Planning and Managing the Development of Complex Software Systems

Dr. Richard Bechtold
Abridge Technology

Wednesday, 30 April 2003
Track 4: 1:50 — 2:30
Ballroom D

With the ongoing evolution of information systems and computer technologies, it is becoming progressively easier to leverage incremental design, development, and testing strategies. This article briefly examines the problems inherent in the traditional "grand, all-at-once" implementation approach. Next, an alternative approach is described that leverages grand and incremental design, incremental development, early incremental testing, rapid risk reduction, and the re-calibration of data used for estimation. Key benefits to this approach include easily developed and highly reusable estimation data, early verification of system feasibility, early management of customer expectations, and early validation of system usability and acceptability.

In the initial decades of the computer industry, systems were built in isolation. Large problems typically were solved through the development of numerous, stand-alone, vertical solutions. As computer systems started covering progressively larger segments of the problem space, they began to overlap and often failed to integrate.

The solution to this problem was part systems engineering, and part *grand design* and *grand implementation*. The premise behind these grand approaches was to think of the entire problem and to implement a *total solution*. Although on the surface this seems perfectly reasonable, the problem is that this strategy, especially when coupled with a Waterfall Model life-cycle development approach, simply does not reliably scale up to implementing large, complex systems.

Depending on whose data you reference, there is anywhere from a 50 percent to an 80 percent likelihood that any given software system project will fail [1]. That is, the project will require substantially more time than originally planned, cost substantially more than originally budgeted, or will deliver substantially less functionality than originally expected, or any combination of the preceding. Furthermore, the larger the project, or the longer the planned duration, the greater the likelihood of failure.

After nearly 25 years in the software industry and 10 years of conducting process appraisals using the Software Engineering Institute's Capability Maturity Model® for Software, I have repeatedly observed that it is exceedingly rare for complex, large-scale, multi-year, grand implementation software projects to deliver all expected functionality within the originally planned schedule and budget. But are these failures truly due to the grand implementation approach, or is it something else?

Grand Implementation Problems

In this article, grand implementation approaches are considered to be coupled with a Waterfall Model life-cycle development approach. Although large, complex systems often need a grand design to ensure overall architectural integrity, having such a design does not mean that the next step must be a grand implementation. Nevertheless, after a grand design is completed development often commences for the entire system. When the entire system is ready, it is put though integration test, system test, and acceptance testing: classic Waterfall Model development. While this can be a very effective approach for small systems or short duration projects, it becomes a much less successful approach as project duration and complexity increase.

"Depending on whose data you reference, there is anywhere from a 50 percent to an 80 percent likelihood that any given software system project will fail."

Part of the problem is requirements volatility. The traditional response to this problem is to require the customer to freeze their requirements – as if that were possible. Frozen requirements make absolutely no sense for the simple reason that all systems are basically built to address one root requirement: solve the customer's problems. If a system is being built during several years, what is the likelihood that the customer's problems will remain unchanged throughout this entire period?

I do not know what the aggregate staff-year transition point is regarding when a grand implementation project crosses over from being more likely to succeed to being more likely to fail, but it does not seem like a very large number. Certainly a grand implementation approach to a four-person, three-month project will likely work. Maybe a grand implementation approach can even work on a 10-person, 18-month project if you are really good. But what about a 100-person, five-year, legacy systems modernization project? Can we reliably apply grand implementation thinking to this scale of a project, or should we consider another approach?

There is still another problem with the grand implementation strategy. Even if you can get it to work, you have likely created an absolute nightmare for whoever will build the eventual replacement system. Too often, people who design systems do not think about an incremental approach to that system's *retirement*. If you do not design the system for incremental retirement, then in all likelihood you will not be able to conduct an incremental retirement.

Typically, this forces the replacement system to also be based upon (yet another) grand implementation, *all-or-nothing* solution. Replacement systems are usually much more complex than the systems they replace, so the problem of *total solu-*

tion replacement – often referred to as *legacy system modernization* – can lead to projects that are progressively more likely to repeatedly fail [2].

Designing and Planning Complex Software Systems

The remaining sections of this article look at a systematic alternative approach for complex systems' construction. For simplicity, this discussion generally focuses on the design, estimation, planning, and management of the construction and delivery of complex Web-based systems or systems that include Web-based subsystems, but the principles are also applicable to the construction of other types of software-intensive systems.

Micro-Incremental Development

Incremental development is not a new concept, but the eXtreme Programming (XP) community has given further definition to some of the principles. In particular, XP advocates using a highly incremental approach, and building testable functionality in much smaller and shorter duration steps [3]. The approach advocated in this article adds additional details to this foundation.

When commencing system decomposition, strive to allocate high-level functionality into regions, partitions, and frames. For this article, regions are defined as major subsystems that can be separately implemented, tested, maintained, and replaced. At the design level, regions capture required system functionality but defer physical implementation details.

Each region consists of several partitions that can be separately implemented, tested, maintained, and replaced. At the design level, partitions reflect not only functional requirements but also capture all important physical implementation details.

Lastly, each partition is divided into several frames. Frames are low-level or atomic software components such as Java class files that can be separately assigned to small software teams for parallel construction and unit testing.

Generally, a large system can be designed with five to 10 distinct regions, each with five to 10 distinct partitions. Completed regions should deliver actual usability to the customer or system end-users. Within each region, completed partitions should deliver actual functionality. Although functionality does not necessarily translate into usability, it does allow the customer or system end-users to evaluate system characteristics and performance, and to provide early feedback to the development team [3].

Hot-Swappable Partitioning

One of the key tenants of this approach is that each partition is *hot-swappable* or capable of being replaced with little or no adverse impact on the rest of the system. Given the preceding guidelines of five to 10 regions, each with five to 10 partitions, your design will contain between 25 and 100 separately implementable, testable, and replaceable partitions. To facilitate this, partitions can be designed to interact with each other primarily through message passing and file input/output.

"Even if you can get it [grand implementation strategy] to work, you have likely created an absolute nightmare for whoever will build the eventual replacement system ... you will not be able to conduct an incremental retirement."

Of course, there are trade-offs to be addressed. Partitions that communicate via file input/output will suffer a severe performance penalty. Additionally, as systems evolve and grow larger, there is a general tendency for them to become increasingly interconnected. Avoiding this will usually require periodic efforts to reduce partition coupling and to increase partition cohesion, both of which translate into increased time, money, and effort. However, by taking steps to ensure hot-swappable partitioning there is an increased likelihood that you can more easily fix or upgrade individual partitions without adversely impacting the rest of the system.

Nevertheless, given the preceding issues it is clear that 100 percent hot-swappable partitioning is more of an ideal than it is a practical reality. At a minimum, however, it is certainly critical to avoid the *all-or-nothing* implementation where the system is either completely working or completely useless. Within the limits of system performance, project budgets, and schedule constraints, hot-swappable partitioning should be a top-priority design objective and built into as many partitions as possible.

Commitment Deferral

A third key concept is the deferral of technical commitments, and especially architectural commitments, to the greatest extent possible. Again, in some areas commitment deferral may be impossible or, indeed, not even desirable. Nevertheless, during design you may have some opportunities to allocate certain technologies to partitions that you plan to build towards the end of the project. By deferring commitments you increase the likelihood that you can more easily respond to changing requirements, changing technologies, or evolving solution alternatives.

Early Detection/Agile Response

Further leveraging XP, a fourth key concept is to plan for a development and implementation approach that allows for early detection of any problems, and for agile and rapid response to those problems. Given that you have designed for both usable releases (regions) and functional releases (partitions), system components can be developed, tested, and immediately delivered to key stakeholders. In addition to early detection and correction of defects and improved customer communications, this approach will allow you to perform highly effective customer expectation management.

Partition and Frame Estimation

Even though the preceding guidelines will result in a system where functionality is implemented across 25 to 100 partitions, these partitions may still be too large to accurately estimate required work. Therefore, as indicated earlier, each partition can be further decomposed into five to 10 frames. The primary objective of frames is to facilitate planning, construction, unit testing and integration testing, and to support risk management (more on this later).

Each frame should be assigned to one person or to a very small team (such as with pair programming). Additionally, each frame should generally appear to require somewhere between one and four weeks of work. When it is obvious that less work will be required, frames can be combined. Conversely, if more work is obviously required, frames can be further

decomposed.

The next step is to analyze the key attributes of each frame and to use those attributes as the basis for schedule and cost estimation. Although key attributes will vary substantially between different software systems, some attributes will be almost universally important, such as a system's diagnostic capability [4].

For example, the following are five key attributes to consider for systems that include Web-based partitions:

- *Artwork* or static content.
- *Logic* or core dynamic content.
- *Diagnostics* or the ability to detect misuse or system intrusions.
- *Security* or the ability to prevent misuse or system intrusions.
- *Containment and Recovery* or the ability to perform damage control and repair.

As shown in Figure 1, each attribute is analyzed and rated using a five-block by five-block grid. To rate the preceding five different attributes, you would use five separate tables. Figure 1 shows an empty grid for estimating required security for a particular frame.

The rows of the grid indicate the relative amount of work to be done. Rows are labeled top-down from *E* to *A* indicating an extremely high amount of work to an extremely low amount of work, respectively. The columns of the grid indicate the relative complexity of the work to be done. Columns are labeled left to right from 1 to 5 indicating extremely low complexity to extremely high complexity, respectively. Since the outermost rows and columns represent extremes, most attributes should be rated in rows B, C, or D and in columns 2, 3, or 4.

As each of the five grids is completed, the estimator determines a confidence level of low, medium, or high and then documents this in the upper left corner. In Figure 2, the estimator indicated they had a low level of confidence in their estimate and that security complexity is rather low (column 2) for this frame. However, the amount of work related to implementing security features is shown as extremely high (row E).

After the five key attribute grids are done, the estimator uses them as *support information* for the composite grid. The composite grid, as shown in Figure 3, is used to estimate overall staff-days for constructing a frame. The cells of the composite grid contain values representing the expected staff days, and the estimator simply circles one of the values. Note that there is a general, *but not algorithmic*, relation between the key attribute basis grids and the composite grid. For example, if most of the basis grids were marked in the upper right regions, it would normally occur that the composite grid would likewise have a cell selected from the upper right region.

As a final step, the estimator uses the upper left square in the composite grid to indicate low, medium, or high confidence in the accuracy of their selection for expected staff-days duration.

The default values shown in Figure 3 were deliberately selected so that *normal* work would span from one to three weeks (the interior three rows and columns). Recall that the work for each frame was initially intended to be *obviously* between one and four weeks. The default values in the composite grid actually accommodate ranges from 40 percent of the obvious minimum to 150 percent of the obvious maximum, or work that spans from two days (A1) to six weeks (E5).

Given these default values, project durations can span from approximately one staff-year (five regions, each with five partitions, each with five frames, each estimated at two days) to approximately 120 staff-years (10 regions, each with 10 partitions, each with 10 frames, each estimated at six weeks). Decomposition can be reduced, or increased, to accommodate shorter or longer project durations, respectively. Similarly, the default values in the composite table can – and indeed should – be adjusted over time to better reflect your actual projects and performance [5].

When you have completed these design and planning steps, you will have a comprehensive and detailed foundation from which to commence managing and controlling your project.

Security					
	1	2	3	4	5
E					
D					
C					
B					
A					

Figure 1: *Key Attribute Analysis*

Security					
Low	1	2	3	4	5
E		X			
D					
C					
B					
A					

Figure 2: *Example Ranking*

Composite (Staff Days)					
	1	2	3	4	5
E	5	10	15	20	30
D	4	10	15	15	20
C	3	5	10	15	15
B	3	5	5	10	10
A	2	3	3	4	5

Figure 3: *Composite Grid*

Managing and Controlling Complex Software Systems Development

As mentioned previously, one of the objectives of the third level decomposition (the frame level) is to support risk management. To accomplish this, it is recommended that you commence actual development with the easiest two regions. Within each of those regions, commence development on the easiest two partitions. Within each of those partitions, commence work on the easiest two frames. The objective of this approach is twofold. First it helps ensure that the development team's learning curve occurs in the least challenging parts of the overall system. Second, you quickly accomplish finished frames.

When you are done with the easiest frames, then commence work on what appear to be the hardest frames. Likewise, commence work in the same pattern with partitions and regions. The objective here is – after much of your learning curve is behind you – to reduce project risk as rapidly as possible. By rapidly undertaking the hardest or most challenging parts of the system, you can more quickly discover whether or not there are any insurmountable hurdles. On the outside chance that you may have to revisit your design, or may need to resort to an alternative solution, these recovery steps are happening much earlier than they might otherwise. Because they are occurring earlier, and because, as discussed earlier, one of your design goals was also to defer commitments even in the event that you need to take an alternate approach, you have a much greater likelihood that you will be able to keep a

greater percentage of the system already developed.

Another benefit to first doing the easiest work, then doing the hardest work, and then doing the average work is that data collected during development rapidly becomes directly usable to recalibrate your estimations and to more accurately predict the remaining work on the project. Depending on the total number of frames, when you are as little as 20 percent into the project you may have a highly reusable set of actual data relating to frames built, estimated time to build them, and actual time to build, test, and debug. This data can be used to further improve the accuracy of estimates relating to the rest of the project.

An important part of this approach is to take a few moments after a frame is completed and do a retrospective estimation worksheet. The format for these worksheets is identical to the original worksheets – the only difference is the time they are completed. By taking this approach, each frame will ultimately have a minimum of two estimation worksheets. One worksheet was done before work commenced, and another one was done after work was completed. By comparing these before and after estimation worksheets, you can analyze and adjust your estimation approach if, for example, you see a clear tendency to underestimate either the amount or complexity of work relating to one or more of the key attribute types.

As you develop the system, to the greatest extent possible strive to rapidly deliver partitions to the primary stakeholders. This will allow for stakeholder evaluation and feedback, and for you to take a proactive approach when addressing conflicting expectations among the stakeholders. Ideally, you can provide early access and insight to end users, program managers, procurement specialists, subject-matter experts, and anyone else who may have a strong and influential opinion about the usability and acceptability of the final system.

Lastly, when tracking and reporting progress, perform only binary accounting at the frame level. That is, do not ask developers for a percent complete estimate on a given frame (we all know what they will tell you). Instead only ask, "Have you started working on it?" and "Have you finished working on it?" Before a frame is completely done, its *percent complete* is zero. After a frame is completely done, its percent complete is 100. At the partition and region level you can easily calculate and report actual percent complete as a function of the percentage of completed underlying frames.

Conclusions

Grand implementation solutions and the Waterfall Model life cycle were perfectly acceptable approaches during the 1970s and even the 1980s. However, as systems continue to become progressively more distributed and exponentially more complicated there are significant opportunities to deliberately design these systems so that they can be incrementally constructed, incrementally tested, incrementally delivered, and incrementally evaluated. This approach directly supports early verification of system feasibility and early validation of system usability and acceptability.

Arguably, at least as important as the preceding benefits, this approach also directly supports the creation of systems that can be incrementally upgraded, incrementally retired, and ultimately incrementally replaced.

Acknowledgments

Various individual aspects of this approach have been in use for years on systems and software engineering projects, XP projects, Capability Maturity Model®-compliant projects, and elsewhere. However, it is hoped that this article presents a new and integrated view of these various individual best practices, and combines them in a way that will provide substantial and reliable risk reductions.

The design, planning, development, and management life cycle described in this article, i.e., *micro-incremental* implementation and evaluation, *hot-swappable partitioning*, *commitment deferral*, *early detection/agile response*, and *partition and frame estimation* is something I have advocated for years. However, the technique for basis estimation of key attributes, and then using that as inputs to composite tables, occurred to me while I was listening to a presentation being given by Rita Hadden on "Credible Estimation for Small Projects" at the 1st International Conference on Software Process Improvement in Washington, D.C., in November 2002. I'm not exactly sure what the specific connection is, but someone else might see one. In any event, her presentation was not only practical and informative, but also a source of inspiration.◆

References

1. The Standish Group. Chaos. Boston, MA: The Standish Group, 1994.
2. Bechtold, Richard. The Fatal Flaw of the Information Systems Industry: Failing to Design for Incremental System Retirement. Accepted for Proc. of the Project Management Institute Seminar, PDS 2003. San Antonio, TX, June 2003.
3. Beck, Kent, and Martin Fowler. Planning eXtreme Programming. Addison-Wesley, 2001.
4. Bechtold, Richard. Diagnostic Software Architectures. Proc. of the Second International Workshop on Development and Evolution of Software Architectures for Product Families. Las Palmas de Gran Canaria, Spain, Feb. 1998.
5. Bechtold, Richard, and Patricia Larsen. Planning and Estimating Complex Web-Based Projects. Proc. of the Software Engineering Process Group Conference. Boston, MA, Feb. 2003.

About the Author

Richard Bechtold, Ph.D., is president of Abridge Technology. He is an independent consultant who assists industry and government with organizational change and systematic process improvement, especially in the area of implementing effective project management. Bechtold has nearly 25 years of experience in the design, development, management, and improvement of complex software systems, architectures, processes, and environments. This experience includes all aspects of organizational change management, process appraisals, process definition and modeling, workflow design and implementation, and managerial and technical training. Bechtold also teaches graduate-level courses in software project management, systems analysis and design, principles of computer architectures, and object-oriented Java programming at George Mason University. The second edition of his latest book, "Essentials of Software Project Management," is scheduled for publication in 2003.

Abridge Technology
42786 Oatyer Court
Ashburn, VA 20148-5000
E-mail: rbechtold@rbechtold.com