Software development lifecycle models

Article *in* ACM SIGSOFT Software Engineering Notes · May 2010

DOI: 10.1145/1764810.1764814 · Source: DBLP

Nayan Ruparelia
Hewlett Packard Enterprise
3 PUBLICATIONS **87** CITATIONS

Software Development Lifecycle Models

Nayan B. Ruparelia
Hewlett-Packard Enterprise Services
< nayan.ruparelia@hp.com >

DOI: 10.1145/1764810.1764814
http://doi.acm.org/10.1145/1764810.1764814

Abstract

This history column article provides a tour of the main software development life cycle (SDLC) models. (A lifecycle covers all the stages of software from its inception with requirements definition through to fielding and maintenance.) System development lifecycle models have drawn heavily on software and so the two terms can be used interchangeably in terms of SDLC, especially since software development in this respect encompasses software systems development. Because the merits of selecting and using an SDLC vary according to the environment in which software is developed as well as its application, I discuss three broad categories for consideration when analyzing the relative merits of SDLC models. I consider the waterfall model before the other models because it has had a profound effect on software development, and has additionally influenced many SDLC models prevalent today. Thereafter, I consider some of the mainstream models and finish with a discussion of what the future could hold for SDLC models.

Keywords: SDLC, SEN History Column, Waterfall, Spiral, Wheel-and-spoke, Unified, RAD, Incremental, B-model, V-model.

Introduction

SDLC is an acronym that is used to describe either software or systems development life-cycles. Although the concepts between the two are the same, one refers to the life-cycle of software whereas the other to that of a system that encompasses software development. In this article, software development is emphasized although the same principles can be transmuted to systems. Also, most of the innovation and thought leadership in terms of devising models and concepts has come from software development, and systems development has borrowed heavily from software development as a result.

SDLC, then, is a conceptual framework or process that considers the structure of the stages involved in the development of an application from its initial feasibility study through to its deployment in the field and maintenance. There are several models that describe various approaches to the SDLC process. An SDLC model is generally used to describe the steps that are followed within the life-cycle framework.

It is necessary to bear in mind that a model is different from a methodology in the sense that the former describes what to do whereas the latter, in addition, describes how to do it. So a model is descriptive whilst a methodology is prescriptive. As a result, we consider SDLC models in this article in terms of their relevance to particular types of software projects. This approach recognizes the context in which a SDLC model is used. For example, the waterfall model may be the best model to use when developing an enterprise relational database but it may not be the optimum model when developing a web-based application. I therefore consider the models for three distinct use cases, or software categories, in order to provide a context for their application, as follows:

Category 1. Software that provides back-end functionality. Typically, this is software that provides a service to other applications.

Category 2. Software that provides a service to an end-user or to an end-user application. Typically, this would be software that encapsulates business logic or formats data to make it more understandable to an end-user.

Category 3. Software that provides a visual interface to an end-user. Typically, this is a front-end application that is a graphical user interface (GUI).

SDLC models, also, may be categorized as falling under three broad groups: linear, iterative and a combination of linear and iterative models. A linear model is a sequential one where one stage, upon its completion, irrevocably leads to the initiation of the next stage. An iterative model, on the other hand, ensures that all the stages of the model shall be revisited in future; the idea being that development remains a constant endeavor for improvement throughout its lifecycle. A combined model recognizes that the iterative development process can be halted at a certain stage.

Although there is an abundance of SDLC models in existence, we shall consider the most important ones or those that have gained popularity. These include: the waterfall, spiral, unified, incrementing, rapid application development, the v and the w models.

Waterfall Model

The waterfall model, also known as the cascade model, was first documented by Benington [1] in 1956 and modified by Winston Royce [2] in 1970. It has underpinned all other models since it created a firm foundation for requirements to be defined and analyzed prior to any design or development.

The original cascade model of Bennington recommended that software be developed in stages: operational analysis → operational specification → design and coding specifications → development → testing → deployment → evaluation. By recognizing that there could be unforeseen design difficulties when a baseline is created at the end of each stage, Royce enhanced this model by providing a feedback loop so that each preceding stage could be revisited. This iteration is shown in Figure 1 using red arrows for the process flow; this allowed for the stages to overlap in addition to the preceding stage to be revisited. But Royce also felt that this arrangement could prove inadequate since the iteration may need to transcend the succeeding-preceding stage pair's iteration. This would be the case at the end of the evaluation (or testing) stage when you may need to iterate to the design stage bypassing the development stage or at the end of the design stage where you may need to revisit the requirements definition stage bypassing the analysis stage. That way, the design and the requirements, respectively, are redefined should the testing or the design warrant. This is illustrated in Figure 1 by the dashed arrows that show a more complex feedback loop. Further, Royce suggested that a preliminary design stage could be inserted between the requirements and analysis stages. He felt that this would address the central role that the design phase plays in minimizing risks by being re-visited several times as the complex feedback loop of Figure 1 shows.

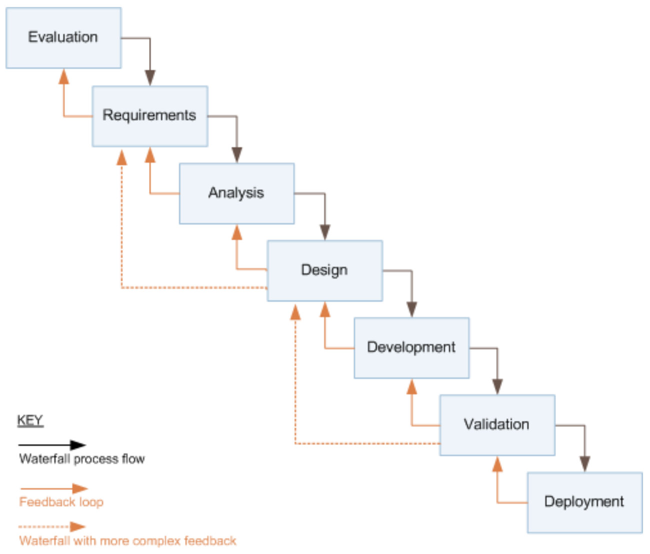

Figure 1: Waterfall model with Royce's iterative feedback.

When referring to the waterfall model in this article, I shall mean Royce's modified version of Benington's cascade model.

One aspect of the waterfall model is its requirement for documentation. Royce suggested that at least six distinct types of document be produced:

1) Requirements document during the requirements stage.
2) Preliminary design specification during the preliminary design stage.
3) Interface design specification during the design stage.
4) Final design specification that is actively revised and updated over each visit of the design stage; this is further updated during the development and validation stages.
5) Test plan during the design stage; this is later updated with test results during the validation or testing stage.
6) Operations manual or instructions during the deployment stage.

Quality assurance is built into the waterfall model by splitting each stage in two parts: one part performs the work as the stage's name suggests and the other validates or verifies it. For instance, the design stage incorporates verification (to assess whether it is fit for purpose), the development stage has unit and integration testing, and the validation stage contains system testing as its part and parcel.

The waterfall model is the most efficient way for creating software in category 1 discussed earlier. Examples for this would be relational databases, compilers or secure operating systems.

B-Model

In 1988, Birrell and Ould discussed an extension to the waterfall model that they called the b-model [3]. By extending the operational life-cycle (denoted as the maintenance cycle) and then attaching it to the waterfall model, they devised the b-model as shown in Figure 2. This was done to ensure that constant improvement of the software or system would become part of the development stages. Also, they felt that an alternative to obsolescence needed to be captured so that enhanced or even newer systems could be developed as spin-offs from the initial system.

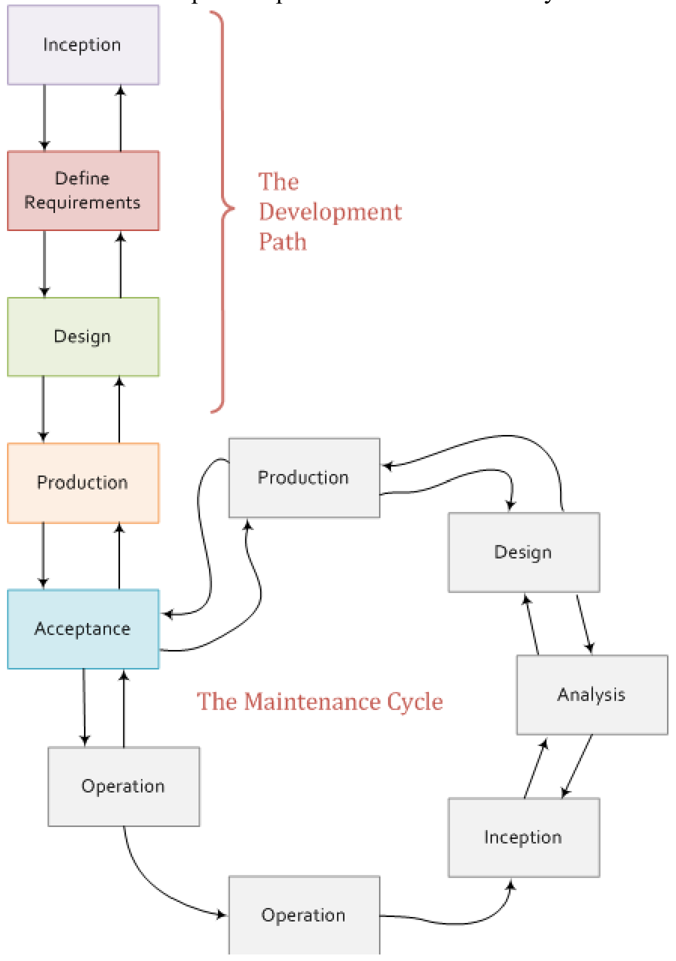

Figure 2: The b-model extends the waterfall model.

In a sense, the b-model was an attempt to modify the waterfall by creating an evolutionary enhancement process that was captured by the spiral model that we shall discuss later. The b-model, however, is more suitable to the development of category 1 software like its cousin, the waterfall.

Incremental Model

The incremental model, also known as the iterative waterfall model, can be viewed as a three dimensional representation of the waterfall model. The z-axis contains a series of waterfall models to represent the number of iterations that would be made in order to improve the functionality of the end product incrementally. The incremental model, in this sense, is a modification to the waterfall that approaches the spiral model.

The main strengths of this model are:

i. Feedback from earlier iterations can be incorporated in the current iteration.
ii. Stakeholders can be involved through the iterations, and so helps to identify architectural risks earlier.
iii. Facilitates delivery of the product with early, incremental releases that evolve to a complete feature-set with each iteration.
iv. Incremental implementation enables changes to be monitored, and issues to be isolated and resolved to mitigate risks.

V-Model

The v-model, also known as the vee model, first presented at the 1991 NCOSE symposium in Chattanooga, Tennessee [4], was developed by NASA. The model is a variation of the waterfall model in a V shape folded in half at the lowest level of decomposition, as Figure 3 shows. The left leg of the V shape represents the evolution of user requirements into ever smaller components through the process of decomposition and definition; the right leg represents the integration and verification of the system components into successive levels of implementation and assembly. The vertical axis depicts the level of decomposition from the system level, at the top, to the lowest level of detail at component level at the bottom. Thus, the more complex a system is the deeper the V shape with correspondingly larger number of stages. The v-model, being three-dimensional, also has an axis that is normal to the plane, the z-axis. This represents components associated with multiple deliveries. Time is therefore represented by two axes: from left to right and into the plane.

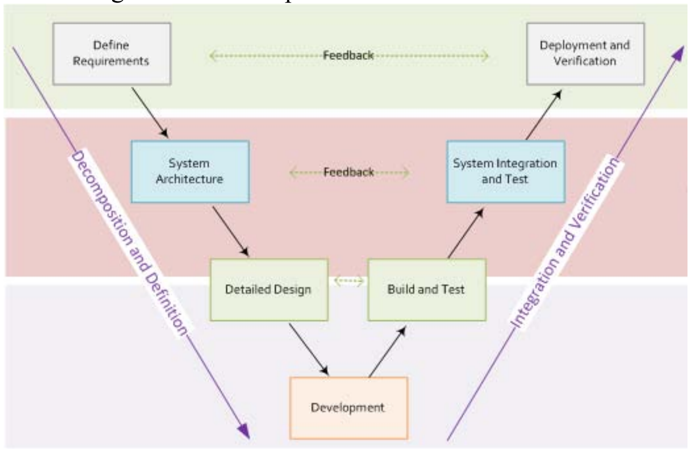
Define Requirements
Feedback
Deployment and Verification
System Architecture
Feedback
System Integration and Test
Detailed Design
Build and Test
Development
Decomposition and Definition
Integration and Verification

Figure 3: V-model's built-in quality assurance.

The v-model is symmetrical across the two legs so that the verification and quality assurance procedures are defined for the right leg during the corresponding stage of the left leg. This ensures that the requirements and the design are verifiable in a SMART (Specific, Measurable, Achievable, Realistic and Time-bound) [5] manner, thus avoiding statements such as "*user friendly,"* which is a valid but non-verifiable requirement.

A variation of the v-model is the vee+ model. This adds user involvement, risks and opportunities to the z-axis of the v-model. Thus, users remain involved until the decompositions are of no interest to them. An application of this is that any anomalies identified during the integration and assembly stages are fed to the user for acceptance or, in the case of rejection, to be resolved at the appropriate levels of decomposition; the anomalies could then be resolved as errors or accepted as design features. In contrast, the spiral model adds user involvement during its risk reduction activities whereas the v and the waterfall models incorporate it during the initial requirements definition phase. Additionally, the risks and opportunities feature of the vee+ mean that certain stages of the v at the lower decomposition levels could be truncated in order to integrate COTS (commercial, off-the-shelf software) packages [6]. This means that varying levels of products, at different levels of decomposition, can be added to the life cycle in a seamless manner.

Another variation of the v-model is the vee++ model [6] that adds intersecting processes to the vee+ model; a decomposition analysis and resolution process is added to the left leg of the v shape, and a verification analysis and decomposition process is added to the right one.

Like the waterfall, the v-model is more suited to category 1 systems; the creators modified the model to the vee+ for categories 1 and 3, and thereafter created the vee++ model for all categories. However, a key strength of the v-model is that it can be used in very large projects where a number of different contractors, sub-contractors and teams are involved; decomposition, integration and verification at each stage with all the parties involved is made possible by the v-model prior to progressing to the next stage. This factor is acknowledged by the time parameter being represented by the y and z axes of the model. Thus, a large number of parties and stakeholders in a very large project become inherently integrated using a requirements-driven approach.

Spiral Model

Boehm modified the waterfall model in 1986 [7] by introducing several iterations that spiral out from small beginnings. An oft- quoted idiom describing the philosophy underlying the spiral method is *start small, think big*.

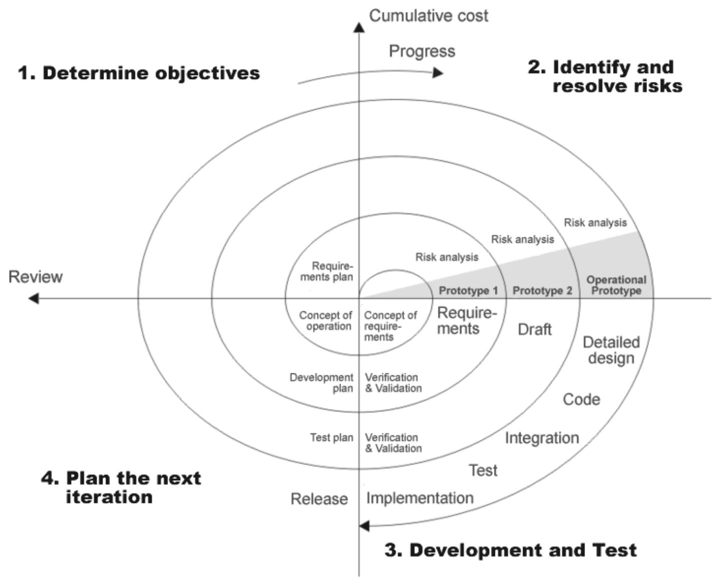

Figure 4: Boehm's spiral life-cycle.

The spiral model attempts to address two main difficulties of the waterfall model:

i. the two design stages can be unnecessary in some situations,
ii. the top-down approach needs to be tempered with a look-ahead step in order to account for software reusability or the identification of risks and issues at an early stage.

The spiral model represents a paradigm shift from the waterfall's specification driven approach to a risk-driven one. Each cycle traverses four quadrants, as illustrated by Figure 4. These are:

i. Determine objectives.
ii. Evaluate alternatives, and identify and resolve risks.
iii. Develop and test.
iv. Plan the next iteration.

As each cycle within the spiral progresses, a prototype is built, verified against its requirements and validated through testing. Prior to this, risks are identified and analyzed in order to manage them. The risks are then categorized as performance (or user-interface) related risks or development risks. If development risks predominate, then the next step follows the incremental waterfall approach. On the other hand, if performance risks predominate, then the next step is to follow the spiral through to the next evolutionary development. This ensures that the prototype produced in the second quadrant has minimal risks associated with it.

Risk management is used to determine the amount of time and effort to be expended for all activities during the cycle, such as planning, project management, configuration management, quality assurance, formal verification and testing. Hence, risk management is used as a tool to control the costs of each cycle.

An important feature of the spiral model is the review that takes place as part of the y-axis of Figure 4. This process occurs at the completion of each cycle and covers all of the products and artifacts produced during the previous cycle including the plans for the next cycle as well as the resources required for it. A primary objective of the review process is to ensure that stakeholders are committed to the approach to be taken during the next cycle.

A set of fundamental questions arise about the spiral model:

i. How does the spiral get started?
ii. How and when do you get off the spiral?
iii. How does system enhancement or maintenance occur?

Boehm's response to these questions was to use a complementary model that he called the Mission Opportunity Model (MOM) [7]. Using the MOM as a central pivot, the operational mission is assessed at key junctures as to whether further development or effort should be expended. This is done by testing the spiral against a set of hypotheses (requirements) at any given time; failure to meet those hypotheses leads to the spiral being terminated. To initiate the spiral, however, the same process is followed using the MOM.

In 1987, Iivari suggested a modification to the spiral model in the January issue of this publication [8] that he called the hierarchical spiral model. He proposed to sub-divide each quadrant of the spiral into two parts to allow for sub-phases. With this approach, he provisioned for baselines and milestones to be created for each cycle in order to have greater control for that cycle. Also, he suggested that the main phases be risk-driven, as with the spiral, but the sub-phases should be specification or process driven as in the case of the waterfall. This combination of the risk-specification paradigms would make the hierarchical spiral model cost conscious (like the spiral) as well as disciplined (like the waterfall).

A key benefit of the spiral model is that it attempts to contain project risks and costs at the outset. Its key application is for the category 1 use case. However, sufficient control of the cycles could make it adoptable for the other categories as well. The main difficulty of the spiral is that it requires very adaptive project management, quite flexible contract mechanisms between the stakeholders and between each cycle of the spiral, and it relies heavily on the systems designers' ability to identify risks correctly for the forthcoming cycle.

Wheel-and-spoke Model

The wheel-and-spoke model is based on the spiral model and is designed to work with smaller teams initially, which then scale up to build value faster. It is a bottom-up approach that retains most of the elements of the spiral model by comprising of multiple iterations.

First, system requirements are established and a preliminary design is created. Thereafter, a prototype is created during the implementation stage. This is then verified against the requirements to form the first spoke of the model. Feedback is propagated back to the development cycle and the next stage adds value to create a more refined prototype. This is evaluated against the requirements and feedback is propagated back to the cycle; this forms the second spoke. Each successive stage goes through a similar verification process to form a spoke. The wheel represents the iteration of the development cycle.

The wheel-and-spoke model has many uses. It can be used to create a set of programs that are related by a common API. The common API then forms the center of the model with each program verifying its conformance to the API (this in essence forms a common set of requirements) at each spoke. Another application of this model is with the Architecture Development Method (ADM) of The Open Group Architecture Framework (TOGAF) [9] where the spokes are used to validate the requirements during the development of the architecture, as Figure 5 shows.

Like the spiral, it is more amenable to category 1 and 2 use cases.

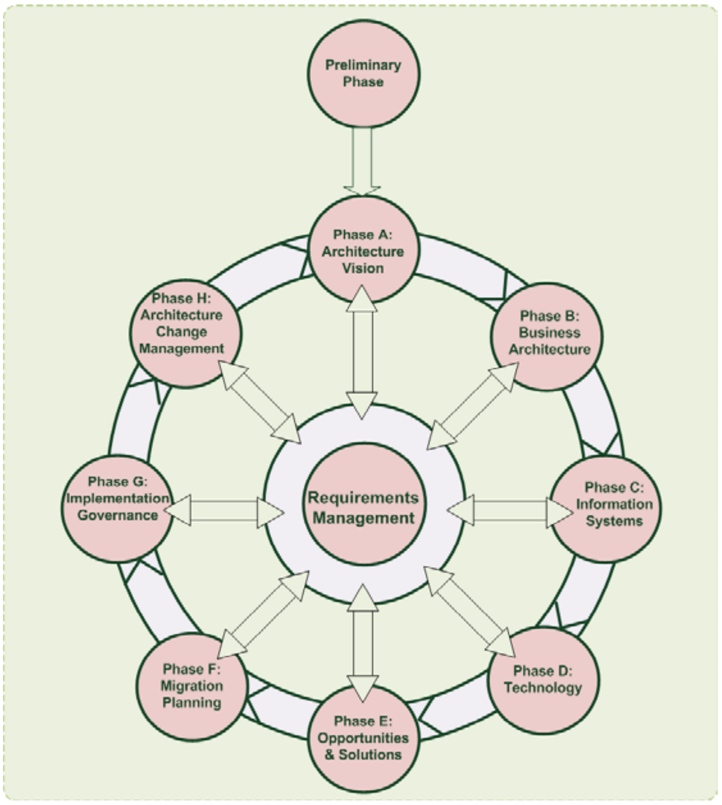
Preliminary Phase
Phase A: Architecture Vision
Phase B: Business Architecture
Phase C: Information Systems
Phase D: Technology
Phase E: Opportunities & Solutions
Phase F: Migration Planning
Phase G: Implementation Governance
Phase H: Architecture Change Management
Requirements Management

Figure 5: TOGAF's ADM as a wheel-and-spoke model.

Unified Process Model

Whereas the waterfall is specification driven and the spiral is risk driven, the unified process model is model (or architecture) based and use case driven [10]; at the same time, it is iterative in nature.

The unified model was created to address the specific development requirements of object-oriented software and its design. A descendent of the Objectory model, it was developed in the 1990s by Rational Software; it is therefore commonly known as the Rational Unified Process (RUP) model. IBM acquired Rational Software in 2003 and continues to develop and market the process as part of various software development toolsets.

RUP encapsulates seven best practices within the model:

i. Develop iteratively using risk management to drive the iterations.
ii. Manage requirements.
iii. Employ a component-based architecture.
iv. Use visual models.
v. Verify quality continuously.
vi. Control changes.
vii. Use customization.

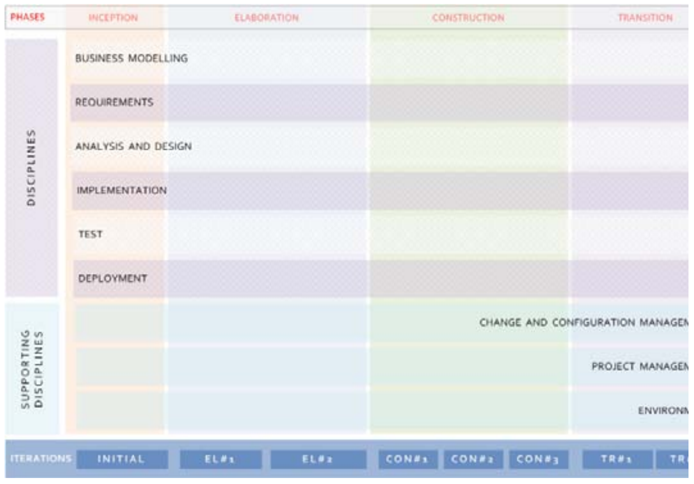

Figure 6: Unified process model.

RUP uses models extensively and these are described using the Unified Modeling Language (UML), which comprises of a collection of semi-formal graphical notations and has become the standard tool for object-oriented modeling. It facilitates the construction of several views of a software system, and supports both static and dynamic modeling. The notations include use case, activity, class, object, interaction and state diagrams amongst others. Use cases are central to RUP because they lay the foundation for subsequent development. They provide a functional view by modeling the way users interact with the system. Each use case describes a single use of the system, and provides a set of high-level requirements for it. Moreover, use cases drive the design, implementation and testing (indeed, the entire development process) of the software system.

RUP is iterative and incremental as it repeats over a series of iterations that make up the life cycle of a system. An iteration occurs during a phase and consists of one pass through the requirements, analysis, design, implementation and test workflows (based on disciplines), building a series of UML models that are inter-related.

As Figure 6 shows, RUP has four phases: the inception, elaboration, construction and transition phases. Each phase may comprise of any number of iterations by engaging with six core disciplines (shown horizontally in the figure). These disciplines are supported by three other disciplines: change and configuration management, project management, and environment.

Although more suitable for the category 2 and 3 applications, RUP could be adapted for mid-scale category 1 applications provided that risk and contractual links are well managed. However, for very large system development, RUP would not be the ideal model to use because of its model-based and use-case driven approach. The iterative waterfall, the V-model (including its cousins, vee+ and vee++), and the spiral models would be more suitable. At the other extreme, for very small projects, RUP could be used provided that the number and type of artifacts are optimized to the minimum required for rapid development.

Rapid Application Development

Principally developed by James Martin in 1991, Rapid Application Development (RAD) is a methodology that uses prototyping as a mechanism, as per Figure 7, for iterative development.

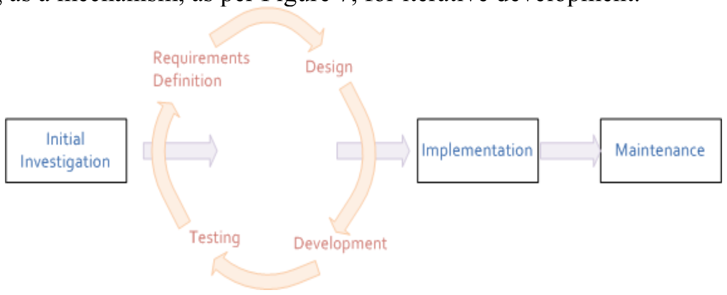

Figure 7: Prototyping approach used by RAD.

RAD promotes a collaborative atmosphere where business stakeholders participate actively in prototyping, creating test cases and performing unit tests. Decision making is devolved away from a centralized structure (usually comprising of the project manager and developers) to the functional team.

The open source software development model (also known as the Cathedral and the Bazaar model that was first documented by Raymond [11]), espousing a '*release early; release often; listen to your customers*' philosophy, is quite similar to RAD and some of its spin-off methodologies such as Agile.

Recently, RAD has come to be used in a broader, generic sense that encompasses a variety of techniques aimed at speeding software development. Of these, five prominent approaches are discussed below in alphabetical order.

Agile

Scope changes, as well as feature creep, are avoided by breaking a project into smaller sub-projects. Development occurs in short intervals and software releases are made to capture small incremental changes.

Applying Agile to large projects can be problematic because it emphasizes real-time communication, preferably on a personal, face-to-face basis. Also, Agile methods produce little documentation during development (requiring a significant amount of post-project documentation) whilst de-emphasizing a formal process-driven steps. Thus, it is more suited to a Category 3 application.

Extreme Programming (XP)

With XP, development takes place incrementally and on the fly with a business champion acting as a conduit for user-driven requirements and functionality; there is not an initial design stage. (This could lead to higher costs later when new, incompatible, requirements surface.) In order to lower costs, new requirements are accounted for in short, fast spiral steps and development takes place with programmers working in pairs.

Joint Application Development (JAD)

JAD advocates collaborative development with the end user or customer by involving him during the design and development phases through workshops (known as JAD sessions). This has the possibility of creating scope creep in a project if the customer's requirements are not managed well.

Lean Development (LD)

In pursuance to the paradigm *'80% today is better than 100% tomorrow'*, Lean Development attempts to deliver a project early with a minimal functionality.

Scrum

Development takes place over a series of short iterations, or sprints, and progress is measured daily. Scrum is more suited to small projects because it places less emphasis on a process driven framework (needed on large projects) and also the central command structure can cause power struggles in large teams.

RAD, as well as its affiliated methodologies described above, is generally much less suited to Category 1 and 2 applications and much more suited to Category 3 applications because of its emphasis on less formal, process independent approaches.

The Future

This history column invariably ends with a consideration of the future; history's importance is in creating a framework for the future. Although in the past software development has influenced systems design in an inordinate way, there should be a fusion of knowledge sharing between the two in both directions in future. Moreover, SDLC models for both disciplines can draw from domains outside their technological boundaries. A large amount of work has been done in other spheres such as behavior analysis, time management, and business management (the Japanese *Kanban* or *Just-In-Time* model, for example) that could be borrowed by future SDLC models. To some extent, this is already happening with some models. For example, the *Behavior Driven Development* technique (facilitates collaboration between various participants) that is part of the Agile model has drawn from behavior analysis. This sharing of ideas from disparate sources to incorporate them in an SDLC model should gather momentum in future.

Since there is a proliferation of SDLC models, it would be good to have a central repository, perhaps hosted on a wiki, where various experiences could be shared. This will fulfill two purposes.

First, the insatiable desire to compare models will be addressed. However, this should be done in the context of the application of the SDLC, much like the three categories proffered in this article. This is important because SDLC models cannot be compared easily without considering their particular category of application.

Second, a repository containing lessons learned will enable key parameters to be distilled from the model and a statistical analysis to be performed on their use and efficacy in various categories of projects. Also, parameters for comparison could include non-functional attributes such as risks and costs at each stage. This will provide us with a better foundation with which to make decisions since the lessons learned analysis will be something that comes from experience on the ground rather than just theory.

Perhaps the ACM could host such a lessons learned wiki and a back-end statistical analysis engine. This can be applied beyond SDLCs and encompass a wide range of techniques and technologies.

References

[1] Benington, H.D. (1956): Production of large computer programs. In *Proceedings, ONR Symposium on Advanced Programming Methods for Digital Computers, June 1956*, pp 15-27.

[2] Royce, Winston W. (1970): Managing the development of large software systems. In *Proceedings, IEEE Wescon, August 1970*, pp 1-9.

[3] Birrell, N. D. and Ould, M.A. (1988): A practical handbook to software development; Cambridge University Press. ISBN 978-0521347921. pp 3-12.

[4] Forsberg, Kevin and Mooz, Harold (1991): The relationship of system engineering to the project cycle. At *NCOSE, Chattanooga, Tennessee, October 21-23, 1991*.

[5] Doran, George T. (1981): There's a S.M.A.R.T. way to write management's goals and objectives. In *Management Review*, vol. 70.11.

[6] Mooz, H and Forsberg, K. (2001): A visual explanation of the development methods and strategies including the waterfall, spiral, vee, vee+, and vee++ models, pp 4-6.

[7] Boehm, Barry W. (1986): A spiral model of software development and enhancement. In *ACM SigSoft Software Engineering Notes, Vol. II, No. 4, 1986*, pp 22-42.

[8] Iivari, Juhani (1987): A hierarchical spiral model for the software process: notes on Boehm's spiral model. In *ACM SIGSOFT Software Engineering Notes, vol. 12, no. 1, January 1, 1987*. Pp 35-37.

[9] TOGAF 9:The Open Group; http://www.opengroup.org/togaf/.

[10] Jacobson I., Booch G. & Rumbaugh J. (1999): The unified software development process; Addison-Wesley, Reading, Massachusetts.

[11] Raymond, Eric (2001): Cathedral and the Bazaar, 1st Edition; O'Reilly Media; ISBN: 978-0596001087.