

A CIRCULAR MODEL FOR SOFTWARE DEVELOPMENT

Marvin J. Carr

McDonnell Douglas Electronic Systems Company
5301 Bolsa Ave.
Huntington Beach, California 92647

INTRODUCTION

Software development in the 50's and 60's consisted of designing to some level of detail sufficient to see how most of the parts fit together, coding and designing at the same time, and then fixing the code to make the system operate to the user's satisfaction. Software developed in this manner was very difficult to maintain and almost impossible to enhance. This led to a successive stagewise model of software development in which the current stage or phase was completed before continuing on to the next stage. In 1970 the Waterfall model [1], Figure 1, was presented which incorporated feedback loops between the stages, thus ensuring that problems were resolved at the correct level. As the computer industry matured and hardware became smaller, faster, and cheaper, computer systems have grown larger and more complex. As a result, the methodology used to produce software has correspondingly grown in complexity. It's no longer a "one shot" process where the project follows the strict design, code, test, and delivery method depicted in the Waterfall model. In a growing number of systems, and certainly in sophisticated weapon systems, it's necessary to build and operate a portion of

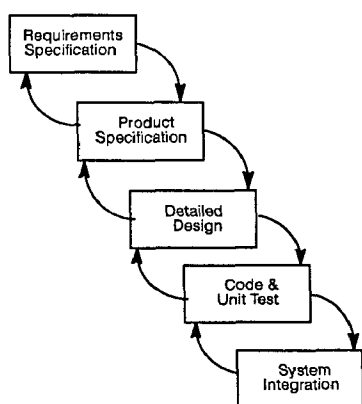


Figure 1. Waterfall Model

the system before the requirements for the entire system can be thoroughly understood. A clear statement of the dilemma that faces all software engineers was made by David Parnas and Paul Clements [2] when they commented that; "People who commission a system don't know exactly what they want." Very often the system specification, in whatever form it takes, contains statements of requirements that are not completely understood, that are ambiguous, and that are subject to change. This "fuzz" factor is the most difficult problem for software engineers. The elimination of the fuzz factor, in most cases, does not occur until later in the life cycle when the customer finally has some first hand operational experience with the subject system. Even then the customer, through experience, will require additional features and/or existing features changed.

Fred Brooks [3] very succinctly described the problem this way;

"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, machines, and other software systems. No other part of the work so cripples the resulting system if done wrong."

Existing software life-cycle models fail to deal adequately with all the problems intrinsic in the development of large, complex, software intensive systems. The Spiral model [4], Figure 2, addresses the limitations of other development models, such as the Waterfall model, in the areas of iterative development and prototyping, however, the spiral model addresses internal development where requirements evolve rather than contract acquisition of software where requirements are specified. The Spiral model also does not address the type of incremental development espoused by Ada design methodologists [5][6]. The Ada development methodology most commonly expressed is one that specifies the requirements, designs an architecture and implements the design at the highest level and then fills in lower levels of abstraction as the system is decomposed. Another aspect that is not thoroughly accounted for is the process of ensuring that requirements are understood prior to implementation.

The software development paradigm presented here proposes a software life cycle model that not only addresses resolving requirements, prototyping and Ada incremental development, but also provides a method to

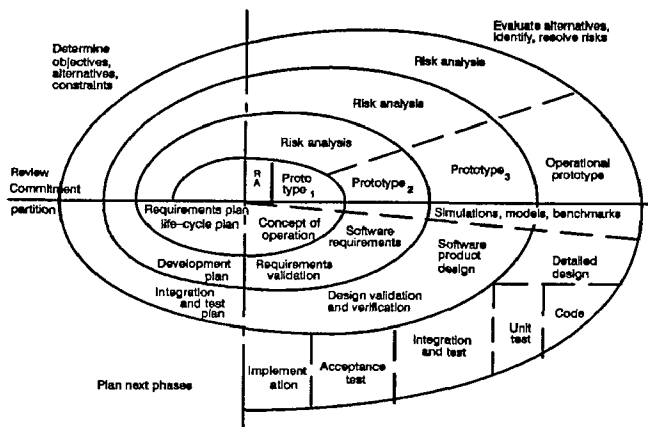


Figure 2. Spiral Model

handle the total software life cycle in a continuum without discontinuities that are inherent in other models.

THE CIRCULAR MODEL

The Circular Model, Figure 3, was developed out of

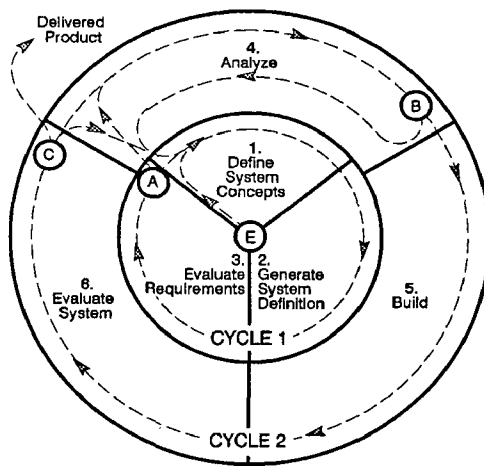


Figure 3. Circular Model

the need for a process that could be used not only for current methodologies, but also for methodologies that are emerging in Object Oriented Development (OOD) and in particular, Ada development. As will be addressed later in this paper, the circular model can accommodate previous models as a matter of course, thus allowing the appropriate combination of models for each software situation.

The circular model consists of two interdependent cycles (see Figure 3). The entry point into the circular model is in the center at point E. Cycle 1 defines the system and plans the overall development. Requirements are analyzed and planning is done to lay out the development activities such that the system is built in the

most efficient manner. Progressing from one stage to another involves a review of the products of the current stage. These reviews involve the customer to ensure that the system is meeting the specified requirements. Cycle 1 consists of three stages (1-3):

1. Define Concepts
 - Feasibility studies
 - Requirements analysis
 - Risk analysis
 - User's interface prototype
2. Generate Definition
 - Requirements analysis
 - Functional analysis
 - Functional prototype
 - Risk analysis
3. Evaluate Requirements
 - Requirements validation
 - Functional prototype
 - Risk analysis

Cycle 2 is concerned with implementing the system defined in cycle 1. Cycle 2 consists of three stages (4-6):

4. Analyze
 - Preliminary design/review
 - Detailed design/review
 - Risk analysis
5. Build (Implementation)
 - Code and unit test
 - Subsystem integration
 - System integration
6. Evaluate System
 - Independent test
 - Acceptance test
 - Operational demonstration

There are three major decision points in the circular model; A, B, and C. Decision point A controls the transition of the process into cycle 2. If the results of evaluating the requirements (stage 3) indicates that the proposed solution generated during stage 2 does not satisfy the customer's requirements then cycle 1 is repeated. This is an extremely important point. The traditional approach is to rush into the design phase before the requirements are thoroughly understood. Many systems have faltered and not met expectations just because of "fuzz" in the requirements. Cycle 1 is repeated as many times as necessary to adequately define and clarify requirements.

Decision point B provides the mechanism to resolve anomalies between requirements and design before coding commences. (In Ada, implementing specifications may be considered part of the design, implementing bodies may be considered coding, depending on the circumstances.) The resolution of requirement-design anomalies are fed into cycle 1 to perform a total system impact analysis. This saves expensive rework of code during integration to account for adverse affects on other parts of the system.

Decision point C, crossing the boundary of cycle 2 to denote completion of the system, brings an orderly end to the life cycle model. The boundary point crossing occurs when the criterion for satisfactory performance of the system is met. Decision point C also provides for re-assessment of the system if the evaluation conducted during stage 6 shows the product did not meet all the system requirements. In the case of incremental delivery, decision point C also provides the mechanism for delivering the completed increment to the customer as well as feeding the completed increment back into cycle 1 for assessment and planning for the next incremental delivery.

CYCLE 1

Cycle 1, Figure 4, is concerned with definition of the system. During the Define Concepts stage (stage 1) the objectives of the system are elaborated through the analysis of the customer supplied system specification. (This customer supplied specification forms the basis for the construction of the system.) A document that describes the operation of the system is written detailing the system's objectives and how the objectives are envisioned to be accomplished. Included in the concepts of operation document are detailed descriptions of the user's interfaces. Stage 1 has a high degree of customer interaction to ensure that the customers needs are fully understood. Prototyping the customer interface is a great aid in stage 1 to help understand and develop the operational aspects of the system. When the operation of the system has been defined and approved by the customer, the system advances to Stage 2.

Stage 2, Generate System Definition, is concerned with producing the top level specifications for the software. A top level system design is formulated by decomposing the total software system into software subsystems (in 2167A terminology the subsystems could be Computer Software Configuration Items (CSCIs).) Frequently this process will identify areas of uncertainty that are significant sources of project risk. If so, the next stage, Evaluate Requirements, formulates a cost effective strategy for resolving the sources of risk. This may involve prototyping, simulation, analytic modeling, or

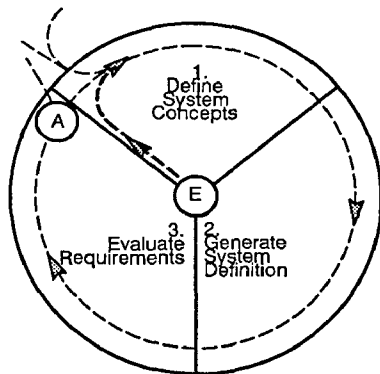


Figure 4. Cycle 1

combinations of these and other risk management techniques. Cycle 1 is then repeated taking into account and analyzing the risk reduction solution's impact on the overall system specification. Cycle 1 is repeated until all aspects of the system are satisfactorily addressed.

The specification of software requirements is one of the products of cycle 1. This specification identifies the major subsystems and their time phased implementation plan. As most often is the case, there are subsystems which are dependent upon capabilities contained in other subsystems being operational before verification of the dependent subsystem can take place. Software development methodologies that are currently being used do not directly allow for this timed phasing of development to take place in an orderly manner. The circular life cycle model provides for time phased development to take place at decision point A. At decision point A, the portion of the software that has been designated for implementation is put into cycle 2 while the rest of the software is held in cycle 1 awaiting the results of implementation of dependent capabilities.

CYCLE 2

The boundary crossing from cycle 1 to cycle 2, decision point A, occurs when the customer and contractor make the decision that the system is adequately understood and specified. (Adequately means that the abstraction of the system that is being transitioned into cycle 2 can be implemented in its entirety.)

Cycle 2, Figure 5, is where the selected proposal for the system is put into action. The Analyze stage consists of designing the software to a detail where coding can easily take place. This is accomplished by performing the design and validating the design against the specifications detailed during cycle 1. Normally this consists of detailing the design in a Program Design Language (PDL) such as Ada PDL.

The Analyze stage ends when the software design satisfactorily passes the design review (decision point B). The decision to enter the Build stage is based upon a complete design that contains no open items, i.e., no

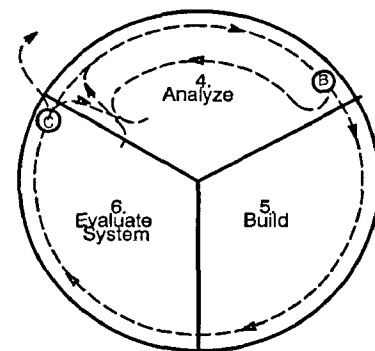


Figure 5. Cycle 2

infamous TBDs (To Be Determined). The Preliminary Design Review (PDR) and the Critical Design Review (CDR) are held in the Analyze stage.

It is possible that at any point during the Analysis stage an anomaly in the design process can occur that has to be corrected by modification of the system specifications before further action is taken. If this is the case then cycle 1 is re-entered (decision point B) to modify the specification and ensure that the modification does not have an adverse affect on the rest of the system. The point stressed here is that cycle 1 must be re-entered to ensure that a complete analysis of the impact that the anomaly has on the rest of the system is accomplished before implementation takes place.

If problems occur during the build stage that reflect back into the requirements or design, the design was not complete and implementation should never have been started. At decision point B all risk factors must be resolved. Risks are resolved through prototyping the risk areas uncovered during design and feeding the results back into the design activities.

The Build stage is entered when the design of the current abstraction has been accepted by the customer or the customer's representative. In this stage, the design is coded, unit tested and integrated to form the product that is to be delivered. It must be re-emphasized here that the build stage should never be entered with existing design/requirements problems.

The Evaluate System stage consists of integrating the product with the rest of the system, independent testing and formal acceptance testing to demonstrate the system satisfies the requirements specification. At the end of the Evaluate System stage the software is delivered to the customer and the project is completed.

The possibility exists that the system will not pass evaluation. In this case it must be determined if further design needs to be done or if further requirements analysis needs to be done. The Analyze stage is entered for further design resolution; cycle 1 is re-entered for requirements analysis.

Two or more software components may be developed concurrently as long as the components are independent and do not rely on another part being complete in order to function properly. Starting cycle 2 of a dependent part of the system before the dependency has been completed poses a risk to the project. This risk has to be evaluated before action is taken.

RECURSIVE/PARALLEL DEVELOPMENT

Recursive development of software, such as that used in Ada design methodologies, is the time-phased development of capabilities such that each succeeding capability either uses part or all of capabilities already implemented; parallel development is concurrent development of software components that have no dependencies on each other. To accomplish this type of software development the circular model is used in a recursive manner (see Figure 6). The first time through cycle 1 defines the overall system and plans the time-ordered implementation of capabilities. The first capability or capabilities to be developed are transition

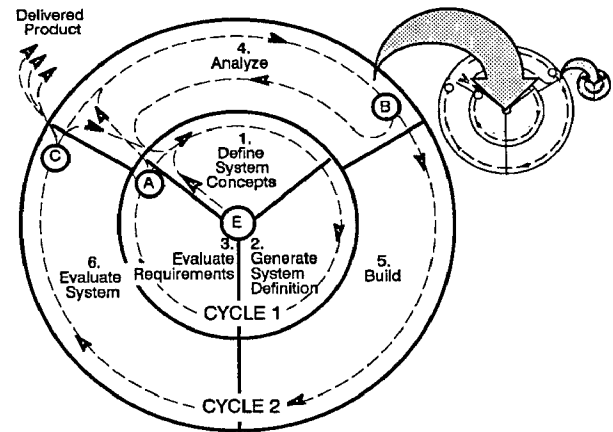


Figure 6. Recursive Use of the Circular Model

into cycle 2. At the end of cycle 2, cycle 1 is re-entered with the newly completed component and the next component is transitioned into cycle 2. The reason for re-entering cycle 1 with the newly developed software is to evaluate the completed software to ensure system requirements and operational concepts are still being met, and to finish the planning for implementation of the next capability.

It is most likely during the analyze phase that the design is abstracted to lower and lower levels, which may be able to be placed in their own recursive/parallel development cycles. This is shown in Figure 6 as the shaded arrow from the analyze stage pointing to a smaller representation of the circular model. At the end of cycle 2, the first capability is delivered and also re-enters cycle 1 for evaluation and integration into succeeding capabilities. The next capability to be developed is then put into cycle 2. This series of events continues until the entire system has been developed and delivered to the customer.

PROTOTYPING

Prototyping of the system can take place at any time during cycle 1 or cycle 2 in parallel with the activities of each stage. During each stage, technical information obtained from the prototype is continually fed into the design process to aid in the design and implementation. An important aspect of cycle 1 is the role that prototyping plays in the evaluation of system requirements. In the past there was no model to represent the evaluation of the prototype to ensure it was solving the correct problem and that the technology to solve the problem was integrated into the main stream design process. In the circular model, the results of the prototyping activity are continually fed into the activities of each stage.

WATERFALL MODEL EMULATION

The circular model adapted to the Waterfall approach of software development is shown in Figure 7. Included are the reviews for each stage of the Waterfall model.

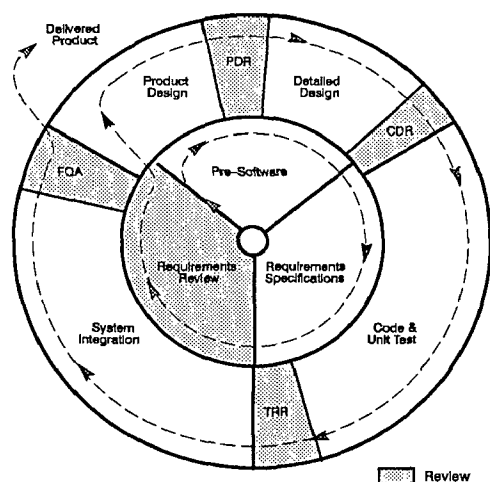


Figure 7. Waterfall Process

The advantage of the circular model can be seen when looking at the Waterfall model in the circular model context. Using the Waterfall model requires that all the requirements be specified before design commences or design is done with some of the requirements still in the "fuzzy" state.

CONCLUSIONS

The circular model is recognized by many McDonnell Douglas Electronic Systems Company (MDESC) projects. Several Ada projects are currently using this model. A corporate metric activity is also using the circular model as the basis for defining the collection points of metric data.

Further elaboration of the steps of recursive/parallel development needs to be performed. In particular, further refinement of the methods of requirements verification and validation need to be worked into the model's methodology.

ACKNOWLEDGMENTS

I would like to thank Bill Halley and Robert Ensey for their patience with me during our frequent theoretical discussions and for their contributions in reviewing this paper.

REFERENCES

- [1] Royce, Winston W., "Managing the Development of Large Software Systems: Concepts and Techniques," Proceedings, WESCON, August 1970.
- [2] Parnas, David L., and Clements, Paul C., "A Rational Design Process: How and Why to Fake It", IEEE Trans. SW Eng., Vol. SE-12, No. 2, February 1986
- [3] Brooks, F.R., Jr., "No Silver Bullet, Essence and Accidents of Software Engineering", Computer, vol 20, no. 14, 1987.
- [4] Boehm, Barry W., "Spiral Model of Software Development and Enhancement," ACM Software Engineering Notes, Vol. 11, No. 4, August 1986.
- [5] Berard, Ed V., Object Oriented Design, 1988, EVB Software Engineering Inc., Frederick, Maryland
- [6] Firesmith, Donald G., Ada Project Management, Version 6.1, 1988, Fort Wayne, Indiana.