Chapter 6

Subject Matter Expert in Different Fields

6.1 Operations management: Special topic: supply chain management[1]

Supply chain management is the business function that coordinates and manages all the activities of the supply chain, including suppliers of raw materials, components and services, transportation providers, internal departments, and information systems. Exhibit 6.1 illustrates a supply chain for providing packaged milk to consumers.

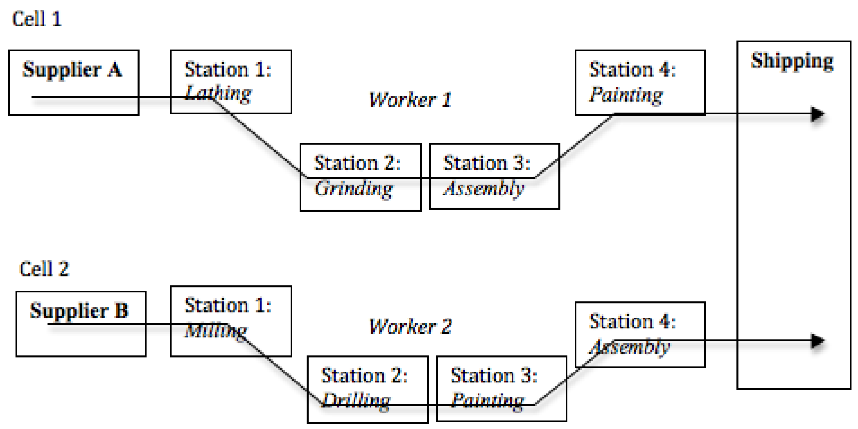


Exhibit 6.1: Illustration of a supply chain

In the manufacturing sector, supply chain management addresses the movement of goods through the supply chain from the supplier to the manufacturer, to wholesalers or warehouse distribution centers, to retailers and finally to the consumer. For example, Apple, Inc uses sophisticated information systems to

[1]This content is available online at <http://cnx.org/content/m35461/1.4/>.

accept orders for custom-built computers from individual customers all over the world. Apple assembles the computers in Shanghai, China, to the customers' specifications. It uses parts and components that are provided by outside suppliers who can deliver the right parts in the right quantity in a timely way to satisfy the immediate production schedule. The completed computers are flown from Shanghai by FedEx, reaching the end-user customers only a few days after the orders were placed. Apple's supply chain allows it to provide fast delivery of high-quality custom computers at competitive prices.

Supply chain concepts also apply to the service sector, where service firms must coordinate equipment, materials, and human resources to provide services to their customers in a timely manner. For example, a retail store that sells electronic products may contract with an outside business to provide installation services to its customers. In many cases, the customer does not even know the installation was done by an outside contractor. Information and communication technologies such as global positioning systems (GPS), barcode technology, customer relationship management (CRM) databases, and the Internet allow service businesses to coordinate external and internal service suppliers to efficiently and effectively respond to customer demand.

The supply chain is not just a one way process that runs from raw materials to the end customer. Although goods tend to flow this way, important data such as forecasts, inventory status, shipping schedules, and sales data are examples of information that is constantly being conveyed to different links in the supply chain. Money also tends to flow "upstream" in the supply chain so goods and service providers can be paid.

6.1.1 Bullwhip effect

A major goal in supply chain management strategy is to minimize the bullwhip effect. The bullwhip effect occurs when inaccurate or distorted information is passed on through the links in the supply chain. As the bad information gets passed from one party to the next, the distortions worsen and cause poor ordering decisions by upstream parties in the supply chain that have little apparent link to the final end-item product demand. As information gets farther from the end customer, the worse the quality of information gets as the supply chain members base their guesses on the bad guesses of their partners. The results are wasteful inventory investments, poor customer service, inefficient distribution, misused manufacturing capacity, and lost revenues for all parties in the supply chain.

For example, Open Range Jeans (a fictitious company) are sold in a popular retail store chain. The retail chain decides to promote Open Range Jeans and reduce the price to boost customer traffic in its stores, but the chain does not tell the Open Range manufacturer of this promotion plan. The manufacturer sees an increase in retail orders, forecasts a long-term growth in demand for its jeans, and places orders with its suppliers for more fabric, zippers, and dye.

Suppliers of fabric, zippers and dye see the increase in orders from the jeans manufacturer and boost their orders for raw cotton, chemicals, etc. Meanwhile, the retail chain has ended its Open Range promotion, and sales of the jeans plummet below normal levels because customers have stocked up to take advantage of the promotion prices. Just as end-customer demand falls, new jeans are being manufactured, and raw materials are being sent to the jeans factory. When the falling end-customer demand is finally realized, manufacturers rush to slash production, cancel orders, and discount inventories.

Not wanting to get burned twice, manufacturers wait until finished goods jean inventories are drawn down to minimal levels. When seasonal demand increases jeans purchases, the retail stores order more Open Range jeans, but the manufacturers cannot respond quickly enough. A stockout occurs at the retail store level just as customers are purchasing jeans during the back-to-school sales season. Retail customers respond to the stockout by purchasing the jeans of a major competitor, causing long-term damage to Open Range's market share.

6.1.1.1 Causes of the bullwhip effect

The bullwhip effect is caused by demand forecast updating, order batching, price fluctuation, and rationing and gaming.

- ***Demand forecast updating*** is done individually by all members of a supply chain. Each member updates its own demand forecast based on orders received from its "downstream" customer. The more members in the chain, the less these forecast updates reflect actual end-customer demand.
- ***Order batching*** occurs when each member takes order quantities it receives from its downstream customer and rounds up or down to suit production constraints such as equipment setup times or truckload quantities. The more members who conduct such rounding of order quantities, the more distortion occurs of the original quantities that were demanded.
- ***Price fluctuations*** due to inflationary factors, quantity discounts, or sales tend to encourage customers to buy larger quantities than they require. This behavior tends to add variability to quantities ordered and uncertainty to forecasts.
- ***Rationing and gaming*** is when a seller attempts to limit order quantities by delivering only a percentage of the order placed by the buyer. The buyer, knowing that the seller is delivering only a fraction of the order placed, attempts to "game" the system by making an upward adjustment to the order quantity. Rationing and gaming create distortions in the ordering information that is being received by the supply chain.

6.1.1.2 Counteracting the bullwhip effect

To improve the responsiveness, accuracy, and efficiency of the supply chain, a number of actions must be taken to combat the bullwhip effect:

- Make real-time end-item demand information available to all members of the supply chain. Information technologies such as electronic data interchange (EDI), bar codes, and scanning equipment can assist in providing all supply chain members with accurate and current demand information.
- Eliminate order batching by driving down the costs of placing orders, by reducing setup costs to make an ordered item, and by locating supply chain members closer to one another to ease transportation restrictions.
- Stabilize prices by replacing sales and discounts with consistent "every-day low prices" at the consumer stage and uniform wholesale pricing at upstream stages. Such actions remove price as a variable in determining order quantities.
- Discourage gaming in rationing situations by using past sales records to determine the quantities that will be delivered to customers.

6.1.2 Other factors affecting supply chain management

In addition to managing the bullwhip effect, supply chain managers must also contend with a variety of factors that pose on-going challenges:

- Increased demands from customers for better performance on cost, quality, delivery, and flexibility. Customers are better informed and have a broader array of options for how they conduct business. This puts added pressure on supply chain managers to continually improve performance.
- Globalization imposes challenges such as greater geographic dispersion among supply chain members. Greater distances create longer lead times and higher transportation costs. Cultural differences, time zones, and exchange rates make communication and decision-making more difficult. Boeing and Airbus have discovered the downside of sourcing from global suppliers. Much smaller suppliers of kitchen galleys, lavatories, and passenger seats have been unable to fulfill orders from Boeing and Airbus, leaving the latter unable to deliver planes to its airline customers.
- Government regulations, tariffs, and environmental rules provide challenges as well. For example, many countries require that products have a minimum percentage of local content. Being environmentally responsible by minimizing waste, properly disposing of dangerous chemicals, and using recyclable materials is rapidly becoming a requirement for doing business.

6.1.3 Supplier selection

Choosing suppliers is one of the most important decisions made by a company. The efficiency and value a supplier provides to an organization is reflected in the end product the organization produces. The supplier must not only provide goods and services that are consistent with the company's mission, it must also provide good value. The three most important factors in choosing a supplier are price, quality, and on-time delivery.

A company must not only choose who it wants as a supplier, it must also decide how many suppliers to use for a given good or service. There are advantages to using multiple suppliers and there are advantages to using one supplier. Whether to single-source or multiple-source often depends on the supply chain structure of the company and the character of the goods or services it produces.

If a company uses a single supplier, it can form a partnership with that supplier. A partnership is a long-term relationship between a supplier and a company that involves trust, information sharing, and financial benefits for both parties. When both parties benefit from a partnership, it is called a "win-win situation". It is easy to see how choosing suppliers is one of the most important decisions a company makes.

There are advantages and disadvantages to using one supplier. One advantage is that the supplier might own patents or processes and be the only source for the product. With one supplier, pricing discounts may be granted because purchases over the long-term are large and unit production costs for the supplier are lower. The supplier may be more responsive if you are the only purchaser of an item, resulting in better supplier relations. Just-in-time ordering is easier to implement, and deliveries may be scheduled more easily. Finally, using a single supplier is necessary to form a partnership. One disadvantage is that if that one supplier experiences a disaster at its warehouse like a fire or a tornado, or its workers go on strike, there is no other ready source for the product. Another possible disadvantage is that a single supplier may not be able to supply a very large quantity if it is suddenly needed. Also, sometimes the government requires the use of multiple suppliers for government projects.

There are also advantages and disadvantages to using multiple suppliers. Suppliers might provide better products and services over time if they know they are competing with other suppliers. Also, if a disaster happens at one supplier's warehouse, other suppliers can make up the loss. If a company uses multiple suppliers, there is more flexibility of volume to match demand fluctuations. One disadvantage with multiple suppliers is that it is more difficult to forge long-term partnerships. Information sharing becomes riskier, lower volumes for each supplier provide fewer opportunities for cost savings, and suppliers tend to be less responsive to emergency situations.

Partnerships are long-term relationships between a supplier and a company that involve trust and sharing and result in benefits for both parties. A good example of a partnership is the partnering between a Deere & Co. farm equipment factory and its suppliers. Deere decided to outsource its sheet metal, bar stock, and castings part families.

When Deere sent requests for bids to 120 companies, 24 companies responded to say they were interested. Deere then sent a team of engineers, quality specialists, and supply chain managers to evaluate each company. One supplier was chosen for each of the three part families. All three of the suppliers that were chosen were located less than two hours of driving time from the Deere plant.

For many years, all three suppliers have continued to provide outstanding quality, delivery, and cost performance to Deere. The suppliers benefited by gaining a long-term customer with a large amount of profitable business. Deere realized a 50 per cent drop in production costs on the three part families and was able to better focus on its mission of manufacturing farm equipment.

6.1.4 Conclusion

Supply chain management concerns the development of communication and information systems to link suppliers together in cooperative partnerships that promote advantage for all participants. Benefits include faster response times, reduced inventory costs, increased accuracy, and improved quality.

6.2 Operations management: Special topic: Total Quality Management[2]

Total Quality Management (TQM) is the organization-wide management of quality that includes facilities, equipment, labor, suppliers, customers, policies, and procedures. TQM promotes the view that quality improvement never ends, quality provides a strategic advantage to the organization, and zero defects is the quality goal that will minimize total quality costs. While this special topic on TQM is not a comprehensive discussion of all aspects of TQM, several key concepts will be discussed.

6.2.1 Quality costs

An important basis for justifying TQM practice is understanding its impact on total quality costs. TQM is rooted in the belief that preventing defects is cheaper than dealing with the costs of quality failures. In other words, total quality costs are minimized when managers strive to reach zero defects in the organization. The four major types of quality costs are prevention, appraisal, internal failure, and external failure.

Prevention costs are the costs created from the effort to reduce poor quality. Examples are designing the products so that they will be durable, training employees so they do a good job, certifying suppliers to ensure that suppliers provide quality in products and services, conducting preventive maintenance on equipment, and documenting quality procedures and improvements. In a traditional organization that does not practice TQM, prevention costs typically comprise the smallest percentage of total quality costs.

A good example of good product design occurs in all Honda products. Honda produces a wide variety of items including automobiles, ATVs, engines, generators, motorcycles, outboard motors, snow blowers, lawn and garden equipment, and even more items. To say the least, Honda engines last a long time. For example, Honda Accords typically run for well over 200,000 miles.

Employee training is also a very important prevention cost. For instance, employees in a vegetable/fruit packaging warehouse need to know what a bad vegetable/fruit looks like, since customers will not want to find spoiled produce in the store. Lifeguards at a swimming pool must know proper procedures for keeping swimmers safe. In many circumstances in both manufacturing and service businesses, the training of employees can make an enormous difference in preventing defects.

Supplier selection and certification are critical prevention activities. A product or service is only as good as the suppliers who partner with an organization to provide the raw materials, parts and components, and supporting services that make up the final products and services that the end customers receive. For example, a home furnishings store might use an outside subcontractor to install carpeting, but if the subcontractor fails to show up on time, tracks mud into the customer's home, or behaves in a rude manner, the store's reputation will suffer. Similarly, a car manufacturer who purchases defective tires from a supplier risks incurring high costs of recalls and lawsuits when the defects are discovered.

Preventive maintenance is necessary for preventing equipment breakdowns. Many manufacturing companies use sophisticated software to track machine usage, and determine optimal schedules for regular machine maintenance, overhauls, and replacement.

Documenting quality is a necessary prevention cost because it helps the organization track quality performance, identify quality problems, collect data, and specify procedures that contribute to the pursuit of zero defects. Documentation is important to communicating good quality practice to all employees and suppliers.

Appraisal costs are a second major type of quality cost. Appraisal costs include the inspection and testing of raw materials, work-in-process, and finished goods. In addition, quality audits, sampling, and statistical process control also fall under the umbrella of appraisal costs.

Inspection and testing of raw materials is very important, since substandard raw materials lead to substandard products. Raw materials used for a bridge determine the strength of the bridge. For example, soft steel will erode away faster than hardened steel. Moreover, the concrete bridge decking needs to be solid, as concrete with air pockets will erode and crumble faster creating an unsafe bridge.

[2]This content is available online at <http://cnx.org/content/m35447/1.4/>.

Finished goods and work-in-process inventory also need inspecting and testing. For example, worker error is quite common in the home construction industry, and this is why inspections occur frequently on newly constructed homes during and after the construction process is complete. Building inspectors ensure that the house has the proper framing, electrical, plumbing, heating, and so forth.

Quality audits and sampling are also important appraisal costs. Quality audits are checks of quality procedures to ensure that employees and suppliers are following proper quality practices. With sampling, a company can ensure with confidence that a batch of products is fit for use. For example, a wooden baseball bat manufacturer may test 10 out of every 100 bats to check that they meet strength standards. One weak bat can signal that quality problems are present.

Statistical process control (SPC) is the final type of appraisal cost. SPC tracks on-going processes in manufacturing or service environments to make sure that they are producing the desired performance. For example, a restaurant might statistically track customer survey results to make sure that customer satisfaction is maintained over time. In manufacturing windshields for automobiles, SPC might be used to track the number of microscopic air bubbles in the glass to make sure the process is performing to standard.

Internal failure costs are a third category of quality costs. This cost occurs when quality defects are discovered before they reach the customer. Examples of internal failure costs include scrapping a product, reworking the product, and lost productivity due to machine breakdowns or labor errors. Internal failure costs are typically more expensive than both prevention and appraisal costs because a great deal of material and labor often has been invested prior to the discovery of the defect. If a book publisher prints 10,000 books, then discovers that one of the chapters is missing from every copy, the cost of reworking or scrapping the books represents a major loss to the company. It would have been much cheaper to have procedures in place to prevent such a mistake from happening in the first place.

In the case of internal failure cost due to machine failures, FedEx, and other courier services cannot keep up with demand when a conveyor belt breaks down in the package distribution center. Major delays and costs occur when such incidents occur. Other examples include a road construction company having a road grader break down, a tool and die shop having a CNC machine break down, and a farmer having a combine break down during harvest time.

External failure costs are the fourth major cost of quality. External failure costs when the defect is discovered after it has reached the customer. This is the most expensive category of quality costs. Examples include product returns, repairs, warranty claims, lost reputation, and lost business. One spectacular example of external failure cost was when the Hubbell telescope was launched into space with mirrors that were ground improperly. When the telescope was turned on, instead of a magnificent view of stars, planets, and galaxies, the scientists could see only blurred images. The price of correcting the problem was over USD 1 billion.

External failure costs also occur when the wrong meal is delivered to a restaurant customer, when a computer breaks down shortly after it was purchased, when the wrong kidney is removed from a patient, and when a poorly designed automobile causes the death of drivers and passengers. Because of the enormous costs of internal and external failures, all companies should strive for zero defects. Successful TQM practice dictates that pursuing zero defects will result in the minimization of total quality costs by spending more on prevention and appraisal activities in order to reduce the much higher costs of internal and external failure.

6.2.2 TQM's seven basic elements

Successful practice of Total Quality Management involves both technical and people aspects that cover the entire organization and extend to relationships with suppliers and customers. Seven basic elements capture the essence of the TQM philosophy: customer focus, continuous improvement, employee empowerment, quality tools, product design, process management, and supplier quality.

- **Customer focus**: Decisions of how to organize resources to best serve customers starts with a clear understanding of customer needs and the measurement of customer satisfaction. For example, the Red Cross surveys its blood donors to determine how it can make the blood donation experience more pleasant and convenient. It collects information on the place, date and time donors came in, and asks donors questions of whether the donation time was convenient, whether they were treated with respect and gratitude, how long they had to wait to donate, and whether parking was adequate. By understanding donors' needs and experiences, Red Cross managers can determine strengths and weaknesses of the donation service process and make adjustments if necessary.

- **Continuous improvement**: An organizational culture that promotes continuous learning and problem solving is essential in the pursuit of zero defects. The Toyota Production System (TPS) is a universal continuous improvement system that has been effectively applied to many different types of organizations, including the health care industry. Essential elements of the TPS culture include studying process flow, collecting data, driving out wasteful non-value-added activities, and making everyone responsible for quality improvement. In the case of health care, the TPS approach enabled one hospital to analyze the causes of patient infections from catheters and pneumonia in patients on ventilators. With simple changes in procedures that prevented patients from getting these secondary illnesses, the hospital was able to save USD 40,000 per patient in these cases.
- **Employee involvement**: Employees in a TQM environment have very different roles and responsibilities than in a traditional organization. They are given responsibility, training, and authority to measure and control the quality of the work they produce, they work together in teams to address quality issues, they are cross-trained to be able to perform multiple tasks and have a greater understanding of the total production process, and they have a more intimate understanding of the operation and maintenance of their equipment. Employees are essential to the building of a continuous improvement organization.
- **Quality tools**: Discussion of the details of quality tools extends beyond the scope of this chapter, but there are seven basic quality tools that are used by front-line workers and managers in monitoring quality performance and gathering data for quality improvement activities. These tools include: cause-and-effect (fishbone) diagrams, flowcharts, checklists, control charts, scatter diagrams, Pareto analysis, and histograms. The beauty of these tools is that they are easy to understand and apply in on-going quality efforts.
- **Product design**: Product design is a key activity to avoid costly internal and external failure costs. For example, when a dental office designs the service process, it might have patients fill out a form that covers important information on general health issues, allergies, and medications. This helps to avoid future complications and problems. Staff, hygienists, and dentists are highly trained to follow proper procedures, the facility is both functional and pleasant, and the equipment and tools are state of the art to ensure that the patient's desired outcome is achieved. In a manufacturing setting, products should be designed to maximize product functionality, reliability, and manufacturability.
- **Process management**: "Quality at the Source" is an important concept in TQM. It means that managers and employees should be focused on the detailed activities in a process where good or bad quality is created. For example, in a Toyota plant in the United States in Georgetown, Kentucky, one of the work stations was responsible for installing seat belts and visors in every vehicle that came along the assembly line. There were 12 possible combinations of visors and seat belts that would go into any particular vehicle and the worker had to select the right combination and install the items in the vehicle in 55 seconds. Even the best workers made several errors during a shift on this activity. After studying the process, the workers came up with an idea to put all the items for a particular vehicle model in a blue plastic tote. With this change, the worker only had to make one decision per vehicle. Almost all the errors from the previous system were eliminated with this simple solution.
- **Supplier quality**: The focus on quality at the source extends to suppliers' processes as well, since the quality of a finished product is only as good as the quality of its individual parts and components, regardless of whether they come from internal or external sources. Sharing your quality and engineering expertise with your suppliers, having a formal supplier certification program, and including your suppliers in the product design stage are important measures to take to ensure that quality at the source extends to the supplier network.

6.2.3 Quality awards and standards

There are several quality awards and standards that are available for organizations to access. The large majority of organizations that use these programs use them as tools to help improve their quality processes and move toward implementing and successfully practicing TQM. The Malcolm Baldrige Award is a United States quality award that covers an extensive list of criteria that are evaluated by independent judges if an organization chooses to compete for the award. In many cases, organizations use the Baldrige criteria as a guide for their internal quality efforts rather than compete directly for the award. The criteria can be accessed from the Internet at: http://www.baldrige.nist.gov/rnet[3] .

The International Organization for Standardization (ISO) sponsors a certification process for organizations that seek to learn and adopt superior methods for quality practice (ISO 9000) and environmentally responsible products and methods of production (ISO 14000). These certifications are increasingly used by organizations of all sizes to compete more effectively in a global marketplace due to the wide acceptance of ISO certification as a criterion for supplier selection. ISO 9000 and ISO 14000 are described on the ISO web page at: http://www.iso.org/iso/home.htm[4] .

> *“The ISO 9000 family addresses "quality management". This means what the organization does to fulfill:*
>
> *the customer's quality requirements, and*
>
> *applicable regulatory requirements, while aiming to*
>
> *enhance customer satisfaction, and*
>
> *achieve continual improvement of its performance in pursuit of these objectives.*
>
> *The ISO 14000 family addresses "environmental management". This means what the organization does to:*
>
> *minimize harmful effects on the environment caused by its activities, and to*
>
> *achieve continual improvement of its environmental performance.”*

Another popular quality award is the Deming Prize, which is a Japanese quality award for which organizations from any country can apply. The Deming Prize was named after W. Edwards Deming, an American statistician, author, and consultant who helped improve United States production capabilities during World War II, but is best known for his work in post-war Japan. He is widely credited with assisting the Japanese in rebuilding their nation's production infrastructure in the areas of product design, product quality, and testing through the application of statistical methods. Florida Power and Electric was the first American company to win the Deming Prize, due to its meticulous use of formal approaches to quality improvement, data-based decision making, quality improvement teams, and the careful documentation of processes and procedures. More information on the Deming Prize can be found at:

http://www.juse.or.jp/e/deming/index.html[5]

[3]http://www.baldrige.nist.gov/rnet
[4]http://www.iso.org/iso/home.htm
[5]http://www.juse.or.jp/e/deming/index.html

6.3 Requirements analysis[6]

6.3.1 Introduction

In systems engineering and software engineering, requirements analysis encompasses those tasks that go into determining the requirements of a new or altered system, taking account of the possibly conflicting requirements of the various stakeholders, such as users. Requirements analysis is critical to the success of a project.

Systematic requirements analysis is also known as requirements engineering. It is sometimes referred to loosely by names such as requirements gathering, requirements capture, or requirements specification. The term "requirements analysis" can also be applied specifically to the analysis proper (as opposed to elicitation or documentation of the requirements, for instance).

Requirements must be measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

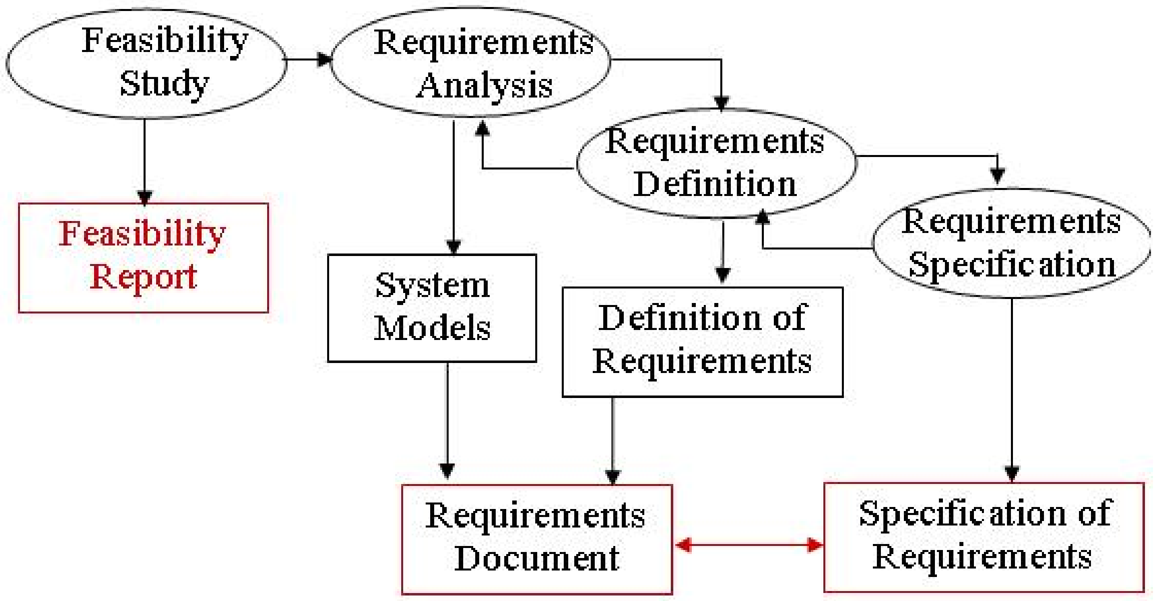


Figure 6.2: Requirements process

6.3.2 Software Requirements Fundamentals

6.3.2.1 Definition of a Software Requirement

At its most basic, a software requirement is a property which must be exhibited in order to solve some problem in the real world. This session refers to requirements on “software” because it is concerned with problems to be addressed by software. Hence, a software requirement is a property which must be exhibited by software developed or adapted to solve a particular problem. The problem may be to automate part of a task of someone who will use the software, to support the business processes of the organization that

[6]This content is available online at <http://cnx.org/content/m14621/1.6/>.

has commissioned the software, to correct shortcomings of existing software, to control a device, and many more. The functioning of users, business processes, and devices is typically complex. By extension, therefore, the requirements on particular software are typically a complex combination of requirements from different people at different levels of an organization and from the environment in which the software will operate.

An essential property of all software requirements is that they be verifiable. It may be difficult or costly to verify certain software requirements. For example, verification of the throughput requirement on the call center may necessitate the development of simulation software. Both the software requirements and software quality personnel must ensure that the requirements can be verified within the available resource constraints.

Requirements have other attributes in addition to the behavioral properties that they express. Common examples include a priority rating to enable trade-offs in the face of finite resources and a status value to enable project progress to be monitored. Typically, software requirements are uniquely identified so that they can be over the entire software life cycle.

6.3.2.2 Product and Process Requirements

A distinction can be drawn between product parameters and process parameters. Product parameters are requirements on software to be developed (for example, “The software shall verify that a student meets all prerequisites before he or she registers for a course.”).

A process parameter is essentially a constraint on the development of the software (for example, “The software shall be written in Ada.”). These are sometimes known as process requirements.

Some software requirements generate implicit process requirements. The choice of verification technique is one example. Another might be the use of particularly rigorous analysis techniques (such as formal specification methods) to reduce faults which can lead to inadequate reliability. Process requirements may also be imposed directly by the development organization, their customer, or a third party such as a safety regulator.

6.3.2.3 Functional and Non-functional Requirements

Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities or statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.

Nonfunctional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements.

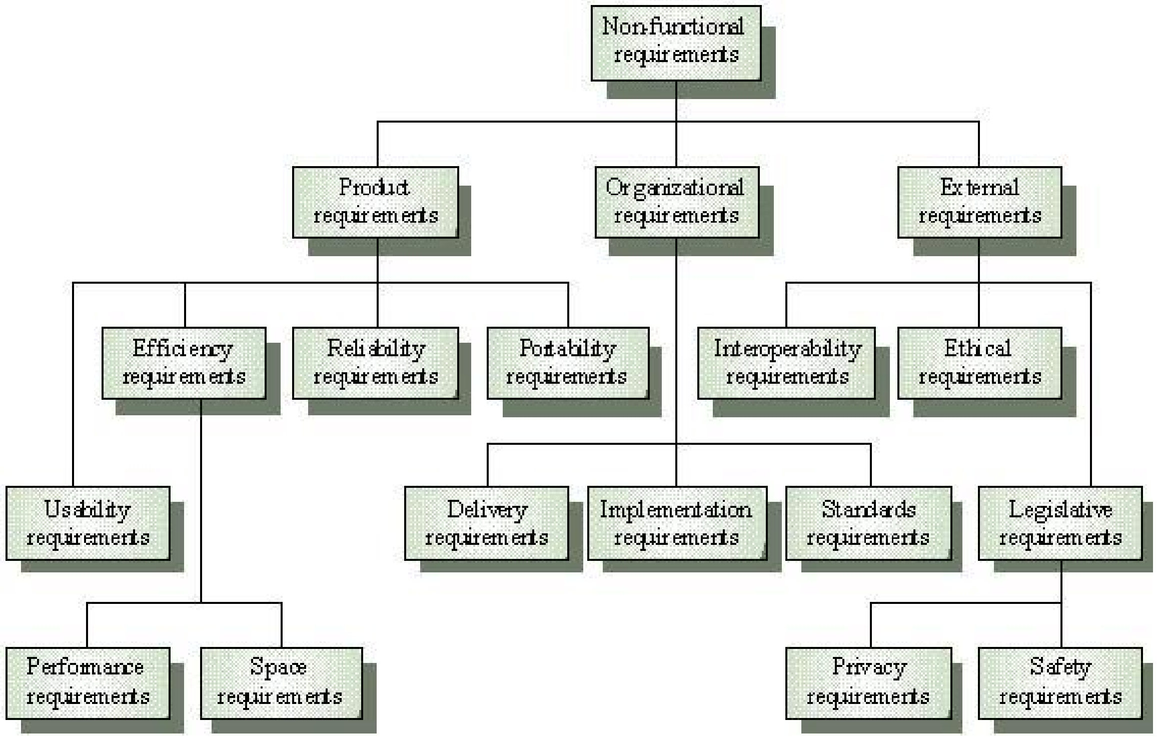


Figure 6.3: Nonfunctional Requirements

They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, or one of many other types of software requirements.

6.3.2.4 Emergent Properties

Some requirements represent emergent properties of software—that is, requirements which cannot be addressed by a single component, but which depend for their satisfaction on how all the software components interoperate. The throughput requirement for a call center would, for example, depend on how the telephone system, information system, and the operators all interacted under actual operating conditions. Emergent properties are crucially dependent on the system architecture.

6.3.2.5 Quantifiable Requirements

Software requirements should be stated as clearly and as unambiguously as possible, and, where appropriate, quantitatively. It is important to avoid vague and unverifiable requirements which depend for their interpretation on subjective judgment ("the software shall be reliable"; "the software shall be user-friendly"). This is particularly important for nonfunctional requirements. Two examples of quantified requirements are the following: a call center's software must increase the center's throughput by 20%; and a system shall have a probability of generating a fatal error during any hour of operation of less than 1 * 10[U+F02D]8. The throughput requirement is at a very high level and will need to be used to derive a number of detailed requirements. The reliability requirement will tightly constrain the system architecture.

6.3.2.6 System Requirements and Software Requirements

In this topic, system means "an interacting combination of elements to accomplish a defined objective. These include hardware, software, firmware, people, information, techniques, facilities, services, and other support elements," as defined by the International Council on Systems Engineering.

System requirements are the requirements for the system as a whole. In a system containing software components, software requirements are derived from system requirements.

The literature on requirements sometimes calls system requirements "user requirements." We can define "user requirements" in a restricted way as the requirements of the system's customers or end-users. System requirements, by contrast, encompass user requirements, requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source.

6.3.3 Requirements Process

This section introduces the software requirements process, orienting the remaining five subareas and showing how the requirements process dovetails with the overall software engineering process.

6.3.3.1 Process Models

The objective of this topic is to provide an understanding that the requirements process

- Is not a discrete front-end activity of the software life cycle, but rather a process initiated at the beginning of a project and continuing to be refined throughout the life cycle
- Identifies software requirements as configuration items, and manages them using the same software configuration management practices as other products of the software life cycle processes
- Needs to be adapted to the organization and project context

In particular, the topic is concerned with how the activities of elicitation, analysis, specification, and validation are configured for different types of projects and constraints.

6.3.3.2 Process Actors

This topic introduces the roles of the people who participate in the requirements process. This process is fundamentally interdisciplinary, and the requirements specialist needs to mediate between the domain of the stakeholder and that of software engineering. There are often many people involved besides the requirements specialist, each of whom has a stake in the software. The stakeholders will vary across projects, but always include users/operators and customers (who need not be the same).

Typical examples of software stakeholders include (but are not restricted to)

- Users: This group comprises those who will operate the software. It is often a heterogeneous group comprising people with different roles and requirements.
- Customers: This group comprises those who have commissioned the software or who represent the software's target market.
- Market analysts: A mass-market product will not have a commissioning customer, so marketing people are often needed to establish what the market needs and to act as proxy customers.
- Regulators: Many application domains such as banking and public transport are regulated. Software in these domains must comply with the requirements of the regulatory authorities.
- Software engineers: These individuals have a legitimate interest in profiting from developing the software by, for example, reusing components in other products. If, in this scenario, a customer of a particular product has specific requirements which compromise the potential for component reuse, the software engineers must carefully weigh their own stake against those of the customer.

It will not be possible to perfectly satisfy the requirements of every stakeholder, and it is the software engineer's job to negotiate trade-offs which are both acceptable to the principal stakeholders and within budgetary, technical, regulatory, and other constraints. A prerequisite for this is that all the stakeholders be identified, the nature of their "stake" analyzed, and their requirements elicited.

6.3.3.3 Process Support and Management

This topic introduces the project management resources required and consumed by the requirements process. It establishes the context for the first subarea (Initiation and scope definition) of the Software Engineering Management KA. Its principal purpose is to make the link between the process activities identified and the issues of cost, human resources, training, and tools.

6.3.3.4 Process Quality and Improvement

This topic is concerned with the assessment of the quality and improvement of the requirements process. Its purpose is to emphasize the key role the requirements process plays in terms of the cost and timeliness of a software product, and of the customer's satisfaction with it. It will help to orient the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement is closely related to both the Software Quality KA and the Software Engineering Process KA. Of particular interest are issues of software quality attributes and measurement, and software process definition. This topic covers

- Requirements process coverage by process improvement standards and models
- Requirements process measures and benchmarking
- Improvement planning and implementation

6.3.4 Requirements Elicitation

Requirements elicitation is concerned with where software requirements come from and how the software engineer can collect them. It is the first stage in building an understanding of the problem the software is required to solve. It is fundamentally a human activity, and is where the stakeholders are identified and relationships established between the development team and the customer. It is variously termed "requirements capture," "requirements discovery," and "requirements acquisition."

One of the fundamental tenets of good software engineering is that there be good communication between software users and software engineers. Before development begins, requirements specialists may form the conduit for this communication. They must mediate between the domain of the software users (and other stakeholders) and the technical world of the software engineer.

6.3.4.1 Requirements Sources

Requirements have many sources in typical software, and it is essential that all potential sources be identified and evaluated for their impact on it. This topic is designed to promote awareness of the various sources of software requirements and of the frameworks for managing them. The main points covered are

- Goals: The term goal (sometimes called "business concern" or "critical success factor") refers to the overall, high-level objectives of the software. Goals provide the motivation for the software, but are often vaguely formulated. Software engineers need to pay particular attention to assessing the value (relative to priority) and cost of goals. A feasibility study is a relatively low-cost way of doing this.
- Domain knowledge: The software engineer needs to acquire, or have available, knowledge about the application domain. This enables them to infer tacit knowledge that the stakeholders do not articulate, assess the trade-offs that will be necessary between conflicting requirements, and, sometimes, to act as a "user" champion.

- Stakeholders: Much software has proved unsatisfactory because it has stressed the requirements of one group of stakeholders at the expense of those of others. Hence, software is delivered which is difficult to use or which subverts the cultural or political structures of the customer organization. The software engineer needs to identify, represent, and manage the "viewpoints" of many different types of stakeholders.
- The operational environment: Requirements will be derived from the environment in which the software will be executed. These may be, for example, timing constraints in real-time software or interoperability constraints in an office environment. These must be actively sought out, because they can greatly affect software feasibility and cost, and restrict design choices.
- The organizational environment: Software is often required to support a business process, the selection of which may be conditioned by the structure, culture, and internal politics of the organization. The software engineer needs to be sensitive to these, since, in general, new software should not force unplanned change on the business process.

6.3.4.2 Elicitation Techniques

Once the requirements sources have been identified, the software engineer can start eliciting requirements from them. This topic concentrates on techniques for getting human stakeholders to articulate their requirements. It is a very difficult area and the software engineer needs to be sensitized to the fact that (for example) users may have difficulty describing their tasks, may leave important information unstated, or may be unwilling or unable to cooperate. It is particularly important to understand that elicitation is not a passive activity, and that, even if cooperative and articulate stakeholders are available, the software engineer has to work hard to elicit the right information. A number of techniques exist for doing this, the principal ones being.

- Interviews: a "traditional" means of eliciting requirements. It is important to understand the advantages and limitations of interviews and how they should be conducted.
- Scenarios: a valuable means for providing context to the elicitation of user requirements. They allow the software engineer to provide a framework for questions about user tasks by permitting "what if" and "how is this done" questions to be asked. The most common type of scenario is the use case.
- Prototypes: a valuable tool for clarifying unclear requirements. They can act in a similar way to scenarios by providing users with a context within which they can better understand what information they need to provide. There is a wide range of prototyping techniques, from paper mock-ups of screen designs to beta-test versions of software products, and a strong overlap of their use for requirements elicitation and the use of prototypes for requirements validation.
- Facilitated meetings: The purpose of these is to try to achieve a summative effect whereby a group of people can bring more insight into their software requirements than by working individually. They can brainstorm and refine ideas which may be difficult to bring to the surface using interviews. Another advantage is that conflicting requirements surface early on in a way that lets the stakeholders recognize where there is conflict. When it works well, this technique may result in a richer and more consistent set of requirements than might otherwise be achievable. However, meetings need to be handled carefully (hence the need for a facilitator) to prevent a situation from occurring where the critical abilities of the team are eroded by group loyalty, or the requirements reflecting the concerns of a few outspoken (and perhaps senior) people are favored to the detriment of others.
- Observation: The importance of software context within the organizational environment has led to the adaptation of observational techniques for requirements elicitation. Software engineers learn about user tasks by immersing themselves in the environment and observing how users interact with their software and with each other. These techniques are relatively expensive, but they are instructive because they illustrate that many user tasks and business processes are too subtle and complex for their actors to describe easily.

6.3.5 Requirements Analysis

This topic is concerned with the process of analyzing requirements to

- Detect and resolve conflicts between requirements
- Discover the bounds of the software and how it must interact with its environment
- Elaborate system requirements to derive software requirements

The traditional view of requirements analysis has been that it be reduced to conceptual modeling using one of anumber of analysis methods such as the Structured Analysis and Design Technique (SADT). While conceptual modeling is important, we include the classification of requirements to help inform trade-offs between requirements (requirements classification) and the process of establishing these trade-offs (requirements negotiation).

Care must be taken to describe requirements precisely enough to enable the requirements to be validated, their implementation to be verified, and their costs to be estimated.

6.3.5.1 Requirements Classification

Requirements can be classified on a number of dimensions:

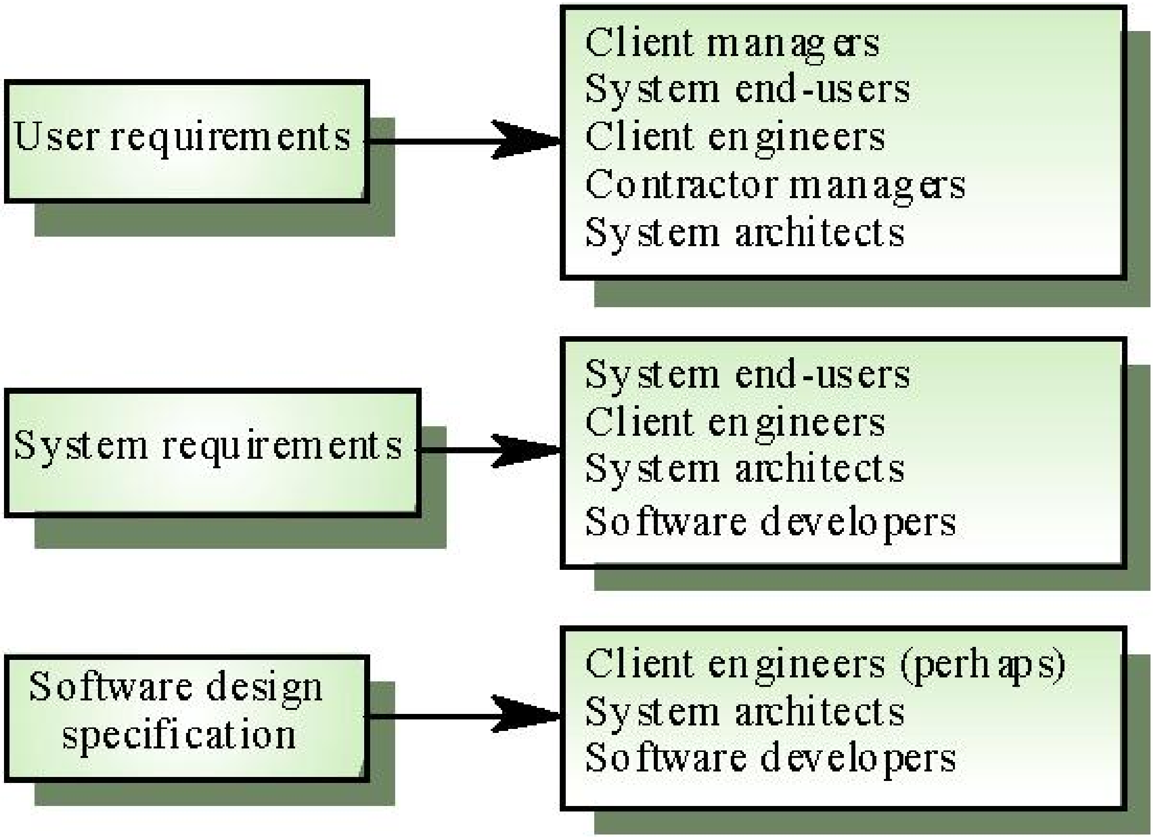


Figure 6.4: Requirement types

Other classifications may be appropriate, depending upon the organization's normal practice and the application itself.

There is a strong overlap between requirements classification and requirements attributes.

6.3.5.2 Conceptual Modeling

The development of models of a real-world problem is key to software requirements analysis. Their purpose is to aid in understanding the problem, rather than to initiate design of the solution. Hence, conceptual models comprise models of entities from the problem domain configured to reflect their real-world relationships and dependencies.

Several kinds of models can be developed. These include data and control flows, state models, event traces, user interactions, object models, data models, and many others. The factors that influence the choice of model include

- The nature of the problem. Some types of software demand that certain aspects be analyzed particularly rigorously. For example, control flow and state models are likely to be more important for real-time software than for management information software, while it would usually be the opposite for data models.
- The expertise of the software engineer. It is often more productive to adopt a modeling notation or method with which the software engineer has experience.
- The process requirements of the customer. Customers may impose their favored notation or method, or prohibit any with which they are unfamiliar. This factor can conflict with the previous factor.
- The availability of methods and tools. Notations or methods which are poorly supported by training and tools may not achieve widespread acceptance even if they are suited to particular types of problems.

Note that, in almost all cases, it is useful to start by building a model of the software context. The software context provides a connection between the intended software and its external environment. This is crucial to understanding the software's context in its operational environment and to identifying its interfaces with the environment.

The issue of modeling is tightly coupled with that of methods. For practical purposes, a method is a notation (or set of notations) supported by a process which guides the application of the notations. There is little empirical evidence to support claims for the superiority of one notation over another. However, the widespread acceptance of a particular method or notation can lead to beneficial industry-wide pooling of skills and knowledge. This is currently the situation with the UML (Unified Modeling Language).

Formal modeling using notations based on discrete mathematics, and which are traceable to logical reasoning, have made an impact in some specialized domains. These may be imposed by customers or standards or may offer compelling advantages to the analysis of certain critical functions or components.

Two standards provide notations which may be useful in performing conceptual modeling–IEEE Std 1320.1, IDEF0 for functional modeling; and IEEE Std 1320.2, IDEF1X97 (IDEFObject) for information modeling.

6.3.5.3 Architectural Design and Requirements Allocation

At some point, the architecture of the solution must be derived. Architectural design is the point at which the requirements process overlaps with software or systems design and illustrates how impossible it is to cleanly decouple the two tasks. This topic is closely related to the Software Structure and Architecture subarea in the Software Design KA. In many cases, the software engineer acts as software architect because the process of analyzing and elaborating the requirements demands that the components that will be responsible for satisfying the requirements be identified. This is requirements allocation–the assignment, to components, of the responsibility for satisfying requirements.

Allocation is important to permit detailed analysis of requirements. Hence, for example, once a set of requirements has been allocated to a component, the individual requirements can be further analyzed to discover further requirements on how the component needs to interact with other components in order to satisfy the allocated requirements. In large projects, allocation stimulates a new round of analysis for each subsystem.

Architectural design is closely identified with conceptual modeling. The mapping from real-world domain entities to software components is not always obvious, so architectural design is identified as a separate topic. The requirements of notations and methods are broadly the same for both conceptual modeling and architectural design.

6.3.5.4 Requirements Negotiation

Another term commonly used for this sub-topic is "conflict resolution." This concerns resolving problems with requirements where conflicts occur between two stakeholders requiring mutually incompatible features, between requirements and resources, or between functional and non-functional requirements. In most cases, it is unwise for the software engineer to make a unilateral decision, and so it becomes necessary to consult with the stakeholder(s) to reach a consensus on an appropriate trade-off. It is often important for contractual reasons that such decisions be traceable back to the customer. We have classified this as a software requirements analysis topic because problems emerge as the result of analysis. However, a strong case can also be made for considering it a requirements validation topic.

6.3.5.5 Requirements Specification

For most engineering professions, the term "specification" refers to the assignment of numerical values or limits to a product's design goals. Typical physical systems have a relatively small number of such values. Typical software has a large number of requirements, and the emphasis is shared between performing the numerical quantification and managing the complexity of interaction among the large number of requirements. So, in software engineering jargon, "software requirements specification" typically refers to the production of a document, or its electronic equivalent, which can be systematically reviewed, evaluated, and approved.

For complex systems, particularly those involving substantial non-software components, as many as three different types of documents are produced: system definition, system requirements, and software requirements. For simple software products, only the third of these is required.

There are some approaches to requirements specification:

- Natural language
- Structured natural language
- Design description language
- Requirements specification language
- Graphical notation
- Formal specification

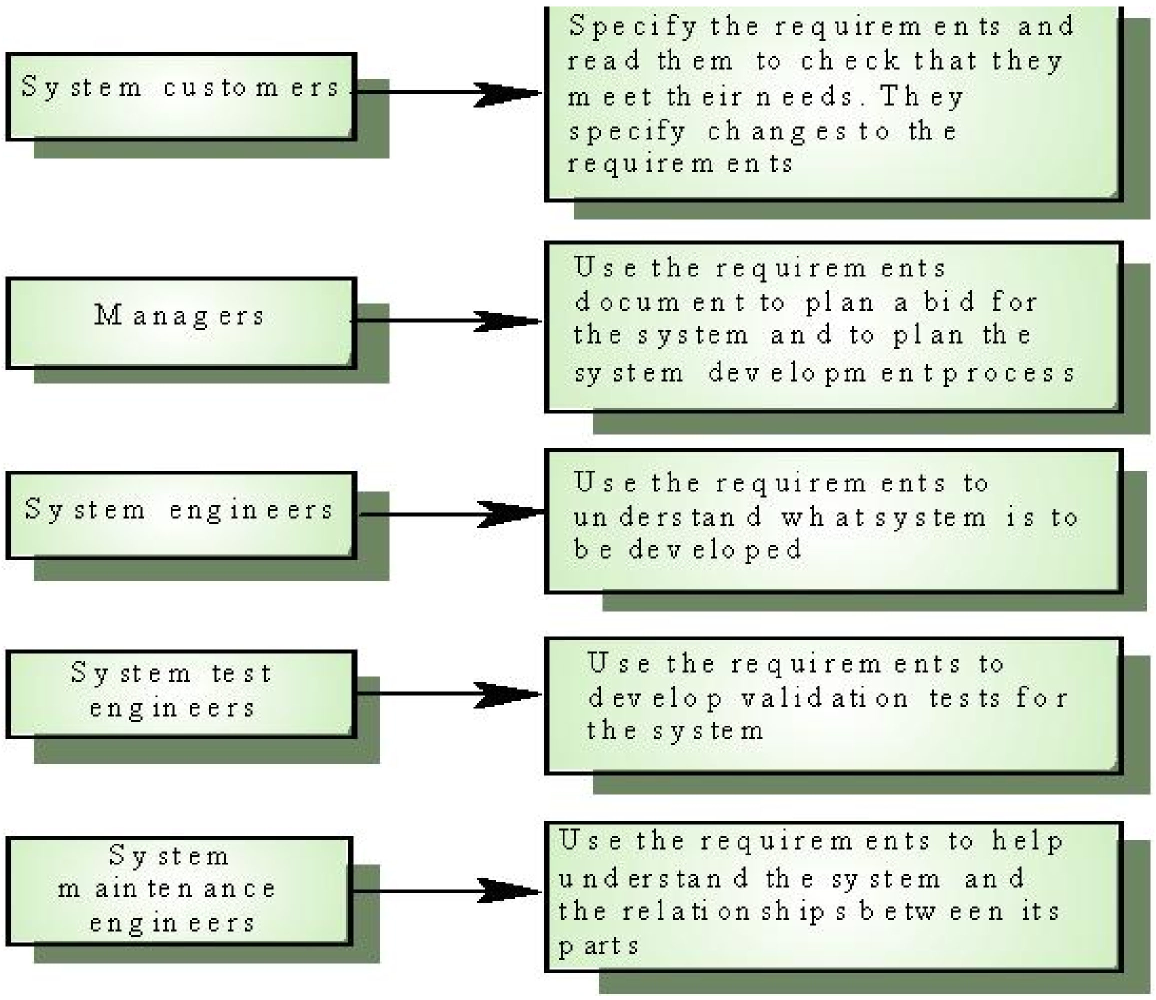

System customers
Specify the requirements and read them to check that they meet their needs. They specify changes to the requirements
Managers
Use the requirements document to plan a bid for the system and to plan the system development process
System engineers
Use the requirements to understand what system is to be developed
System test engineers
Use the requirements to develop validation tests for the system
System maintenance engineers
Use the requirements to help understand the system and the relationships between its parts

Figure 6.5: Types of requirement document

6.3.5.6 The System Definition Document

This document (sometimes known as the user requirements document or concept of operations) records the system requirements. It defines the high-level system requirements from the domain perspective. Its readership includes representatives of the system users/customers (marketing may play these roles for market-driven software), so its content must be couched in terms of the domain. The document lists the system requirements along with background information about the overall objectives for the system, its target environment and a statement of the constraints, assumptions, and non-functional requirements. It may include conceptual models designed to illustrate the system context, usage scenarios and the principal domain entities, as well as data, information, and workflows. IEEE Std 1362, Concept of Operations Document, provides advice on the preparation and content of such a document. (IEEE1362-98)

6.3.5.7 System Requirements Specification

Developers of systems with substantial software and non-software components, a modern airliner, for example, often separate the description of system requirements from the description of software requirements. In this view, system requirements are specified, the software requirements are derived from the system requirements, and then the requirements for the software components are specified. Strictly speaking, system requirements specification is a systems engineering activity and falls outside the scope of this Guide. IEEE Std 1233 is a guide for developing system requirements.

6.3.5.8 Software Requirements Specification

Software requirements specification establishes the basis for agreement between customers and contractors or suppliers (in market-driven projects, these roles may be played by the marketing and development divisions) on what the software product is to do, as well as what it is not expected to do. For non-technical readers, the software requirements specification document is often accompanied by a software requirements definition document.

Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign. It should also provide a realistic basis for estimating product costs, risks, and schedules.

Organizations can also use a software requirements specification document to develop their own validation and verification plans more productively.

Software requirements specification provides an informed basis for transferring a software product to new users or new machines. Finally, it can provide a basis for software enhancement.

Software requirements are often written in natural language, but, in software requirements specification, this may be supplemented by formal or semi-formal descriptions. Selection of appropriate notations permits particular requirements and aspects of the software architecture to be described more precisely and concisely than natural language. The general rule is that notations should be used which allow the requirements to be described as precisely as possible. This is particularly crucial for safety-critical and certain other types of dependable software. However, the choice of notation is often constrained by the training, skills and preferences of the document's authors and readers.

A number of quality indicators have been developed which can be used to relate the quality of software requirements specification to other project variables such as cost, acceptance, performance, schedule, reproducibility, etc. Quality indicators for individual software requirements specification statements include imperatives, directives, weak phrases, options, and continuances. Indicators for the entire software requirements specification document include size, readability, specification, depth, and text structure.

6.3.6 Requirements validation

The requirements documents may be subject to validation and verification procedures. The requirements may be validated to ensure that the software engineer has understood the requirements, and it is also important to verify that a requirements document conforms to company standards, and that it is understandable, consistent, and complete. Formal notations offer the important advantage of permitting the last two properties to be proven (in a restricted sense, at least). Different stakeholders, including representatives of the customer and developer, should review the document(s). Requirements documents are subject to the same software configuration management practices as the other deliverables of the software life cycle processes.

It is normal to explicitly schedule one or more points in the requirements process where the requirements are validated. The aim is to pick up any problems before resources are committed to addressing the requirements. Requirements validation is concerned with the process of examining the requirements document to ensure that it defines the right software (that is, the software that the users expect).

6.3.6.1 Requirements Reviews

Perhaps the most common means of validation is by inspection or reviews of the requirements document(s). A group of reviewers is assigned a brief to look for errors, mistaken assumptions, lack of clarity, and deviation from standard practice. The composition of the group that conducts the review is important (at least one representative of the customer should be included for a customer-driven project, for example), and it may help to provide guidance on what to look for in the form of checklists.

Reviews may be constituted on completion of the system definition document, the system specification document, the software requirements specification document, the baseline specification for a new release, or at any other step in the process.

6.3.6.2 Prototyping

Prototyping is commonly a means for validating the software engineer's interpretation of the software requirements, as well as for eliciting new requirements. As with elicitation, there is a range of prototyping techniques and a number of points in the process where prototype validation may be appropriate. The advantage of prototypes is that they can make it easier to interpret the software engineer's assumptions and, where needed, give useful feedback on why they are wrong. For example, the dynamic behavior of a user interface can be better understood through an animated prototype than through textual description or graphical models. There are also disadvantages, however. These include the danger of users' attention being distracted from the core underlying functionality by cosmetic issues or quality problems with the prototype. For this reason, several people recommend prototypes which avoid software, such as flip-chart-based mock-ups. Prototypes may be costly to develop. However, if they avoid the wastage of resources caused by trying to satisfy erroneous requirements, their cost can be more easily justified.

6.3.6.3 Model Validation

It is typically necessary to validate the quality of the models developed during analysis. For example, in object models, it is useful to perform a static analysis to verify that communication paths exist between objects which, in the stakeholders' domain, exchange data. If formal specification notations are used, it is possible to use formal reasoning to prove specification properties.

6.3.6.4 Acceptance Tests

An essential property of a software requirement is that it should be possible to validate that the finished product satisfies it. Requirements which cannot be validated are really just "wishes." An important task is therefore planning how to verify each requirement. In most cases, designing acceptance tests does this.

Identifying and designing acceptance tests may be difficult for non-functional requirements. To be validated, they must first be analyzed to the point where they can be expressed quantitatively.

6.3.7 Practical Considerations

The first level of decomposition of subareas presented in this KA may seem to describe a linear sequence of activities. This is a simplified view of the process.

The requirements process spans the whole software life cycle. Change management and the maintenance of the requirements in a state which accurately mirrors the software to be built, or that has been built, are key to the success of the software engineering process.

Not every organization has a culture of documenting and managing requirements. It is frequent in dynamic start-up companies, driven by a strong "product vision" and limited resources, to view requirements documentation as an unnecessary overhead. Most often, however, as these companies expand, as their customer base grows, and as their product starts to evolve, they discover that they need to recover the requirements that motivated product features in order to assess the impact of proposed changes. Hence, requirements documentation and change management are key to the success of any requirements process.

6.3.7.1 Iterative Nature of the Requirements Process

There is general pressure in the software industry for ever shorter development cycles, and this is particularly pronounced in highly competitive market-driven sectors. Moreover, most projects are constrained in some way by their environment, and many are upgrades to, or revisions of, existing software where the architecture is a given. In practice, therefore, it is almost always impractical to implement the requirements process as a linear, deterministic process in which software requirements are elicited from the stakeholders, baselined, allocated, and handed over to the software development team. It is certainly a myth that the requirements for large software projects are ever perfectly understood or perfectly specified.

Instead, requirements typically iterate towards a level of quality and detail which is sufficient to permit design and procurement decisions to be made. In some projects, this may result in the requirements being baselined before all their properties are fully understood. This risks expensive rework if problems emerge late in the software engineering process. However, software engineers are necessarily constrained by project management plans and must therefore take steps to ensure that the "quality" of the requirements is as high as possible given the available resources. They should, for example, make explicit any assumptions which underpin the requirements, as well as any known problems.

In almost all cases, requirements understanding continues to evolve as design and development proceeds. This often leads to the revision of requirements late in the life cycle. Perhaps the most crucial point in understanding requirements engineering is that a significant proportion of the requirements will change. This is sometimes due to errors in the analysis, but it is frequently an inevitable consequence of change in the "environment": for example, the customer's operating or business environment, or the market into which software must sell. Whatever the cause, it is important to recognize the inevitability of change and take steps to mitigate its effects. Change has to be managed by ensuring that proposed changes go through a defined review and approval process, and, by applying careful requirements tracing, impact analysis, and software configuration management. Hence, the requirements process is not merely a front-end task in software development, but spans the whole software life cycle. In a typical project, the software requirements activities evolve over time from elicitation to change management.

6.3.7.2 Change Management

Change management is central to the management of requirements. This topic describes the role of change management, the procedures that need to be in place, and the analysis that should be applied to proposed changes. It has strong links to the Software Configuration Management KA.

6.3.7.3 Requirements Attributes

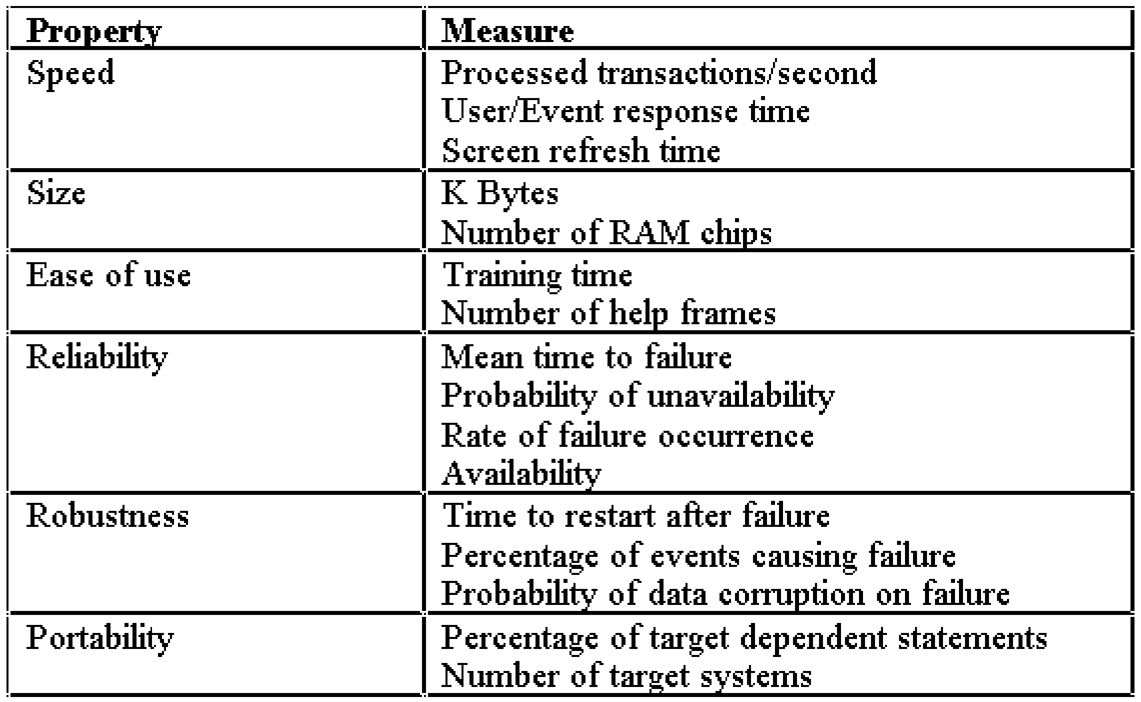
Requirements should consist not only of a specification of what is required, but also of ancillary information which helps manage and interpret the requirements. This should include the various classification dimensions of the requirement and the verification method or acceptance test plan. It may also include additional information such as a summary rationale for each requirement, the source of each requirement, and a change history. The most important requirements attribute, however, is an identifier which allows the requirements to be uniquely and unambiguously identified.

6.3.7.4 Requirements Tracing

Requirements tracing is concerned with recovering the source of requirements and predicting the effects of requirements. Tracing is fundamental to performing impact analysis when requirements change. A requirement should be traceable backwards to the requirements and stakeholders which motivated it (from a software requirement back to the system requirement(s) that it helps satisfy, for example). Conversely, a requirement should be traceable forwards into the requirements and design entities that satisfy it (for example, from a system requirement into the software requirements that have been elaborated from it, and on into the code modules that implement it).

6.3.7.5 Measuring Requirements

As a practical matter, it is typically useful to have some concept of the "volume" of the requirements for a particular software product. This number is useful in evaluating the "size" of a change in requirements, in estimating the cost of a development or maintenance task, or simply for use as the denominator in other measurements.



Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 6.6: Requirement measurements

6.4 Software quality management[7]

6.4.1 Introduction

What is software quality, and why is it so important that it is pervasive in the Software Engineering Body of Knowledge? Within an information system, software is a tool, and tools have to be selected for quality and for appropriateness. That is the role of equirements. But software is more than a tool. It dictates the performance of the system, and it is therefore important to the system quality.

The notion of "quality" is not as simple as it may seem. For any engineered product, there are many desired qualities relevant to a particular project, to be discussed and determined at the time that the product requirements are determined. Qualities may be present or absent, or may be matters of degree, with tradeoffs among them, with practicality and cost as major considerations. The software engineer has a responsibility to

[7]This content is available online at <http://cnx.org/content/m28899/1.1/>.

elicit the system's quality requirements that may not be explicit at the outset and to discuss their importance and the difficulty of attaining them. All processes associated with software quality (e.g. building, checking, improving quality) will be designed with these in mind

and carry costs based on the design. Thus, it is important to have in mind some of the possible attributes of quality.

Various researchers have produced models (usually taxonomic) of software quality characteristics or attributes that can be useful for discussing, planning, and rating the quality of software products. The models often include metrics to "measure" the degree of each quality attribute the product attains.

Usually these metrics may be applied at any of the product levels. They are not always direct measures of the quality characteristics of the finished product, but may be relevant to the achievement of overall quality. Each model may have a different set of attributes at the highest level of the taxonomy, and selection of and definitions for the attributes at all levels may differ. The important point is that the system software requirements define the quality requirements and the definitions of the attributes for them.

6.4.2 Software Quality Fundamentals

Agreement on quality requirements, as well as clear communication to the software engineer on what constitutes quality, require that the many aspects of quality be formally defined and discussed.

A software engineer should understand the underlying meanings of quality concepts and characteristics and their value to the software under development or to maintenance.

The important concept is that the software requirements define the required quality characteristics of the software and influence the measurement methods and acceptance criteria for assessing these characteristics.

6.4.2.1 Software Engineering Culture and Ethics

Software engineers are expected to share a commitment to software quality as part of their culture.

Ethics can play a significant role in software quality, the culture, and the attitudes of software engineers. The IEEE Computer Society and the ACM have developed a code of ethics and professional practice based on eight principles to help software engineers reinforce attitudes related to quality and to the independence of their work.

6.4.2.2 Value and Costs of Quality

The notion of "quality" is not as simple as it may seem. For any engineered product, there are many desired qualities relevant to a particular perspective of the product, to be discussed and determined at the time that the product requirements are set down. Quality characteristics may be required or not, or may be required to a greater or lesser degree, and trade-offs may be made among them.

The cost of quality can be differentiated into prevention cost, appraisal cost, internal failure cost, and external failure cost.

A motivation behind a software project is the desire to create software that has value, and this value may or may not be quantified as a cost. The customer will have some maximum cost in mind, in return for which it is expected that the basic purpose of the software will be fulfilled. The customer may also have some expectation as to the quality of the software. Sometimes customers may not have thought through the quality issues or their related costs. Is the characteristic merely decorative, or is it essential to the software? If the answer lies somewhere in between, as is almost always the case, it is a matter of making the customer a part of the decision process and fully aware of both costs and benefits. Ideally, most of these decisions will be made in the software requirements process, but these issues may arise throughout the software life cycle. There is no definite rule as to how these decisions should be made, but the software engineer should be able to present quality alternatives and their costs.

6.4.2.3 Models and Quality Characteristics

Terminology for software quality characteristics differs from one taxonomy (or model of software quality) to another, each model perhaps having a different number of hierarchical levels and a different total number of characteristics. Various authors have produced models of software quality characteristics or attributes which can be useful for discussing, planning, and rating the quality of software products. ISO/IEC has defined three related models of software product quality (internal quality, external quality, and quality in use) (ISO9126-01) and a set of related parts (ISO14598-98).

6.4.2.3.1 Software engineering process quality

Software quality management and software engineering process quality have a direct bearing on the quality of the software product.

Models and criteria which evaluate the capabilities of software organizations are primarily project organization and management considerations, and, as such, are covered in the Software Engineering Management and Software Engineering Process.

Of course, it is not possible to completely distinguish the quality of the process from the quality of the product.

Process quality influences the quality characteristics of software products, which in turn affect quality-in-use as perceived by the customer.

Two important quality standards are TickIT and one which has an impact on software quality, the ISO9001-00 standard, along with its guidelines for application to software.

Another industry standard on software quality is CMMI. CMMI intends to provide guidance for improving processes. Specific process areas related to quality management are process and product quality assurance, process verification, and process validation. CMMI classifies reviews and audits as methods of verification, and not as specific processes like.

There was initially some debate over whether ISO9001 or CMMI should be used by software engineers to ensure quality. This debate is widely published, and, as a result, the position has been taken that the two are complementary and that having ISO9001 certification can help greatly in achieving the higher maturity levels of the CMMI.

6.4.2.3.2 Software product quality

The software engineer needs, first of all, to determine the real purpose of the software. In this regard, it is of prime importance to keep in mind that the customer's requirements come first and that they include quality requirements, not just functional requirements. Thus, the software engineer has a responsibility to elicit quality requirements which may not be explicit at the outset and to discuss their importance as well as the level of difficulty in attaining them. All processes associated with software quality (for example, building, checking, and improving quality) will be designed with these requirements in mind, and they carry additional costs.

Standard ISO9126-01 defines, for two of its three models of quality, the related quality characteristics and sub-characteristics, and measures which are useful for assessing software product quality.

The meaning of the term "product" is extended to include any artifact which is the output of any process used to build the final software product. Examples of a product include, but are not limited to, an entire system requirements specification, a software requirements specification for a software component of a system, a design module, code, test documentation, or reports produced as a result of quality analysis tasks. While most treatments of quality are described in terms of the final software and system performance, sound engineering practice requires that intermediate products relevant to quality be evaluated throughout the software engineering process.

6.4.2.4 Quality Improvement

The quality of software products can be improved through an iterative process of continuous improvement which requires management control, coordination, and feedback from many concurrent processes: the software life cycle processes; the process of error/defect detection, removal, and prevention; and the quality improvement process.

The theory and concepts behind quality improvement, such as building in quality through the prevention and early detection of errors, continuous improvement, and customer focus, are pertinent to software engineering. These concepts are based on the work of experts in quality who have stated that the quality of a product is directly linked to the quality of the process used to create it.

Approaches such as the Total Quality Management (TQM) process of Plan, Do, Check, and Act (PDCA) are tools by which quality objectives can be met. Management sponsorship supports process and product evaluations and the resulting findings. Then, an improvement program is developed identifying detailed actions and improvement projects to be addressed in a feasible time frame. Management support implies that each improvement project has enough resources to achieve the goal defined for it. Management sponsorship must be solicited frequently by implementing proactive communication activities. The involvement of work groups, as well as middle-management support and resources allocated at project level.

6.4.3 Software Quality Management Processes

Software quality management (SQM) applies to all perspectives of software processes, products, and resources. It defines processes, process owners, and requirements for those processes, measurements of the process and its outputs, and feedback channels. Software quality management processes consist of many activities. Some may find defects directly, while others indicate where further examination may be valuable. The latter are also referred to as direct-defect-finding activities. Many activities often serve as both.

Planning for software quality involves:

- Defining the required product in terms of its quality characteristics.
- Planning the processes to achieve the required product.

These aspects differ from, for instance, the planning SQM processes themselves, which assess planned quality characteristics versus actual implementation of those plans. The software quality management processes must address how well software products will, or do, satisfy customer and stakeholder requirements, provide value to the customers and other stakeholders, and provide the software quality needed to meet software requirements.

SQM can be used to evaluate the intermediate products as well as the final product.

Some of the specific SQM processes are defined in standard (IEEE12207.0-96):

- Quality assurance process
- Verification process
- Validation process
- Review process
- Audit process

These processes encourage quality and also find possible problems. But they differ somewhat in their emphasis.

SQM processes help ensure better software quality in a given project. They also provide, as a by-product, general information to management, including an indication of the quality of the entire software engineering process. The Software Engineering Process and Software Engineering Management KAs discuss quality programs for the organization developing the software. SQM can provide relevant feedback for these areas.

SQM processes consist of tasks and techniques to indicate how software plans (for example, management, development, configuration management) are being implemented and how well the intermediate and final products are meeting their specified requirements. Results from these tasks are assembled in reports for management before corrective action is taken. The management of an SQM process is tasked with ensuring that the results of these reports are accurate.

As described in this KA, SQM processes are closely related; they can overlap and are sometimes even combined. They seem largely reactive in nature because they address the processes as practiced and the products as produced; but they have a major role at the planning stage in being proactive in terms of the processes and procedures needed to attain the quality characteristics and degree of quality needed by the stakeholders in the software.

Risk management can also play an important role in delivering quality software. Incorporating disciplined risk analysis and management techniques into the software life cycle processes can increase the potential for producing a quality product.

6.4.3.1 Software Quality Assurance

SQA processes provide assurance that the software products and processes in the project life cycle conform to their specified requirements by planning, enacting, and performing a set of activities to provide adequate confidence that quality is being built into the software. This means ensuring that the problem is clearly and adequately stated and that the solution's requirements are properly defined and expressed. SQA seeks to maintain the quality throughout the development and maintenance of the product by the execution of a variety of activities at each stage which can result in early identification of problems, an almost inevitable feature of any complex activity. The role of SQA with respect to process is to ensure that planned processes are appropriate and later implemented according to plan, and that relevant measurement processes are provided to the appropriate organization.

The SQA plan defines the means that will be used to ensure that software developed for a specific product satisfies the user's requirements and is of the highest quality possible within project constraints. In order to do so, it must first ensure that the quality target is clearly defined and understood. It must consider management, development, and maintenance plans for the software. Refer to standard (IEEE730-98) for details.

The specific quality activities and tasks are laid out, with their costs and resource requirements, their overall management objectives, and their schedule in relation to those objectives in the software engineering management, development, or maintenance plans. The SQA plan should be consistent with the software configuration management plan. The SQA plan identifies documents, standards, practices, and conventions governing the project and how they will be checked and monitored to ensure adequacy and compliance. The SQA plan also identifies measures, statistical techniques, procedures for problem reporting and corrective action, resources such as tools, techniques, and methodologies, security for physical media, training, and SQA reporting and documentation. Moreover, the SQA plan addresses the software quality assurance activities of any other type of activity described in the software plans, such as procurement of supplier software to the project or commercial off-the-shelf software (COTS) installation, and service after delivery of the software. It can also contain acceptance criteria as well as reporting and management activities which are critical to software quality.

6.4.3.2 Verification & Validation

For purposes of brevity, Verification and Validation (V&V) are treated as a single topic in this Guide rather than as two separate topics as in the standard (IEEE12207.0-96). "Software V&V is a disciplined approach to assessing software products throughout the product life cycle. A V&V effort strives to ensure that quality is built into the software and that the software satisfies user requirements" (IEEE1059-93).

V&V addresses software product quality directly and uses testing techniques which can locate defects so that they can be addressed. It also assesses the intermediate products, however, and, in this capacity, the intermediate steps of the software life cycle processes.

The V&V process determines whether or not products of a given development or maintenance activity conform to the requirement of that activity, and whether or not the final software product fulfills its intended purpose and meets user requirements. Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose. Both the verification process and the validation process begin early in the development or maintenance phase. They provide an examination of key product features in relation both to the product's immediate predecessor and to the specifications it must meet.

The purpose of planning V&V is to ensure that each resource, role, and responsibility is clearly assigned. The resulting V&V plan documents and describes the various resources and their roles and activities, as well as the techniques and tools to be used. An understanding of the different purposes of each V&V activity will help in the careful planning of the techniques and resources needed to fulfill their purposes.

The plan also addresses the management, communication, policies, and procedures of the V&V activities and their interaction, as well as defect reporting and documentation requirements.

6.4.3.3 Reviews and Audits

For purposes of brevity, reviews and audits are treated as a single topic in this Guide, rather than as two separate topics as in (IEEE12207.0-96). The review and audit process is broadly defined in (IEEE12207.0-96) and in more detail in (IEEE1028-97). Five types of reviews or audits are presented in the IEEE1028-97 standard:

- Management reviews
- Technical reviews
- Inspections
- Walk-throughs
- Audits

6.4.3.3.1 Management reviews

The purpose of a management review is to monitor progress, determine the status of plans and schedules, confirm requirements and their system allocation, or evaluate the effectiveness of management approaches used to achieve fitness for purpose. They support decisions about changes and corrective actions that are required during a software project. Management reviews determine the adequacy of plans, schedules, and requirements and monitor their progress or inconsistencies. These reviews may be performed on products such as audit reports, progress reports, V&V reports, and plans of many types, including risk management, project management, software configuration management, software safety, and risk assessment, among others.

6.4.3.3.2 Technical reviews

"The purpose of a technical review is to evaluate a software product to determine its suitability for its intended use. The objective is to identify discrepancies from approved specifications and standards. The results should provide management with evidence confirming (or not) that the product meets the specifications and adheres to standards, and that changes are controlled" (IEEE1028-97).

Specific roles must be established in a technical review: a decision-maker, a review leader, a recorder, and technical staff to support the review activities. A technical review requires that mandatory inputs be in place in order to proceed:

- Statement of objectives
- A specific software product
- The specific project management plan
- The issues list associated with this product
- The technical review procedure

The team follows the review procedure. A technically qualified individual presents an overview of the product, and the examination is conducted during one or more meetings. The technical review is completed once all the activities listed in the examination have been completed.

6.4.3.3.3 Inspections

The purpose of an inspection is to detect and identify software product anomalies. Two important differentiators of inspections as opposed to reviews are as follows:

- An individual holding a management position over any member of the inspection team shall not participate in the inspection.
- An inspection is to be led by an impartial facilitator who is trained in inspection techniques.

Software inspections always involve the author of an intermediate or final product, while other reviews might not. Inspections also include an inspection leader, a recorder, a reader, and a few (2 to 5) inspectors. The members of an inspection team may possess different expertise, such as domain expertise, design method expertise, or language expertise. Inspections are usually conducted on one relatively small section of the product at a time. Each team member must examine the software product and other review inputs prior to the review meeting, perhaps by applying an analytical technique to a small section of the product, or to the entire product with a focus only on one aspect, for example, interfaces. Any anomaly found is documented and sent to the inspection leader. During the inspection, the inspection leader conducts the session and verifies that everyone has prepared for the inspection. A checklist, with anomalies and questions germane to the issues of interest, is a common tool used in inspections. The resulting list often classifies the anomalies and is reviewed for completeness and accuracy by the team. The inspection exit decision must correspond to one of the following three criteria:

- Accept with no or at most minor reworking
- Accept with rework verification
- Reinspect

Inspection meetings typically last a few hours, whereas technical reviews and audits are usually broader in scope and take longer.

6.4.3.3.4 Walk-throughs

The purpose of a walk-through is to evaluate a software product. A walk-through may be conducted for the purpose of educating an audience regarding a software product. The major objectives are to:

- Find anomalies
- Improve the software product
- Consider alternative implementations
- Evaluate conformance to standards and specifications

The walk-through is similar to an inspection but is typically conducted less formally. The walk-through is primarily organized by the software engineer to give his teammates the opportunity to review his work, as an assurance technique.

6.4.3.3.5 Audits

The purpose of a software audit is to provide an independent evaluation of the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures. The audit is a formally organized activity, with participants having specific roles, such as lead auditor, another auditor, a recorder, or an initiator, and includes a representative of the audited organization. The audit will identify instances of nonconformance and produce a report requiring the team to take corrective action.

While there may be many formal names for reviews and audits such as those identified in the standard (see IEEE1028- 97), the important point is that they can occur on almost any product at any stage of the development or maintenance process.

6.4.4 Practical Considerations

6.4.4.1 Software Quality Requirements

6.4.4.1.1 Influence factors

Various factors influence planning, management, and selection of SQM activities and techniques, including:

- The domain of the system in which the software will reside (safety-critical, mission-critical, business-critical)
- System and software requirements
- The commercial (external) or standard (internal) components to be used in the system
- The specific software engineering standards applicable
- The methods and software tools to be used for development and maintenance and for quality evaluation and improvement
- The budget, staff, project organization, plans, and scheduling of all the processes
- The intended users and use of the system
- The integrity level of the system

Information on these factors influences how the SQM processes are organized and documented, how specific SQM activities are selected, what resources are needed, and which will impose bounds on the efforts.

6.4.4.1.2 Dependability

In cases where system failure may have extremely severe consequences, overall dependability (hardware, software, and human) is the main quality requirement over and above basic functionality. Software dependability includes such characteristics as fault tolerance, safety, security, and usability. Reliability is also a criterion which can be defined in terms of dependability (ISO9126).

The body of literature for systems must be highly dependable ("high confidence" or "high integrity systems"). Terminology for traditional mechanical and electrical systems which may not include software has been imported for discussing threats or hazards, risks, system integrity, and related concepts, and may be found in the references cited for this section.

6.4.4.1.3 Integrity levels of software

The integrity level is determined based on the possible consequences of failure of the software and the probability of failure. For software in which safety or security is important, techniques such as hazard analysis for safety or threat analysis for security may be used to develop a planning activity which would identify where potential trouble spots lie. The failure history of similar software may also help in identifying which techniques will be most useful in detecting faults and assessing quality.

6.4.4.2 Defect Characterization

SQM processes find defects. Characterizing those defects leads to an understanding of the product, facilitates corrections to the process or the product, and informs project management or the customer of the status of the process or product. Many defect (fault) taxonomies exist, and, while attempts have been made to gain consensus on a fault and failure taxonomy, the literature indicates. Defect (anomaly) characterization is also used in audits and reviews, with the review leader often presenting a list of anomalies provided by team members for consideration at a review meeting.

As new design methods and languages evolve, along with advances in overall software technologies, new classes of defects appear, and a great deal of effort is required to interpret previously defined classes. When tracking defects, the software engineer is interested in not only the number of defects but also the types. Information alone, without some classification, is not really of any use in identifying the underlying causes of the defects, since specific types of problems need to be grouped together in order for determinations to be made about them. The point is to establish a defect taxonomy that is meaningful to the organization and to the software engineers.

SQM discovers information at all stages of software development and maintenance. Typically, where the word "defect" is used, it refers to a "fault" as defined below. However, different cultures and standards may use somewhat different meanings for these terms, which have led to attempts to define them. Partial definitions taken from standard (IEEE610.12-90) are:

- Error: "A difference…between a computed result and the correct result"
- Fault: "An incorrect step, process, or data definition in a computer program"
- Failure: "The [incorrect] result of a fault"
- Mistake: "A human action that produces an incorrect result"
- Failures found in testing as a result of software faults are included as defects in the discussion in this section. Reliability models are built from failure data collected during software testing or from software in service, and thus can be used to predict future failures and to assist in decisions on when to stop testing.

One probable action resulting from SQM findings is to remove the defects from the product under examination. Other actions enable the achievement of full value from the findings of SQM activities. These actions include analyzing and summarizing the findings, and using measurement techniques to improve the product and the process as well as to track the defects and their removal.

Data on the inadequacies and defects found during the implementation of SQM techniques may be lost unless they are recorded. For some techniques (for example, technical reviews, audits, inspections), recorders are present to set down such information, along with issues and decisions. When automated tools are used, the tool output may provide the defect information. Data about defects may be collected and recorded on an SCR (software change request) form and may subsequently be entered into some type of database, either manually or automatically, from an analysis tool. Reports about defects are provided to the management of the organization.

6.4.4.3 Software Quality Management Techniques

SQM techniques can be categorized in many ways: static, people-intensive, analytical, dynamic.

6.4.4.3.1 Static techniques

Static techniques involve examination of the project documentation and software, and other information about the software products, without executing them. These techniques may include people-intensive activities or analytical activities conducted by individuals, with or without the assistance of automated tools.

6.4.4.3.2 People-intensive techniques

The setting for people-intensive techniques, including reviews and audits, may vary from a formal meeting to an informal gathering or a desk-check situation, but (usually, at least) two or more people are involved. Preparation ahead of time may be necessary. Resources other than the items under examination may include checklists and results from analytical techniques and testing. These activities are discussed in (IEEE1028-97) on reviews and audits.

6.4.4.3.3 Analytical techniques

A software engineer generally applies analytical techniques. Sometimes several software engineers use the same technique, but each applies it to different parts of the product. Some techniques are tool-driven; others are manual. Some may find defects directly, but they are typically used to support other techniques. Some also include various assessments as part of overall quality analysis. Examples of such techniques include complexity analysis, control flow analysis, and algorithmic analysis.

Each type of analysis has a specific purpose, and not all types are applied to every project. An example of a support technique is complexity analysis, which is useful for determining whether or not the design or implementation is too complex to develop correctly, to test, or to maintain. The results of a complexity analysis may also be used in developing test cases. Defect-finding techniques, such as control flow analysis, may also be used to support another activity. For software with many algorithms, algorithmic analysis is important, especially when an incorrect algorithm could cause a catastrophic result. There are too many analytical techniques to list them all here. The list and references provided may offer insights into the selection of a technique, as well as suggestions for further reading.

Other, more formal, types of analytical techniques are known as formal methods. They are used to verify software requirements and designs. Proof of correctness applies to critical parts of software. They have mostly been used in the verification of crucial parts of critical systems, such as specific security and safety requirements.

6.4.4.3.4 Dynamic techniques

Different kinds of dynamic techniques are performed throughout the development and maintenance of software. Generally, these are testing techniques, but techniques such as simulation, model checking, and symbolic execution may be considered dynamic. Code reading is considered a static technique, but experienced software engineers may execute the code as they read through it. In this sense, code reading may also qualify as a dynamic technique. This discrepancy in categorizing indicates that people with different roles in the organization may consider and apply these techniques differently.

Some testing may thus be performed in the development process, SQA process, or V&V process, again depending on project organization. Because SQM plans address testing, this section includes some comments on testing.

6.4.4.3.5 Testing

The assurance processes described in SQA and V&V examine every output relative to the software requirement specification to ensure the output's traceability, consistency, completeness, correctness, and performance. This confirmation also includes the outputs of the development and maintenance processes, collecting, analyzing, and measuring the results. SQA ensures that appropriate types of tests are planned, developed, and implemented, and V&V develops test plans, strategies, cases, and procedures.

Two types of testing may fall under the headings SQA and V&V, because of their responsibility for the quality of the materials used in the project:

- Evaluation and test of tools to be used on the project (IEEE1462-98)
- Conformance test (or review of conformance test) of components and COTS products to be used in the product; there now exists a standard for software packages (IEEE1465-98)

Sometimes an independent V&V organization may be asked to monitor the test process and sometimes to witness the actual execution to ensure that it is conducted in accordance with specified procedures. Again, V&V may be called upon to evaluate the testing itself: adequacy of plans and procedures, and adequacy and accuracy of results.

Another type of testing that may fall under the heading of V&V organization is third-party testing. The third party is not the developer, nor is in any way associated with the development of the product. Instead, the third party is an independent facility, usually accredited by some body of authority. Their purpose is to test a product for conformance to a specific set of requirements.

6.4.4.4 Software Quality Measurement

The models of software product quality often include measures to determine the degree of each quality characteristic attained by the product.

If they are selected properly, measures can support software quality (among other aspects of the software life cycle processes) in multiple ways. They can help in the management decision-making process. They can find problematic areas and bottlenecks in the software process; and they can help the software engineers assess the quality of their work for SQA purposes and for longer-term process quality improvement.

With the increasing sophistication of software, questions of quality go beyond whether or not the software works to how well it achieves measurable quality goals.

There are a few more topics where measurement supports SQM directly. These include assistance in deciding when to stop testing. For this, reliability models and benchmarks, both using fault and failure data, are useful.

The cost of SQM processes is an issue which is almost always raised in deciding how a project should be organized. Often, generic models of cost are used, which are based on when a defect is found and how much effort it takes to fix the defect relative to finding the defect earlier in the development process. Project data may give a better picture of cost.

Finally, the SQM reports themselves provide valuable information not only on these processes, but also on how all the software life cycle processes can be improved.

While the measures for quality characteristics and product features may be useful in themselves (for example, the number of defective requirements or the proportion of defective requirements), mathematical and graphical techniques can be applied to aid in the interpretation of the measures. These fit into the following categories:

- Statistically based (for example, Pareto analysis, runcharts, scatter plots, normal distribution)
- Statistical tests (for example, the binomial test, chi-squared test)
- Trend analysis
- Prediction (for example, reliability models)

The statistically based techniques and tests often provide a snapshot of the more troublesome areas of the softwareproduct under examination. The resulting charts andgraphs are visualization aids which the decision-makerscan use to focus resources where they appear most needed. Results from trend analysis may indicate that a schedule has not been respected, such as in testing, or that certain classes of faults will become more intense unless some corrective action is taken in development. The predictive techniques assist in planning test time and in predicting failure.

References:

http://en.wikipedia.org/wiki/Software_quality_assurance, http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-171Fall2003/CourseHome/, http://www.cs.cornell.edu/courses/cs501/2008sp/, http://www.comp.lancs.ac.uk/computing/resources/IanS/SE7/, http://www.ee.unb.ca/kengleha/courses/CMPE3213/IntroT
http://www.softwarecertifications.org/qai_cmsq.htm, http://satc.gsfc.nasa.gov/assure/agbsec3.txt, etc...

6.5 Software configuration management[8]

6.5.1 Introduction

A system can be defined as a collection of components organized to accomplish a specific function or set of functions. The configuration of a system is the functional and/or physical characteristics of hardware, firmware, or software, or a combination of these, as set forth in technical documentation and achieved in a product. It can also be thought of as a collection of specific versions of hardware, firmware, or software items combined according to specific build procedures to serve a particular purpose. Configuration management (CM), then, is the discipline of identifying the configuration of a system at distinct points in time for the purpose of systematically controlling changes to the configuration, and maintaining the integrity and traceability of the configuration throughout the system life cycle. It is formally defined as "A discipline applying technical and administrative direction and surveillance to: identify and document the functional

[8]This content is available online at <http://cnx.org/content/m14730/1.1/>.

and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements."

Software configuration management (SCM) is a critical element of software engineering.

Unfortunately, in practice it is often ignored until absolutely necessary. It may be introduced at first customer release, possibly through customer pressure. Tool support for SCM is limited in that only certain aspects of software development and maintenance are accommodated. SCM methods and tools are often viewed as intrusive by developers, a management tool that imposes additional work with little perceived benefit to the tasks of the developer.

Software configuration management (SCM) is a supporting software life cycle process which benefits project management, development and maintenance activities, assurance activities, and the customers and users of the end product.

The concepts of configuration management apply to all items to be controlled, although there are some differences in implementation between hardware CM and software CM.

SCM is closely related to the software quality assurance (SQA) activity. SQA processes provide assurance that the software products and processes in the project life cycle conform to their specified requirements by planning, enacting, and performing a set of activities to provide adequate confidence that quality is being built into the software. SCM activities help in accomplishing these SQA goals.

The SCM activities are: management and planning of the SCM process, software configuration identification, software configuration control, software configuration status accounting, software configuration auditing, and software release management and delivery.

The figure following shows a stylized representation of these activities:

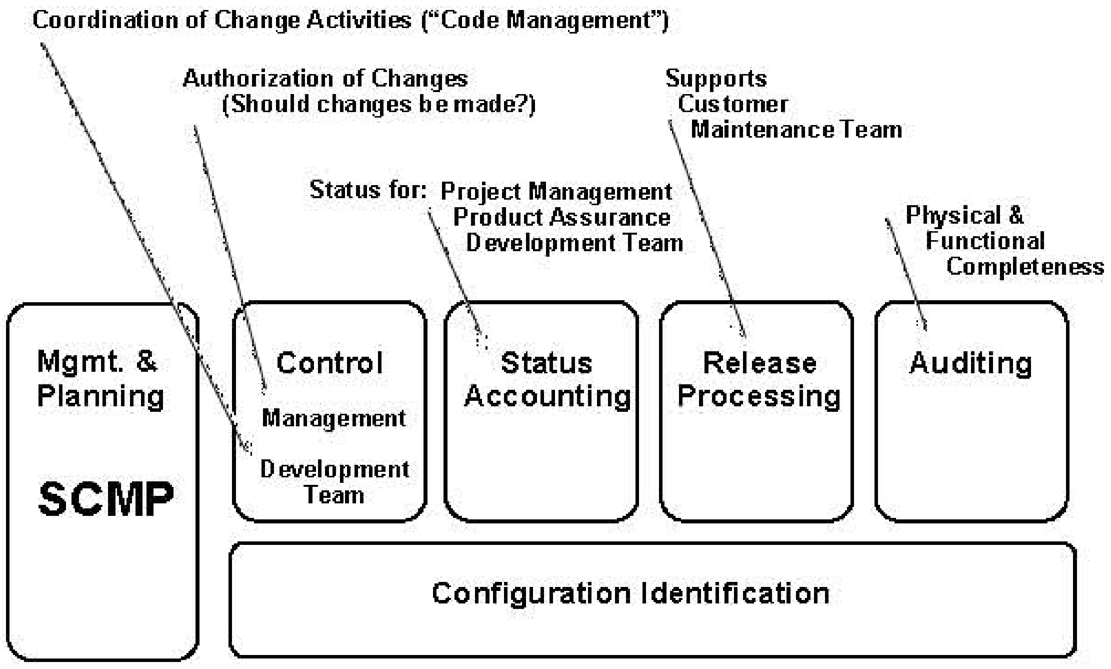

Coordination of Change Activities ("Code Management")
Authorization of Changes (Should changes be made?)
Supports Customer Maintenance Team
Status for: Project Management Product Assurance Development Team
Physical & Functional Completeness
Mgmt. & Planning
SCMP
Control
Management
Development Team
Status Accounting
Release Processing
Auditing
Configuration Identification

Figure 6.7

6.5.2 Management of the SCM Process

SCM controls the evolution and integrity of a product by identifying its elements, managing and controlling change, and verifying, recording, and reporting on configuration information. From the software engineer's perspective, SCM facilitates development and change implementation activities. A successful SCM implementation requires careful planning and management. This, in turn, requires an understanding of the organizational context for, and the constraints placed on, the design and implementation of the SCM process.

6.5.2.1 Organizational Context for SCM

To plan an SCM process for a project, it is necessary to understand the organizational context and the relationships among the organizational elements. SCM interacts with several other activities or organizational elements.

The organizational elements responsible for the software engineering supporting processes may be structured in various ways. Although the responsibility for performing certain SCM tasks might be assigned to other parts of the organization such as the development organization, the overall responsibility for SCM often rests with a distinct organizational element or designated individual.

Software is frequently developed as part of a larger system containing hardware and firmware elements. In this case, SCM activities take place in parallel with hardware and firmware CM activities, and must be consistent with system-level CM. Buckley describes SCM within this context. Note that firmware contains hardware and software, therefore both hardware and software CM concepts are applicable.

SCM might interface with an organization's quality assurance activity on issues such as records management and non-conforming items. Regarding the former, some tems under SCM control might also be project records subject to provisions of the organization's quality assurance program. Managing nonconforming items is usually the responsibility of the quality assurance activity; however, SCM might assist with tracking and reporting on software configuration items falling into this category.

Perhaps the closest relationship is with the software development and maintenance organizations.

It is within this context that many of the software configuration control tasks are conducted. Frequently, the same tools support development, maintenance, and SCM purposes.

6.5.2.2 Constraints and Guidance for the SCM Process

Constraints affecting, and guidance for, the SCM process come from a number of sources. Policies and procedures set forth at corporate or other organizational levels might influence or prescribe the design and implementation of the SCM process for a given project. In addition, the contract between the acquirer and the supplier might contain provisions affecting the SCM process. For example, certain configuration audits might be required, or it might be specified that certain items be placed under CM. When software products to be developed have the potential to affect public safety, external regulatory bodies may impose constraints. Finally, the particular software life cycle process chosen for a software project and the tools selected to implement the software affect the design and implementation of the SCM process.

Guidance for designing and implementing an SCM process can also be obtained from "best practice," as reflected in the standards on software engineering issued by the various standards organizations. Moore provides a roadmap to these organizations and their standards. Best practice is also reflected in process improvement and process assessment models such as the Software Engineering Institute's Capability Maturity Model Integration (SEI/CMMI) and ISO/IEC15504 Software Engineering–Process Assessment (ISO/IEC 15504-98).

6.5.2.3 Planning for SCM

The planning of an SCM process for a given project should be consistent with the organizational context, applicable constraints, commonly accepted guidance, and the nature of the project (for example, size and criticality). The major activities covered are: Software Configuration Identification, Software Configuration Control, Software Configuration Status Accounting, Software Configuration Auditing, and Software Release

Management and Delivery. In addition, issues such as organization and responsibilities, resources and schedules, tool selection and implementation, vendor and subcontractor control, and interface control are typically considered. The results of the planning activity are recorded in an SCM Plan (SCMP), which is typically subject to SQA review and audit.

6.5.2.3.1 SCM organization and responsibilities

To prevent confusion about who will perform given SCM activities or tasks, organizations to be involved in the SCM process need to be clearly identified. Specific responsibilities for given SCM activities or tasks also need to be assigned to organizational entities, either by title or by organizational element. The overall authority and reporting channels for SCM should also be identified, although this might be accomplished at the project management or quality assurance planning stage.

6.5.2.3.2 SCM resources and schedules

Planning for SCM identifies the staff and tools involved in carrying out SCM activities and tasks. It addresses scheduling questions by establishing necessary sequences of SCM tasks and identifying their relationships to the project schedules and milestones established at the project management planning stage. Any training requirements necessary for implementing the plans and training new staff members are also specified.

6.5.2.3.3 Tool selection and implementation

Different types of tool capabilities, and procedures for their use, support SCM activities. Depending on the situation, these tool capabilities can be made available with some combination of manual tools, automated tools providing a single SCM capability, automated tools integrating a range of SCM (and perhaps other) capabilities, or integrated tool environments which serve the needs of multiple participants in the software engineering process (for example, SCM, development, V&V). Automated tool support becomes increasingly important, and increasingly difficult to establish, as projects grow in size and as project environments become more complex. These tool capabilities provide support for:

- the SCM Library
- the software change request (SCR) and approval procedures
- code (and related work products) and change management tasks
- reporting software configuration status and collecting SCM measurements
- software configuration auditing
- managing and tracking software documentation
- performing software builds
- managing and tracking software releases and their delivery

The tools used in these areas can also provide measurements for process improvement. Royce describes seven core measures of value in managing software engineering processes. Information available from the various SCM tools relates to Royce's Work and Progress management indicator and to his quality indicators of Change Traffic and Stability, Breakage and Modularity, Rework and Adaptability, and MTBF (mean time between failures) and Maturity. Reporting on these indicators can be organized in various ways, such as by software configuration item or by type of change requested.

We can represent a mapping of tool capabilities and procedures to SCM Activities:

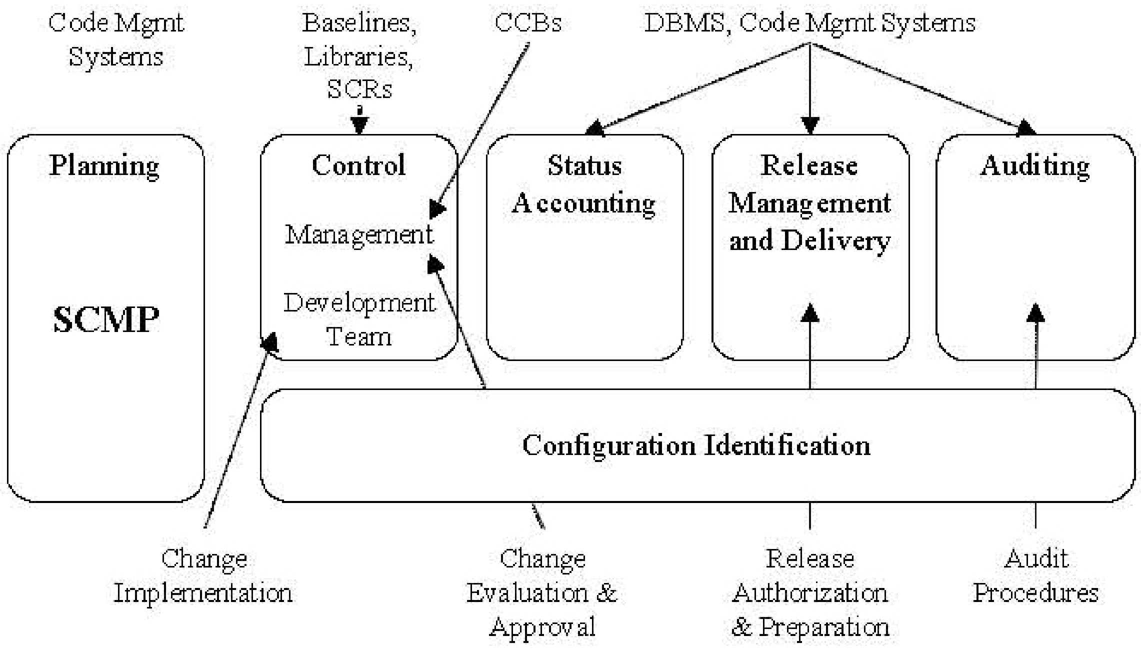


Figure 6.8

In this example, code management systems support the operation of software libraries by controlling access to library elements, coordinating the activities of multiple users, and helping to enforce operating procedures. Other tools support the process of building software and release documentation from the software elements contained in the libraries. Tools for managing software change requests support the change control procedures applied to controlled software items. Other tools can provide database management and reporting capabilities for management, development, and quality assurance activities. As mentioned above, the capabilities of several tool types might be integrated into SCM systems, which in turn are closely coupled to various other software activities.

In planning, the software engineer picks SCM tools fit for the job. Planning considers issues that might arise in the implementation of these tools, particularly if some form of culture change is necessary.

6.5.2.3.4 Vendor/Subcontractor Control

A software project might acquire or make use of purchased software products, such as compilers or other tools. SCM planning considers if and how these items will be taken under configuration control (for example, integrated into the project libraries) and how changes or updates will be evaluated and managed.

Similar considerations apply to subcontracted software. In this case, the SCM requirements to be imposed on the subcontractor's SCM process as part of the subcontract and the means for monitoring compliance also need to be established. The latter includes consideration of what SCM information must be available for effective compliance monitoring.

6.5.2.3.5 Interface control

When a software item will interface with another software or hardware item, a change to either item can affect the other. The planning for the SCM process considers how the interfacing items will be identified and how changes to the items will be managed and communicated. The SCM role may be part of a larger, system-level process for interface specification and control, and may involve interface specifications, interface control plans, and interface control documents. In this case, SCM planning for interface control takes place within the context of the system-level process.

6.5.2.4 SCM Plan

The results of SCM planning for a given project are recorded in a Software Configuration Management Plan (SCMP), a "living document" which serves as a reference for the SCM process. It is maintained (that is, updated and approved) as necessary during the software life cycle. In implementing the SCMP, it is typically necessary to develop a number of more detailed, subordinate procedures defining how specific requirements will be carried out during day-to-day activities.

Guidance on the creation and maintenance of an SCMP, based on the information produced by the planning activity, is available from a number of sources, such as IEEE828-98. This reference provides requirements for the information to be contained in an SCMP. It also defines and describes six categories of SCM information to be included in an SCMP:

- Introduction (purpose, scope, terms used)
- SCM Management (organization, responsibilities, authorities, applicable policies, directives, and procedures)
- SCM Activities (configuration identification, configuration control, and so on)
- SCM Schedules (coordination with other project activities)
- SCM Resources (tools, physical resources, and humanresources)
- SCMP Maintenance

6.5.2.5 Surveillance of Software Configuration Management

After the SCM process has been implemented, some degree of surveillance may be necessary to ensure that the provisions of the SCMP are properly carried out. There are likely to be specific SQA requirements for ensuring compliance with specified SCM processes and procedures. This could involve an SCM authority ensuring that those with the assigned responsibility perform the defined SCM tasks correctly. The software quality assurance authority, as part of a compliance auditing activity, might also perform this surveillance.

The use of integrated SCM tools with process control capability can make the surveillance task easier. Some tools facilitate process compliance while providing flexibility for the software engineer to adapt procedures. Other tools enforce process, leaving the software engineer with less flexibility. Surveillance requirements and the level of flexibility to be provided to the software engineer are important considerations in tool selection.

6.5.2.5.1 SCM measures and measurement

SCM measures can be designed to provide specific information on the evolving product or to provide insight into the functioning of the SCM process. A related goal of monitoring the SCM process is to discover opportunities for process improvement. Measurements of SCM processes provide a good means for monitoring the effectiveness of SCM activities on an ongoing basis. These measurements are useful in characterizing the current state of the process, as well as in providing a basis for making comparisons over time. Analysis of the measurements may produce insights leading to process changes and corresponding updates to the SCMP.

Software libraries and the various SCM tool capabilities provide sources for extracting information about the characteristics of the SCM process (as well as providing project and management information). For example, information about the time required to accomplish various types of changes would be useful in an evaluation of the criteria for determining what levels of authority are optimal for authorizing certain types of changes.

Care must be taken to keep the focus of the surveillance on the insights that can be gained from the measurements, not on the measurements themselves.

6.5.2.5.2 In-process audits of SCM

Audits can be carried out during the software engineering process to investigate the current status of specific elements of the configuration or to assess the implementation of the SCM process. In-process auditing of SCM provides a more formal mechanism for monitoring selected aspects of the process and may be coordinated with the SQA function.

6.5.3 Software Configuration Identification

The software configuration identification activity identifies items to be controlled, establishes identification schemes for the items and their versions, and establishes the tools and techniques to be used in acquiring and managing controlled items. These activities provide the basis for the other SCM activities.

6.5.3.1 Identifying Items to Be Controlled

A first step in controlling change is to identify the software items to be controlled. This involves understanding the software configuration within the context of the system configuration, selecting software configuration items, developing a strategy for labeling software items and describing their relationships, and identifying the baselines to be used, along with the procedure for a baseline's acquisition of the items.

6.5.3.1.1 Software configuration

A software configuration is the set of functional and physical characteristics of software as set forth in the technical documentation or achieved in a product. It can be viewed as a part of an overall system configuration.

6.5.3.1.2 Software configuration item

A software configuration item (SCI) is an aggregation of software designated for configuration management and is treated as a single entity in the SCM process. A variety of items, in addition to the code itself, is typically controlled by SCM. Software items with potential to become SCIs include plans, specifications and design documentation, testing materials, software tools, source and executable code, code libraries, data and data dictionaries, and documentation for installation, maintenance, operations, and software use.

Selecting SCIs is an important process in which a balance must be achieved between providing adequate visibility for project control purposes and providing a manageable number of controlled items.

6.5.3.1.3 Software configuration item relationships

The structural relationships among the selected SCIs, and their constituent parts, affect other SCM activities or tasks, such as software building or analyzing the impact of proposed changes. Proper tracking of these relationships is also important for supporting traceability. The design of the identification scheme for SCIs should consider the need to map the identified items to the software structure, as well as the need to support the evolution of the software items and their relationships.

6.5.3.1.4 Software version

Software items evolve as a software project proceeds. A version of a software item is a particular identified and specified item. It can be thought of as a state of an evolving item. A revision is a new version of an item that is intended to replace the old version of the item. A variant is a new version of an item that will be added to the configuration without replacing the old version.

6.5.3.1.5 Baseline

A software baseline is a set of software configuration items formally designated and fixed at a specific time during the software life cycle. The term is also used to refer to a particular version of a software configuration item that has been agreed on. In either case, the baseline can only be changed through formal change control procedures. A baseline, together with all approved changes to the baseline, represents the current approved configuration.

Commonly used baselines are the functional, allocated, developmental, and product baselines. The functional baseline corresponds to the reviewed system requirements. The allocated baseline corresponds to the reviewed software requirements specification and software interface requirements specification. The developmental baseline represents the evolving software configuration at selected times during the software life cycle. Change authority for this baseline typically rests primarily with the development organization, but may be shared with other organizations (for example, SCM or Test). The product baseline corresponds to the completed software product delivered for system integration. The baselines to be used for a given project, along with their associated levels of authority needed for change approval, are typically identified in the SCMP.

6.5.3.1.6 Acquiring software configuration items

Software configuration items are placed under SCM control at different times; that is, they are incorporated into a particular baseline at a particular point in the software life cycle. The triggering event is the completion of some form of formal acceptance task, such as a formal review.

This is an acquisition of items:

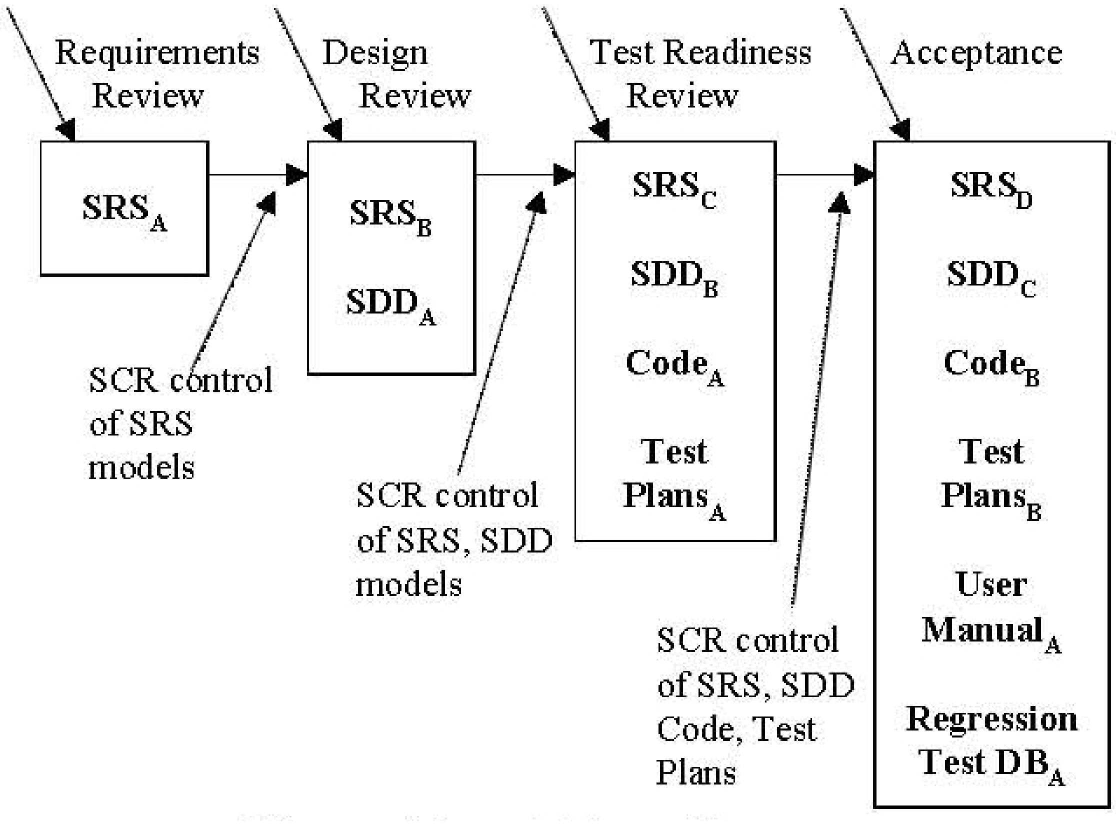


Figure 6.9

Following the acquisition of an SCI, changes to the item must be formally approved as appropriate for the SCI and the baseline involved, as defined in the SCMP. Following approval, the item is incorporated into the software baseline according to the appropriate procedure.

6.5.3.2 Software Library

A software library is a controlled collection of software and related documentation designed to aid in software development, use, and maintenance. It is also instrumental in software release management and delivery activities. Several types of libraries might be used, each corresponding to a particular level of maturity of the software item. For example, a working library could support coding and a project support library could support testing, while a master library could be used for finished products. An appropriate level of SCM control (associated baseline and level of authority for change) is associated with each library. Security, in terms of access control and the backup facilities, is a key aspect of library management.

The tool(s) used for each library must support the SCM control needs for that library, both in terms of controlling SCIs and controlling access to the library. At the working library level, this is a code management capability serving developers, maintainers, and SCM. It is focused on managing the versions of software items while supporting the activities of multiple developers. At higher levels of control, access is more restricted and SCM is the primary user.

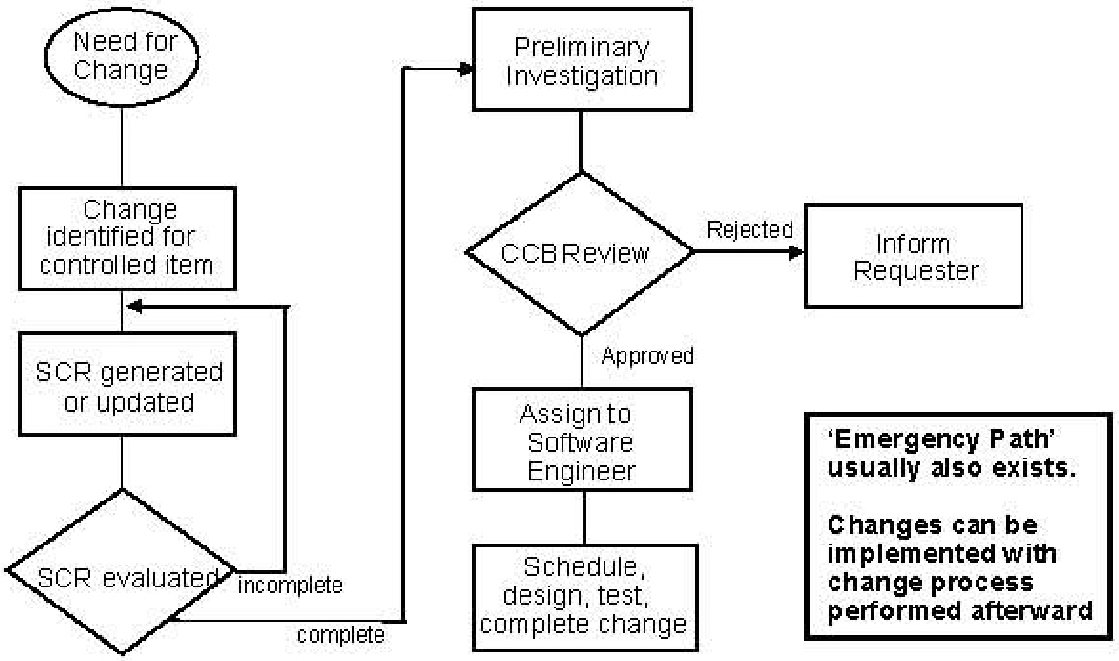
These libraries are also an important source of information for measurements of work and progress.

6.5.4 Software Configuration Control

Software configuration control is concerned with managing changes during the software life cycle. It covers the process for determining what changes to make, the authority for approving certain changes, support for the implementation of those changes, and the concept of formal deviations from project requirements, as well as waivers of them. Information derived from these activities is useful in measuring change traffic and breakage, and aspects of rework.

6.5.4.1 Requesting, Evaluating, and Approving Software Changes

The first step in managing changes to controlled items is determining what changes to make. The software change request process provides formal procedures for submitting and recording change requests, evaluating the potential cost and impact of a proposed change, and accepting, modifying, or rejecting the proposed change. Requests for changes to software configuration items may be originated by anyone at any point in the software life cycle and may include a suggested solution and requested priority. One source of change requests is the initiation of corrective action in response to problem reports. Regardless of the source, the type of change (for example, defect or enhancement) is usually recorded on the SCR.


Need for Change
Change identified for controlled item
SCR generated or updated
SCR evaluated
incomplete
complete
Preliminary Investigation
CCB Review
Rejected
Inform Requester
Approved
Assign to Software Engineer
Schedule, design, test, complete change
'Emergency Path' usually also exists.
Changes can be implemented with change process performed afterward

This flow of a Change Control Process provides an opportunity for tracking defects and collecting change activity measurements by change type. Once an SCR is received, a technical evaluation (also known as an impact analysis) is performed to determine the extent of the modifications that would be necessary should the change request be accepted. A good understanding of the relationships among software (and possibly, hardware) items is important for this task. Finally, an established authority, commensurate with the affected baseline, the SCI involved, and the nature of the change, will evaluate the technical and managerial aspects of the change request and either accept, modify, reject, or defer the proposed change.

6.5.4.1.1 Software Configuration Control Board

The authority for accepting or rejecting proposed changes rests with an entity typically known as a Configuration Control Board (CCB). In smaller projects, this authority may actually reside with the leader or an assigned individual rather than a multi-person board. There can be multiple levels of change authority depending on a variety of criteria, such as the criticality of the item involved, the nature of the change (for example, impact on budget and schedule), or the current point in the life cycle. The composition of the CCBs used for a given system varies depending on these criteria (an SCM representative would always be present). All stakeholders, appropriate to the level of the CCB, are represented. When the scope of authority of a CCB is strictly software, it is known as a Software Configuration Control Board (SCCB). The activities of the CCB are typically subject to software quality audit or review.

6.5.4.1.2 Software change request process

An effective software change request (SCR) process requires the use of supporting tools and procedures ranging from paper forms and a documented procedure to an electronic tool for originating change requests, enforcing the flow of the change process, capturing CCB decisions, and reporting change process information. A link between this tool capability and the problem-reporting system can facilitate the tracking of solutions for reported problems. Change process descriptions and supporting forms (information) are given in a variety of references.

6.5.4.2 Implementing Software Changes

Approved SCRs are implemented using the defined software procedures in accordance with the applicable schedule requirements. Since a number of approved SCRs might be implemented simultaneously, it is necessary to provide a means for tracking which SCRs are incorporated into particular software versions and baselines. As part of the closure of the change process, completed changes may undergo configuration audits and software quality verification. This includes ensuring that only approved changes have been made. The change request process described above will typically document the SCM (and other) approval information for the change.

The actual implementation of a change is supported by the library tool capabilities, which provide version management and code repository support. At a minimum, these tools provide check-in/out and associated version control capabilities. More powerful tools can support parallel development and geographically distributed environments. These tools may be manifested as separate specialized applications under the control of an independent SCM group. They may also appear as an integrated part of the software engineering environment. Finally, they may be as elementary as a rudimentary change control system provided with an operating system.

6.5.4.3 Deviations and Waivers

The constraints imposed on a software engineering effort or the specifications produced during the development activities might contain provisions which cannot be satisfied at the designated point in the life cycle. A deviation is an authorization to depart from a provision prior to the development of the item. A waiver is an authorization to use an item, following its development, that departs from the provision in some way. In these cases, a formal process is used for gaining approval for deviations from, or waivers of, the provisions.

6.5.5 Software Configuration Status Accounting

Software configuration status accounting (SCSA) is the recording and reporting of information needed for effective management of the software configuration.

6.5.5.1 Software Configuration Status Information

The SCSA activity designs and operates a system for the capture and reporting of necessary information as the life cycle proceeds. As in any information system, the configuration status information to be managed for the evolving configurations must be identified, collected, and maintained. Various information and measurements are needed to support the SCM process and to meet the configuration status reporting needs of management, software engineering, and other related activities. The types of information available include the approved configuration identification, as well as the identification and current implementation status of changes, deviations, and waivers.

Some form of automated tool support is necessary to accomplish the SCSA data collection and reporting tasks. This could be a database capability, or it could be a standalone tool or a capability of a larger, integrated tool environment.

6.5.5.2 Software Configuration Status Reporting

Reported information can be used by various organizational and project elements, including the development team, the maintenance team, project management, and software quality activities. Reporting can take the form of ad hoc queries to answer specific questions or the periodic production of predesigned reports. Some information produced by the status accounting activity during the course of the life cycle might become quality assurance records.

In addition to reporting the current status of the configuration, the information obtained by the SCSA can serve as a basis for various measurements of interest to management, development, and SCM. Examples include the number of change requests per SCI and the average time needed to implement a change request.

6.5.6 Software Configuration Auditing

A software audit is an activity performed to independently evaluate the conformance of software products and processes to applicable regulations, standards, guidelines, plans, and procedures. Audits are conducted according to a well-defined process consisting of various auditor roles and responsibilities. Consequently, each audit must be carefully planned. An audit can require a number of individuals to perform a variety of tasks over a fairly short period of time. Tools to support the planning and conduct of an audit can greatly facilitate the process.

The software configuration auditing activity determines the extent to which an item satisfies the required functional and physical characteristics. Informal audits of this type can be conducted at key points in the life cycle. Two types of formal audits might be required by the governing contract (for example, in contracts covering critical software): the Functional Configuration Audit (FCA) and the Physical Configuration Audit (PCA). Successful completion of these audits can be a prerequisite for the establishment of the product baseline. Buckley contrasts the purposes of the FCA and PCA in hardware versus software contexts, and recommends careful evaluation of the need for a software FCA and PCA before performing them.

6.5.6.1 Software Functional Configuration Audit

The purpose of the software FCA is to ensure that the audited software item is consistent with its governing specifications. The output of the software verification and validation activities is a key input to this audit.

6.5.6.2 Software Physical Configuration Audit

The purpose of the software physical configuration audit (PCA) is to ensure that the design and reference documentation is consistent with the as-built software product.

6.5.6.3 In-process Audits of a Software Baseline

As mentioned above, audits can be carried out during the development process to investigate the current status of specific elements of the configuration. In this case, an audit could be applied to sampled baseline items to ensure that performance is consistent with specifications or to ensure that evolving documentation continues to be consistent with the developing baseline item.

6.5.7 Software Release Management and Delivery

The term "release" is used in this context to refer to the distribution of a software configuration item outside the development activity. This includes internal releases as well as distribution to customers. When different versions of a software item are available for delivery, such as versions for different platforms or versions with varying capabilities, it is frequently necessary to recreate specific versions and package the correct materials for delivery of the version. The software library is a key element in accomplishing release and delivery tasks.

6.5.7.1 Software Building

Software building is the activity of combining the correct versions of software configuration items, using the appropriate configuration data, into an executable program for delivery to a customer or other recipient, such as the testing activity. For systems with hardware or firmware, the executable program is delivered to the system-building activity. Build instructions ensure that the proper build steps are taken and in the correct sequence. In addition to building software for new releases, it is usually also necessary for SCM to have the capability to reproduce previous releases for recovery, testing, maintenance, or additional release purposes.

Software is built using particular versions of supporting tools, such as compilers. It might be necessary to rebuild an exact copy of a previously built software configuration item. In this case, the supporting tools and associated build instructions need to be under SCM control to ensure availability of the correct versions of the tools.

A tool capability is useful for selecting the correct versions of software items for a given target environment and for automating the process of building the software from the selected versions and appropriate configuration data. For large projects with parallel development or distributed development environments, this tool capability is necessary. Most software engineering environments provide this capability. These tools vary in complexity from requiring the software engineer to learn a specialized scripting language to graphics-oriented approaches that hide much of the complexity of an "intelligent" build facility.

The build process and products are often subject to software quality verification. Outputs of the build process might be needed for future reference and may become quality assurance records.

6.5.7.2 Software Release Management

Software release management encompasses the identification, packaging, and delivery of the elements of a product, for example, executable program, documentation, release notes, and configuration data. Given that product changes can occur on a continuing basis, one concern for release management is determining when to issue a release. The severity of the problems addressed by the release and measurements of the fault densities of prior releases affect this decision. The packaging task must identify which product items are to be delivered, and then select the correct variants of those items, given the intended application of the product. The information documenting the physical contents of a release is known as a version description document. The release notes typically describe new capabilities, known problems, and platform requirements necessary for proper product operation. The package to be released also contains installation or upgrading instructions. The latter can be complicated by the fact that some current users might have versions that are several releases old. Finally, in some cases, the release management activity might be required to track the distribution of the product to various customers or target systems. An example would be a case where the supplier was required to notify a customer of newly reported problems.

A tool capability is needed for supporting these release management functions. It is useful to have a connection with the tool capability supporting the change request process in order to map release contents to the SCRs that have been received. This tool capability might also maintain information on various target platforms and on various customer environments.

6.6 Starting a new Web project[9]

Before Beginning Website Construction

Before beginning development on a site it is important to be able to answer a few questions in order to understand what you need to be developing and why. Below I will offer my suggestions. My suggestions are based in part on the concepts of John December, who wrote The World Wide Web Unleashed; HTML 3.2 and Cgi Unleashed; and Presenting Java.

John December's methodology involves six sets of information called elements:

1. Audience information - a store of knowledge about the target audience for the web as well as the actual audience who uses the web.
2. Purpose statement - defines the reason for and scope of the web's existence.
3. Objectives list defines the specific goals the web should accomplish.
4. Domain information - a collection of knowledge and information about the subject domain the web covers.
5. Web specification - a detailed description of the constraints and elements that will go into the web.
6. Web presentation - the full description of the technical structures (hypertext and other media) by which the web is delivered to the users.

You can view more about John December at http://www.december.com[10] .

Steps to Take Before Beginning Construction

1. Determine who your audience is. Who are you building it to service? What are assumptions you make about this audience? When might members of the audience have conflicting needs? How many people do you hope to have hitting your site?Example: Our intended audience is students interested in distance education. They may be any age range although generally over 20 years old. They may be any gender and may come from anywhere in the United States. Some may come from overseas but we will focus on United States. They may or may not have disabilities such as sight impairment. They likely will have completed high school and are looking for options for continuing their education. We assume the user:1) has familiarity with the Web; 2) is interested in pursuing additional education; 3) are looking to find out more about us and have found us either due to a search on distance education or by knowing our direct URL.
2. Determine the stakeholders. Determine who internally will be impacted by a new site. Discuss ways in which they could be impacted. Try to include members from each main stakeholder group in all discussions. Support form all stakeholder groups can make or break your project.
3. Determine your overall purpose for the site. This may or may not match the companies overall purpose, but should definitely complement the companies overall purpose. This should be a few sentences.Example: The purpose of our site is to disseminate information on courses and degrees offered at our institution
4. Determine your goals and objectives. Goals are basically from the viewpoint of the company and what you want to see happen, whereas objectives are commonly what you want the end user to end up doing.Example:Goal1: Inspire students to view our online list of courses and entice them to then register for a courseObjective1: Have the end user go to our online list of courses and either register for a course or inquire for additional information
5. Evaluate your resources. Determine what type of manpower, machinery and budget you have to work with. Are there system limitations? Will you have maintainability issues (such as if contracting out)? What programmers do you have in house? What are their skills? What is their current workload?
6. Create a wish list of what you would like. Often creating this manner of a list is done in a brainstorming session. This should be done with members from various parts of the organization. Don't worry about limiting the list at this point. You may even end up with requests that directly conflict with one another based on the needs of different audiences or stakeholder groups.

[9]This content is available online at <http://cnx.org/content/m15998/1.1/>.
[10]http://www.december.com/

7. Set priorities. Determine what on the wish list is a requirement and what is nice to have. Determine what your bare minimum would be, and then determine what you would incorporate in phases after that (level 2 requests, level 3 requests). Iron out problems with conflicting requests (such as whether or not to list prices)
8. Determine feasibility based on currently available resources.
9. Develop a "straw man". This is basically a mapped out diagram or text document showing what you envision and how you envision it being organized. Bear in mind that there are two types of structures: 1) what is visible to the end user and 2) what the structure is on the underlying system (where things are located physically for example), so you may want more than one document/chart. Also determine at this point what type of depth you are envisioning. How much detail will there be? How much will users be able to drill down? What parts would use multimedia or interactivity? Does this answer to your goals and objectives?
10. Create a detailed specification of what you want. The more precise and detailed you can be the better. This will be an offshoot of your "straw man" but will be written with the developers in mind so they know what you are looking for. It will also be to ensure that all stakeholders have the same view of what is being requested.
11. Determine who is responsible for what. Who will be responsible for the content development now and in the future? Who will be responsible for the actual coding? Who will maintain it? Who will address security concerns? Who will maintain the server? Who has final say where there is disagreement about content? Who does one go to if they experience problems or have questions? You may want to include this in your detailed specification.
12. Ensure all parties are in agreement to the detailed specification. Do developers agree it is feasible? If so, do they know how they will proceed? Does management agree with the structure and what will be presented? Do they agree with the anticipated sticker price?

Once you have completed the above steps you will be in a good position to begin building your first drafts of the framework. Remember before you start though and throughout the process to ensure you are keeping your primary goals and objectives in sight.

6.7 Flowcharting[11]

6.7.1 Flowcharting Symbols

6.7.1.1 Terminal

The rounded rectangles, or terminal points, indicate the flowchart's starting and ending points.

[11] This content is available online at <http://cnx.org/content/m19623/1.7/>.

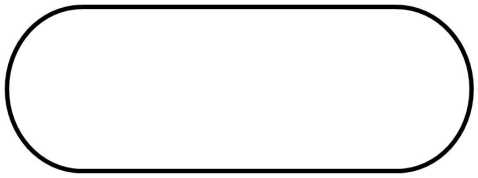


Figure 6.10

6.7.1.2 Process

The rectangle depicts a process such as a mathematical computation, or a variable assignment.
Note: the C++ language equivalent is the statement.

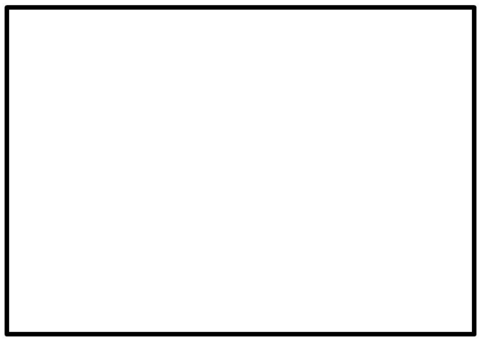


Figure 6.11

6.7.1.3 Input/Output

The parallelograms designate input or output operations.
Note: the C++ language equivalent is cin or cout.

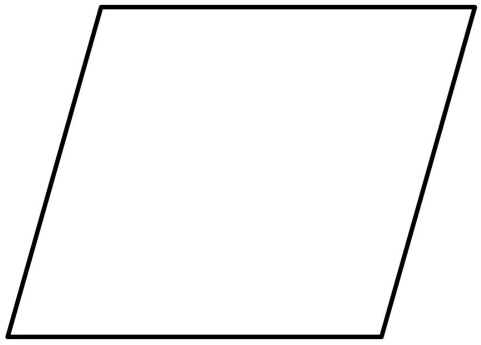


Figure 6.12

6.7.1.4 Connectors

Sometimes a flowchart is broken into two or more smaller flowcharts. This is usually done when a flowchart does not fit on a single page, or must be divided into sections. A connector symbol, which is a small circle with a letter or number inside it, allows you to connect two flowcharts on the same page. A connector symbol that looks like a pocket on a shirt, allows you to connect to a flowchart on a different page.

On-Page Connector

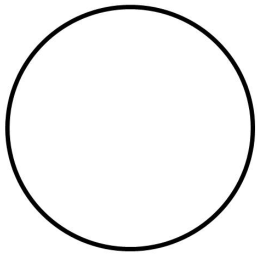


Figure 6.13

Off-Page Connector

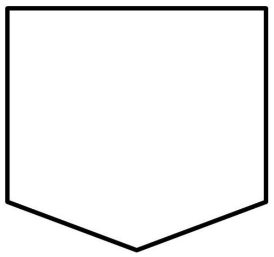


Figure 6.14

6.7.1.5 Decision

The diamond is used to represent the true/false statement being tested in a decision symbol.

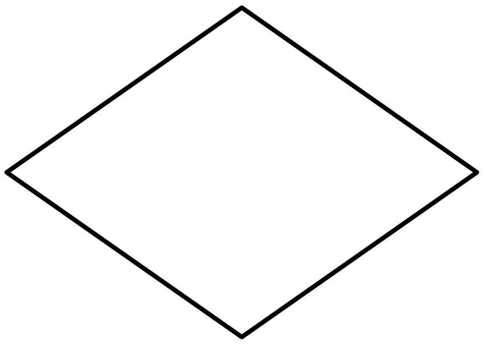


Figure 6.15

6.7.1.6 Module Call

A program module is represented in a flowchart by rectangle with some lines to distinguish it from process symbol. Often programmers will make a distinction between program control and specific task modules as shown below.

Note: C++ equivalent is the function.

Local module: usually a program control function.

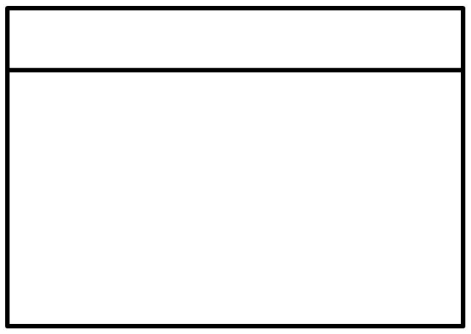


Figure 6.16

Library module: usually a specific task function.

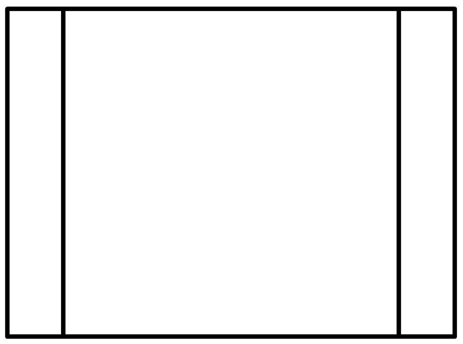


Figure 6.17

6.7.1.7 Flow Lines

Note: The default flow is left to right and top to bottom (the same way you read English). To save time arrowheads are often only drawn when the flow lines go contrary the normal.

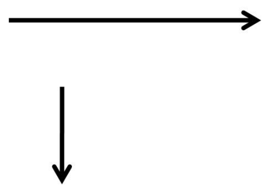


Figure 6.18

6.7.2 Examples

We will demonstrate various flowcharting items by showing the flowchart for some pseudocode.

6.7.2.1 Functions

Example 6.1: pseudocode: Function with no parameter passing

```
Function clear monitor
  Pass In: nothing
  Direct the operating system to clear the monitor
  Pass Out: nothing
Endfunction
```

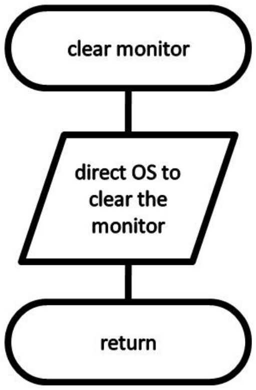


Figure 6.19: Function clear monitor

Example 6.2: pseudocode: Function main calling the clear monitor function

```
Function main
  Pass In: nothing
  Doing some lines of code
  Call: clear monitor
  Doing some lines of code
  Pass Out: value zero to the operating system
Endfunction
```

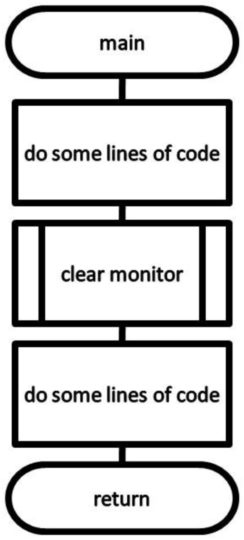


Figure 6.20: Function main

6.7.2.2 Sequence Control Structures

The next item is pseudocode for a simple temperature conversion program. This demonstrates the use of both the on-page and off-page connectors. It also illustrates the sequence control structure where nothing unusually happens. Just do one instruction after another in the sequence listed.

Example 6.3: pseudocode: Sequence control structure

```
Filename: Solution_Lab_04_Pseudocode.txt
Purpose:  Convert Temperature from Fahrenheit to Celsius
Author:   Ken Busbee; © 2008 Kenneth Leroy Busbee
Date:     Dec 24, 2008

Pseudocode = IPO Outline

input
  display a message asking user for the temperature in Fahrenheit
  get the temperature from the keyboard
processing
  calculate the Celsius by subtracting 32 from the Fahrenheit
  temperature then multiply the result by 5 then
  divide the result by 9. Round up or down to the whole number.
  HINT: Use 32.0 when subtracting to ensure floating-point accuracy.
output
  display the celsius with an appropriate message
```

```
pause so the user can see the answer
```

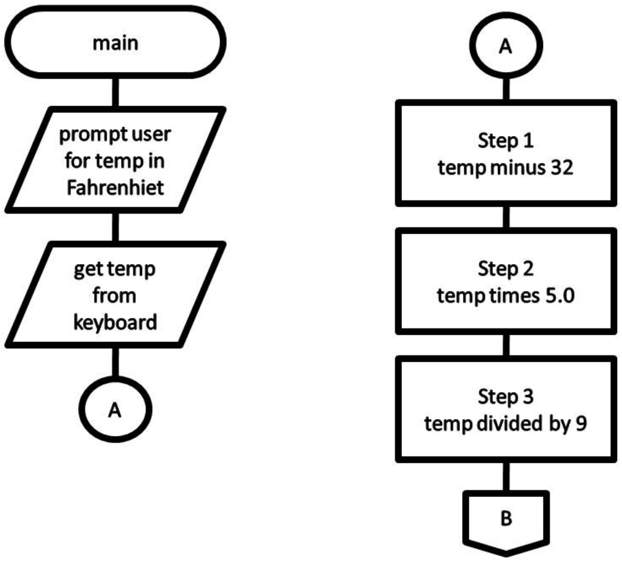


Figure 6.21: Sequence control structure

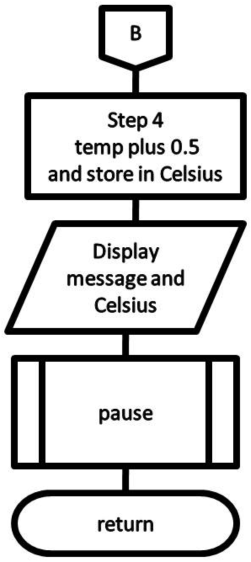


Figure 6.22: Sequence control structured continued

6.7.2.3 Selection Control Structures

Example 6.4: pseudocode: If then Else

```
If age > 17
  Display a message indicating you can vote.
Else
  Display a message indicating you can't vote.
Endif
```

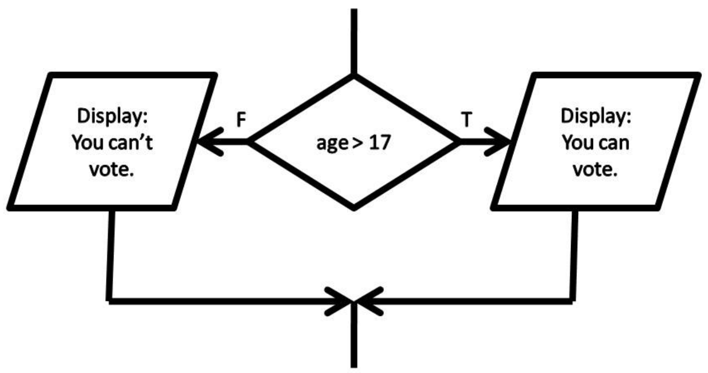


Figure 6.23: If then Else control structure

Example 6.5: pseudocode: Case

```
Case of age
  0 to 17   Display "You can't vote."
  18 to 64  Display "Your in your working years."
  65 +      Display "You should be retired."
Endcase
```

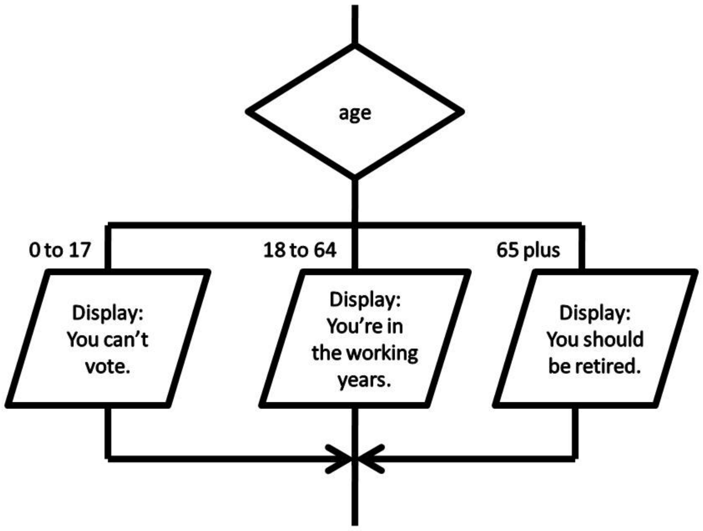

age
0 to 17
18 to 64
65 plus
Display: You can't vote.
Display: You're in the working years.
Display: You should be retired.

Figure 6.24: Case control structure

6.7.2.4 Iteration (Repetition) Control Structures

Example 6.6: pseudocode: While

```
count assigned zero
While count < 5
  Display "I love computers!"
  Increment count
Endwhile
```

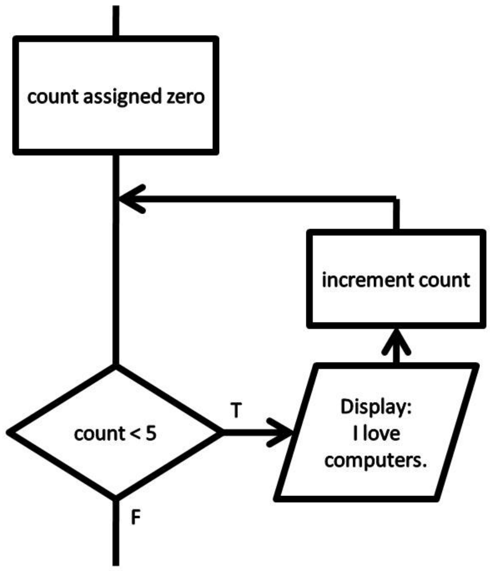


Figure 6.25: While control structure

Example 6.7: pseudocode: For

```
For x starts at 0, x < 5, increment x
  Display "Are we having fun?"
Endfor
```

The for loop does not have a standard flowcharting method and you will find it done in different ways. The for loop as a counting loop can be flowcharted similar to the while loop as a counting loop.

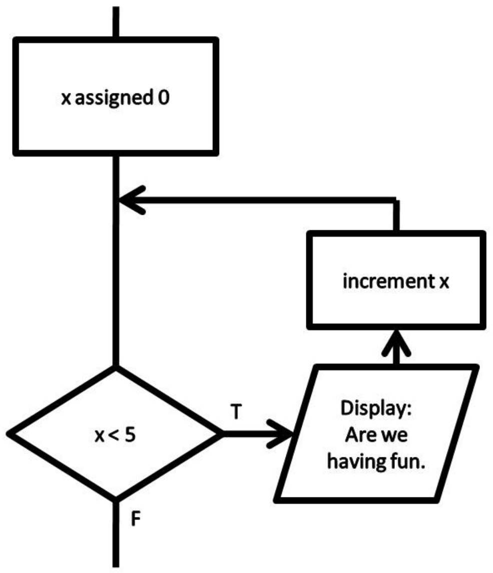


Figure 6.26: For control structure

Example 6.8: pseudocode: Do While

```
count assigned five
Do
  Display "Blast off is soon!"
  Decrement count
While count > zero
```

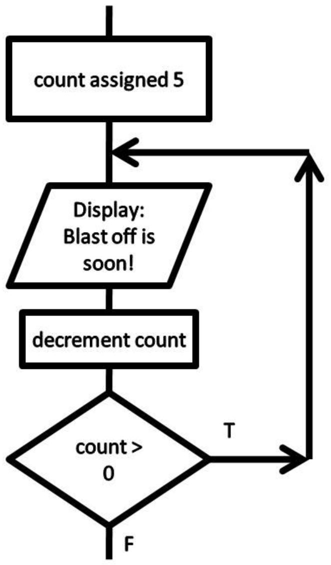


Figure 6.27: Do While control structure

Example 6.9: pseudocode: Repeat Until

```
count assigned five
Repeat
  Display "Blast off is soon!"
  Decrement count
Until count < one
```

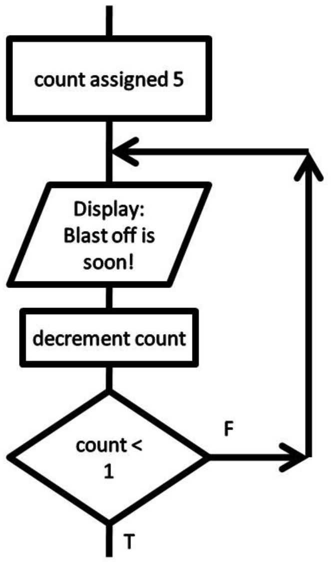


Figure 6.28: Repeat Until control structure

6.7.3 Definitions

Definition 6.1: flowcharting
A programming design tool that uses graphical elements to visually depict the flow of logic within a function.

Definition 6.2: process symbol
A rectangle used in flowcharting for normal processes such as assignment.

Definition 6.3: input/output symbol
A parallelogram used in flowcharting for input/output interactions.

Definition 6.4: decision symbol
A diamond used in flowcharting for asking a question and making a decision.

Definition 6.5: flow lines
Lines (sometimes with arrows) that connect the various flowcharting symbols.