Software Tech News

SOFTWARE RISK MANAGEMENT — THE PRACTICAL APPROACH

by George Holt, Mei Technology Corporation

INTRODUCTION:

This paper discusses a practical approach to risk management that will accommodate flexibility and adaptability to diverse software projects by stressing early prototyping, frequent functional builds, and a set of metrics to provide management insight during software development.

Flexible Planning: One characteristic of excellence in software development is the ability to make optimal decisions, as new information becomes available. One must guard against inflexible plans and schedules, which, all too often, present an artificial version of reality. Plans and schedules are necessary to measure progress, but must be flexible enough to adapt to change as necessary. Risk mitigation emphasizes flexibility instead of rigidity in the planning process. I'll explain how to build flexibility into program plans and how to maintain flexibility to adapt to program changes.

Software Development Planning: The primary plan for software development is the Software Development Plan. One should spend sufficient time from the start of each project putting it together. This is the road map that guides the software effort. However, it should not be so rigid that it handicaps the development effort by stifling initiative or being inflexible.

Sound Development Principles Promote Flexibility: One software development principle that promotes flexibility is to reduce code complexity and modularize common functions. The first step is to segregate the software code that will not be common to all target platforms or operating systems. The remaining code that is now common to all similar systems is then optimized by grouping common functions and reducing code redundancy. Future benefits of this approach are readily apparent. As "bugs" are discovered or modifications required, the fix will be relatively easy when the area affected is in the common code, because the software fix can be addressed in one place rather than multiple places. Then when new hardware platforms are required, gains will increase because the fix will have already been implemented. Also, when rehosting to a new platform this approach allows the common code to remain untouched, and finding areas to change for the new target system will be straightforward. Now let's take a look at other flexible approaches one can use, especially those to reduce risk stemming from changing requirements or completely unknown factors.

Risk Mitigation Planning: Exploratory and evolutionary prototype process models are excellent risk mitigation techniques: (1) For proof of concept; or (2) When requirements are ill defined or likely to change during the life of the project either through redefinition or unknown factors.

Exploratory Prototyping: This technique is valuable in saving the costs of full up R&D by first answering the questions: "Is the concept sound? Is it worth proceeding further?" Exploratory prototyping should be used to clarify requirements, discover desirable features of the target system, and encourage discussion of alternative solutions.

Evolutionary Prototyping: Evolutionary prototyping is a combination of exploratory prototyping and incremental development. Evolutionary prototypes focus on the best-understood parts of the system and build upon its strengths. Prototypes are then used operationally by end users to better define the remaining requirements. As a result, the system is implemented in stages of increasing capability. The initial prototype is developed quickly, and the requirements effort is minimized because the best understood parts of the system are being implemented first. Each succeeding version, or functional build, of the prototype then resolves a more difficult problem and, if necessary, with the aid of additional limited exploratory prototypes.

Process Improvement: Process improvement is on going and evolutionary. It's important to continually look at your process to see if there is a better way of getting the job done. This can't be done in a vacuum. Communication with everyone involved in the project is a must. Communicate on all levels: With your customers you should participate in Integrated Product (Development) Teams (IPTs) and System Management Teams (SMTs). They are invaluable to the process. With your engineers you should hold very informal but focused discussions as the need arises. One thing to guard against is extended meetings that cut into an engineers work time. This is frustrating and counter-productive to the project. As an alternative, develop and distribute weekly status reports to them so they gain insight to everyone else's work and how they fit into the big picture. I'd like to highlight the fact that one may have the very best process in place and yet fail miserably in developing good software. A motivated, goal oriented work force will succeed even when the process is sub-optimal.

Quality Management: Companies have developed and maintained software under strict government standards. But we know that companies can follow those standards to the letter and the software product can still fail. Conversely, companies could ignore the standards (as in commercial software development) and the product could be a masterpiece, pleasing the customer beyond expectations. The difference is in the quality and motivation of the people and how the project is managed — or quality management.

Quality Management: I've found that the best quality management approach uses empowerment, ownership, and consensus to gain maximum software productivity from the workforce.

Empowerment: Empowerment allows people to be their best. It requires teaching and learning and providing the right resources and then allowing the engineers to apply their talents to the task. Management should not dictate the solution. Engineers should have the freedom to think, apply judgment, and take reasonable risks without fear of undue punitive consequences. Empowered software teams are capable of analyzing alternatives and optimizing the solution. Empowerment grows the intellectual capacity of these teams and allows the customer to reap the benefits.

Ownership: Ownership or accountability gets everyone on the team signed up as owners of the deliverable. One should become totally committed to delivering the highest quality product to the customer. Because all of the team members are mutually accountable for design, development, testing, and quality assurance, they can offer constructive criticism. More importantly, any one individual becomes more receptive to accepting criticism from the rest of the team.

Consensus: Consensus has two aspects. At the start of each task, sufficient time should be allocated for all participants to agree on the functionality of the end product and the development of — for the lack of a better term — the "theme". The theme describes the purpose of the product rather than just a bunch of features put together without rhyme or reason. Clarify the theme, or purpose, to eliminate the tendency for added features that would not contribute to the product and would just lengthen the development process. A good theme, that everyone can sign up to, guides the priority and order of development and provides focus for the team.

Its also important to gain consensus with the end user of the product on the theme very early in the development cycle. On software block updates, for example, requirements are often generic. They need to be clearly defined and understood by both the developer and the end user. This requires frequent meetings where, through an iterative process, all requirements come to the surface. One should then clarify and codify them in documents such as a Software Requirements Baseline Document (SRBD), a Software Requirements Specification (SRS), or as a minimum, in letters of agreement or memorandums of understanding.

Guarding Against Dependencies: Another important principle to follow to help ensure success is guarding against dependencies. A dependency is anything not under the software developer's control, e.g., a software module or driver being developed by some entity outside the team. Your team may do everything well with regard to timeliness and yet the other party may not, either because his proficiency is low or his priorities are different. Try not to accept a dependency unless it is fully justified.

Flexibility Accommodates Change: Realize that during the development phase, there will be many very important unknowns that can be critical to success. These will unfold only as the development proceeds. The key to accommodating unknowns is being able to admit that you don't know. Guard against either individuals or teams pretending they know all the answers and try to breed a respect for lucid ignorance. Often there is a tendency to look for some semblance of order as a defense against uncertainty—reports are completed; schedules are met; yet no one may have insight into the real progress of the program. Recognize that the real goal on a software development project is not to have the correct plan in advance, but to make the right decisions every day, as the unknowns become known. Software requires intellectual prowess throughout the development phase and flexibility is the key. Once the project starts, It is imperative to monitor progress continually — sometimes known as "keeping the lights on." This is where metrics and risk management come into play.

Metrics: Project managers and task leaders should monitor collected metrics to gain insight to the progress of the project. Then development practices should be modified if metric results suggest that a process improvement is required. Some metrics that pay big dividends are:

> **REVIC**: This model estimates the cost of software projects and has good predictive capabilities provided that environmental factors are carefully estimated and applied to the model.
>
> **Schedule Adherence Measures**: Plot actual task progress against planned progress for all major and sub tasks.
>
> **McCabe's Cyclomatic Complexity Metric**: This tool measures the complexity of single code modules.
>
> **Fan Out**: This model measures system or structural intermodule complexity. Measuring such factors as fan-out—the number of modules called by a given module—will show high correlation between overall system complexity and development defect rate.

Software Cleanroom Process: The Cleanroom software process stresses proof of correctness in the design and coding phases of development.

Readiness, Maturity, Growth Model: This model provides management a quick look at the state of the software development effort. It has good predictive value in determining when software is ready for Formal Qualification Testing (FQT).

Curvilinear Relationship between Defect Rate and Module Size: This metric provides the optimum size of modules to minimize overall defects.

Early Defect Removal: Software engineers should concentrate heavily on early defect removal in the design and coding phases of software development. You'll find this has a substantial impact on reducing total program costs. It's also important during development to conduct informal analyses to eliminate error-causing processes.

SW Error/Fix Ratio and Defect Removal Efficiency: This measure is used to gain insight into how proficient and timely the software team is in resolving errors ("bugs").

The Risk Management Process

This section describes the discipline necessary for a good risk management program. Risk management is dynamic and ongoing throughout the development process. It requires active participation from the entire team from management down to the working level. The following is an idealized process and should not be followed, lock step, for each and every software undertaking. Modify the process depending on the type of work to be performed. For example; software rehosting or software block updates may not require the same degree of discipline as a new development effort. Risk management consists of two broad categories of activities: risk assessment and risk control.

RISK ASSESSMENT: Risk Assessment contains three functions: Risk Identification, Risk Analysis, and Risk Prioritization.

1. **Risk Identification**: Identify items or events (such as changes in customer requirements, new development technologies, or a change in target systems) that may have significant negative impact on the project. Input from project participants and lessons learned from past projects constitute the inputs to this first step. Risk identification checklists should be developed to guide the process.
2. **Risk Analysis**: Once risks are identified, decision analysis, cost risk analysis, schedule analysis, reliability analysis, and similar techniques and models can be used to analyze the risks. Each risk is then evaluated to assess the potential impact of the risk on the project. Then each risk is rated in two ways: the likelihood that the risk event will actually occur, and the consequences to the project if the risk event occurs.
3. **Risk Prioritization**: Risk exposure is then determined, using statistically-based decision mechanisms to determine how to best handle the risk and then the risks are prioritized.

RISK CONTROL: Risk Control contains the following three functions:

1. **Risk Management Planning**: Risk management steps, for each risk, are based on the likelihood of occurrence and impact. Risk management steps can incur additional project cost, in terms of both resources and project duration. As a result, cost-benefit analysis should be used to evaluate when benefits, gained by the risk management steps, might become out-weighed by costs associated with implementing them.

2. **Risk Resolution**: Once the risks have been identified the next step is to resolve or reduce those risks. Techniques such as staffing decisions, cost/schedule estimates, quality-monitoring, evaluation of new technologies, prototyping, scrubbing requirements, benchmarking, and simulation/modeling are employed. Experience in project management has shown that 80 percent of the potential for project failure can be accounted for by only 20 percent of the identified risks. That's why risk analysis and risk prioritization directs energies to focus on the 20 percent that cause the most trouble. Employ early prototyping to gain insight on the big risks, and frequent functional system builds to provide management visibility by 'keeping the lights on.' These are two techniques with big payoffs.

3. **Risk Monitoring**: This provides timely risk visibility and resolution. Incorporate techniques such as milestone tracking, tracking of top risks, guarding against new vulnerabilities from prior fixes, and continual risk reassessment. Insist that at any one point in time the program manager, the principal investigator/technical lead, and each developer be able to state his three top risks (i.e., priorities, or watch items). These are dynamic, and as each one is resolved, another should move up to take its place. Finally, ensure that the feedback loop stays active.

4. **Integrated Product Teams: T**he use of the Integrated Product Team (IPT) is an additional approach for containing costs and reducing risk — especially schedule risk. The IPT facilitates problem solving, allows rapid response to changing requirements, and allows the Team to work within cost and schedule. It also allows the Government to understand the cost and schedule impacts of changing requirements (or requests for new features) and allows the developer the opportunity to provide acceptable solutions.

Evolutionary development works to contain costs when one starts with poorly stated requirements (or high technical risk). You should build prototype functional software for customer evaluation, and then change it based on the customer's feedback. Build a little and demo it, fix the first, build some more, and demo it again. This, in itself, is very important as it reduces full up program costs and schedule risk, because everyone understands what is being built and when they will see the part that interests them. They too become "owners" of the program and its greatest supporters. The IPT cannot meet too often. In this manner, costs can be controlled. When new requirements or features are brought up, the developer can say, "Yes, we can do that, but it will cost this much because.… Here are some cost alternatives." This dialog is very important for controlling costs and keeping within budget.

SUMMARY: A viable risk management approach should be flexible. It should not stifle initiative. Mitigate schedule and cost risk by: looking ahead, using rapid prototyping when necessary, following a defined risk mitigation process, using a good set of metrics, keeping the customer in the loop, and "keeping the lights on". This risk management approach will contribute to success in developing and maintaining quality software over time.

****

BACKGROUND NOTE ON MEI TECHNOLOGY:

Mei Technology has produced quality software for many years. Our strength is due to many factors: We have a repeatable software development process. We have a flat and therefore responsive management structure with established quality control and review procedures to ensure quality products. There is an ongoing, evolutionary software process improvement; a proven risk management process; and a unique workforce management approach that attracts, trains, and retains top engineers.

The Mei Technology Advantage: Mei Technology has a unique blend of DoD and MIL-STD discipline and an in-depth understanding of what it takes to produce quality software. We have an established and repeatable process for quality assurance and risk management. We have the added advantage of understanding and being flexible and responsive to changes in requirements. We also understand the iterative nature of software development in a military environment, where frequent communication with the end user is an absolute necessity. Mei believes that the product we provide to our customers is something more than a software application. It is the building and maintaining of an intellectual product composed of skilled designers, programmers, and testers and the guiding principles that our company employs that allow them to achieve excellence in any software task we undertake.

The Lightweight Howitzer Program, for which Mei prototyped a Digital Fire Control System, is now a recipient of reusable software stemming from the software Mei developed for the Paladin Self Propelled Howitzer software rehost project. PM Lightweight Howitzer has reaped the benefits of Mei's risk containment approach to software reuse. Also the Government has gained by reducing life cycle costs by having common, easily maintainable software.