28.1 Reactive versus Proactive Risk Strategies

Quote:

"If you don't actively attack the risks, they will actively attack you."

Tom Gilb

Reactive risk strategies have been laughingly called the "Indiana Jones school of risk management" [Tho92]. In the movies that carried his name, Indiana Jones, when faced with overwhelming difficulty, would invariably say, "Don't worry, I'll think of something!" Never worrying about problems until they happened, Indy would react in some heroic way.

Sadly, the average software project manager is not Indiana Jones and the members of the software project team are not his trusty sidekicks. Yet, the majority of software teams rely solely on reactive risk strategies. At best, a reactive strategy monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a *fire-fighting mode*. When this fails, "crisis management" [Cha92] takes over and the project is in real jeopardy.

A considerably more intelligent strategy for risk management is to be proactive. A *proactive* strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner. Throughout the remainder of this chapter, I discuss a proactive strategy for risk management.

28.2 Software Risks

Although there has been considerable debate about the proper definition for software risk, there is general agreement that risk always involves two characteristics: *uncertainty*—the risk may or may not happen; that is, there are no 100 percent

probable risks[1]—and *loss*—if the risk becomes a reality, unwanted consequences or losses will occur [Hig95]. When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

? What types of risks are you likely to encounter as software is built?

Project risks threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project. In Chapter 26, project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced. If a technical risk becomes a reality, implementation may become difficult or impossible. Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors. Technical risks occur because the problem is harder to solve than you thought it would be.

Business risks threaten the viability of the software to be built and often jeopardize the project or the product. Candidates for the top five business risks are (1) building an excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn't understand how to sell (sales risk), (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks).

It is extremely important to note that simple risk categorization won't always work. Some risks are simply unpredictable in advance.

Quote:

"Projects with no real risks are losers. They are almost always devoid of benefit; that's why they weren't done years ago."

Tom DeMarco and Tim Lister

Another general categorization of risks has been proposed by Charette [Cha89]. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment). *Predictable risks* are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced). *Unpredictable risks* are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

1 A risk that is 100 percent probable is a constraint on the software project.

Seven Principles of Risk Management

INFO

The Software Engineering Institute (SEI) (**www.sei.cmu.edu**) identifies seven principles that "provide a framework to accomplish effective risk management." They are:

Maintain a global perspective—view software risks within the context of a system in which it is a component and the business problem that it is intended to solve.

Take a forward-looking view—think about the risks that may arise in the future (e.g., due to changes in the software); establish contingency plans so that future events are manageable.

Encourage open communication—if someone states a potential risk, don't discount it. If a risk is proposed in an informal manner, consider it. Encourage all stakeholders and users to suggest risks at any time.

Integrate—a consideration of risk must be integrated into the software process.

Emphasize a continuous process—the team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.

Develop a shared product vision—if all stakeholders share the same vision of the software, it is likely that better risk identification and assessment will occur.

Encourage teamwork—the talents, skills, and knowledge of all stakeholders should be pooled when risk management activities are conducted.

28.3 Risk Identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories that have been presented in Section 28.2: generic risks and product-specific risks. *Generic risks* are a potential threat to every software project. *Product-specific risks* can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built. To identify product-specific risks, the project plan and the software statement of scope are examined, and an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"

ADVICE

Although generic risks are important to consider, it's the product-specific risks that cause the most headaches. Be certain to spend the time to identify as many product-specific risks as possible.

One method for identifying risks is to create a risk item checklist. The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

- *Product size*—risks associated with the overall size of the software to be built or modified.
- *Business impact*—risks associated with constraints imposed by management or the marketplace.

- *Stakeholder characteristics*—risks associated with the sophistication of the stakeholders and the developer's ability to communicate with stakeholders in a timely manner.
- *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.
- *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.
- *Technology to be built*—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow you to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of "risk components and drivers" [AFC88] are listed along with their probability of occurrence. Drivers for performance, support, cost, and schedule are discussed in answer to later questions.

A number of comprehensive checklists for software project risk are available on the Web (e.g., [Baa07], [NAS07], [Wor04]). You can use these checklists to gain insight into generic risks for software projects.

28.3.1 Assessing Overall Project Risk

The following questions have been derived from risk data obtained by surveying experienced software project managers in different parts of the world [Kei98]. The questions are ordered by their relative importance to the success of a project.

? Is the software project we're working on at serious risk?

1. Have top software and customer managers formally committed to support the project?
2. Are end users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and its customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end users have realistic expectations?
6. Is the project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be implemented?

WebRef
[illegible] is a database and tools that help managers identify, track, and communicate project risks. It can be found at www.spmn.com

10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

If any one of these questions is answered negatively, mitigation, monitoring, and management steps should be instituted without fail. The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

28.3.2 Risk Components and Drivers

The U.S. Air Force [AFC88] has published a pamphlet that contains excellent guidelines for software risk identification and abatement. The Air Force approach requires that the project manager identify the risk drivers that affect software risk components—performance, cost, support, and schedule. In the context of this discussion, the risk components are defined in the following manner:

Quote:
"Risk management is project management for adults."
Tim Lister

- *Performance risk*—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- *Cost risk*—the degree of uncertainty that the project budget will be maintained.
- *Support risk*—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- *Schedule risk*—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic. Referring to Figure 28.1 [Boe89], a characterization of the potential consequences of errors (rows labeled 1) or a failure to achieve a desired outcome (rows labeled 2) are described. The impact category is chosen based on the characterization that best fits the description in the table.

28.4 Risk Projection

Risk projection, also called *risk estimation*, attempts to rate each risk in two ways—(1) the likelihood or probability that the risk is real and (2) the consequences of the problems associated with the risk, should it occur. You work along with other managers and technical staff to perform four risk projection steps:

1. Establish a scale that reflects the perceived likelihood of a risk.
2. Delineate the consequences of the risk.
3. Estimate the impact of the risk on the project and the product.
4. Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

FIGURE 28.1 Impact assessment. Source: [Boe89].

Category \ Components		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

Note: (1) The potential consequence of undetected software errors or faults.
(2) The potential consequence if the desired outcome is not achieved.

The intent of these steps is to consider risks in a manner that leads to prioritization. No software team has the resources to address every possible risk with the same degree of rigor. By prioritizing risks, you can allocate resources where they will have the most impact.

ADVICE

Think hard about the software you're about to build and ask yourself, "what can go wrong?" Create your own list and ask other members of the team to do the same.

28.4.1 Developing a Risk Table

A risk table provides you with a simple technique for risk projection.[2] A sample risk table is illustrated in Figure 28.2.

You begin by listing all risks (no matter how remote) in the first column of the table. This can be accomplished with the help of the risk item checklists referenced in Section 28.3. Each risk is categorized in the second column (e.g., PS implies a

2 The risk table can be implemented as a spreadsheet model. This enables easy manipulation and sorting of the entries.

FIGURE 28.2 Sample risk table prior to sorting

Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
Σ				
Σ				
Σ				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

project size risk, BU implies a business risk). The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually. One way to accomplish this is to poll individual team members in round-robin fashion until their collective assessment of risk probability begins to converge.

Next, the impact of each risk is assessed. Each risk component is assessed using the characterization presented in Figure 28.1, and an impact category is determined. The categories for each of the four risk components—performance, support, cost, and schedule—are averaged[3] to determine an overall impact value.

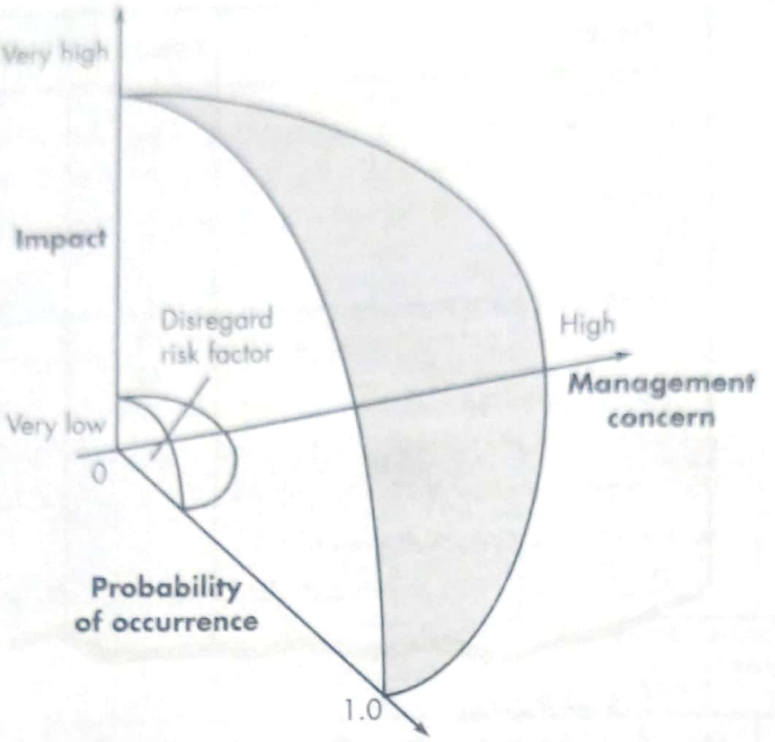
KEY POINT
A risk table is sorted by probability and impact to rank risks.

Once the first four columns of the risk table have been completed, the table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.

You can study the resultant sorted table and define a cutoff line. The *cutoff line* (drawn horizontally at some point in the table) implies that only risks that lie above the line will be given further attention. Risks that fall below the line are reevaluated to accomplish second-order prioritization. Referring to Figure 28.3, risk impact and

3 A weighted average can be used if one risk component has more significance for a project.

FIGURE 28.3 Risk and management concern



probability have a distinct influence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

All risks that lie above the cutoff line should be managed. The column labeled RMMM contains a pointer into a *risk mitigation, monitoring, and management plan* or, alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff. The RMMM plan and risk information sheets are discussed in Sections 28.5 and 28.6.

Quote:

"[Today,] no one has the luxury of getting to know a task so well that it holds no surprises, and surprises mean risk."

Stephen Grey

Risk probability can be determined by making individual estimates and then developing a single consensus value. Although that approach is workable, more sophisticated techniques for determining risk probability have been developed [AFC88]. Risk drivers can be assessed on a qualitative probability scale that has the following values: impossible, improbable, probable, and frequent. Mathematical probability can then be associated with each qualitative value (e.g., a probability of 0.7 to 0.99 implies a highly probable risk).

28.4.2 Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs. For example, a poorly defined external interface to customer hardware

(a technical risk) will preclude early design and testing and will likely lead to system integration problems late in a project. The scope of a risk combines the severity (just how serious is it?) with its overall distribution (how much of the project will be affected or how many stakeholders are harmed?). Finally, the timing of a risk considers when and for how long the impact will be felt. In most cases, you want the "bad news" to occur as soon as possible, but in some cases, the longer the delay, the better.

? How do we assess the consequences of a risk?

Returning once more to the risk analysis approach proposed by the U.S. Air Force [AFC88], you can apply the following steps to determine the overall consequences of a risk: (1) determine the average probability of occurrence value for each risk component; (2) using Figure 28.1, determine the impact for each component based on the criteria shown, and (3) complete the risk table and analyze the results as described in the preceding sections.

The overall *risk exposure* RE is determined using the following relationship [Hal98]:

$$\text{RE} = P \times C$$

RE = P x C

where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

For example, assume that the software team defines a project risk in the following manner:

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. 80 percent (likely).

Risk impact. Sixty reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be $18 \times 100 \times 14 = \$25{,}200$.

Risk exposure. RE = $0.80 \times 25{,}200 \sim \$20{,}200$.

ADVICE

Compare RE for all risks to the cost estimate for the project. If RE is greater than 50 percent of the project cost, the viability of the project must be evaluated.

Risk exposure can be computed for each risk in the risk table, once an estimate of the cost of the risk is made. The total risk exposure for all risks (above the cutoff in the risk table) can provide a means for adjusting the final cost estimate for a project. It can also be used to predict the probable increase in staff resources required at various points during the project schedule.

The risk projection and analysis techniques described in Sections 28.4.1 and 28.4.2 are applied iteratively as the software project proceeds. The project team

should revisit the risk table at regular intervals, reevaluating each risk to determine when new circumstances cause its probability and impact to change. As a consequence of this activity, it may be necessary to add new risks to the table, remove some risks that are no longer relevant, and change the relative positions of still others.

SafeHome

Risk Analysis

The scene: Doug Miller's office prior to the initiation of the *SafeHome* software project.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: I'd like to spend some time brainstorming risks for the *SafeHome* project.

Jamie: As in what can go wrong?

Doug: Yep. Here are a few categories where things can go wrong. [He shows everyone the categories noted in the introduction to Section 28.3.]

Vinod: Umm . . . do you want us to just call them out, or . . .

Doug: No here's what I thought we'd do. Everyone make a list of risks . . . right now . . ."

[Ten minutes pass, everyone is writing.]

Doug: Okay, stop.

Jamie: But I'm not done!

Doug: That's okay. We'll revisit the list again. Now, for each item on your list, assign a percent likelihood that the risk will occur. Then, assign an impact to the project on a scale of 1 (minor) to 5 (catastrophic).

Vinod: So if I think that the risk is a coin flip, I specify a 50 percent likelihood, and if I think it'll have a moderate project impact, I specify a 3, right?

Doug: Exactly.

[Five minutes pass, everyone is writing.]

Doug: Okay, stop. Now we'll make a group list on the white board. I'll do the writing; we'll call out one entry from your list in round-robin format.

[Fifteen minutes pass; the list is created.]

Jamie (pointing at the board and laughing): Vinod, that risk (pointing toward an entry on the board) is ridiculous. There's a higher likelihood that we'll all get hit by lightning. We should remove it.

Doug: No, let's leave it for now. We consider all risks, no matter how weird. Later we'll winnow the list.

Jamie: But we already have over 40 risks . . . how on earth can we manage them all?

Doug: We can't. That's why we'll define a cut-off after we sort these guys. I'll do that off-line and we'll meet again tomorrow. For now, get back to work . . . and in your spare time, think about any risks that we've missed.

28.5 Risk Refinement

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

? What's a good way to describe a risk?

One way to do this is to represent the risk in *condition-transition-consequence* (CTC) format [Glu94]. That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted in Section 28.4.2, you could write:

Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

This general condition can be refined in the following manner:

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

The consequences associated with these refined subconditions remain the same (i.e., 30 percent of software components must be custom engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

28.6 Risk Mitigation, Monitoring, and Management

Quote:

"If I take so many precautions, it is because I leave nothing to chance."

Napoleon

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation*. For example, assume that high staff turnover is noted as a project risk r_1. Based on past history and management intuition, the likelihood l_1 of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact x_1 is projected as critical. That is, high turnover will have a critical impact on project cost and schedule.

To mitigate this risk, you would develop a strategy for reducing turnover. Among the possible steps to be taken are:

? What can we do to mitigate a risk?

- Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under your control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.

- Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.
- Conduct peer reviews of all work (so that more than one person is "up to speed").
- Assign a backup staff member for every critical technologist.

As the project proceeds, *risk monitoring* activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the general attitude of team members based on project pressures, the degree to which the team has jelled, interpersonal relationships among team members, potential problems with compensation and benefits, and the availability of jobs within the company and outside it are all monitored.

In addition to monitoring these factors, a project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of work product standards and mechanisms to be sure that work products are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor work products carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well under way and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, you can temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to "get up to speed." Those individuals who are leaving are asked to stop all work and spend their last weeks in "knowledge transfer mode." This might include video-based knowledge capture, the development of "commentary documents or Wikis," and/or meeting with other team members who will remain on the project.

If RE for a specific risk is less than the cost of risk mitigation, don't try to mitigate the risk but continue to monitor it.

It is important to note that risk mitigation, monitoring, and management (RMMM) steps incur additional project cost. For example, spending the time to back up every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, you perform a classic cost-benefit analysis. If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is "backup," management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent, management will likely put all into place.

For a large project, 30 or 40 risks may be identified. If between three and seven risk management steps are identified for each, risk management may become a project in itself! For this reason, you should adapt the Pareto 80-20 rule to software risk. Experience indicates that 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help you to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure). For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don't fall into the critical 20 percent (the risks with highest project priority).

Risk is not limited to the software project itself. Risks can occur after the software has been successfully developed and delivered to the customer. These risks are typically associated with the consequences of software failure in the field.

Software safety and hazard analysis (e.g., [Dun02], [Her00], [Lev95]) are software quality assurance activities (Chapter 16) that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

28.7 THE RMMM PLAN

A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate *risk mitigation, monitoring, and management plan* (RMMM). The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.

Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a *risk information sheet* (RIS) [Wil97]. In most cases, the RIS is maintained using a database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily. The format of the RIS is illustrated in Figure 28.4.

Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. As I have already discussed, risk mitigation is a problem avoidance activity. Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to collect information that can be used for future risk analysis. In many cases, the problems that occur during a project can be traced to more than one risk. Another job of risk monitoring is to attempt to allocate origin [what risk(s) caused which problems throughout the project].

FIGURE 28.4 Risk information sheet. Source: [MIN97].

Risk information sheet			
Risk ID: P02-4-32	Date: 5/9/09	Prob: 80%	Impact: high

Description:
Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Refinement/context:
Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards.
Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.
Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.

Mitigation/monitoring:
1. Contact third party to determine conformance with design standards.
2. Press for interface standards completion; consider component structure when deciding an interface protocol.
3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.

Management/contingency plan/trigger:
RE computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly.
Trigger: Mitigation steps unproductive as of 7/1/09.

Current status:
5/12/09: Mitigation steps initiated.

Originator: D. Gagne	Assigned: B. Laster

SOFTWARE TOOLS

Risk Management

Objective: The objective of risk management tools is to assist a project team in defining risks, assessing their impact and probability, and tracking risks throughout a software project.

Mechanics: In general, risk management tools assist in generic risk identification by providing a list of typical project and business risks, provide checklists or other "interview" techniques that assist in identifying project specific risks, assign probability and impact to each risk, support risk mitigation strategies, and generate many different risk-related reports.

Representative Tools:[4]

@risk, developed by Palisade Corporation (**www.palisade.com**), is a generic risk analysis tool that uses Monte Carlo simulation to drive its analytical engine.

Riskman, distributed by ABS Consulting (**www.absconsulting.com/riskmansoftware/index.html**), is a risk evaluation expert system that identifies project-related risks.

Risk Radar, developed by SPMN (**www.spmn.com**), assists project managers in identifying and managing project risks.

4 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

22.1 Software Configuration Management

The output of the software process is information that may be divided into three broad categories: (1) computer programs (both source level and executable forms), (2) work products that describe the computer programs (targeted at various stakeholders), and (3) data or content (contained within the program or external to it). The items that comprise all information produced as part of the software process are collectively called a *software configuration*.

As software engineering work progresses, a hierarchy of *software configuration items* (SCIs)—a named element of information that can be as small as a single UML diagram or as large as the complete design document—is created. If each SCI simply led to other SCIs, little confusion would result. Unfortunately, another variable enters the process—*change*. Change may occur at any time, for any reason. In fact, the *First Law of System Engineering* [Ber80] states: "No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle."

Note:
"There is nothing permanent except change."
Heraclitus, 500 B.C.

What is the origin of these changes? The answer to this question is as varied as the changes themselves. However, there are four fundamental sources of change:

- New business or market conditions dictate changes in product requirements or business rules.
- New stakeholder needs demand modification of data produced by information systems, functionality delivered by products, or services delivered by a computer-based system.
- Reorganization or business growth/downsizing causes changes in project priorities or software engineering team structure.
- Budgetary or scheduling constraints cause a redefinition of the system or product.

? What is the origin of changes that are requested for software?

Software configuration management is a set of activities that have been developed to manage change throughout the life cycle of computer software. SCM can be viewed as a software quality assurance activity that is applied throughout the software process. In the sections that follow, I describe major SCM tasks and important concepts that can help you to manage change.

22.1.1 An SCM Scenario[1]

A typical CM operational scenario involves a project manager who is in charge of a software group, a configuration manager who is in charge of the CM procedures and policies, the software engineers who are responsible for developing and maintaining the software product, and the customer who uses the product. In the scenario, assume that the product is a small one involving about 15,000 lines of code being developed by a team of six people. (Note that other scenarios of smaller or larger teams are possible, but, in essence, there are generic issues that each of these projects face concerning CM.)

? What are the goals of and the activities performed by each of the constituencies involved in change management?

At the operational level, the scenario involves various roles and tasks. For the project manager, the goal is to ensure that the product is developed within a certain time frame. Hence, the manager monitors the progress of development and recognizes and reacts to problems. This is done by generating and analyzing reports about the status of the software system and by performing reviews on the system.

The goals of the configuration manager are to ensure that procedures and policies for creating, changing, and testing of code are followed, as well as to make information about the project accessible. To implement techniques for maintaining control over code changes, this manager introduces mechanisms for making official requests for changes, for evaluating them (via a Change Control Board that is responsible for approving changes to the software system), and for authorizing changes. The manager creates and disseminates task lists for the engineers and basically creates the project context. Also, the manager collects statistics about components in the software system, such as information determining which components in the system are problematic.

KEY POINT

There must be a mechanism to ensure that simultaneous changes to the same component are properly tracked, managed, and executed.

For the software engineers, the goal is to work effectively. This means engineers do not unnecessarily interfere with each other in the creation and testing of code and in the production of supporting work products. But, at the same time, they try to communicate and coordinate efficiently. Specifically, engineers use tools that help build a consistent software product. They communicate and coordinate by notifying one another about tasks required and tasks completed. Changes are propagated across each other's work by merging files. Mechanisms exist to ensure that, for components that undergo simultaneous changes, there is some way of resolving conflicts and

1 This section is extracted from [Dar01]. Special permission to reproduce "Spectrum of Functionality in CM System" by Susan Dart [Dar01], © 2001 by Carnegie Mellon University is granted by the Software Engineering Institute.

merging changes. A history is kept of the evolution of all components of the system along with a log with reasons for changes and a record of what actually changed. The engineers have their own workspace for creating, changing, testing, and integrating code. At a certain point, the code is made into a baseline from which further development continues and from which variants for other target machines are made.

The customer uses the product. Since the product is under CM control, the customer follows formal procedures for requesting changes and for indicating bugs in the product.

Ideally, a CM system used in this scenario should support all these roles and tasks; that is, the roles determine the functionality required of a CM system. The project manager sees CM as an auditing mechanism; the configuration manager sees it as a controlling, tracking, and policy making mechanism; the software engineer sees it as a changing, building, and access control mechanism; and the customer sees it as a quality assurance mechanism.

22.1.2 Elements of a Configuration Management System

In her comprehensive white paper on software configuration management, Susan Dart [Dar01] identifies four important elements that should exist when a configuration management system is developed:

- *Component elements*—a set of tools coupled within a file management system (e.g., a database) that enables access to and management of each software configuration item.
- *Process elements*—a collection of actions and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.
- *Construction elements*—a set of tools that automate the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- *Human elements*—a set of tools and process features (encompassing other CM elements) used by the software team to implement effective SCM.

These elements (to be discussed in more detail in later sections) are not mutually exclusive. For example, component elements work in conjunction with construction elements as the software process evolves. Process elements guide many human activities that are related to SCM and might therefore be considered human elements as well.

22.1.3 Baselines

Change is a fact of life in software development. Customers want to modify requirements. Developers want to modify the technical approach. Managers want to modify the project strategy. Why all this modification? The answer is really quite simple.

ADVICE

Most software changes are justified, so there's no point in complaining about them. Rather, be certain that you have mechanisms in place to handle them.

As time passes, all constituencies know more (about what they need, which approach would be best, and how to get it done and still make money). This additional knowledge is the driving force behind most changes and leads to a statement of fact that is difficult for many software engineering practitioners to accept: *Most changes are justified!*

A *baseline* is a software configuration management concept that helps you to control change without seriously impeding justifiable change. The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:

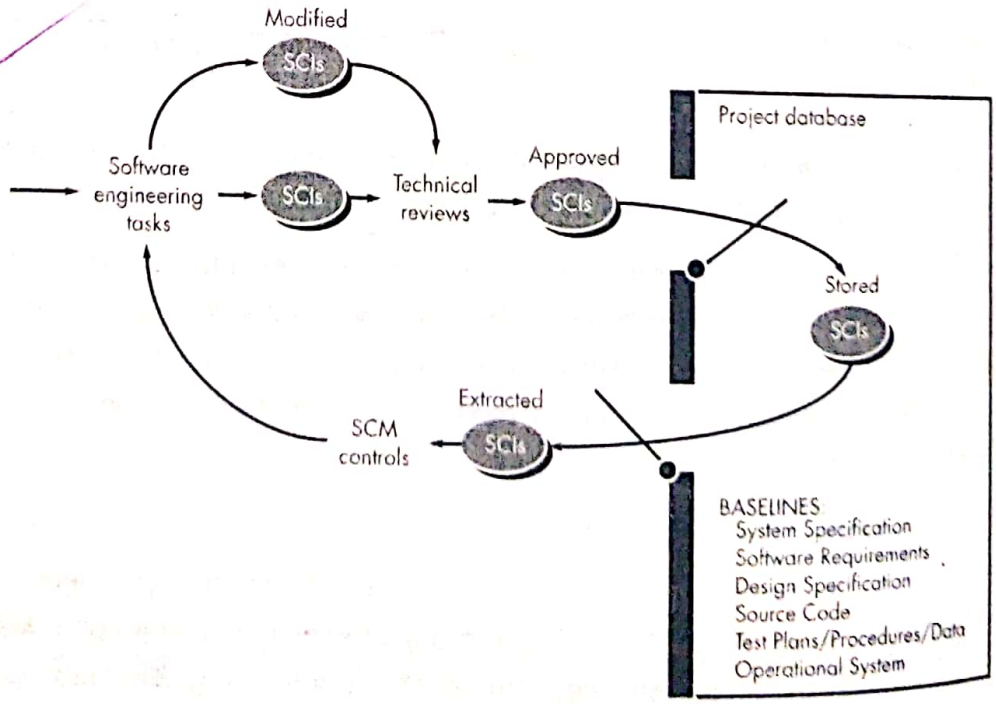
> A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.

Before a software configuration item becomes a baseline, changes may be made quickly and informally. However, once a baseline is established, changes can be made, but a specific, formal procedure must be applied to evaluate and verify each change.

In the context of software engineering, a baseline is a milestone in the development of software. A baseline is marked by the delivery of one or more software configuration items that have been approved as a consequence of a technical review (Chapter 15). For example, the elements of a design model have been documented and reviewed. Errors are found and corrected. Once all parts of the model have been reviewed, corrected, and then approved, the design model becomes a baseline. Further changes to the program architecture (documented in the design model) can be made only after each has been evaluated and approved. Although baselines can be defined at any level of detail, the most common software baselines are shown in Figure 22.1.

FIGURE 22.1

Baselined SCIs and the project database



The progression of events that lead to a baseline is also illustrated in Figure 22.1. Software engineering tasks produce one or more SCIs. After SCIs are reviewed and approved, they are placed in a *project database* (also called a *project library* or *software repository* and discussed in Section 22.2). When a member of a software engineering team wants to make a modification to a baselined SCI, it is copied from the project database into the engineer's private workspace. However, this extracted SCI can be modified only if SCM controls (discussed later in this chapter) are followed. The arrows in Figure 22.1 illustrate the modification path for a baselined SCI.

22.1.4 Software Configuration Items

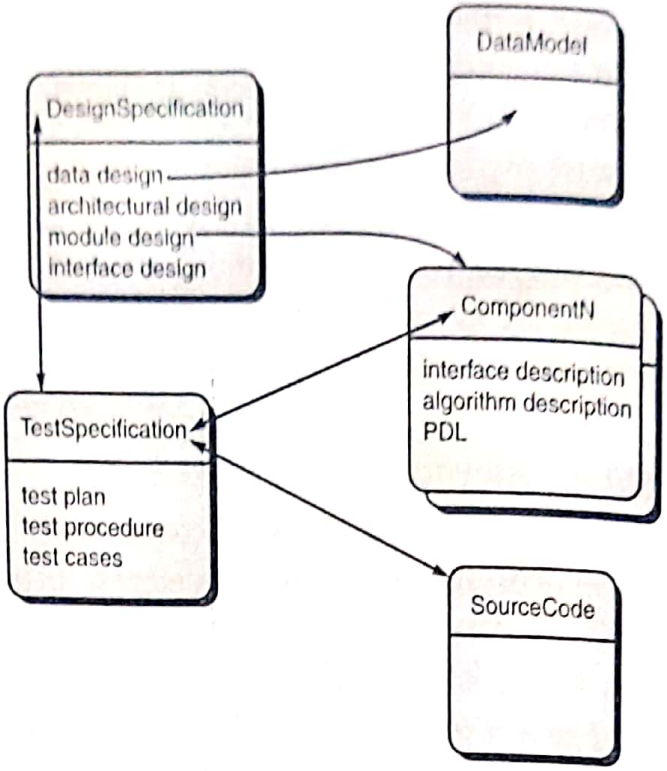
I have already defined a software configuration item as information that is created as part of the software engineering process. In the extreme, a SCI could be considered to be a single section of a large specification or one test case in a large suite of tests. More realistically, an SCI is all or part of a work product (e.g., a document, an entire suite of test cases, or a named program component).

In addition to the SCIs that are derived from software work products, many software engineering organizations also place software tools under configuration control. That is, specific versions of editors, compilers, browsers, and other automated tools are "frozen" as part of the software configuration. Because these tools were used to produce documentation, source code, and data, they must be available when changes to the software configuration are to be made. Although problems are rare, it is possible that a new version of a tool (e.g., a compiler) might produce different results than the original version. For this reason, tools, like the software that they help to produce, can be baselined as part of a comprehensive configuration management process.

In reality, SCIs are organized to form configuration objects that may be cataloged in the project database with a single name. A *configuration object* has a name, attributes, and is "connected" to other objects by relationships. Referring to Figure 22.2, the configuration objects, **DesignSpecification, DataModel, ComponentN, SourceCode,** and **TestSpecification** are each defined separately. However, each of the objects is related to the others as shown by the arrows. A curved arrow indicates a compositional relation. That is, **DataModel** and **ComponentN** are part of the object **DesignSpecification.** A double-headed straight arrow indicates an interrelationship. If a change were made to the **SourceCode** object, the interrelationships enable you to determine what other objects (and SCIs) might be affected.[2]

2 These relationships are defined within the database. The structure of the database (repository) is discussed in greater detail in Section 22.2.

FIGURE 22.2
Configuration objects



22.2 THE SCM REPOSITORY

In the early days of software engineering, software configuration items were maintained as paper documents (or punched computer cards!), placed in file folders or three-ring binders, and stored in metal cabinets. This approach was problematic for many reasons: (1) finding a configuration item when it was needed was often difficult, (2) determining which items were changed, when and by whom was often challenging, (3) constructing a new version of an existing program was time consuming and error prone, and (4) describing detailed or complex relationships between configuration items was virtually impossible.

Today, SCIs are maintained in a project database or repository. *Webster's Dictionary* defines the word *repository* as "any thing or person thought of as a center of accumulation or storage." During the early history of software engineering, the repository was indeed a person—the programmer who had to remember the location of all information relevant to a software project, who had to recall information that was never written down and reconstruct information that had been lost. Sadly, using a person as "the center for accumulation and storage" (although it conforms to Webster's definition) does not work very well. Today, the repository is a "thing"—a database that acts as the center for both accumulation and storage of software engineering information. The role of the person (the software engineer) is to interact with the repository using tools that are integrated with it.

22.2.1 The Role of the Repository

The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner. It provides the obvious

functions of a modern database management system by ensuring data integrity, sharing, and integration. In addition, the SCM repository provides a hub for the integration of software tools, is central to the flow of the software process, and can enforce uniform structure and format for software engineering work products.

To achieve these capabilities, the repository is defined in terms of a meta-model. The *meta-model* determines how information is stored in the repository, how data can be accessed by tools and viewed by software engineers, how well data security and integrity can be maintained, and how easily the existing model can be extended to accommodate new needs.

22.2.2 General Features and Content

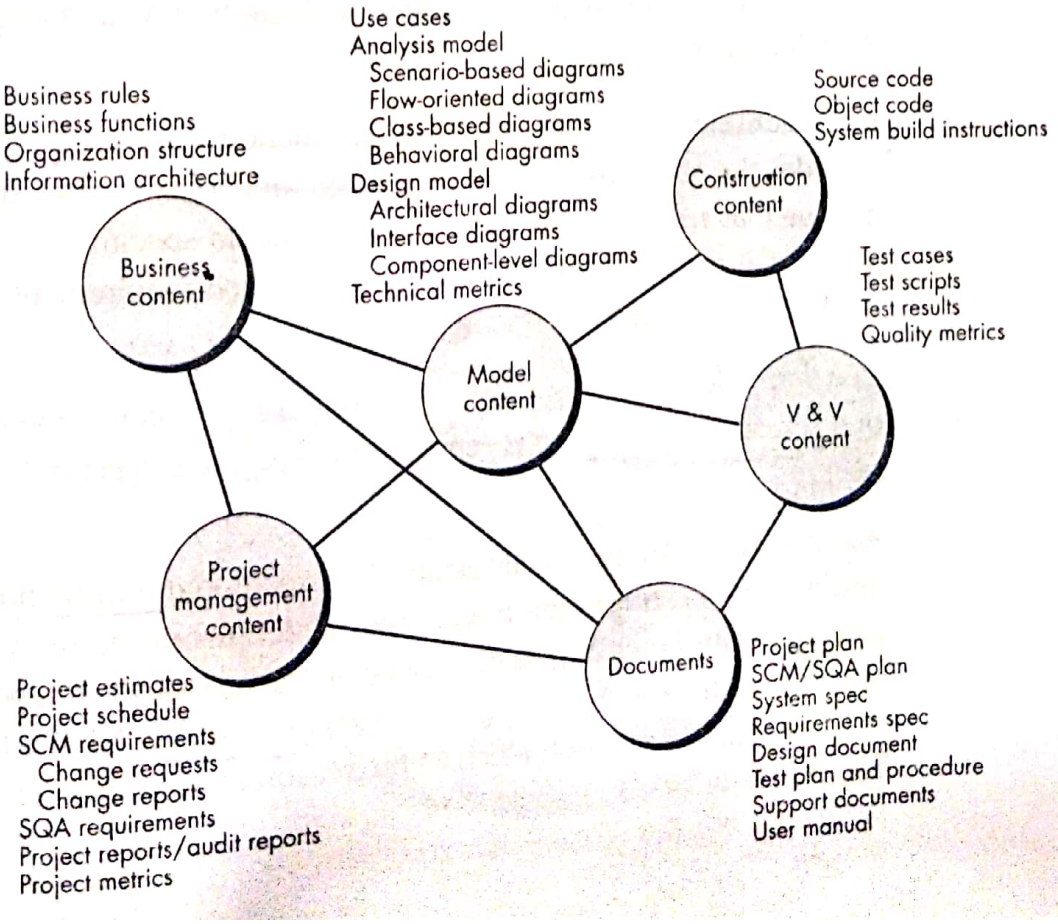
The features and content of the repository are best understood by looking at it from two perspectives: what is to be stored in the repository and what specific services are provided by the repository. A detailed breakdown of types of representations, documents, and other work products that are stored in the repository is presented in Figure 22.3.

WebRef
An example of a commercially available repository can be obtained at www.oracle.com/technology/products/repository/index.html

A robust repository provides two different classes of services: (1) the same types of services that might be expected from any sophisticated database management system and (2) services that are specific to the software engineering environment.

A repository that serves a software engineering team should also (1) integrate with or directly support process management functions, (2) support specific rules that govern the SCM function and the data maintained within the repository, (3) provide an interface to other software engineering tools, and (4) accommodate storage of sophisticated data objects (e.g., text, graphics, video, audio).

FIGURE 22.3 Content of the repository



22.2.3 SCM Features

KEY POINT

The repository must be capable of maintaining SCIs related to many different versions of the software. More important, it must provide the mechanisms for assembling these SCIs into a version-specific configuration.

To support SCM, the repository must have a tool set that provides support for the following features:

Versioning. As a project progresses, many versions (Section 22.3.2) of individual work products will be created. The repository must be able to save all of these versions to enable effective management of product releases and to permit developers to go back to previous versions during testing and debugging.

The repository must be able to control a wide variety of object types, including text, graphics, bit maps, complex documents, and unique objects like screen and report definitions, object files, test data, and results. A mature repository tracks versions of objects with arbitrary levels of granularity; for example, a single data definition or a cluster of modules can be tracked.

Dependency tracking and change management. The repository manages a wide variety of relationships among the data elements stored in it. These include relationships between enterprise entities and processes, among the parts of an application design, between design components and the enterprise information architecture, between design elements and deliverables, and so on. Some of these relationships are merely associations, and some are dependencies or mandatory relationships.

The ability to keep track of all of these relationships is crucial to the integrity of the information stored in the repository and to the generation of deliverables based on it, and it is one of the most important contributions of the repository concept to the improvement of the software process. For example, if a UML class diagram is modified, the repository can detect whether related classes, interface descriptions, and code components also require modification and can bring affected SCIs to the developer's attention.

Requirements tracing. This special function depends on link management and provides the ability to track all the design and construction components and deliverables that result from a specific requirements specification (forward tracing). In addition, it provides the ability to identify which requirement generated any given work product (backward tracing).

Configuration management. A configuration management facility keeps track of a series of configurations representing specific project milestones or production releases.

Audit trails. An audit trail establishes additional information about when, why, and by whom changes are made. Information about the source of changes can be entered as attributes of specific objects in the repository. A repository trigger mechanism is helpful for prompting the developer or the tool that is being used to initiate entry of audit information (such as the reason for a change) whenever a design element is modified.

22.3 THE SCM PROCESS

Quote:
"Any change, even a change for the better, is accompanied by drawbacks and discomforts."
Arnold Bennett

The software configuration management process defines a series of tasks that have four primary objectives: (1) to identify all items that collectively define the software configuration, (2) to manage changes to one or more of these items, (3) to facilitate the construction of different versions of an application, and (4) to ensure that software quality is maintained as the configuration evolves over time.

A process that achieves these objectives need not be bureaucratic or ponderous, but it must be characterized in a manner that enables a software team to develop answers to a set of complex questions:

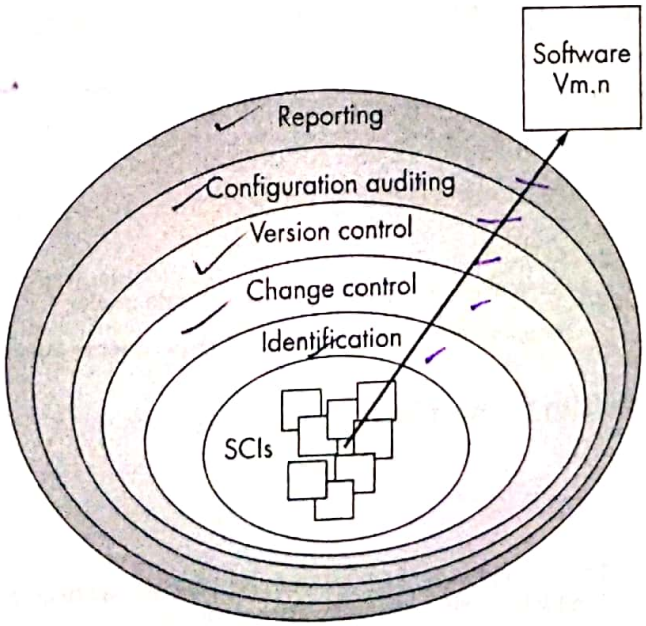
? What questions should the SCM process be designed to answer?

- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and ranking requested changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to apprise others of changes that are made?

These questions lead to the definition of five SCM tasks—identification, version control, change control, configuration auditing, and reporting—illustrated in Figure 22.4.

Referring to the figure, SCM tasks can viewed as concentric layers. SCIs flow outward through these layers throughout their useful life, ultimately becoming part

FIGURE 22.4 Layers of the SCM process



of the software configuration of one or more versions of an application or system. As an SCI moves through a layer, the actions implied by each SCM task may or may not be applicable. For example, when a new SCI is created, it must be identified. However, if no changes are requested for the SCI, the change control layer does not apply. The SCI is assigned to a specific version of the software (version control mechanisms come into play). A record of the SCI (its name, creation date, version designation, etc.) is maintained for configuration auditing purposes and reported to those with a need to know. In the sections that follow, we examine each of these SCM process layers in more detail.

22.3.1 Identification of Objects in the Software Configuration

To control and manage software configuration items, each should be separately named and then organized using an object-oriented approach. Two types of objects can be identified [Cho89]: basic objects and aggregate objects.[3] A *basic object* is a unit of information that you create during analysis, design, code, or test. For example, a basic object might be a section of a requirements specification, part of a design model, source code for a component, or a suite of test cases that are used to exercise the code. An *aggregate object* is a collection of basic objects and other aggregate objects. For example, a **DesignSpecification** is an aggregate object. Conceptually, it can be viewed as a named (identified) list of pointers that specify aggregate objects such as **ArchitecturalModel** and **DataModel,** and *basic objects* such as **ComponentN** and **UMLClassDiagramN**.

KEY POINT

The interrelationships established for configuration objects allow you to assess the impact of change.

Each object has a set of distinct features that identify it uniquely: a name, a description, a list of resources, and a "realization." The object name is a character string that identifies the object unambiguously. The object description is a list of data items that identify the SCI type (e.g., model element, program, data) represented by the object, a project identifier, and change and/or version information. Resources are "entities that are provided, processed, referenced or otherwise required by the object" [Cho89]. For example, data types, specific functions, or even variable names may be considered to be object resources. The realization is a pointer to the "unit of text" for a basic object and null for an aggregate object.

Configuration object identification can also consider the relationships that exist between named objects. For example, using the simple notation:

Class diagram <part-of> requirements model;
Requirements model <part-of> requirements specification;

you can create a hierarchy of SCIs.

3 The concept of an aggregate object [Gus89] has been proposed as a mechanism for representing a complete version of a software configuration.

...f the project ...ose provides the ...h to establish ...se relationships, ...n are time consum- ...e to establish and ...ult to keep up-to- ...e, although very ...ful for impact ...ysis, they are not ...ired for overall ...ware management.

In many cases, objects are interrelated across branches of the object hierarchy. These cross-structural relationships can be represented in the following manner:

DataModel <interrelated> DataFlowModel
DataModel <interrelated> TestCaseClassM

In the first case, the interrelationship is between a composite object, while the second relationship is between an aggregate object (**DataModel**) and a basic object (**TestCaseClassM**).

The identification scheme for software objects must recognize that objects evolve throughout the software process. Before an object is baselined, it may change many times, and even after a baseline has been established, changes may be quite frequent.

22.3.2 Version Control

Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process. A version control system implements or is directly integrated with four major capabilities: (1) a project database (repository) that stores all relevant configuration objects, (2) a *version management* capability that stores all versions of a configuration object (or enables any version to be constructed using differences from past versions), (3) a *make facility* that enables you to collect all relevant configuration objects and construct a specific version of the software. In addition, version control and change control systems often implement an *issues tracking* (also called *bug tracking*) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

A number of version control systems establish a *change set*—a collection of all changes (to some baseline configuration) that are required to create a specific version of the software. Dart [Dar91] notes that a change set "captures all changes to all files in the configuration along with the reason for changes and details of who made the changes and when."

A number of named change sets can be identified for an application or system. This enables you to construct a version of the software by specifying the change sets (by name) that must be applied to the baseline configuration. To accomplish this, a *system modeling* approach is applied. The system model contains: (1) a *template* that includes a component hierarchy and a "build order" for the components that describes how the system must be constructed, (2) construction rules, and (3) verification rules.[4]

A number of different automated approaches to version control have been proposed over the last few decades. The primary difference in approaches is the sophistication of the attributes that are used to construct specific versions and variants of a system and the mechanics of the process for construction.

4 It is also possible to query the system model to assess how a change in one component will impact other components.

SOFTWARE TOOLS

The Concurrent Versions System (CVS)

The use of tools to achieve version control is essential for effective change management. The *Concurrent Versions System* (CVS) is a widely used tool for version control. Originally designed for source code, but useful for any text-based file, the CVS system (1) establishes a simple repository, (2) maintains all versions of a file in a single named file by storing only the differences between progressive versions of the original file, and (3) protects against simultaneous changes to a file by establishing different directories for each developer, thus insulating one from another. CVS merges changes when each developer completes her work.

It is important to note that CVS is not a "build" system; that is, it does not construct a specific version of the software. Other tools (e.g., *Makefile*) must be integrated with CVS to accomplish this. CVS does not implement a change control process (e.g., change requests, change reports, bug tracking).

Even with these limitations, CVS "is a dominant open-source network-transparent version control system [that] is useful for everyone from individual developers to large, distributed teams" [CVS07]. Its client-server architecture allows users to access files via Internet connections, and its open-source philosophy makes it available on most popular platforms.

CVS is available at no cost for Windows, Mac OS, LINUX, and UNIX environments. See [CVS07] for further details.

22.3.3 Change Control

Quote:

"The art of progress is to preserve order amid change and to preserve change amid order."

Alfred North Whitehead

The reality of change control in a modern software engineering context has been summed up beautifully by James Bach [Bac98]:

> Change control is vital. But the forces that make it necessary also make it annoying. We worry about change because a tiny perturbation in the code can create a big failure in the product. But it can also fix a big failure or enable wonderful new capabilities. We worry about change because a single rogue developer could sink the project; yet brilliant ideas originate in the minds of those rogues, and a burdensome change control process could effectively discourage them from doing creative work.

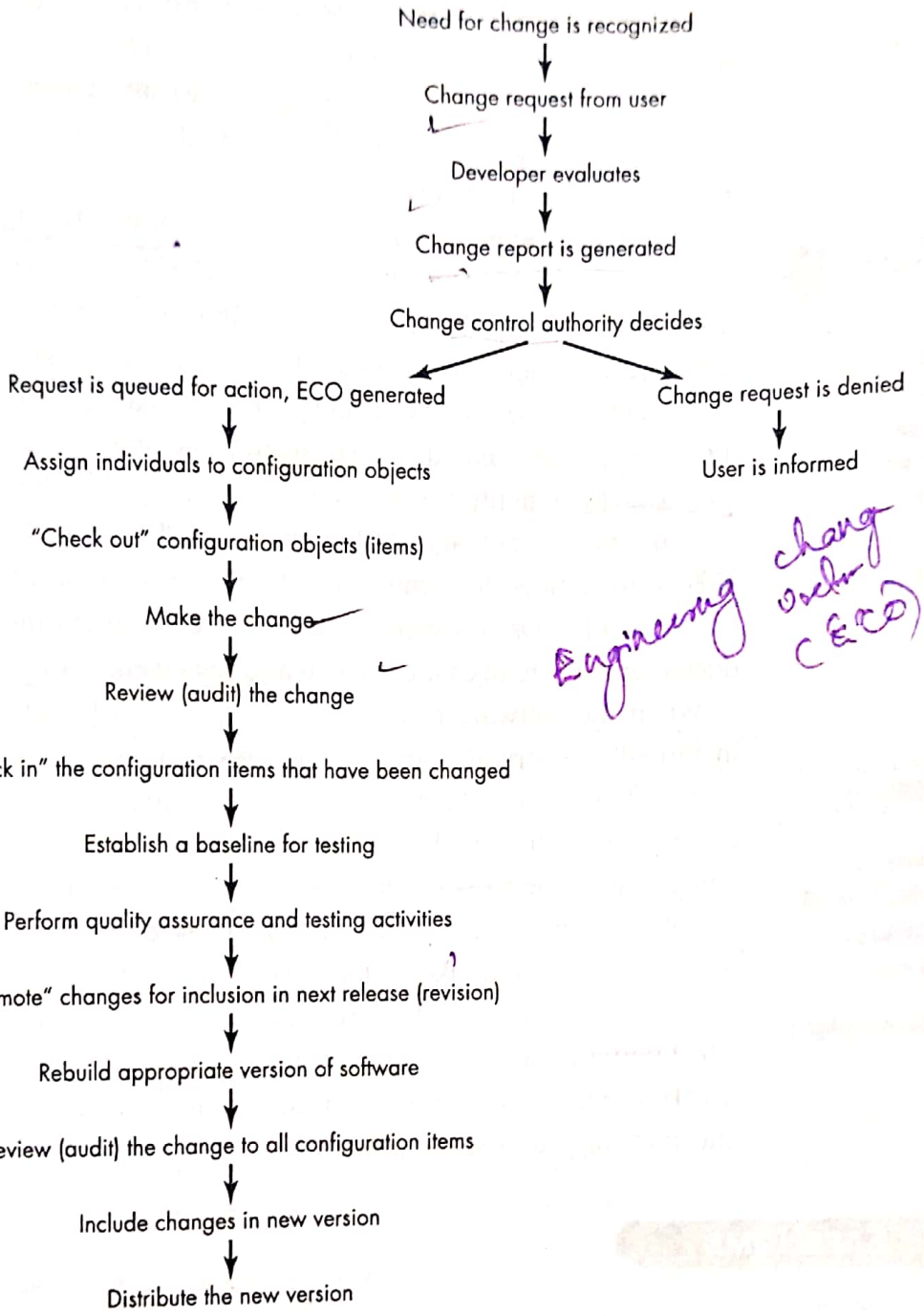
Bach recognizes that we face a balancing act. Too much change control and we create problems. Too little, and we create other problems.

KEY POINT

It should be noted that a number of change requests may be combined to result in a single ECO and that ECOs typically result in changes to multiple configuration objects.

For a large software project, uncontrolled change rapidly leads to chaos. For such projects, change control combines human procedures and automated tools to provide a mechanism for the control of change. The change control process is illustrated schematically in Figure 22.5. A *change request* is submitted and evaluated to assess technical merit, potential side effects, overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a *change report*, which is used by a *change control authority* (CCA)—a person or group that makes a final decision on the status and priority of the change. An *engineering change order* (ECO) is generated for each approved change. The ECO describes the change to be made, the constraints that must be respected, and the criteria for review and audit.

The object(s) to be changed can be placed in a directory that is controlled solely by the software engineer making the change. A version control system (see the CVS sidebar) updates the original file once the change has been made. As an alternative,



the object(s) to be changed can be "checked out" of the project database (repository), the change is made, and appropriate SQA activities are applied. The object(s) is (are) then "checked in" to the database and appropriate version control mechanisms (Section 22.3.2) are used to create the next version of the software.

These version control mechanisms, integrated within the change control process, implement two important elements of change management—access control and synchronization control. *Access control* governs which software engineers have the authority to access and modify a particular configuration object. *Synchronization control* helps to ensure that parallel changes, performed by two different people, don't overwrite one another.

You may feel uncomfortable with the level of bureaucracy implied by the change control process description shown in Figure 22.5. This feeling is not uncommon. Without proper safeguards, change control can retard progress and create unnecessary red tape. Most software developers who have change control mechanisms (unfortunately, many have none) have created a number of layers of control to help avoid the problems alluded to here.

ADVICE

Opt for a bit more change control than you think you'll need. It's likely that too much will be the right amount.

Prior to an SCI becoming a baseline, only *informal change control* need be applied. The developer of the configuration object (SCI) in question may make whatever changes are justified by project and technical requirements (as long as changes do not affect broader system requirements that lie outside the developer's scope of work). Once the object has undergone technical review and has been approved, a baseline can be created.[5] Once an SCI becomes a baseline, *project level change control* is implemented. Now, to make a change, the developer must gain approval from the project manager (if the change is "local") or from the CCA if the change affects other SCIs. In some cases, formal generation of change requests, change reports, and ECOs is dispensed with. However, assessment of each change is conducted and all changes are tracked and reviewed.

When the software product is released to customers, *formal change control* is instituted. The formal change control procedure has been outlined in Figure 22.5.

Quote:

"Change is inevitable, except for vending machines."

Bumper sticker

The change control authority plays an active role in the second and third layers of control. Depending on the size and character of a software project, the CCA may be composed of one person—the project manager—or a number of people (e.g., representatives from software, hardware, database engineering, support, marketing). The role of the CCA is to take a global view, that is, to assess the impact of change beyond the SCI in question. How will the change affect hardware? How will the change affect performance? How will the change modify customers' perception of the product? How will the change affect product quality and reliability? These and many other questions are addressed by the CCA.

SafeHome

SCM Issues

The scene: Doug Miller's office as the *SafeHome* software project begins.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: I know it's early, but we've got to talk about change management.

Vinod (laughing): Hardly. Marketing called this morning with a few "second thoughts." Nothing major, but it's just the beginning.

5 A baseline can be created for other reasons as well. For example, when "daily builds" are created, all components checked in by a given time become the baseline for the next day's work.

ie: We've been pretty informal about change ement on past projects.

g: I know, but this is bigger and more visible, and ecall.

ed (nodding): We got killed by uncontrolled ges on the home lighting control project . . . remember things that

g (frowning): A nightmare that I'd prefer not to

ie: So what do we do?

g: As I see it, three things. First we have to elop—or borrow—a change control process.

ie: You mean how people request changes?

Vinod: Yeah, but also how we evaluate the change, decide when to do it (if that's what we decide), and how we keep records of what's affected by the change.

Doug: Second, we've got to get a really good SCM tool for change and version control.

Jamie: We can build a database for all of our work products.

Vinod: They're called SCIs in this context, and most good tools provide some support for that.

Doug: That's a good start, now we have to . . .

Jamie: Uh, Doug, you said there were three things . . .

Doug (smiling): Third—we've all got to commit to follow the change management process and use the tools—no matter what, okay?

22.3.4 Configuration Audit

Identification, version control, and change control help you to maintain order in what would otherwise be a chaotic and fluid situation. However, even the most successful control mechanisms track a change only until an ECO is generated. How can a software team ensure that the change has been properly implemented? The answer is twofold: (1) technical reviews and (2) the software configuration audit.

The technical review (Chapter 15) focuses on the technical correctness of the configuration object that has been modified. The reviewers assess the SCI to determine consistency with other SCIs, omissions, or potential side effects. A technical review should be conducted for all but the most trivial changes.

A *software configuration audit* complements the technical review by assessing a configuration object for characteristics that are generally not considered during review. The audit asks and answers the following questions:

hat are the imary ns that we ing a uration

1. Has the change specified in the ECO been made? Have any additional modifications been incorporated?
2. Has a technical review been conducted to assess technical correctness?
3. Has the software process been followed and have software engineering standards been properly applied?
4. Has the change been "highlighted" in the SCI? Have the change date and change author been specified? Do the attributes of the configuration object reflect the change?
5. Have SCM procedures for noting the change, recording it, and reporting it been followed?
6. Have all related SCIs been properly updated?

In some cases, the audit questions are asked as part of a technical review. However, when SCM is a formal activity, the configuration audit is conducted separately by the quality assurance group. Such formal configuration audits also ensure that the correct SCIs (by version) have been incorporated into a specific build and that all documentation is up-to-date and consistent with the version that has been built.

22.3.5 Status Reporting

Configuration status reporting (sometimes called *status accounting*) is an SCM task that answers the following questions: (1) What happened? (2) Who did it? (3) When did it happen? (4) What else will be affected?

ADVICE

Develop a "need to know" list for every configuration object and keep it up-to-date. When a change is made, be sure that everyone on the list is notified.

The flow of information for configuration status reporting (CSR) is illustrated in Figure 22.5. Each time an SCI is assigned new or updated identification, a CSR entry is made. Each time a change is approved by the CCA (i.e., an ECO is issued), a CSR entry is made. Each time a configuration audit is conducted, the results are reported as part of the CSR task. Output from CSR may be placed in an online database or website, so that software developers or support staff can access change information by keyword category. In addition, a CSR report is generated on a regular basis and is intended to keep management and practitioners appraised of important changes.

SOFTWARE TOOLS

SCM Support

Objective: SCM tools provide support to one or more of the process activities discussed in Section 22.3.

Mechanics: Most modern SCM tools work in conjunction with a repository (a database system) and provide mechanisms for identification, version and change control, auditing, and reporting.

Representative Tools:[6]

CCC/Harvest, distributed by Computer Associates (**www.cai.com**), is a multiplatform SCM system.

ClearCase, developed by Rational, provides a family of SCM functions (**www-306.ibm.com/software/awdtools/clearcase/index.html**).

Serena ChangeMan ZMF, distributed by Serena (**www.serena.com/US/products/zmf/index.aspx**), provides a full set of SCM tools that are applicable for both conventional software and WebApps.

SourceForge, distributed by VA Software (**sourceforge.net**), provides version management, build capabilities, issue/bug tracking, and many other management features.

SurroundSCM, developed by Seapine Software, provides complete change management capabilities (**www.seapine.com**).

Vesta, distributed by Compac, is a public domain SCM system that can support both small (<10 KLOC) and large (10,000 KLOC) projects (**www.vestasys.org**).

A comprehensive list of commercial SCM tools and environments can be found at **www.cmtoday.com/yp/commercial.html**.

6 Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.