INFORMATION SYSTEMS DEVELOPMENT TECHNIQUES AND THEIR APPLICATION TO THE HYDROLOGIC DATABASE DERIVATION APPLICATION

By Paul Davidson, Hydrologic Engineer, USBR Upper Colorado Regional Office, Salt Lake City, UT; Rich Hedrich, Hydrologic Engineer, USBR, Lower Colorado Regional Office, Boulder City, NV; Tom Leavy, Professional Research Assistant, University of Colorado, Center for Advanced Decision Support for Water and Environmental Systems; Warren Sharp, Hydrologic Engineer, USBR Columbia Area Office, Yakima, WA; Neil Wilson, Professional Research Assistant, University of Colorado, Center for Advanced Decision Support for Water and Environmental Systems

Abstract: The proper collection, storage and availability of data are fundamental to the USBR decision-making process. Key elements of these data are consistency and completeness. In order to achieve these goals, the USBR has retained CADSWES to develop a data derivation application that will provide user-defined data transformations on sets of detail data to populate more aggregated, higher levels of data storage in a timely and consistent manner. This project requires the analysis, design and rollout of a sophisticated derivation application machine, along with support utilities to allow for setup and maintenance of the processing. This paper will discuss the development process from an information technology development perspective and the use of such proven tools and techniques as top-down, structured analysis, design and quality assurance methodologies. A primary goal in using these structured development techniques is to ensure the early and continued involvement of the subject matter experts, developers and project management. By embracing a philosophy of collaboration, it is possible to produce a requirements definition that captures the functionality of the product without becoming bogged down in design and implementation tradeoffs. Later, a development cycle can be executed to produce prototypes that test the proof of concept early and often. These initial development efforts lead to a cycle of implementation of additional details and functionality, along with their quality assurance and verification against the agreed upon requirements specification. This development cycle allows for the early and continual testing of the entire system and provides a feedback loop allowing for quick detection of high-level system problems and their fixes. Utilizing these methods, a complex data processing system can be developed and shipped in a timely manner that works according to users' understanding and needs, and also functions in a correct and consistent manner.

INTRODUCTION

Solving the problem of sparse and inconsistent data in the HDB: The HDB has been used in a production mode in several USBR sites since about 1995. The data in the HDB have become a critical part of the planning and operational decision-making process, as well as a data warehouse used to mine information to support various research projects. The reliability of these data is a direct function of their availability and consistency within several time series data stores supported by the HDB. By availability, we mean

the complete filling-in of a series of data with respect to all applicable time steps. For instance, average data gathered at an hourly interval should be properly aggregated to provide average data for daily, monthly and yearly intervals. Furthermore, these higher intervals should consistently reflect the values stored in the lower intervals. The process of completely populating time steps from a set of base data and guaranteeing consistency had previously been left to the *ad hoc* execution of an incomplete application. This resulted in HDBs that were sparsely populated and potentially inconsistent over different time steps. The solution decided upon by a team of HDB experts was to create an application that would intelligently use base data to derive higher-level, time aggregated data on a scheduled basis, using formal, predefined source data and methods.

WHY SHOULD DEVELOPMENT BE STRUCTURED?

Complex applications benefit from more formal system development process methods: It is important to note that a formal methodology is not always necessary for, or appropriate to, software production. There are times when an *ad hoc* software development process is justifiable. For instance, if the project is small or non-critical, the use of a formal methodology may not be important and may even tend to impede the delivery schedule. The challenge is to understand when a project has reached critical mass and when a more formal methodology will enhance the product and the delivery schedule. Several factors affect the overall complexity of a development effort. However, one benchmark, established in the 1970s, deems a project to be small if it contains less than 1000 to 2000 lines of code. Once this level is reached, the complexity of the process tends to grow faster than does the measurement of the number of lines of code required to build the application. It had been suspected that the amount of effort required to roll out an application requiring 100,000 lines of code was much more than 10 times the effort of an application requiring 10,000 lines of code. As it turns out, the scaling of effort may be more like a factor of 50 to 75. There are many reasons for this, but some of the common factors that contribute to this non-linear scaling are:

- Error density tends to increase as the total amount of code increases. In other words, the number of errors per unit of code grows with the total amount of code written.

- As the requirements become more complex, the number of people required for system implementation increases. This increases the number of communication paths according to the formula $n*(n-2)/2$, where n is the number of people assigned to the project. This increase in the number of communication paths leads to a higher probability that errors of misunderstanding will be introduced.

Possible pitfalls of trying to build a complex system using *ad hoc* or poorly defined methods are:

- Complicated manual procedures are necessary in order to prepare and use the data in the system.

- Users lack confidence in the correctness of the system and its information, often resulting in the creation of parallel backup systems to verify results.

- Using the new system is more cumbersome and time-consuming than the one(s) it replaced.

- Some of the functionality remains unused due to ineffective testing and lack of user understanding.

- Users need to know internal quirks of the system in order to get required information.

- Manual processes are used to pass information because it is difficult to get from the new system.

These problems can be avoided with the proper application of a more formal development technique resulting in a system that performs according to expectations, and that can be delivered in a reasonable amount of time and with a reasonable budget. After considering the Derivation Application and reviewing information from an initial analysis effort, it was decided that this system was more than trivial and would require the use of a more formal methodology in order to provide an effective product in a reasonable amount of time.

AVAILABLE METHODOLOGIES

Basic structures: Given the need for a system development model, what techniques are available and what are the associated advantages and disadvantages? The common methodologies are variations on a similar theme that attempts to provide a clear definition of the problem and proposed solution, and involve the correct talent mix to deliver according to expectations. Some of the techniques are very formalized and define a series of steps that assume that each step is fully completed and is 100% correct before proceeding to the next. Other techniques use a formalized approach while allowing room for innovation through phase overlap, feedback and refinement loops. However, all of the models recognize four main elements or phases of system development and deployment. Those are:

- functional requirements or analysis
- system design
- system build
- system maintenance

Our research has found that the most commonly defined and used system development life cycle models are:

- Build and fix
- Staged or waterfall

- Spiral
- Recursive/parallel or object oriented

In determining this list of possible life cycle models, two others were considered. They are the rapid prototype model and the incremental model. Each of these seemed to be simple variations of the waterfall model. The rapid prototype model replaces the requirements analysis phase of the waterfall model with the development of a throw-away prototype which is used to drive out functional requirements by focusing on user interface specifications. The incremental model replaces the system build phase of the waterfall model with a series of build phases for those parts of the product that can be isolated and delivered separately. It seemed to us that the basic waterfall model can easily be adopted to provide these two other models, based on specific projects and project team mixes. Additionally, the Microsoft synchronize and stabilize model was not considered since we feel that developing commercial software is a different activity than is developing in-house products, and this model simply recognizes that the customer is more of a survey respondent than part of the project team and then uses a variation of the incremental model to ship product.

The build and fix model is not technically a life cycle model but, rather, represents the lack of one. This was the methodology typical of system development in the old days, when computers were expensive and people were cheap. The use of this "technique" resulted in expending the system funds primarily on the back end of the system life, in the maintenance mode. This is a good place to make a point of how using a formal method for system development leads to more robust systems and frees up expensive team resources to work on solving new problems and challenges. Typical breakdowns of the relative costs of each of the four major phases have been documented to be these approximate percentages. For a system development using the build and fix, or lack of formal approach, the time and cost spent is approximately 5% on analysis, 5% on design, 20% on construction and 70% on maintenance. So, given a development team of 10 people, their typical activity mix would be that 7 of the 10 are spending full time fixing existing systems. For systems developed with an effective use of a more formal approach, the time and cost breakdown is more typically 50% analysis, 10% design, 10% construction and 30% maintenance. So, for that same team of 10 people, now 7 of them are devoting their resources to developing new systems to improve the business.

Having decided that our application is complex enough to warrant the use of a more formal methodology, we can now investigate the various available techniques and weigh their tradeoffs and applicability.

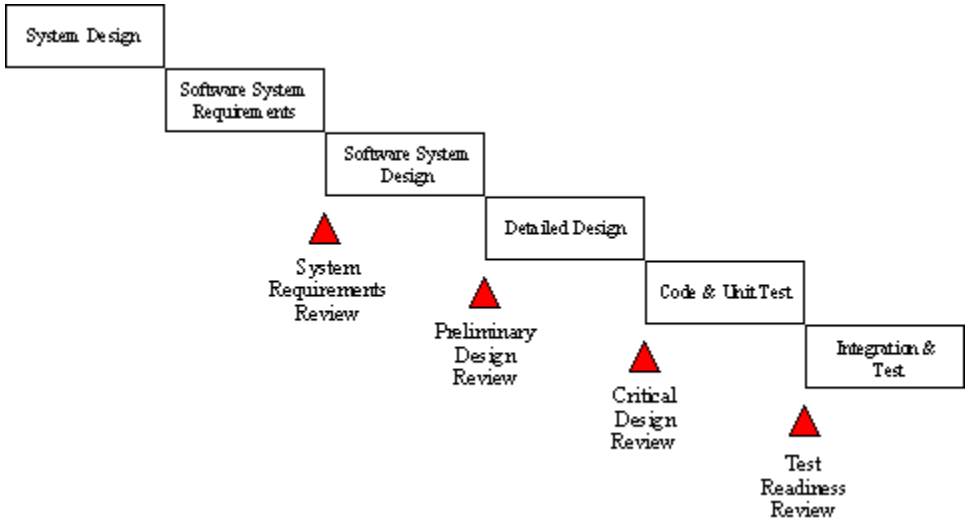
Waterfall model: The waterfall life cycle model divides the development process into a series of sequential steps. These steps are typically defined as:

- Requirements definition
- Preliminary design
- Detailed design
- Implementation

- Unit testing
- Integration testing
- System testing
- System rollout
- Maintenance

One of the fundamental aspects of the waterfall model is that each step is assumed to stand alone and must be completed 100% before moving to the next step. This type of model is comfortable in large, formal organizations and on projects that have high risk in the areas of budget and schedule predictability and control. The original model did not allow for returning to earlier stages and making modifications. Later, when this was found to be too restrictive, a feedback cycle was incorporated which allowed the team to revisit and modify earlier phases. However, allowed only for changes, and not enhancements, to be made, and the affected step had to be "closed out" and assumed to be complete before the changes were propagated to following steps. This implies, for example, that the requirements analysis is 100% complete before moving into any design or build work. This forces the development team to work in an artificial manner that studies have shown to be counterproductive because the team's natural creativity is stifled. These studies have shown that the software development process done by experts tends to move between very formal and very informal sessions that can blur the boundaries defined by the system development stages.

Steps in a typical waterfall development method life cycle:

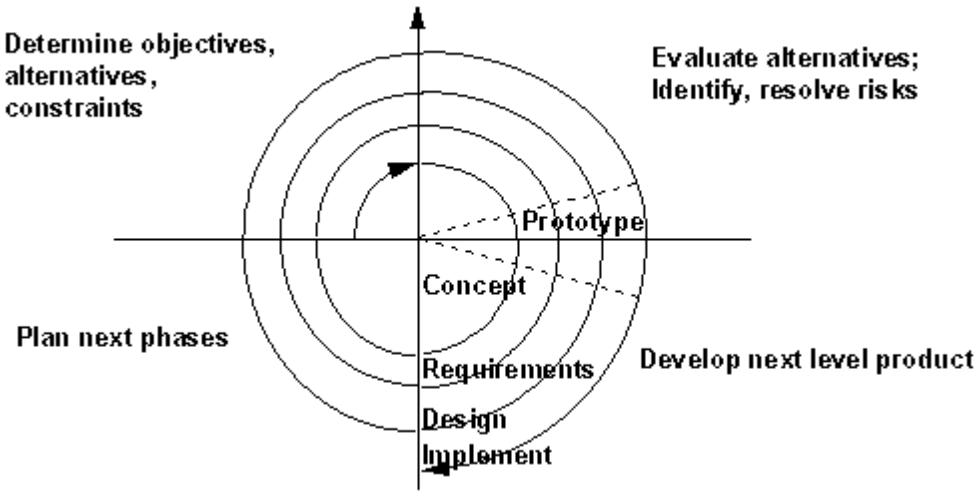


Spiral model: The spiral model uses concepts of prototyping and evolutionary system implementation to primarily identify and evaluate risk and cost. Using this model, before any work is started, a risk analysis and cost/benefit analysis are completed and evaluated. This model divides the development activities into four quadrants through which the effort proceeds. Each time a quadrant is visited, the scope is increased based on go/no go

decisions made in the previous efforts. Thus, an expanding spiral effect that finally leads to a deliverable system is created. The four quadrants of activity are:

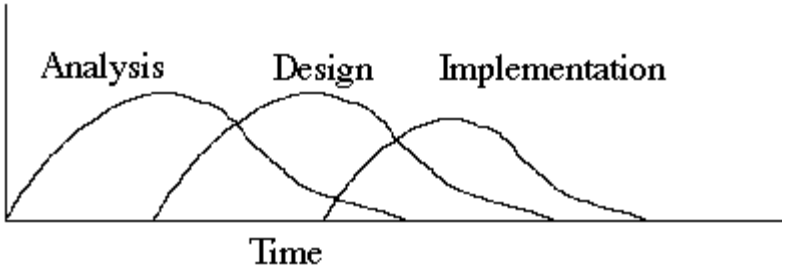
- Determine project objectives, such review alternatives as build or buy, and identify such constraints as time available, budget and staff talent availability
- Evaluate the defined alternatives and identify the risks, and choose alternatives to minimize those risks
- Develop a prototype based on previous work and verify its effectiveness
- Reevaluate and plan the next cycle

Each trip around the quadrants is called a "round" and a commitment to proceed or a decision to abandon is made at the end of each round. Rounds zero and one are generally concerned with project feasibility and risk assessment. If there is a "go" decision at the end of round one, a more detailed analysis and concept of operations is begun. At this point, the spiral model borrows many of the concepts of the waterfall model, and rounds two and three result in functional requirements specifications and preliminary design decisions. However, each round from here produces a working prototype and a new feasibility study and risk tradeoff analysis. Thus, the spiral model borrows several concepts from the waterfall model but incorporates them in a framework of feedback loops to assess the viability of the proposed system in an ongoing manner, adding to the flexibility of changing the implementation based on outside influences. It also borrows heavily on the rapid prototype cycle or methodology to provide end users with previews of the system quickly so that they can continue refining requirements and determining if the finished product will meet their needs. In other words, the spiral life cycle model defines an organized approach to prototyping. This seems to work well in an environment where many of the options, requirements and constraints are not known or not well understood initially, such as a research project. If this is a development effort where the product, options, requirements and constraints are fairly well-defined and understood, the spiral approach is not as appropriate or necessary as are other less risk assessment driven techniques.



Recursive/parallel or object oriented model: The recursive/parallel model incorporates the idea that software is developed by applying work from all the major development cycle phases in differing amounts depending on the progress of the project. It has been characterized as "analyze a little, design a little, build a little, test a little" and then repeat the process. It is actually an evolution of the waterfall model that incorporates ideas about the psychology of system development mentioned in the discussion of the waterfall method. The recursive/parallel model considers the "real" way that expert system developers work and allows for more creativity by the project team while still understanding that different phases of the project have different deliverables. This approach evolved from people attempting to apply object-oriented thinking to projects. Projects of significant size produce requirements that are at different levels of abstraction. While some are broad and high-level, others are very detailed. This approach allows for all the requirements to be examined and for choosing some whose detailed analysis can be deferred until later in the project timeline. Also, in traditional project methodologies, like the waterfall life cycle, analysis and design are very different phases and disciplines using very different techniques. The recursive/parallel method tends to smooth the distinction between these phases. Since many people who worked in a build and fix mode are comfortable thinking in design terms, this approach can take advantage of those talents while still producing an effective set of functional requirements. The diagram indicates how the different phases of the project life cycle now interact and overlap using the recursive/parallel approach.

Life cycle phases intermix in recursive/parallel model:



ADVANTAGES AND DISADVANTAGES

Each of these models offers advantages and incurs disadvantages and these tradeoffs should be understood in order to select the best methodology of work for a specific system implementation project. We feel that it is important to realize that there are no magic bullets available for software system development and that each project should work to choose a method based on its particular personality. We will now look at some of the tradeoffs involved in each of the life cycle models we have looked at.

Build and fix: As mentioned earlier, this is really the lack of a life cycle model. About the only advantage that might be attributed to this approach is that it defers doing the hard requirements analysis until the project has reached a crisis stage and is either abandoned or totally restructured. For systems with any level of complexity, this approach should not even be considered. There is always a danger that this lack of methodology is chosen by default.

Waterfall model: The waterfall approach is very structured and disciplined. Formal documentation and sign-off procedures enforce this disciplined approach. A phase is not considered completed until all documentation is done and approved. The documentation provides concrete evidence of progress and milestone achievement, which is embraced by management. This tends to add cost to the front end of the project while cutting time in the build and delivery phases, since all decisions will be made before the implementation begins. The biggest problems with this method occur because the subject matter experts are required to have perfect foresight. Once the system is specified, there is no easy method to modify the functionality. This results in delivered products that may not meet the real needs. This very rigid approach can work well if the requirements are stable and completely understood.

Spiral model: The spiral model's biggest advantage is in early detection of high-risk situations and the ability to make a go/no go decision at the end of each round. This is because there is an explicit risk analysis designed to identify potential problems as more is learned about the system requirements and implementation. This relies heavily on the team being experienced and able to accurately assess risks. A major drawback to this approach is that risk analysis itself is very expensive and adds significantly to the overall delivery cost. Also, some environments may not lend themselves to being able to easily cancel a project, such as one that has been contracted to third parties.

Recursive/parallel model: This model's strength is that it recognizes the "real" way that development team members think about software projects. That is, the phases tend to overlap and there should be an allowance for iterations of the entire cycle. It allows for non-critical specification and design decisions to be postponed until the infrastructure will support an appropriate development. This also tends to involve all different skill sets of the project team on an early and continuing basis. The danger associated with this approach is that it can easily degenerate to a build and fix scenario, or to a continual cycle of "code a bit, test a bit," where a final product is never completely delivered.

Thus, each of the models has significant strengths and weaknesses, and often the type of project is not clear enough to definitely recommend one over the other. Also, other criteria should be considered, such as the type of organization, management maturity and style and the skills of the team members, before trying to implement a particular methodology. Often the best suggestion is to weigh the advantages of each model in relation to what is known of your project and then to select ideas from each model to build a custom technique.

DERIVATION APPLICATION APPROACH

In the opinion of one of the authors, a key plus in choosing a more formal system life cycle is that it provides project visibility to the entire team early and often. In deciding to use some structure, we are pushed to enlist the input of subject matter experts, systems analysts, developers, and project management from the very beginning and throughout the life of the project. This leads to the concept of joint application development. Joint application development has the major goal of emphasizing and involving the entire team for the entire duration of the project in an environment that is conducive to focusing on the new project in planned, dedicated work sessions. This increased participation leads to the following benefits:

- Improves relationship between users, management, and developers.
- Improves system literacy of users, and subject understanding of developers.
- Conflict resolution becomes responsibility of entire team, with emphasis on leads.
- Improves systems analysis time by focusing work sessions and gathering all project resources simultaneously.
- Lowers cost of systems development by defining requirements more completely and correctly.
- Increases team satisfaction, confidence, and support.
- Reduces maintenance time due to earlier completeness and correctness of specification.

Thus, during the early ramp up of the Derivation Application development effort, a team was formed that represented the main user community of the HDB, USBR and

CADSWES project management representatives, and the CADSWES and contract developer resources assigned to the project. It was initially felt that the user interface portion of the project was relatively small and well-understood, so an early prototyping approach was not necessary. Instead, we felt that the underlying functionality of the application needed to be investigated in some detail to provide for a more complete implementation and to allow the product to be easily extensible. We therefore produced a working document that described in functional terms the requirements of the new system, tradeoff decisions that were purposely made, and outstanding issues that require discussion and resolution. This document has been used as an informal contract and guide for the development and deployment of the application. The document was produced as a result of work sessions involving the entire team, mostly via conference phone calls due to geographic constraints of the team members.

As this document evolved, it became clear that while a portion of the requirements was very fundamental and not subject to much change or interpretation, other aspects were either not fully understood or of a much lower priority. Using this information, we constructed a subset of the requirements document that we called phase 1. We felt that phase 1 could be implemented in a few months, would provide a solid foundation for additional increments, and would probably not change as the application extended. Using the phase 1 requirements subset, we were able to deliver a working sample within 6 months of the initial project kickoff. This foundation product is being deployed in a parallel test mode in two of the three HDB user sites participating in the effort. During this parallel test phase, the application will continue to be extended and tested until it is felt that all functionality required to move it to production status is available and proven. Our methodology then is a hybrid, using ideas borrowed from all of the models.

We initially constructed a fairly formal requirements document with the understanding that it may need to change as we learn about the system. We used ideas from the spiral model to do some risk assessment in defining a rollout plan for the new system. In addition, we are employing some incremental build ideas for the construction phase in order to quickly provide test product that can be verified by real users. Almost the entire cycle has incorporated a lot of the recursive/parallel thinking process in that we recognize that some design issues must be addressed in order to fully understand functionality implementation issues and so that early coding can occur on foundation parts. However, we specifically have not needed to concern ourselves with a lot of physical design or infrastructure issues. This is because our platform mix for the application was defined by the current HDB implementation. We feel that, although we haven't embraced one particular development methodology, we have successfully blended the concepts and produced a technique that works for this project. We are confident that the key to its success is really due to the common thread in all the life cycle methods of early and continuous involvement of the entire team. This affords the opportunity to define the right solution with the right tradeoff decisions and ensures that all team members have a good understanding of the process and the product.

REFERENCES

Topper, A., Ouelette, D., Jorgensen, P., 1994, Structured Methods, Chapter 2

Bellin, D., Suchman, S., 1990, Structured Systems Development Manual, Chapter 1

Denenberg, D., 2002, Notes from IFSM 636, Structured Systems Analysis

Berard, E., 2001, Life Cycle Approaches, Information Technology White Paper

Landay, J., 2001, The Software Life Cycle, notes from class lectures for CS169 Software Engineering