain, etc. In short, the traditional models are context-free rather than context-sensitive. But as all of these life cycle models have been in use for some time, we refer to them as the traditional models, and characterize each in turn.

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 Such models resemble finite state machine descriptions of software evolution. However, these models have been perhaps most useful in helping to structure, staff, and manage large software development projects in complex organizational settings, which was one of the primary purposes Alternatively, these classic models have been widely characterized as both poor descriptive and prescriptive models of how software development "in-the-small" or "in-the-large" can or should occur. provides a common view of the waterfall model for software development attributed to Royce.

Industrial and Military Standards, and Capability Models

Industrial firms often adopt some variation of the classic model as the basis for standardizing their software development practices Such standardization is often motivated by needs to simplify or eliminate complications that emerge during large software development or project management. From the 1970's through the present, many government contractors organized their software development activities according to succession of military software standards such now the standard that most such contractors now follow.

These standards are an outgrowth of the classic life cycle activities, together with the documents required by clients who procure either software systems or complex platforms with embedded software systems. Military software system are often constrained in ways not found in industrial or academic practice, required use of military standard computing equipment (which is often technologically dated and possesses limited processing are embedded in larger submarines, missiles, command and control systems) which are mission- components or products is a recent direction for government contractors, and thus represents new challenges for how to incorporate a component-based development into the overall software life cycle. Accordingly, new software life cycle models that exploit COTS components will continue to appear in the next few years.

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 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.

Prototyping technologies usually take some form of software functional specifications as their starting point or input, which in turn is simulated, analyzed, or directly executed. These technologies can allow developers to rapidly construct early or primitive versions of software systems that users can evaluate. User evaluations can then be incorporated as feedback to refine the emerging system specifications and designs. Further, depending on the prototyping technology, the complete working system can be developed through a continual revising/refining the input specifications. This has the advantage of always providing a working version of the emerging system, while redefining software design and testing activities to input specification refinement and execution. Alternatively, other prototyping approaches are best suited for developing throwaway or demonstration systems, or for building prototypes by reusing part/all of some existing software systems. Subsequently, it becomes clear why modern models of software development like the Spiral Model (described later) and the ISO 12207 expect that prototyping will be a common activity that facilitates the capture and refinement of software requirements, as well as overall software development.

Software Production Process Models

There are two kinds of software production process models: non-operational and operational. Both are software process models. The difference between the two primarily stems from the fact that the operational models can be viewed as computational scripts or programs: programs that implement a particular regimen of software engineering and development. Non-operational models on the other hand denote conceptual approaches that have not yet been sufficiently articulated in a form suitable for codification or automated processing.

Non-Operational Process Models

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 riskdriven approach to software process analysis and structuring 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. The radial dimension denotes cumulative development costs, and the angular dimension denotes progress made in accomplishing each development spiral. Risk analysis, which seeks to identify situations that might cause a development effort to fail or go over budget/schedule, occurs during each spiral cycle.

In each cycle, it represents roughly the same amount of angular displacement, while the displaced sweep volume denotes increasing levels of effort required for risk analysis. System development in this model therefore spirals out only so far as needed according to the risk that must be managed. Alternatively, the spiral model indicates that the classic software life cycle model need only be followed when risks are greatest, and after early system prototyping as a way of reducing these risks, albeit at increased cost. The insights that the Spiral Model offered has in turned influenced the standard software life cycle process models, such as noted earlier. Finally, efforts are now in progress to integrate computer-based support for stakeholder negotiations and capture of trade-off rationales into an operational form of the WinWin Spiral Model.

Operational Process Models

In contrast to the preceding non-operational process models, many models are now beginning to appear which codify software engineering processes in computational terms--as programs or executable models. Three classes of operational software process models can be identified and examined. Following this, we can also identify a number of emerging trends that exploit and extend the use of operational process models for software engineering. Operational specifications for rapid prototyping. The operational approach to software development assumes the existence of a formal specification language and processing environment that supports the evolutionary development of specifications into an prototype implementation Specifications in the language are coded, and when computationally evaluated, constitute a functional prototype of the specified system.

When such specifications can be developed and processed incrementally, the resulting system prototypes can be refined and evolved into functionally more complete systems. However, the emerging software systems are always operational in some form during their development. Variations within this approach represent either efforts where the prototype is the end sought, or where specified prototypes are kept operational but refined into a complete system. The specification language determines the power underlying operational specification technology. Simply stated, if the specification language is a conventional programming language, then nothing new in the way of software development is realized. However, if the specification incorporates (or extends to) syntactic and semantic language constructs that are specific to the application domain, which usually are not part of conventional programming languages, then domain-specific rapid prototyping can be supported.

An interesting twist worthy of note is that it is generally within the capabilities of many operational specification languages to specify "systems" whose purpose is to serve as a model of an arbitrary abstract process, such as a software process model. In this way, using a prototyping language and environment, one might be able to specify an abstract model of some software engineering processes as a system that produces and consumes certain types of documents, as well as the classes of development transformations applied to them. Thus, in this regard, it may be possible to construct operational software process models that can be executed or simulated using software prototyping technology.

Humphrey and Kellner describe one such application and give an example using the graphic-based state-machine notation provided in the STATECHARTS environment operational specifications, successively transforming and refining these specifications into an implemented system, assimilating maintenance requests by incorporating the new/enhanced specifications into the current development derivation, then replaying the revised development toward implementation has been limited to demonstrating such mechanisms and specifications on software coding, maintenance, project communication and management as well as to software component catalogs and formal models of software development Last, recent research has shown how to combine different life cycle, product, and production process models within a process-driven framework that integrates both conventional and knowledge-based software engineering tools and environments.

Software process automation and programming

Process automation and programming are concerned with developing formal specifications of how a system or family of software systems should be developed. Such specifications therefore provide an account for the organization and description of various software production task chains, how they interrelate, when then can iterate, etc., as well as what software tools to use to support different tasks, and how these tools should be used Focus then converges on characterizing the constructs incorporated into the language for specifying and programming software processes. Accordingly, discussion then turns to examine the appropriateness of language constructs for expressing rules for backward and forward-chaining, behavior, object type structures, process dynamism, constraints, goals, policies, modes of user interaction, plans, off-line activities, resource commitments, etc. across various This in turn implies that conventional mechanisms such as operating system shell scripts do not support the kinds of software process automation these constructs portend.

An Actor Dependency model

The basic features of the Actor Dependency (AD) model have been presented in an earlier and are briefly reviewed in The concepts are illustrated using a simple example of a software project organization. extends the basic model by distinguishing roles and positions from agents. Dependencies across role/position/agent relationships reflect the more elaborate and subtle aspects ofsoftware processes.

The basic model

An Actor Dependency model consists of a set of nodes and links. Each node represents an actor, and each link between two actors indicates that one actor depends on the other for something in order that the former may attain some goal. We call the depending actor the depender, and the actor who is depended upon the dependee. The object around which the dependency relationship centresis called the dependum. By depending on another actor for a dependum, an actor (the depender) is able to achieve goals that it was not able to do without the dependency, or not as easily or as well. At the same time, the depender becomes vulnerable. If the dependee fails to deliver the dependum, the depender would be adversely affected in its ability to achieve its goals. shows an Actor Dependency model for a hypothetical (and simplistic) software engineering project organization.

A customer depends on a project manager to have a system developed. The project manager in turn depends on a designer, a programmer, and a tester to do the technical work and be on schedule. Technical team members depend on each other for intermediate work products such as the design, code, and test results. The manager is also depended on by his boss for no project overrun, and by the quality assurance manager for the system to be maintainable. The user depends on the project manager for a user-friendly and high performance system. The Actor Dependency model distinguishes among three main types of dependencies, based on the ontological category of the dependum, namely, assertion, activity, or entity.

Models play a crucial role in managing the vast amount of data and allowing us to address specific groups of problems instead of individual issues. The quantity of observations is much reduced, and we deal with specific groups of issues rather than with millions of individual problems, thanks to the mapping of visible and unseen occurrences to ideas (in German: Begriffe). This allows us to amass data, which can then be used to make judgments and formulate plans for coping with the actual world. The capacity for reflection, which is regarded the distinction between man and animal, is closely tied to the usage of models. The specific strength of models is founded on the notion of abstraction: a model is typically not connected to one single item or phenomena alone, but to many, perhaps to a limitless number of them, it is related to a class. Those who take notice of the regularity with which high tides give way to low and low tides give way to high might either anticipate these changes or take advantage of them. Those that discover that a given class of creatures rather than one single living species is swift, powerful, and deadly, have enhanced their prospects for survival. While we live, i.e. act and respond, we utilize models all the time, typically unknowingly. In research and engineering, however, the situation is quite different; the development of models is discussed at length; it is central to the goals of both disciplines and a crucial stage in the production of tangible results. Research produces hypotheses. To further clarify, a theory is a subset of a model. The more prominent a theory is in the world, the higher it is estimated. The effects of this kind of influence may alter how we see the world in many ways.

CONCLUSION

The models of software development must account for software the interrelationships 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.

REFERENCES

1. Ambriola, V., R. Conradi and A. Fuggetta, Assessing process-centered software engineering environments, ACM Trans. Softw. Eng. Metho dol. 6, 3, 283-328, 1997.
2. Balzer, R., Transformational Implementation: An Example, IEEE Trans. Software Engineering, 7, 1, 3-14,1981.
3. Balzer, R., A 15 Year Perspective on Automatic Programming, IEEE Trans. Software Engineering, 11,11,1257-1267, 1985.
4. Balzer, R., T. Cheatham, and C. Green, Software Technology in the 1990's: Using a New Paradigm, Computer,16,11, 39-46, 1983.
5. Basili, V.R. and H.D. Rombach, The TAME Project: Towards Improvement-Oriented Software Environments, IEEE Trans. Soft. Engr., 14, 6, 759-773, 1988.
6. Basili, V. R., and A. J. Turner, Iterative Enhancement: A Practical Technique for Software Development, IEEE Trans. Software Engineering, 1,4, 390-396, 1975.
7. Batory, D., V. Singhal, J. Thomas, S. Dasari, B. Geraci, M. Sirkin, The GenVoca model of software-system generators, IEEE Software, 11(5), 89-94, September 1994.
8. Bauer, F. L., Programming as an Evolutionary Process, Proc. 2nd. Intern. Conf. Software Engineering, IEEE Computer Society, 223-234, January, 1976.
9. Beck, K. Extreme Programming Explained, Addison-Wesley, Palo Alto, CA, 1999.
10. Bendifallah, S., and W. Scacchi, Understanding Software Maintenance Work, IEEE Trans. Software Engineering, 13,3, 311-323, 1987.
11. Bendifallah, S. and W. Scacchi, Work Structures and Shifts: An Empirical Analysis of Software Specification Teamwork, Proc. 11th. Intern. Conf. Software Engineering, IEEE Computer Society, 260-270, 1989.
12. Biggerstaff, T., and A. Perlis (eds.), Special Issues on Software Reusability, IEEE Trans. Software Engineering, 10, ,5, 1984.
13. Boehm, B., Software Engineering, IEEE Trans. Computer, C-25,12,1226-1241, 1976.

14. Boehm, B. W., Software Engineering Economics, Prentice-Hall, Englewood Cliffs, N. J., 1981 20 Boehm, B., A Spiral Model of Software Development and Enhancement, Computer, 20(9), 61- 72, 1987.
15. Boehm, B., A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy, Using the WinWin Spiral Model: A Case Study, Computer, 31(7), 33-44, 1998.
16. Bolcer, G.A., R.N. Taylor, Advanced workflow management technologies, Software Process-- Improvement and Practice, 4,3, 125-171, 1998.