MODULE - 4

SOFTWARE PROJECT MANAGEMENT

- Software project management is an essential part of software engineering.
- The success criteria **for project management** obviously vary from project to project, but, for most projects, **important goals are:**
    1. to deliver the software to the customer at the agreed time;
    2. to keep overall costs within budget
    3. to deliver software that meets the customer’s expectations;
    4. to maintain a coherent and well-functioning development team.
- Software engineering is different from other types of engineering in a number of ways:
    1. The product is intangible.
    2. Large software projects are often “one-off” projects.
    3. Software processes are variable and organization-specific.

- It is impossible to write a standard job description for a software project manager.
- Some of the most **important factors that affect how software projects are managed are:**
  1. Company size
  2. Software customers
  3. Software size
  4. Software type
  5. Organizational culture
  6. Software development processes

- The **fundamental project management activities** that are common to all organizations:
    1. **Project planning** → Project managers are responsible for planning, estimating, and scheduling project development and assigning people to tasks.
    2. **Risk management** → Project managers have to assess the risks that may affect a project, monitor these risks, and take action when problems arise.
    3. **People management** → Project managers are responsible for managing a team of people. They have to choose people for their team and establish ways of working that lead to effective team performance.
    4. **Reporting** → Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software.
    5. **Proposal writing** → The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates and justifies why the project contract should be awarded to a particular organization or team.

Risk Management

- Risk management is one of the most important jobs for a project manager.
- Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks.
- Risks can be categorized according to type of risk (technical, organizational, etc.)

- **Classification of risks according to what these risks affect:**
  1. **Project risks** → affect the project schedule or resources. An example of a project risk is the loss of an experienced system architect.
  2. **Product risks** → affect the quality or performance of the software being developed. An example of a product risk is the failure of a purchased component to perform as expected.
  3. **Business risks** → affect the organization developing or procuring the software. For example, a competitor introducing a new product is a business risk.
- For large projects, you should record the results of the risk analysis in a risk register along with a consequence analysis. This sets out the consequences of the risk for the project, product, and business.

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of company management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
Software tool underperformance	Product	Software tools that support the project do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Fig: Examples of common project, product, and business risks

- Effective risk management makes it easier to cope with problems and to ensure that these do not lead to unacceptable budget or schedule slippage.
- For small projects, formal risk recording may not be required, but the project manager should be aware of them.
- The specific risks that may affect a project depend on the project and the organizational environment in which the software is being developed.
- Software risk management is important because of the inherent uncertainties in software development.

- An outline of the process of risk management is presented in Figure. It involves several **stages:**
  1. **Risk identification** → You should identify possible project, product, and business risks.
  2. **Risk analysis** → You should assess the likelihood and consequences of these risks.
  3. **Risk planning** → You should make plans to address the risk, either by avoiding it or by minimizing its effects on the project.
  4. **Risk monitoring** → You should regularly assess the risk and your plans for risk mitigation and revise these plans when you learn more about the risk.
- The risk management process is an iterative process that continues throughout a project.

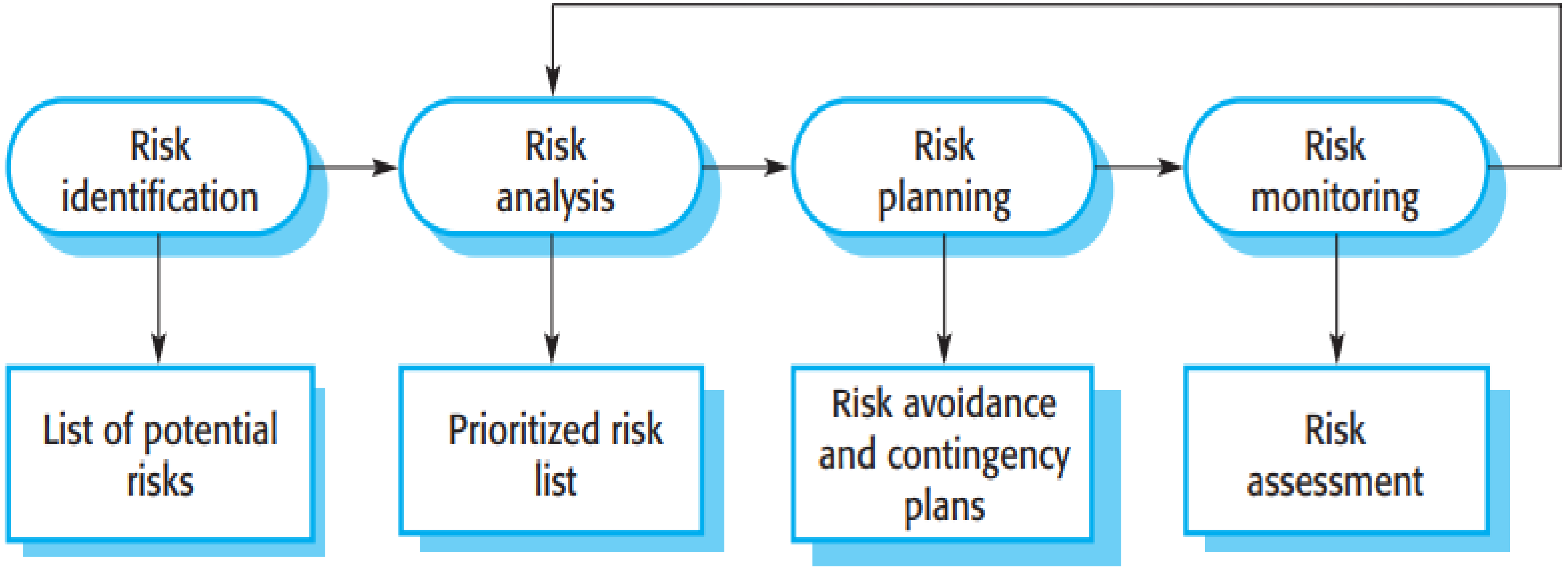
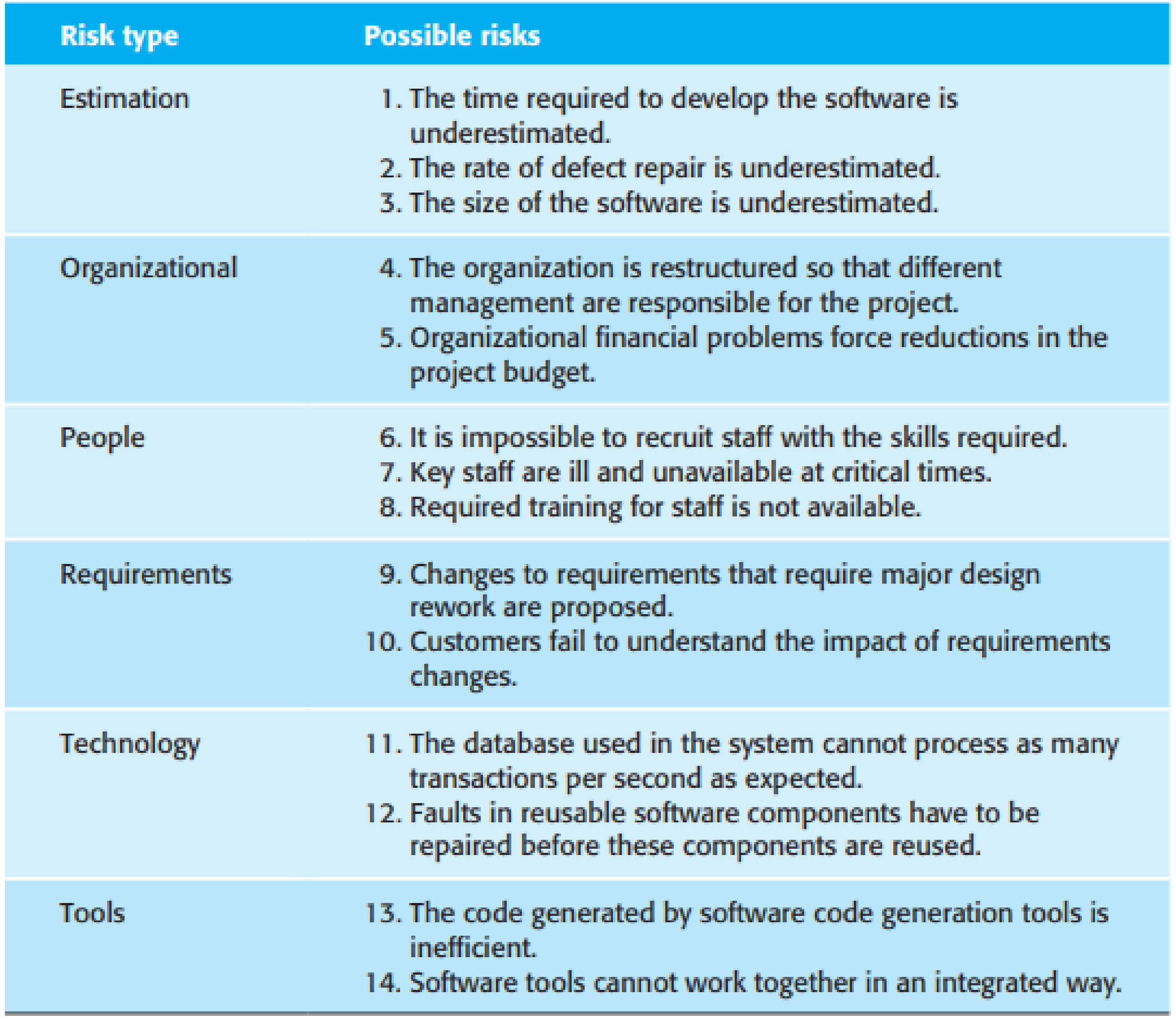


Fig: The Risk Management Process

Risk Identification:

- Risk identification is the first stage of the risk management process.
- It is concerned with identifying the risks that could pose a major threat to the software engineering process, the software being developed, or the development organization.
- Risk identification may be a team process in which a team gets together to brainstorm possible risks.
- As a starting point for risk identification, a checklist of different types of risk may be used.

- **6 types of risk** may be included in a risk checklist:
  1. **Estimation risks** → arise from the management estimates of the resources required to build the system.
  2. **Organizational risks** → arise from the organizational environment where the software is being developed.
  3. **People risks** → are associated with the people in the development team.
  4. **Requirements risks** → come from changes to the customer requirements and the process of managing the requirements change.
  5. **Technology risks** → come from the software or hardware technologies that are used to develop the system.
  6. **Tools risks** → come from the software tools and other support software used to develop the system.



Risk type	Possible risks
Estimation	1. The time required to develop the software is underestimated. 2. The rate of defect repair is underestimated. 3. The size of the software is underestimated.
Organizational	4. The organization is restructured so that different management are responsible for the project. 5. Organizational financial problems force reductions in the project budget.
People	6. It is impossible to recruit staff with the skills required. 7. Key staff are ill and unavailable at critical times. 8. Required training for staff is not available.
Requirements	9. Changes to requirements that require major design rework are proposed. 10. Customers fail to understand the impact of requirements changes.
Technology	11. The database used in the system cannot process as many transactions per second as expected. 12. Faults in reusable software components have to be repaired before these components are reused.
Tools	13. The code generated by software code generation tools is inefficient. 14. Software tools cannot work together in an integrated way.

Figure 22.3 Examples of different types of risk

Risk Analysis:

- During the risk analysis process, you have to consider each identified risk and make a judgment about the probability and seriousness of that risk.
- It is not possible to make precise, numeric assessment of the probability and seriousness of each risk.
- You should assign the risk to one of a number of bands:
    1. The probability of the risk might be assessed as insignificant, low, moderate, high, or very high.
    2. The effects of the risk might be assessed as catastrophic (threaten the survival of the project), serious (would cause major delays), tolerable (delays are within allowed contingency), or insignificant.
- You may then tabulate the results of this analysis process using a table ordered according to the seriousness of the risk.

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (5).	Low	Catastrophic
It is impossible to recruit staff with the skills required (6).	High	Catastrophic
Key staff are ill at critical times in the project (7).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused (12).	Moderate	Serious
Changes to requirements that require major design rework are proposed (9).	Moderate	Serious
The organization is restructured so that different managements are responsible for the project (4).	High	Serious
The database used in the system cannot process as many transactions per second as expected (11).	Moderate	Serious
The time required to develop the software is underestimated (1).	High	Serious
Software tools cannot be integrated (14).	High	Tolerable
Customers fail to understand the impact of requirements changes (10).	Moderate	Tolerable
Required training for staff is not available (8).	Moderate	Tolerable
The rate of defect repair is underestimated (2).	Moderate	Tolerable
The size of the software is underestimated (3).	High	Tolerable
Code generated by code generation tools is inefficient (13).	Moderate	Insignificant

Figure 22.4 Risk types and examples

- Both the probability and the assessment of the effects of a risk may change as more information about the risk becomes available and as risk management plans are implemented. You should therefore update this table during each iteration of the risk management process.
- Once the risks have been analyzed and ranked, you should assess which of these risks are most significant.
- In general, catastrophic risks should always be considered, as should all serious risks that have more than a moderate probability of occurrence.

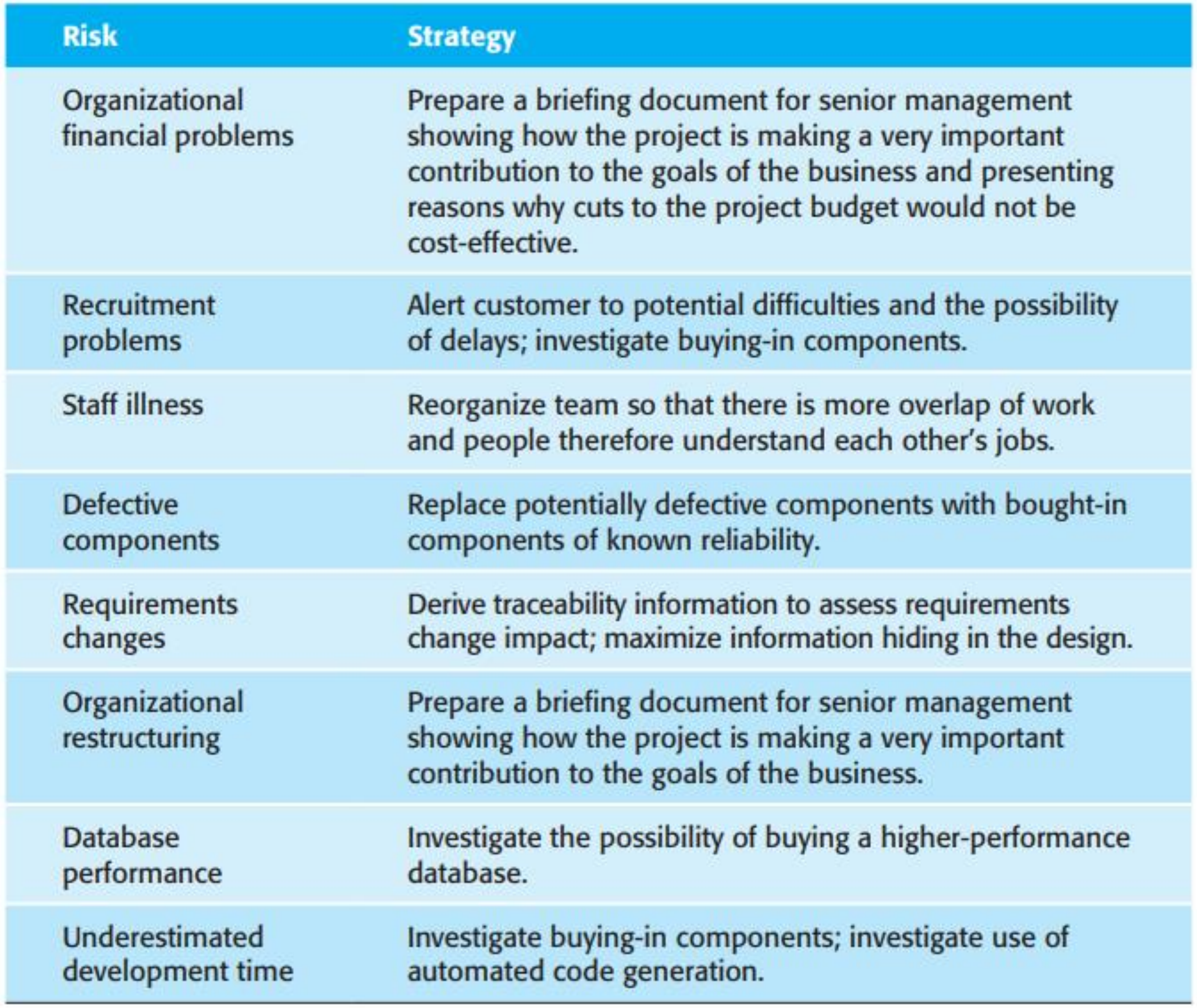
Risk Planning:

- The risk planning process develops strategies to manage the key risks that threaten the project.
- For each risk, you have to think of actions that you might take to minimize the disruption to the project if the problem identified in the risk occurs.
- You should also think about the information that you need to collect while monitoring the project so that emerging problems can be detected before they become serious.

- In risk planning, you have to ask “what-if” questions that consider both individual risks, combinations of risks, and external factors that affect these risks. For example, questions that you might ask are:
    1. What if several engineers are ill at the same time?
    2. What if an economic downturn leads to budget cuts of 20% for the project?
    3. What if the performance of open-source software is inadequate and the only expert on that open-source software leaves?
    4. What if the company that supplies and maintains software components goes out of business?
    5. What if the customer fails to deliver the revised requirements as predicted?
- Based on the answers to these “what-if” questions, you may devise strategies for managing the risks.

- The **possible risk management strategies** fall into 3 categories:
  1. **Avoidance strategies** → Following these strategies means that the probability that the risk will arise is reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components.
  2. **Minimization strategies** → Following these strategies means that the impact of the risk is reduced. An example of a risk minimization strategy is the strategy for staff illness.
  3. **Contingency plans** → Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial problems.
- The strategies used in critical systems ensure reliability, security, and safety, where you must avoid, tolerate, or recover from failures.

- It is best to use a strategy that avoids the risk.
- If this is not possible, you should use a strategy that reduces the chances that the risk will have serious effects.
- Finally, you should have strategies in place to cope with the risk if it arises. These should reduce the overall impact of a risk on the project or product.



Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of automated code generation.

Figure 22.5 Strategies to help manage risk

Risk Monitoring:

- Risk monitoring is the process of checking that your assumptions about the product, process, and business risks have not changed.
- You should regularly assess each of the identified risks to decide whether or not that risk is becoming more or less probable.
- You should also think about whether or not the effects of the risk have changed.
- To do this, you have to look at other factors, such as the number of requirements change requests, which give you clues about the risk probability and its effects. These factors are dependent on the types of risk.

Risk type	Potential indicators
Estimation	Failure to meet agreed schedule; failure to clear reported defects.
Organizational	Organizational gossip; lack of action by senior management.
People	Poor staff morale; poor relationships among team members; high staff turnover.
Requirements	Many requirements change requests; customer complaints.
Technology	Late delivery of hardware or support software; many reported technology problems.
Tools	Reluctance by team members to use tools; complaints about software tools; requests for faster computers/more memory, and so on.

Figure 22.6 Risk indicators

- You should monitor risks regularly at all stages in a project.
- At every management review, you should consider and discuss each of the key risks separately.
- You should decide if the risk is more or less likely to arise and if the seriousness and consequences of the risk have changed.

Managing People

- The people working in a software organization are its greatest assets.
- It is expensive to recruit and retain good people.
- Software managers have to ensure that the engineers working on a project are as productive as possible.
- It is important that software project managers understand the technical issues that influence the work of software development.
- Software engineers often have strong technical skills but may lack the softer skills that enable them to motivate and lead a project development team.
- As a project manager, you should be aware of the potential problems of people management and should try to develop people management skills.

- **4 critical factors that influence the relationship between a manager and the people** that he or she manages:
  1. **Consistency** → All the people in a project team should be treated in a comparable way. No one expects all rewards to be identical, but people should not feel that their contribution to the organization is undervalued.
  2. **Respect** → Different people have different skills, and managers should respect these differences.
  3. **Inclusion** → People contribute effectively when they feel that others listen to them and take account of their proposals. It is important to develop a working environment where all views, even those of the least experienced staff, are considered.
  4. **Honesty** → As a manager, you should always be honest about what is going well and what is going badly in the team. You should also be honest about your level of technical knowledge and be willing to defer to staff with more knowledge when necessary.

Motivating People:

- As a project manager, you need to motivate the people who work with you so that they will contribute to the best of their abilities.
- In practice, **motivation** means organizing work and its environment to encourage people to work as effectively as possible.
- To provide this encouragement, you should understand a little about what motivates people.
- People are motivated by satisfying their needs. These needs are arranged in a series of levels, as shown in Figure.

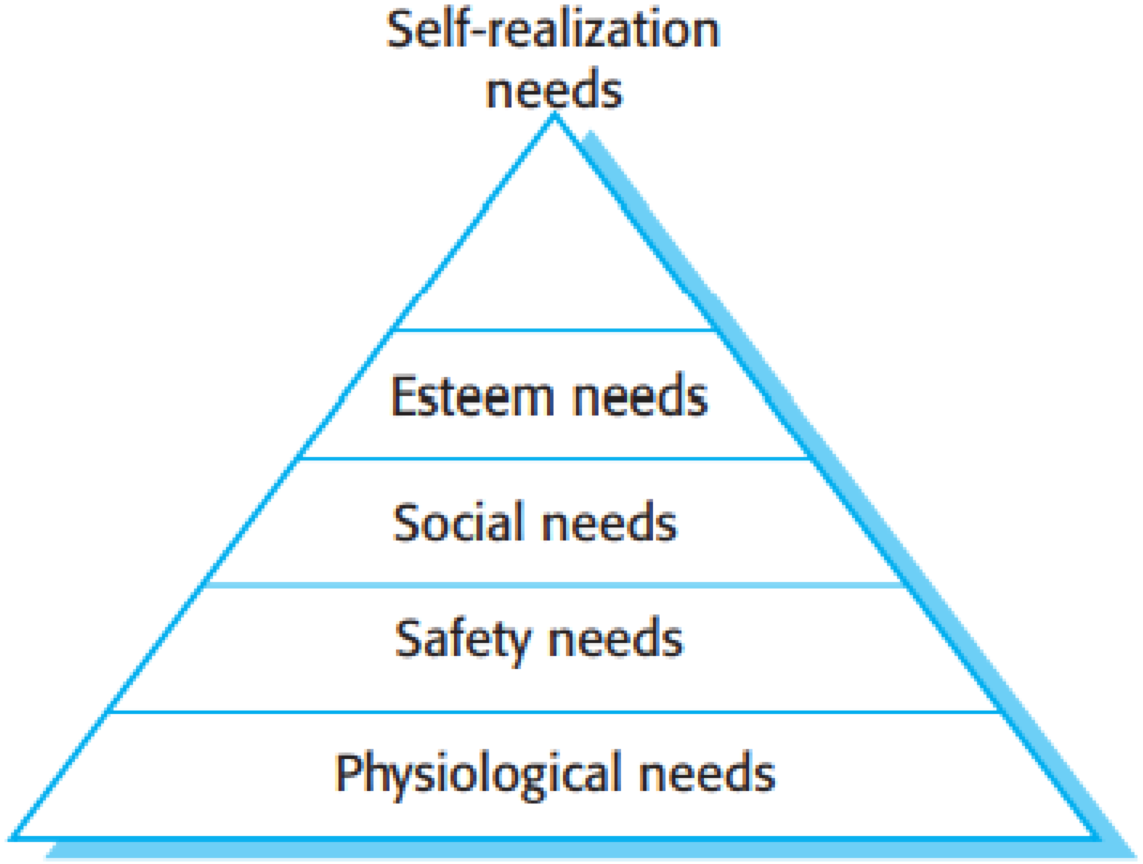
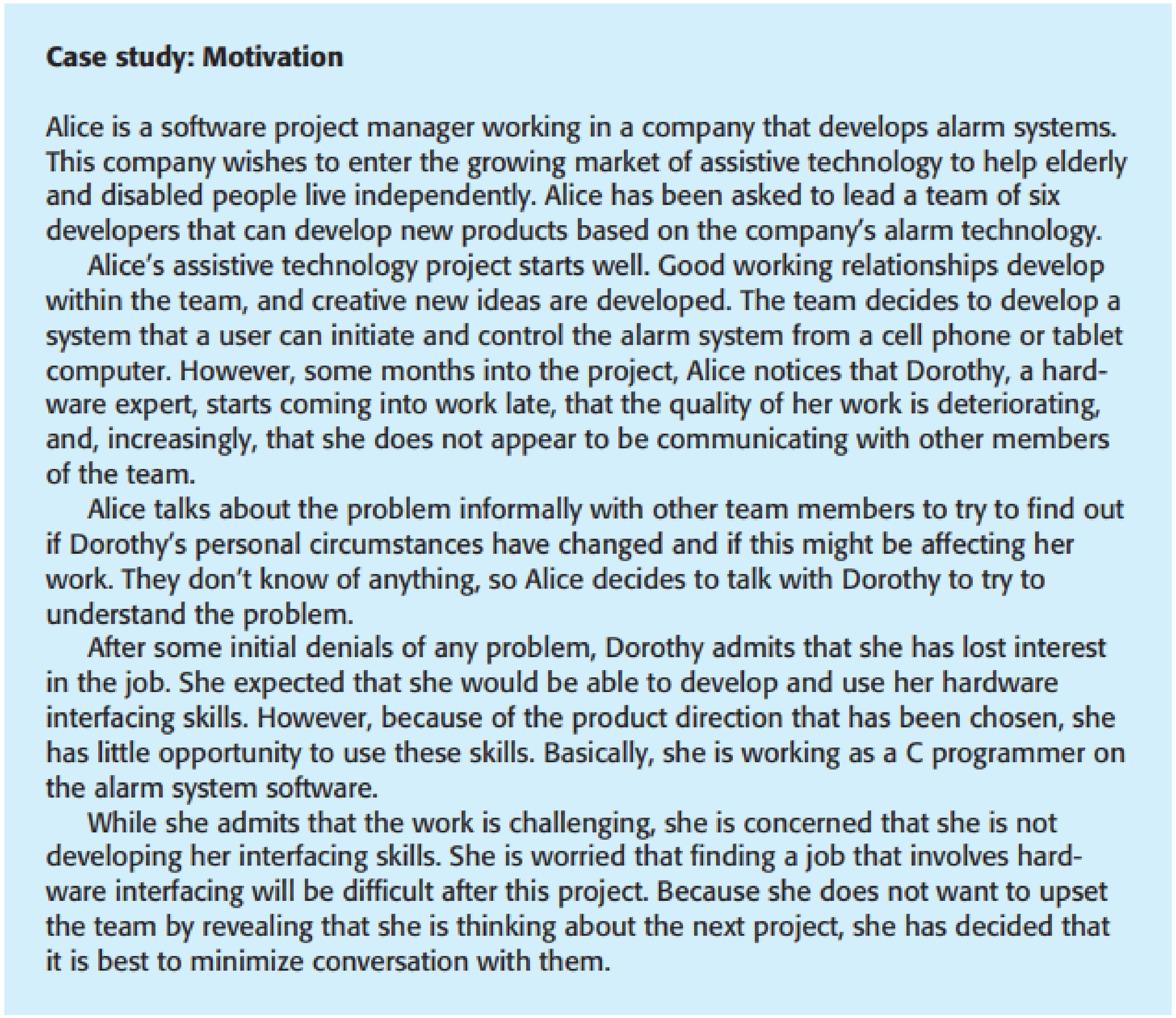


Figure 22.7 Human needs hierarchy

- The lower levels of this hierarchy represent fundamental needs for food, sleep, and so on, and the need to feel secure in an environment.
- **Social need** is concerned with the need to feel part of a social grouping.
- **Esteem need** represents the need to feel respected by others, and **self-realization need** is concerned with personal development.
- People need to satisfy lower-level needs such as hunger before the more abstract, higher-level needs.
- People working in software development organizations are not usually hungry, thirsty, or physically threatened by their environment. Therefore, making sure that peoples' social, esteem, and self-realization needs are satisfied is most important from a management point of view.

1. **To satisfy social needs**, you need to give people time to meet their co-workers and provide places for them to meet. This is relatively easy when all of the members of a development team work in the same place. Social networking systems and teleconferencing can be used for remote communications.
2. **To satisfy esteem needs**, you need to show people that they are valued by the organization. Public recognition of achievements is a simple and effective way of doing this.
3. Finally, **to satisfy self-realization needs**, you need to give people responsibility for their work, assign them demanding (but not impossible) tasks, and provide opportunities for training and development where people can enhance their skills. Training is an important motivating influence as people like to gain new knowledge and learn new skills.

- Maslow’s model of motivation takes an exclusively personal viewpoint on motivation.
- It does not take adequate account of the fact that people feel themselves to be part of an organization, a professional group, and one or more cultures.
- Being a member of a cohesive group is highly motivating for most people.
- Therefore, as a manager, you also have to think about how a group as a whole can be motivated.



Case study: Motivation

Alice is a software project manager working in a company that develops alarm systems. This company wishes to enter the growing market of assistive technology to help elderly and disabled people live independently. Alice has been asked to lead a team of six developers that can develop new products based on the company's alarm technology.

Alice's assistive technology project starts well. Good working relationships develop within the team, and creative new ideas are developed. The team decides to develop a system that a user can initiate and control the alarm system from a cell phone or tablet computer. However, some months into the project, Alice notices that Dorothy, a hardware expert, starts coming into work late, that the quality of her work is deteriorating, and, increasingly, that she does not appear to be communicating with other members of the team.

Alice talks about the problem informally with other team members to try to find out if Dorothy's personal circumstances have changed and if this might be affecting her work. They don't know of anything, so Alice decides to talk with Dorothy to try to understand the problem.

After some initial denials of any problem, Dorothy admits that she has lost interest in the job. She expected that she would be able to develop and use her hardware interfacing skills. However, because of the product direction that has been chosen, she has little opportunity to use these skills. Basically, she is working as a C programmer on the alarm system software.

While she admits that the work is challenging, she is concerned that she is not developing her interfacing skills. She is worried that finding a job that involves hardware interfacing will be difficult after this project. Because she does not want to upset the team by revealing that she is thinking about the next project, she has decided that it is best to minimize conversation with them.

Figure 22.8 Individual motivation

- Psychological personality type also influences motivation.
- Bass and Dunteman (Bass and Dunteman 1963) identified **3 classifications for professional workers:**
  1. **Task-oriented people** → who are motivated by the work they do. In software engineering, these are people who are motivated by the intellectual challenge of software development.
  2. **Self-oriented people** → who are principally motivated by personal success and recognition. They are interested in software development as a means of achieving their own goals. They often have longer-term goals and they wish to be successful in their work to help realize these goals.
  3. **Interaction-oriented people** → who are motivated by the presence and actions of co-workers.

- Research has shown that interaction-oriented personalities usually like to work as part of a group, whereas task-oriented and self-oriented people usually prefer to act as individuals.
- **People Capability Maturity Model (P-CMM) →** is a framework for assessing how well organizations manage the development of their staff. It highlights best practice in people management and provides a basis for organizations to improve their people management processes. It is best suited to large rather than small, informal companies.

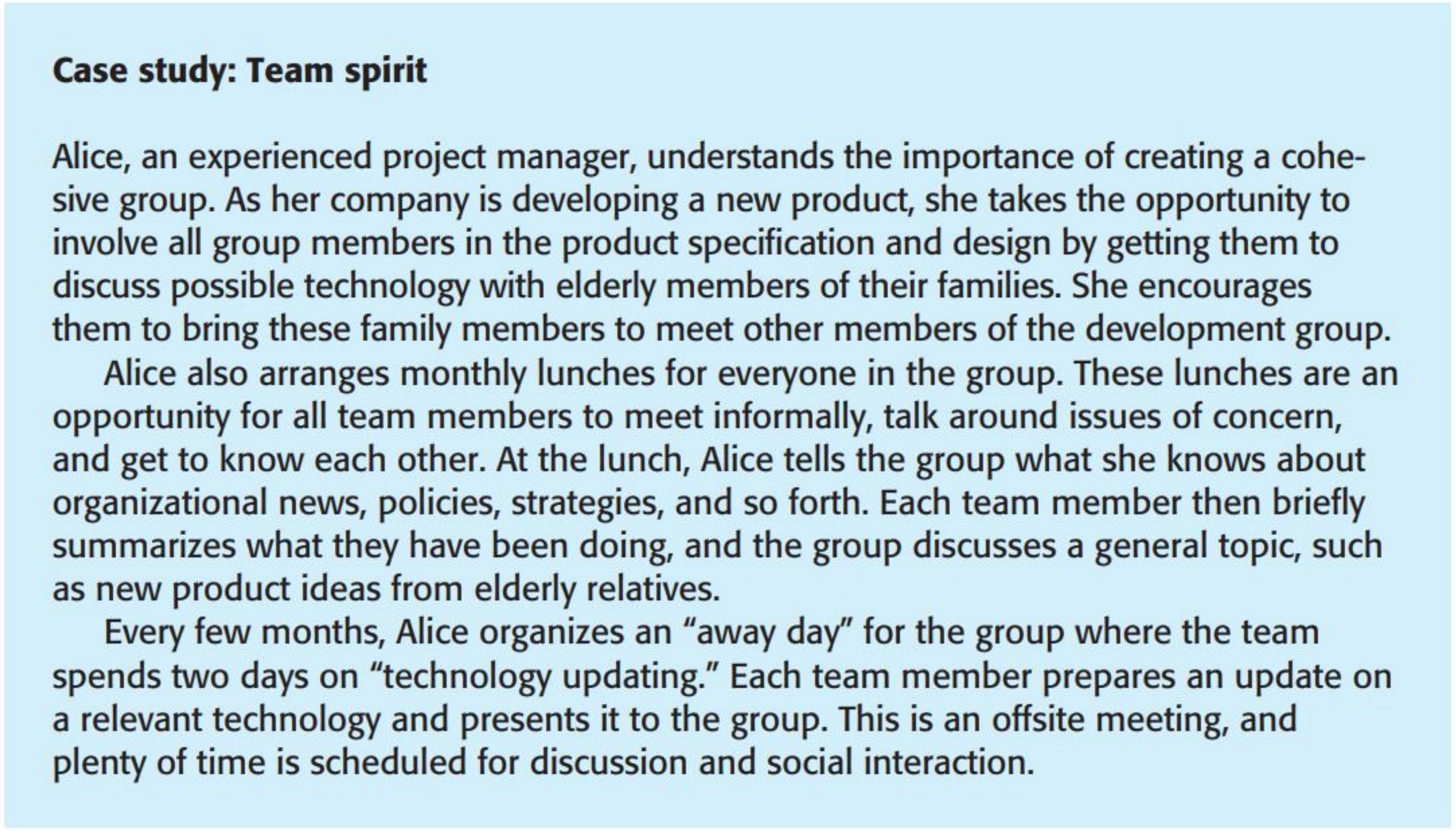
TEAMWORK

- As it is impossible for everyone in a large group to work together on a single problem, large teams are usually split into a number of smaller groups.
- Each group is responsible for developing part of the overall system.
- The best size for a software engineering group is 4 to 6 members, and they should never have more than 12 members.
- When groups are small, communication problems are reduced.

- Putting together a group that has the right balance of technical skills, experience, and personalities is a critical management task.
- A good group is cohesive and thinks of itself as a strong, single unit.
- The people involved are motivated by the success of the group as well as by their own personal goals.
- In a **cohesive group**, members think of the group as more important than the individuals who are group members.
  - ➢They are loyal to the group.
  - ➢They identify with group goals and other group members.
  - ➢They attempt to protect the group, as an entity, from outside interference. This makes the group robust and able to cope with problems and unexpected situations.

- The **benefits of creating a cohesive group** are:
    1. The group can establish its own quality standards.
    2. Individuals learn from and support each other.
    3. Knowledge is shared.
    4. Refactoring and continual improvement is encouraged.
- Good project managers should always try to encourage group cohesiveness.
- They may try to establish a sense of group identity by naming the group and establishing a group identity and territory.
- One of the most effective ways of promoting cohesion is **to be inclusive** i.e., you should treat group members as responsible and trustworthy, and make information freely available.

- An effective way of making people feel valued and part of a group is to make sure that they know what is going on.



Case study: Team spirit

Alice, an experienced project manager, understands the importance of creating a cohesive group. As her company is developing a new product, she takes the opportunity to involve all group members in the product specification and design by getting them to discuss possible technology with elderly members of their families. She encourages them to bring these family members to meet other members of the development group.

Alice also arranges monthly lunches for everyone in the group. These lunches are an opportunity for all team members to meet informally, talk around issues of concern, and get to know each other. At the lunch, Alice tells the group what she knows about organizational news, policies, strategies, and so forth. Each team member then briefly summarizes what they have been doing, and the group discusses a general topic, such as new product ideas from elderly relatives.

Every few months, Alice organizes an "away day" for the group where the team spends two days on "technology updating." Each team member prepares an update on a relevant technology and presents it to the group. This is an offsite meeting, and plenty of time is scheduled for discussion and social interaction.

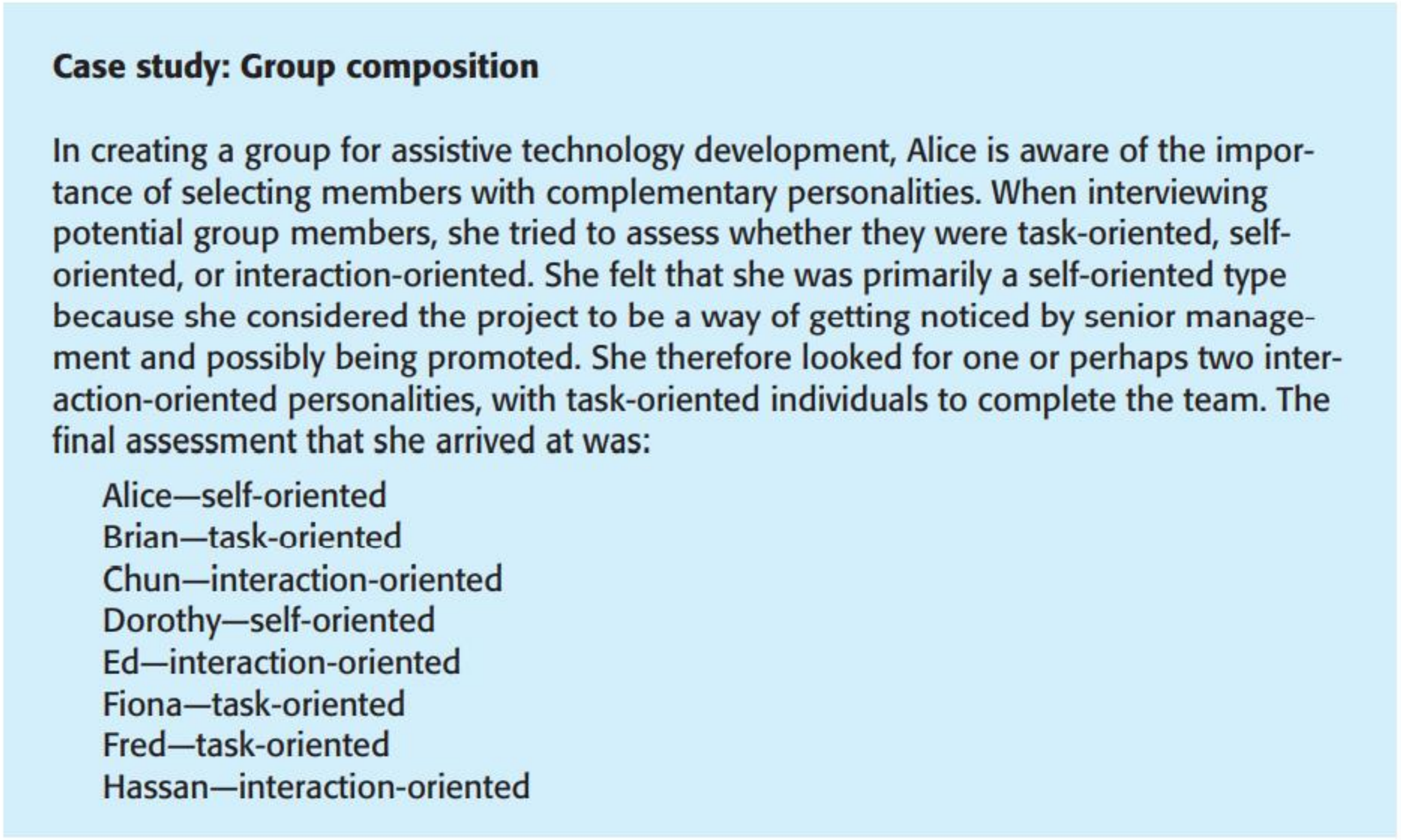
Figure 22.9 Group cohesion

- Given a stable organizational and project environment, the **3 factors that have the biggest effect on team working are:**
  1. The people in the group (**Selecting group members**)
  2. The way the group is organized (**Group organizations**)
  3. Technical and managerial communications (**Group communications**)

Selecting Group Members:

- A manager or team leader's job is to create a cohesive group and organize that group so that they work together effectively.
- This task involves selecting a group with the right balance of technical skills and personalities.
- Technical knowledge and ability should not be the only factor used to select group members.
- People who are motivated by the work are likely to be the strongest technically.
- People who are self-oriented will probably be best at pushing the work forward to finish the job.
- People who are interaction-oriented help facilitate communications within the group.

- The project manager has to control the group so that individual goals do not take precedence over organizational and group objectives.
- This control is easier to achieve if all group members participate in each stage of the project.
- Individual initiative is most likely to develop when group members are given instructions without being aware of the part that their task plays in the overall project.
- If all the members of the group are involved in the design from the start, they are more likely to understand why design decisions have been made. They may then identify with these decisions rather than oppose them.



Case study: Group composition

In creating a group for assistive technology development, Alice is aware of the importance of selecting members with complementary personalities. When interviewing potential group members, she tried to assess whether they were task-oriented, self-oriented, or interaction-oriented. She felt that she was primarily a self-oriented type because she considered the project to be a way of getting noticed by senior management and possibly being promoted. She therefore looked for one or perhaps two interaction-oriented personalities, with task-oriented individuals to complete the team. The final assessment that she arrived at was:

Alice—self-oriented
Brian—task-oriented
Chun—interaction-oriented
Dorothy—self-oriented
Ed—interaction-oriented
Fiona—task-oriented
Fred—task-oriented
Hassan—interaction-oriented

Figure 22.10 Group composition

Group Organization:

- The way a group is organized affects the group's decisions, the ways information is exchanged, and the interactions between the development group and external project stakeholders.
- Project managers are often responsible for selecting the people in the organization who will join their software engineering team.
- Getting the best possible people in this process is very important as poor selection decisions may be a serious risk to the project.
- Key factors that should influence the selection of staff are education and training, application domain and technology experience, communication ability, adaptability, and problem solving ability.

- **Important organizational questions for project managers** include the following:
  1. Should the project manager be the technical leader of the group?
  2. Who will be involved in making critical technical decisions, and how will these decisions be made? Will decisions be made by the system architect or the project manager or by reaching consensus among a wider range of team members?
  3. How will interactions with external stakeholders and senior company management be handled?
  4. How can groups integrate people who are not co-located?
  5. How can knowledge be shared across the group?

Informal Groups	Hierarchical Groups
1. Small programming groups are usually organized. 2. Group leader gets involved in the software development with the other group members. 3. The group as a whole discusses the work to be carried out, and tasks are allocated according to ability and experience. 4. More senior group members may be responsible for the architectural design. 5. Detailed design and implementation is the responsibility of the team member who is allocated to a particular task. 6. Groups are very successful, particularly when most group members are experienced and competent. Such a group makes decisions which improves cohesiveness and performance. 7. With no experienced engineers to direct the work, the result can be a lack of coordination between group members and, possibly, eventual project failure.	1. Group leader is at the top of the hierarchy. 2. Group leader has more formal authority than the group members and so can direct their work. 3. There is a clear organizational structure. 4. Decisions are made toward the top of the hierarchy and implemented by people lower down. 5. Communications are primarily instructions from senior staff; the people at lower levels of the hierarchy have relatively little communication with the managers at the upper levels. 6. These groups can work well when a well-understood problem can be easily broken down into software components that can be developed in different parts of the hierarchy. 7. This grouping allows for rapid decision making.

- In software development, effective team communications at all levels is essential:
    1. Changes to the software often require changes to several parts of the system, and this requires discussion and negotiation at all levels in the hierarchy.
    2. Software technologies change so fast that more junior staff may know more about new technologies than experienced staff. Top-down communications may mean that the project manager does not find out about the opportunities of using these new technologies. More junior staff may become frustrated because of what they see as old-fashioned technologies being used for development.
- A major challenge facing project managers is the difference in technical ability between group members.
- i.e., adopting a group model that is based on individual experts can pose significant risks.

Group Communications:

- It is absolutely essential that group members communicate effectively and efficiently with each other and with other project stakeholders.
- Good communication also helps strengthen group cohesiveness.
- Group members:
    1. Exchange information on the status of their work, the design decisions that have been made, and changes to previous design decisions.
    2. Resolve problems that arise with other stakeholders and inform these stakeholders of changes to the system, the group, and delivery plans.
    3. Come to understand the motivations, strengths, and weaknesses of other people in the group.

- The effectiveness and efficiency of communications are influenced by:
  1. **Group size** → As a group gets bigger, it gets harder for members to communicate effectively. The number of one-way communication links is $n * (n - 1)$, where n is the group size.
  2. **Group structure** → People in informally structured groups communicate more effectively than people in groups with a formal, hierarchical structure.
  3. **Group composition** → People with the same personality may clash, and, as a result, communications can be inhibited.
  4. **The physical work environment** → The organization of the workplace is a major factor in facilitating or inhibiting communications.
  5. **The available communication channels** → There are many different forms of communication—face to face, email messages, formal documents, telephone, and technologies such as social networking and wikis.

- Effective communication is achieved when communications are two-way and the people involved can discuss issues and information and establish a common understanding of proposals and problems.
- All this can be done through meetings, although these meetings are often dominated by powerful personalities.
- Informal discussions when a manager meets with the team for coffee are sometimes more effective.
- Wikis and blogs allow project members and external stakeholders to exchange information, irrespective of their location. They help manage information and keep track of discussion threads, which often become confusing when conducted by email.
- You can also use instant messaging and teleconferences, which can be easily arranged, to resolve issues that need discussion.

MODULE 4

Software Project Management - Risk management, Managing people, Teamwork. **Project Planning, Software pricing, Plan-driven development, Project scheduling, Agile planning**. **Estimation techniques, COCOMO cost modeling.** Configuration management, Version management, System building, Change management, Release management, Agile software management - SCRUM framework. Kanban methodology and lean approaches.

Project planning

Software pricing

Project scheduling

Agile planning

Estimation techniques

COCOMO cost modeling

Project planning

- Project planning is one of the most important jobs of a software project manager.
- As a manager, you have to break down the work into parts and assign them to project team members, anticipate problems that might arise, and prepare tentative solutions to those problems.
- The project plan, which is created at the start of a project and updated as the project progresses, is used to show how the work will be done and to assess progress on the project.
- Project planning takes place at three stages in a project life cycle:

1. At the proposal stage, when you are bidding for a contract to develop or provide a software system. You need a plan at this stage to help you decide if you have the resources to complete the work and to work out the price that you should quote to a customer.
2. During the project startup phase, when you have to plan who will work on the project, how the project will be broken down into increments, how resources will be allocated across your company, and so on. Here, you have more information than at the proposal stage, and you can therefore refine the initial effort estimates that you have prepared.
3. Periodically throughout the project, when you update your plan to reflect new information about the software and its development. You learn more about the system being implemented and the capabilities of your development team. As software requirements change, the work breakdown has to be altered and the schedule extended.

Project planning

- Three main parameters should be used when computing the costs of a software development project:
  - ■ effort costs (the costs of paying software engineers and managers);
  - ■ hardware and software costs, including hardware maintenance and software support; and
  - ■ travel and training costs.

❖For most projects, the biggest cost is the effort cost.

❖You have to estimate the total effort (in person-months) that is likely to be required to complete the work of a project.

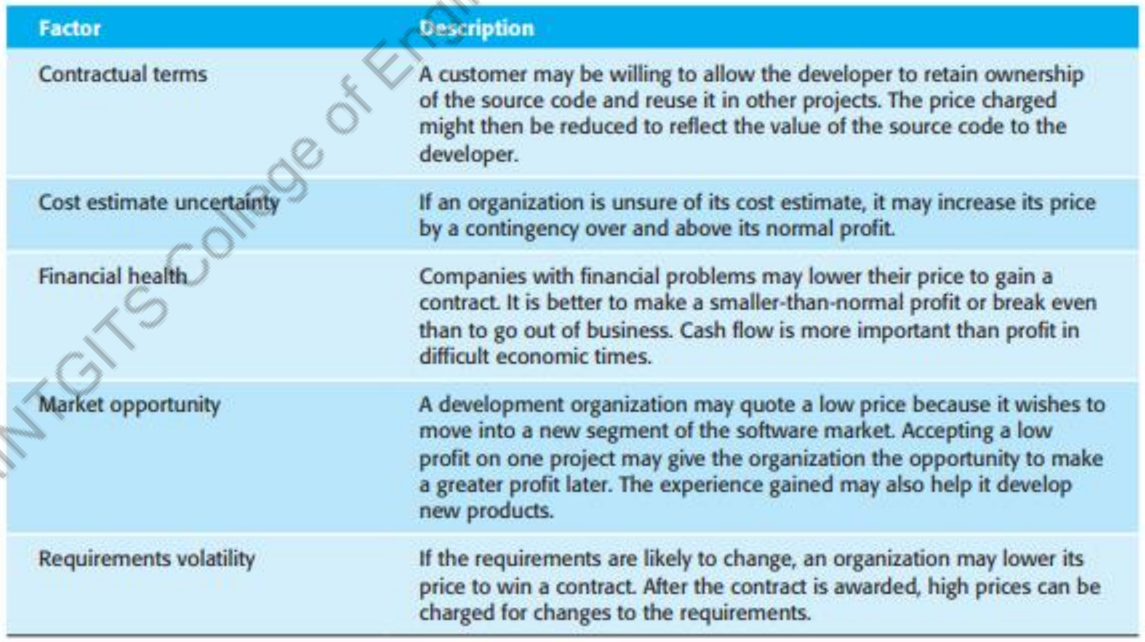
❖Obviously, you have limited information to make such an estimate. You therefore make the best possible estimate and then add contingency (extra time and effort) in case your initial estimate is optimistic

Project planning

- For commercial systems, you normally use commodity hardware, which is relatively cheap. However, software costs can be significant if you have to license middleware and platform software.
- Extensive travel may be needed when a project is developed at different sites. While travel costs themselves are usually a small fraction of the effort costs, the time spent traveling is often wasted and adds significantly to the effort costs of the project. You can use electronic meeting systems and other collaborative software to reduce travel.
- Once a contract to develop a system has been awarded, the outline project plan for the project has to be refined to create a project startup plan. At this stage, you should know more about the requirements for this system. You use this plan as a basis for allocating resources to the project from within the organization and to help decide if you need to hire new staff.
- The plan should also define project monitoring mechanisms. You must keep track of the progress of the project and compare actual and planned progress and costs.
- The project plan always evolves during the development process because of requirements changes, technology issues, and development problems.
- .If an agile method is used, there is still a need for a project startup plan because regardless of the approach used, the company still needs to plan how resources will be allocated to a project

Software pricing

- In principle, **the price of a software system developed for a customer is simply the cost of development plus profit for the developer**.
- In practice, however, the relationship between the project cost and the price quoted to the customer is not usually so simple.
- When **calculating a price, you take broader organizational, economic, political, and business considerations into account** (Figure).
- You need to think about organizational concerns, the risks associated with the project, and the type of contract that will be used. These issues may cause the price to be adjusted upward or downward



Factor	Description
Contractual terms	A customer may be willing to allow the developer to retain ownership of the source code and reuse it in other projects. The price charged might then be reduced to reflect the value of the source code to the developer.
Cost estimate uncertainty	If an organization is unsure of its cost estimate, it may increase its price by a contingency over and above its normal profit.
Financial health	Companies with financial problems may lower their price to gain a contract. It is better to make a smaller-than-normal profit or break even than to go out of business. Cash flow is more important than profit in difficult economic times.
Market opportunity	A development organization may quote a low price because it wishes to move into a new segment of the software market. Accepting a low profit on one project may give the organization the opportunity to make a greater profit later. The experience gained may also help it develop new products.
Requirements volatility	If the requirements are likely to change, an organization may lower its price to win a contract. After the contract is awarded, high prices can be charged for changes to the requirements.

Factors affecting software pricing

Software pricing

- Pricing to win means that a company has some idea of the price that the customer expects to pay and makes a bid for the contract based on the customer’s expected price. This may seem unethical and unbusinesslike, but it does have advantages for both the customer and the system provider.
- A **project cost is agreed** on the basis of an **outline proposa**l. Negotiations then take place between client and customer to establish the detailed project specification. This specification is constrained by the agreed cost. The buyer and seller must agree on what is acceptable system functionality.
- The fixed factor in many projects is not the project requirements but the cost. The requirements may be changed so that the project costs remain within budget.
- This approach has advantages for both the software developer and the customer. The requirements are negotiated to avoid requirements that are difficult to implement and potentially very expensive. Flexible requirements make it easier to reuse software

Plan-driven development

Project plans

The planning process

Plan-driven development

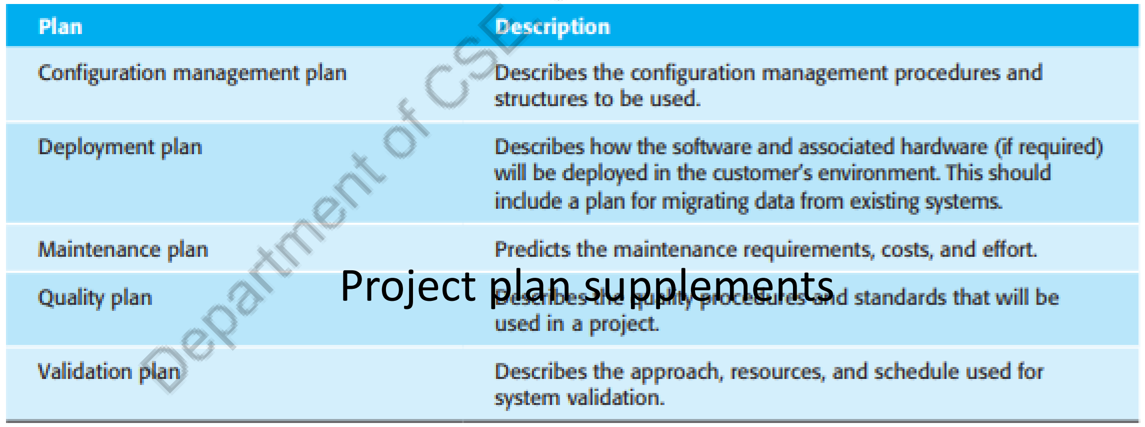
- Plan-driven or plan-based development is an approach to software engineering where the development process is planned in detail.
- A project plan is created that records the work to be done, who will do it, the development schedule, and the work products.
- Managers use the plan to support project decision making and as a way of measuring progress.
- Agile development involves a different planning process, where decisions are delayed.
- The problem with plan-driven development is that early decisions have to be revised because of changes to the environments in which the software is developed and used.
- Delaying planning decisions avoids unnecessary rework.
- However, the arguments in favor of a plan-driven approach are that early planning allows organizational issues (availability of staff, other projects, etc.) to be taken into account. Potential problems and dependencies are discovered before the project starts, rather than once the project is underway.
- The best approach to project planning involves a sensible mixture of plan-based and agile development.

Project plans

- In a plan-driven development project, a project plan sets out the resources available to the project, the work breakdown, and a schedule for carrying out the work.
- The plan should identify the approach that is taken to risk management as well as risks to the project and the software under development.
- The details of project plans vary depending on the type of project and organization but plans normally include the following sections:
    1. Introduction: Briefly describes the objectives of the project and sets out the constraints (e.g., budget, time) that affect the management of the project.
    2. Project organization :Describes the way in which the development team is organized, the people involved, and their roles in the team.
    3. Risk analysis: Describes possible project risks, the likelihood of these risks arising, and the risk reduction strategies that are proposed.
    4. Hardware and software resource requirements: Specifies the hardware and support software required to carry out the development. If hardware has to be purchased, estimates of the prices and the delivery schedule may be included.
    5. Work breakdown: Sets out the breakdown of the project into activities and identifies the inputs to and the outputs from each project activity.
    6. Project schedule: Shows the dependencies between activities, the estimated time required to reach each milestone, and the allocation of people to activities. The ways in which the schedule may be presented are discussed in the next section of the chapter.
    7. Monitoring and reporting mechanisms: Defines the management reports that should be produced, when these should be produced, and the project monitoring mechanisms to be used.

Project plans

The main project plan should always include a project risk assessment and a schedule for the project.
In addition, you may develop a number of supplementary plans for activities such as testing and configuration management.
Figure shows some supplementary plans that may be developed.
These are all usually needed in large projects developing large, complex systems



Plan	Description
Configuration management plan	Describes the configuration management procedures and structures to be used.
Deployment plan	Describes how the software and associated hardware (if required) will be deployed in the customer's environment. This should include a plan for migrating data from existing systems.
Maintenance plan	Predicts the maintenance requirements, costs, and effort.
Quality plan	Describes the quality procedures and standards that will be used in a project.
Validation plan	Describes the approach, resources, and schedule used for system validation.

Project plan supplements

The planning process

- Project planning is an iterative process that starts when you create an initial project plan during the project startup phase.
- Figure is a UML activity diagram that shows a typical workflow for a project planning process.
- Plan changes are inevitable. As more information about the system and the project team becomes available during the project, you should regularly revise the plan to reflect requirements, schedule, and risk changes. Changing business goals also leads to changes in project plans.

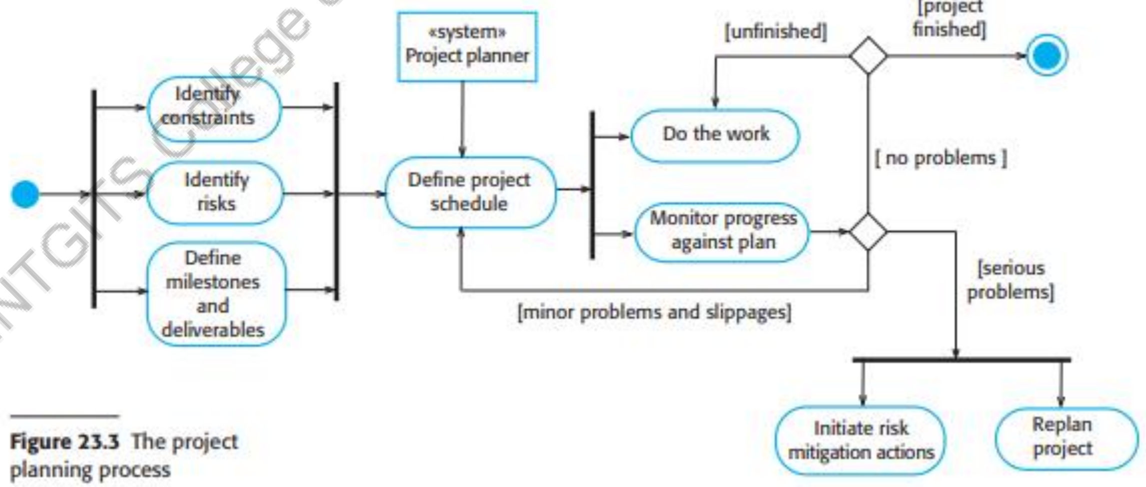


Figure 23.3 The project planning process

The planning process

- At the beginning of a planning process, you should **assess the constraints** affecting the project. These constraints are the required delivery date, staff available, overall budget, available tools, and so on.
- In conjunction with this assessment, you should also **identify the project milestones and deliverables**. Milestones are points in the schedule against which you can assess progress, for example, the handover of the system for testing. Deliverables are work products that are delivered to the customer, for example, a requirements document for the system.
- The process then **enters a loop** that terminates when the project is complete.
- You draw up an **estimated schedule for** the project, and the activities defined in the schedule are initiated or are approved to continue.
- After some time (usually about two to three weeks), you should **review progress and note discrepancies from the planned schedule**. Because initial estimates of project parameters are inevitably approximate, minor slippages are normal and you will have to make **modifications to the original plan**.
- [Problems of some description always arise during a project, and these lead to project delays. Your initial assumptions and scheduling should therefore be pessimistic and take unexpected problems into account.]
- [You should include contingency in your plan so that if things go wrong, then your delivery schedule is not seriously disrupted.]
- If there are serious problems with the development work that are likely to lead to significant delays, you need to **initiate risk mitigation actions to reduce the risks of project failure**. In conjunction with these actions, you also have **to re-plan the projec**t. This may involve renegotiating the project constraints and deliverables with the customer.
- A **new schedule** of when work should be completed also has to be established and agreed to with the customer.
- If this **renegotiation is unsuccessful or the risk mitigation actions are ineffective**, then you should arrange for a **formal project technical review**. The objectives of this review are to find an **alternative approach** that will allow the project to continue.
- The outcome of a review may be a decision to **cancel a project**.
- Management may then decide to stop software development or to make major changes to the project to reflect the changes in the organizational objectives

Project scheduling

Schedule presentation

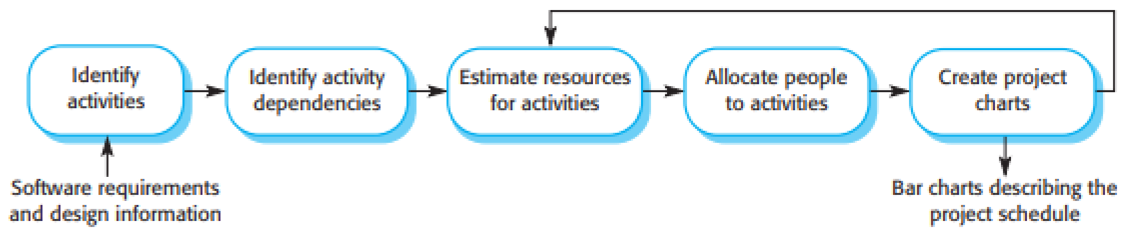
Project scheduling

- Project scheduling is the process of deciding how the work in a project will be organized as separate tasks, and when and how these tasks will be executed.
- You estimate the calendar time needed to complete each task and the effort required, and you suggest who will work on the tasks that have been identified.
- You also have to estimate the hardware and software resources that are needed to complete each task.
- An initial project schedule is usually created during the project startup phase. This schedule is then refined and modified during development planning

Project scheduling

- Both plan-based and agile processes need an initial project schedule, although less detail is included in an agile project plan.
- This initial schedule is used to plan how people will be allocated to projects and to check the progress of the project against its contractual commitments.
- In traditional development processes, the complete schedule is initially developed and then modified as the project progresses. In agile processes, there has to be an overall schedule that identifies when the major phases of the project will be completed. An iterative approach to scheduling is then used to plan each phase.

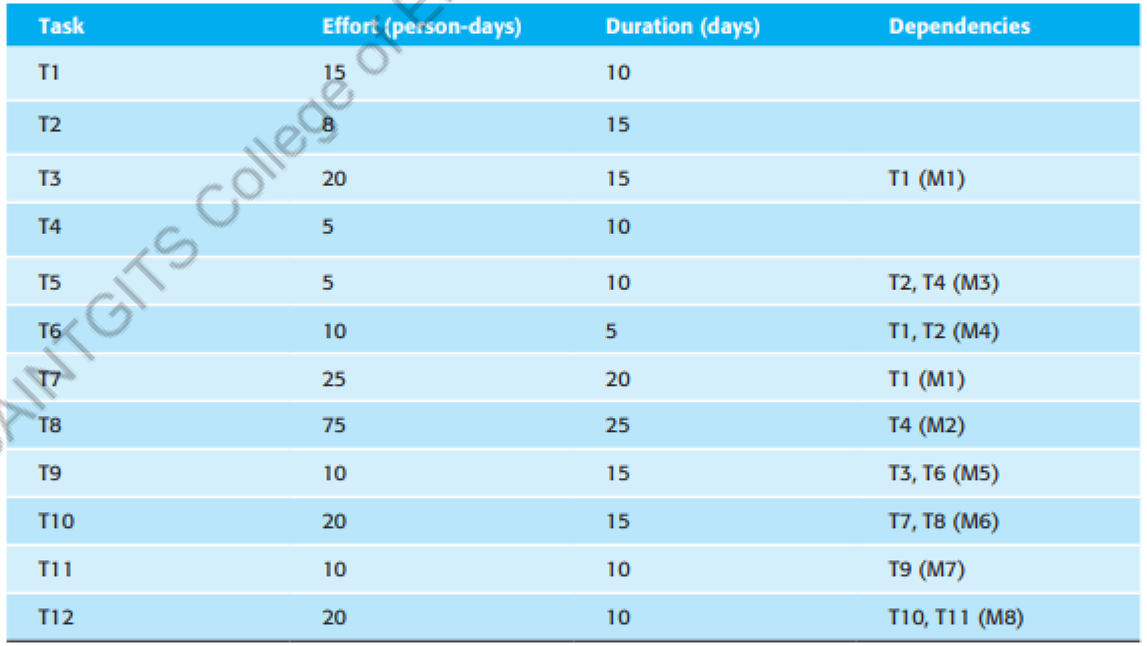
Project scheduling


Identify activities
Identify activity dependencies
Estimate resources for activities
Allocate people to activities
Create project charts
Software requirements and design information
Bar charts describing the project schedule

- Scheduling in plan-driven projects (Figure) involves breaking down the total work involved in a project into separate tasks and estimating the time required to complete each task.
- Tasks should normally last at least a week and no longer than 2 months.
- The maximum amount of time for any task should be 6 to 8 weeks. If a task will take longer than this, it should be split into subtasks for project planning and scheduling.
- Some of these tasks are carried out in parallel, with different people working on different components of the system. You have to coordinate these parallel tasks and organize the work so that the workforce is used optimally and you don’t introduce unnecessary dependencies between the tasks.
- It is important to avoid a situation where the whole project is delayed because a critical task is unfinished.
- When you are estimating schedules, you must take into account the possibility that things will go wrong. People working on a project may fall ill or leave, hardware may fail, and essential support software or hardware may be delivered late.
- If the project is new and technically advanced, parts of it may turn out to be more difficult and take longer than originally anticipated.
- A good rule of thumb is to estimate as if nothing will go wrong and then increase your estimate to cover anticipated problems.
- A further contingency factor to cover unanticipated problems may also be added to the estimate. This extra contingency factor depends on the type of project, the process parameters (deadline, standards, etc.), and the quality and experience of the software engineers working on the project.

Project scheduling

- Project schedules may simply be documented in a table or spreadsheet showing the tasks, estimated effort, duration, and task dependencies (Figure 11).
- However, this style of presentation makes it difficult to see the relationships and dependencies between the different activities.
- For this reason, alternative graphical visualizations of project schedules have been developed that are often easier to read and understand. Two types of visualization are commonly used:



Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

Figure 11: Tasks, durations, and dependencies

Project scheduling

- Two types of visualization are commonly used:

1. Calendar-based bar charts show who is responsible for each activity, the expected elapsed time, and when the activity is scheduled to begin and end. Bar charts are also called Gantt charts, after their inventor, Henry Gantt.
2. Activity networks show the dependencies between the different activities making up a project. These networks are described in an associated web section.

Project activities are the basic planning element.

Each activity has:

■ a duration in calendar days or months;

■ an effort estimate, which shows the number of person-days or person-months to complete the work;

■ a deadline by which the activity should be complete; and

■ a defined endpoint, which might be a document, the holding of a review meeting, the successful execution of all tests, or the like

Project scheduling

- When planning a project, you may decide to define project milestones.
- A **milestone** is a logical end to a stage of the project where the progress of the work can be reviewed.
- Each milestone should be documented by a brief report (often simply an email) that summarizes the work done and whether or not the work has been completed as planned.
- **Milestones** may be associated with a single task or with groups of related activities.
- Some activities create project **deliverables**—outputs that are delivered to the software customer. Usually, the deliverables that are required are specified in the project contract, and the customer's view of the project's progress depends on these deliverables.
- Milestones and deliverables are not the same thing. Milestones are short reports that are used for progress reporting, whereas deliverables are more substantial project outputs such as a requirements document or the initial implementation of a system.
- The estimated duration for some tasks is more than the effort required and vice versa. If the effort is less than the duration, the people allocated to that task are not working full time on it. If the effort exceeds the duration, this means that several team members are working on the task at the same time.

Project scheduling

- Figure 12 takes the information in Figure 11 and presents the project schedule as a bar chart showing a project calendar and the start and finish dates of tasks.
- Reading from left to right, the bar chart clearly shows when tasks start and end. The milestones (M1, M2, etc.) are also shown on the bar chart. Notice that tasks that are independent may be carried out in parallel. For example, tasks T1, T2, and T4 all start at the beginning of the project

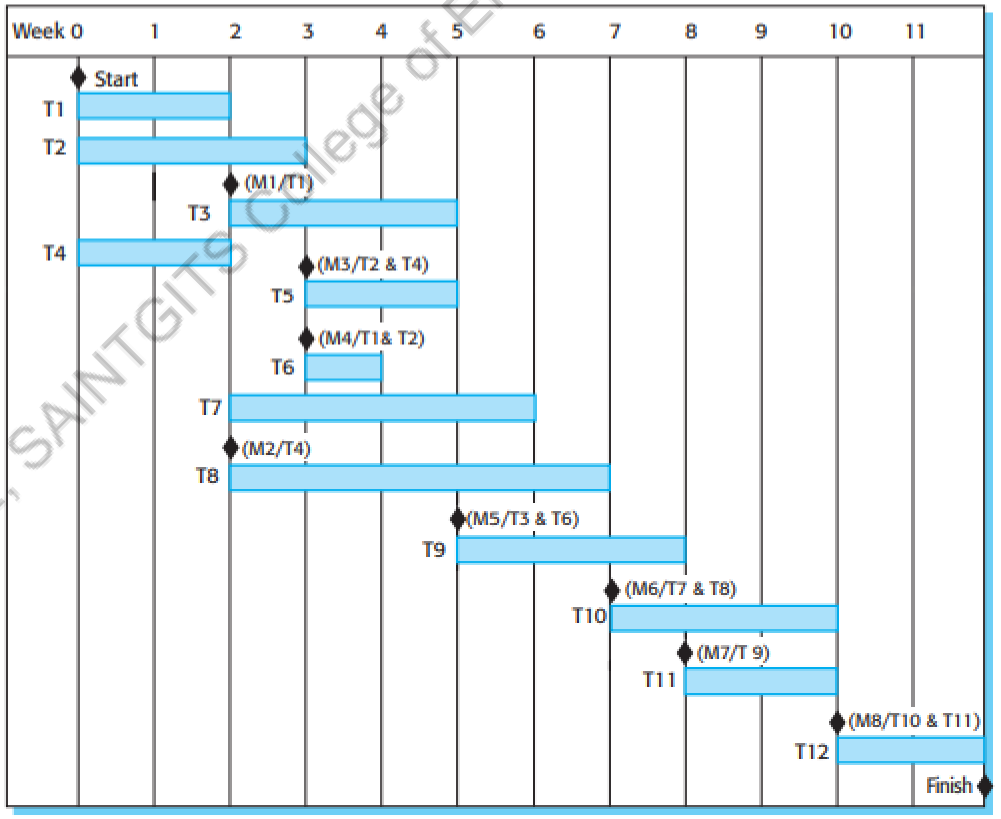


Figure 12: Activity bar chart

Project scheduling

- As well as planning the delivery schedule for the software, project managers have to allocate resources to tasks.
- The key resource is, of course, the **software engineers** who will do the work. They have to be assigned to project activities.
- The resource allocation can be analyzed by project management tools, and a bar chart can be generated showing when staff are working on the project (Figure 13).
- People may be working on more than one task at the same time, and sometimes they are not working on the project.
- They may be on holiday, working on other projects, or attending training courses. Part-time assignments are shown using a diagonal line crossing the bar

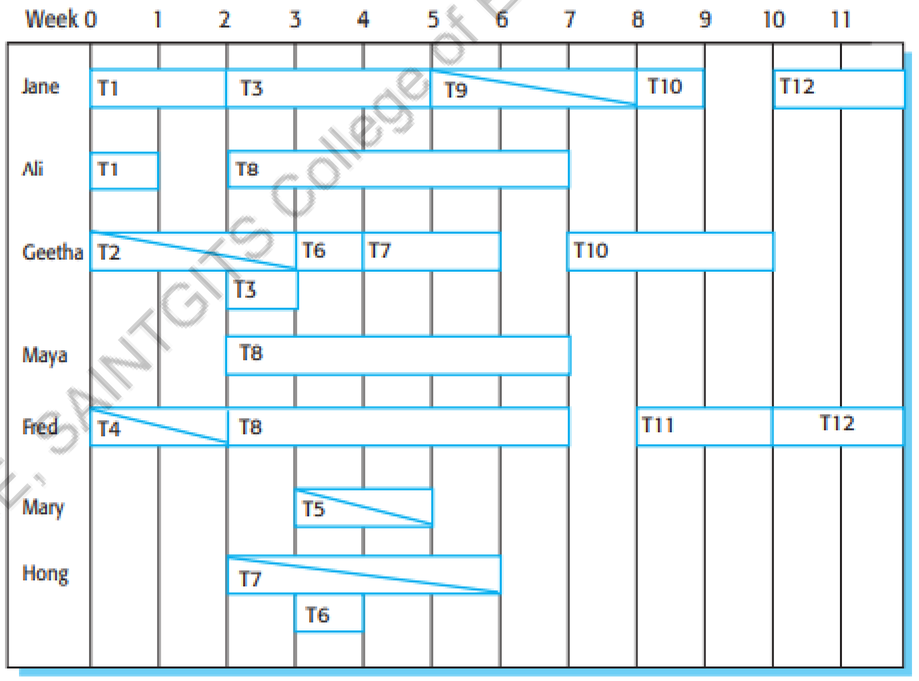


Figure 13: Staff allocation chart

Project scheduling

- Large organizations usually employ a number of specialists who work on a project when needed.
- The use of specialists is unavoidable when complex systems are being developed, but it can lead to scheduling problems.
- If one project is delayed while a specialist is working on it, this may affect other projects where the specialist is also required. These projects may be delayed because the specialist is not available.
- If a task is delayed, later tasks that are dependent on it may be affected. They cannot start until the delayed task is completed.
- Delays can cause serious problems with staff allocation, especially when people are working on several projects at the same time.
- If a task (T) is delayed, the people allocated to it may be assigned to other work (W). To complete this work may take longer than the delay, but, once assigned, they cannot simply be reassigned back to the original task. This may then lead to further delays in T as they complete W.
- Normally, you should use a project planning tool, such as the Basecamp or Microsoft project, to create, update, and analyze project schedule information.
- Project management tools usually expect you to input project information into a table, and they create a database of project information.
- Bar charts and activity charts can then be generated automatically from this database.

Agile planning

Agile planning

- Agile methods of software development are iterative approaches where the software is developed and delivered to customers in increments.
- Unlike plan-driven approaches, the functionality of these increments is not planned in advance but is decided during the development.
- The decision on what to include in an increment depends on progress and on the customer’s priorities.
- The argument for this approach is that the customer’s priorities and requirements change, so it makes sense to have a flexible plan that can accommodate these changes

Agile planning

- Agile development methods such as Scrum and Extreme Programming have a two-stage approach to planning, corresponding to the startup phase in plan-driven development and development planning:

1. Release planning, which looks ahead for several months and decides on the features that should be included in a release of a system.
2. Iteration planning, which has a shorter term outlook and focuses on planning the next increment of a system. This usually represents 2 to 4 weeks of work for the team.

Agile planning

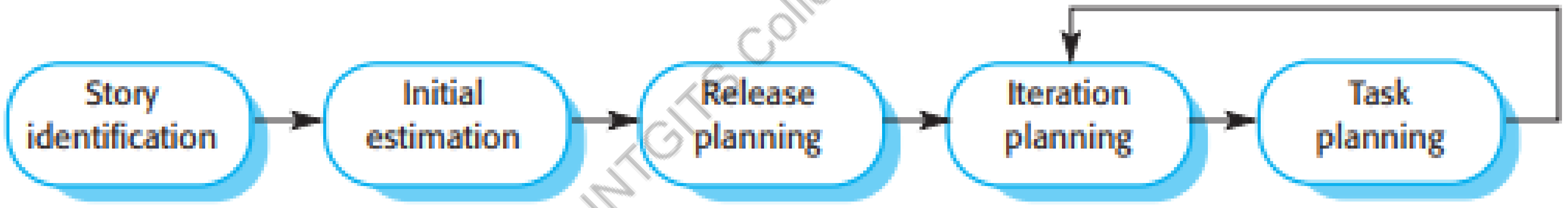


Figure 2: The planning game

Agile planning

- Scrum approach- iteration planning
- Extreme programming- User stories
- The basis of the planning game (Figure 2- previous slide) is a set of **user stories** that cover all of the functionality to be included in the final system. The development team and the software customer work together to develop these stories. The team members read and discuss the stories and rank them based on the amount of time they think it will take to implement the story.
- Some stories may be too large to implement in a single iteration, and these are broken down into smaller stories. The problem with ranking stories is that people often find it difficult to estimate how much effort or time is needed to do something. To make this easier, relative ranking may be used.
- The team compares stories in pairs and decides which will take the most time and effort, without assessing exactly how much effort will be required.
- At the end of this process, the list of stories has been ordered, with the stories at the top of the list taking the most effort to implement.
- The team then allocates notional effort points to all of the stories in the list. A complex story may have 8 points and a simple story 2 points.
- Once the stories have been estimated, the relative effort is translated into the first estimate of the total effort required by using the idea of "velocity."
- Velocity is the number of effort points implemented by the team, per day. This can be estimated either from previous experience or by developing one or two stories to see how much time is required.
- The velocity estimate is approximate but is refined during the development process.
- Once you have a velocity estimate, you can calculate the total effort in person-days to implement the system.

Agile planning

- **Release planning** involves selecting and refining the stories that will reflect the features to be implemented in a release of a system and the order in which the stories should be implemented. The customer has to be involved in this process. A release date is then chosen, and the stories are examined to see if the effort estimate is consistent with that date. If not, stories are added or removed from the list.
- **Iteration planning** is the first stage in developing a deliverable system increment. Stories to be implemented during that iteration are chosen, with the number of stories reflecting the time to deliver an workable system (usually 2 or 3 weeks) and the team’s velocity. When the delivery date is reached, the development iteration is complete, even if all of the stories have not been implemented. The team considers the stories that have been implemented and adds up their effort points. The velocity can then be recalculated, and this measure is used in planning the next version of the system.

Agile planning

- At the start of each development iteration, there is a task planning stage where the developers break down stories into development tasks. A development task should take 4–16 hours. All of the tasks that must be completed to implement all of the stories in that iteration are listed. The individual developers then sign up for the specific planning tasks that they will implement.
- Each developer knows their individual velocity and so should not sign up for more tasks than they can implement in the time allotted
- This approach to task allocation has two important benefits:

1. The whole team gets an overview of the tasks to be completed in an iteration. They therefore have an understanding of what other team members are doing and who to talk to if task dependencies are identified.
2. Individual developers choose the tasks to implement; they are not simply allocated tasks by a project manager. They therefore have a sense of ownership in these tasks, and this is likely to motivate them to complete the task.

Halfway through an iteration, progress is reviewed. At this stage, half of the story effort points should have been completed.

Agile planning

Advantages:

- This approach to planning has the advantage that a software increment is always delivered at the end of each project iteration.
- If the features to be included in the increment cannot be completed in the time allowed, the scope of the work is reduced.
- The delivery schedule is never extended..

Disadvantages

- A major difficulty in agile planning is that it relies on customer involvement and availability.
- This involvement can be difficult to arrange, as customer representatives sometimes have to prioritize other work and are not available for the planning game.
- Furthermore, some customers may be more familiar with traditional project plans and may find it difficult to engage in an agile planning process.

❑ Agile planning works well with small, stable development teams that can get together and discuss the stories to be implemented.

❑ However, where teams are large and/or geographically distributed, or when team membership changes frequently, it is practically impossible for everyone to be involved in the collaborative planning that is essential for agile project management.

❑ Consequently, large projects are usually planned using traditional approaches to project management

Estimation techniques

Experience-based techniques

Algorithmic cost modeling

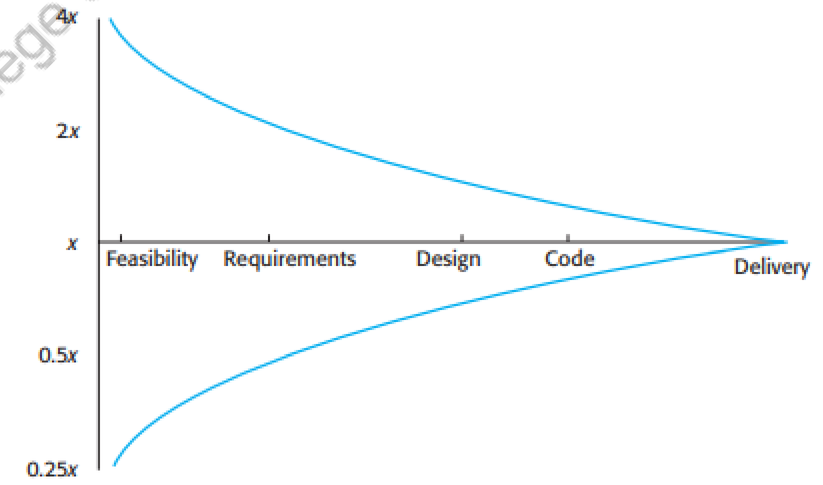
Estimation techniques

- Estimating project schedules is difficult. You have to make initial estimates on the basis of an incomplete user requirements definition.
- There are so many uncertainties that it is impossible to estimate system development costs accurately during the early stages of a project. Nevertheless, organizations need to make software effort and cost estimates.
- Two types of techniques can be used for making estimates:

1. **Experience-based techniques**: The estimate of future effort requirements is based on the manager's experience of past projects and the application domain. Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.
2. **Algorithmic cost modeling:** In this approach, a formulaic approach is used to compute the project effort based on estimates of product attributes, such as size, process characteristics, and experience of staff involvedIn

❑ In both cases, you need to use your judgment to estimate either the effort directly or the project and product characteristics. In the startup phase of a project, these estimates have a wide margin of error.

❑ During development planning, estimates become more and more accurate as the project progresses



Estimate uncertainty

Estimation techniques- **Experience-based techniques**

- **Experience-based techniques** rely on the manager's experience of past projects and the actual effort expended in these projects on activities that are related to software development.
- Typically, you identify the deliverables to be produced in a project and the different software components or systems that are to be developed.
- You document these in a spreadsheet, estimate them individually, and compute the total effort required.
- It usually helps to get a group of people involved in the effort estimation and to ask each member of the group to explain their estimate.
- This often reveals factors that others have not considered, and you then iterate toward an agreed group estimate.

Estimation techniques

- The **difficulty with experience-based techniques** is that a new software project may not have much in common with previous projects.
- Software development changes very quickly, and a project will often use unfamiliar techniques such as web services, application system configuration, or HTML5.
- If you have not worked with these techniques, your previous experience may not help you to estimate the effort required, making it more difficult to produce accurate costs and schedule estimates.
- It is impossible to say whether experience-based or algorithmic approaches are more accurate.
- Project estimates are often self-fulfilling. The estimate is used to define the project budget, and the product is adjusted so that the budget figure is realized.

Algorithmic cost modeling

- Algorithmic cost modeling uses a mathematical formula to predict project costs based on estimates of the project size, the type of software being developed, and other team, process, and product factors.
- Algorithmic cost models are developed by analyzing the costs and attributes of completed projects, then finding the closest-fit formula to the actual costs incurred. Algorithmic cost models are primarily used to make estimates of software development costs.
- Most algorithmic models for estimating effort in a software project are based on a simple formula:

$$\text{Effort} = A \times \text{Size}^B \times M$$

- A: a constant factor, which depends on local organizational practices and the type of software that is developed.
- Size: an assessment of the code size of the software or a functionality estimate expressed in function or application points.
- B: represents the complexity of the software and usually lies between 1 and 1.5.
- M: is a factor that takes into account process, product and development attributes, such as the dependability requirements for the software and the experience of the development team.

These attributes may increase or decrease the overall difficulty of developing the system.

Algorithmic cost modeling

- The number of lines of source code (SLOC) in the delivered system is the fundamental size metric that is used in many algorithmic cost models.
- To estimate the number of lines of code in a system, you may use a combination of approaches:

1. Compare the system to be developed with similar systems and use their code size as the basis for your estimate.
2. Estimate the number of function or application points in the system and formulaically convert these to lines of code in the programming language used.
3. Rank the system components using judgment of their relative sizes and use a known reference component to translate this ranking to code sizes.

Algorithmic cost modeling

- Most algorithmic estimation models have an exponential component (B in the above equation) that increases with the size and complexity of the system.
- This reflects the fact that costs do not usually increase linearly with project size.
- As the size and complexity of the software increase, extra costs are incurred because of the communication overhead of larger teams, more complex configuration management, more difficult system integration, and so on.
- The more complex the system, the more these factors affect the cost.

Algorithmic cost modeling

- The idea of using a scientific and objective approach to cost estimation is an attractive one, but all algorithmic cost models suffer from two key problems:

1. It is practically impossible to estimate Size accurately at an early stage in a project, when only the specification is available. Function-point and application point estimates are easier to produce than estimates of code size but are also usually inaccurate.
2. The estimates of the complexity and process factors contributing to B and M are subjective. Estimates vary from one person to another, depending on their background and experience of the type of system that is being developed.

Accurate code size estimation is difficult at an early stage in a project because the size of the final program depends on design decisions that may not have been made when the estimate is required.

The programming language used for system development also affects the number of lines of code to be developed. A language like Java might mean that more lines of code are necessary than if C (say) was used. However, this extra code allows more compile-time checking, so validation costs are likely to be reduced. It is not clear how this should be taken into account in the estimation process.

Algorithmic cost models are a systematic way to estimate the effort required to develop a system. However, these models are complex and difficult to use.

Another barrier that discourages the use of algorithmic models is the need for calibration. Model users should calibrate their model and the attribute values using their own historical project data, as this reflects local practice and experience.

If you use an algorithmic cost estimation model, you should develop a range of estimates (worst, expected, and best) rather than a single estimate and apply the costing formula to all of them.

COCOMO cost modeling

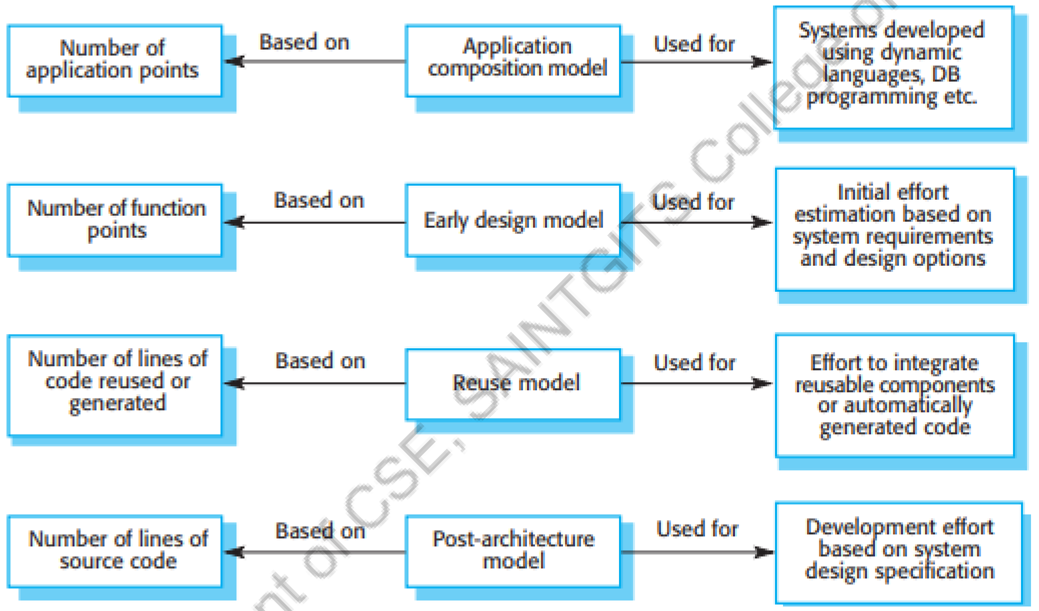
The application composition model
The early design model
The reuse model
The post-architecture level

COCOMO cost modeling

- The best known algorithmic cost modeling technique and tool is the COCOMO II model.
- This empirical model was derived by collecting data from a large number of software projects of different sizes.
- These data were analyzed to discover the formulas that were the best fit to the observations.
- These formulas linked the size of the system and product, project, and team factors to the effort to develop the system.
- COCOMO II is a freely available model that is supported with open-source tools.
- COCOMO II was developed from earlier COCOMO (Constructive Cost Modeling) cost estimation models, which were largely based on original code development (B. W. Boehm 1981; B. Boehm and Royce 1989).
- The COCOMO II model takes into account modern approaches to software development, such as rapid development using dynamic languages, development with reuse, and database programming.
- COCOMO II embeds several submodels based on these techniques, which produce increasingly detailed estimates.

The **submodels that are part of the COCOMO II** model are: (figure in the next slide)

1. An application composition model: This models the effort required to develop systems that are created from reusable components, scripting, or database programming. Software size estimates are based on application points, and a simple size/productivity formula is used to estimate the effort required.
2. An early design model: This model is used during early stages of the system design after the requirements have been established. The estimate is based on the standard estimation formula, with a simplified set of seven multipliers. Estimates are based on function points, which are then converted to number of lines of source code.
3. A reuse model: This model is used to compute the effort required to integrate reusable components and/or automatically generated program code. It is normally used in conjunction with the post-architecture model.
4. A post-architecture model: Once the system architecture has been designed, a more accurate estimate of the software size can be made. Again, this model uses the standard formula for cost estimation discussed above. However, it includes a more extensive set of 17 multipliers reflecting personnel capability, product, and project characteristics


Number of application points
Based on
Application composition model
Used for
Systems developed using dynamic languages, DB programming etc.
Number of function points
Based on
Early design model
Used for
Initial effort estimation based on system requirements and design options
Number of lines of code reused or generated
Based on
Reuse model
Used for
Effort to integrate reusable components or automatically generated code
Number of lines of source code
Based on
Post-architecture model
Used for
Development effort based on system design specification

COCOMO Estimation Models

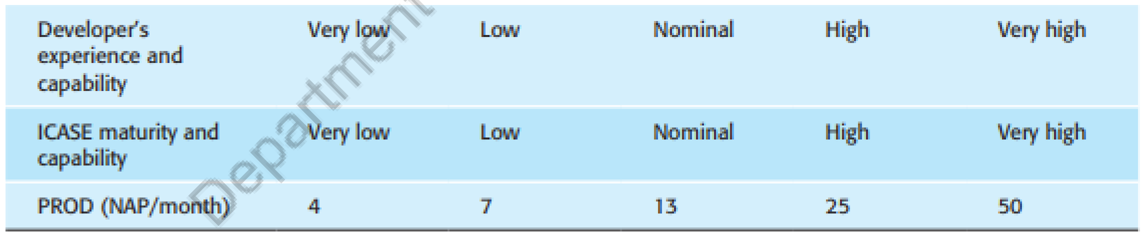
- Of course, in large systems, different parts of the system may be developed using different technologies, and you may not have to estimate all parts of the system to the same level of accuracy.
- In such cases, you can use the appropriate submodel for each part of the system and combine the results to create a composite estimate.

The application composition model

- The application composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing existing components.
- It is based on an estimate of weighted application points (sometimes called object points), divided by a standard estimate of application point productivity.
- The number of application points in a program is derived from four simpler estimates:
  - ■ the number of separate screens or web pages that are displayed;
  - ■ the number of reports that are produced;
  - ■ the number of modules in imperative programming languages (such as Java); and
  - ■ the number of lines of scripting language or database programming code.

  This estimate is then adjusted according to the difficulty of developing each application point.

Productivity depends on the developer's experience and capability as well as the capabilities of the software tools (ICASE) used to support development. Figure (next slide) shows the levels of application-point productivity suggested by the COCOMO model developers.



Developer's experience and capability	Very low	Low	Nominal	High	Very high
ICASE maturity and capability	Very low	Low	Nominal	High	Very high
PROD (NAP/month)	4	7	13	25	50

Application point productivity

The application composition model

- Application composition usually relies on reusing existing software and configuring application systems.
- Some of the application points in the system will therefore be implemented using reusable components. Consequently, you have to adjust the estimate to take into account the percentage of reuse expected.

Therefore, the final formula for effort computation for system prototypes is:

$$PM = (NAP \times (1 - \%reuse/100)) \, / \, PROD$$

PM: the effort estimate in person-months.

NAP: the total number of application points in the delivered system.

%reuse: an estimate of the amount of reused code in the development.

PROD: the application-point productivity as shown in Figure (previous slide)

The early design model

- This model may be used during the early stages of a project, before a detailed architectural design for the system is available.
- The early design model assumes that user requirements have been agreed and initial stages of the system design process are underway.
- Your goal at this stage should be to make a quick and approximate cost estimate.
- Therefore, you have to make simplifying assumptions, such as the assumpion that there is no effort involved in integrating reusable code.
- Early design estimates are most useful for option exploration where you need to compare different ways of implementing the user requirements.
- The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

$$\text{Effort} = A \times \text{Size}^{B} \times M$$

- The co-efficient A should be 2.94.
- The size of the system is expressed in KSLOC, which is the number of thousands of lines of source code. KSLOC is calculated by estimating the number of function points in the software.
- You then use standard tables, which relate software size to function points for different programming languages (QSM 2014) to compute an initial estimate of the system size in KSLOC.
- The exponent B reflects the increased effort required as the size of the project increases. This can vary from 1.1 to 1.24 depending on the novelty of the project, the development flexibility, the risk resolution processes used, the cohesion of the development team, and the process maturity level of the organization.

The early design model

- This results in an effort computation as follows:

$$PM = 2.94 \times Size^{(1.1 \text{ to } 1.24)} \times M$$

$$M = PERS \times PREX \times RCPX \times RUSE \times PDIF \times SCED \times FSIL$$

- PERS: personnel capability
- PREX: personnel experience
- RCPX: product reliability and complexity
- RUSE: reuse required
- PDIF: platform difficulty
- SCED: schedule
- FSIL: support facilities
- The multiplier M is based on seven project and process attributes that increase or decrease the estimate. You estimate values for these attributes using a six-point scale, where 1 corresponds to "very low" and 6 corresponds to "very high"; for example, PERS = 6 means that expert staff are available to work on the project

The reuse model

- The COCOMO reuse model is used to estimate the effort required to integrate reusable or generated code.
- Most large systems include a significant amount of code that has been reused from previous development projects.
- COCOMO II considers two types of reused code.

Black-box code

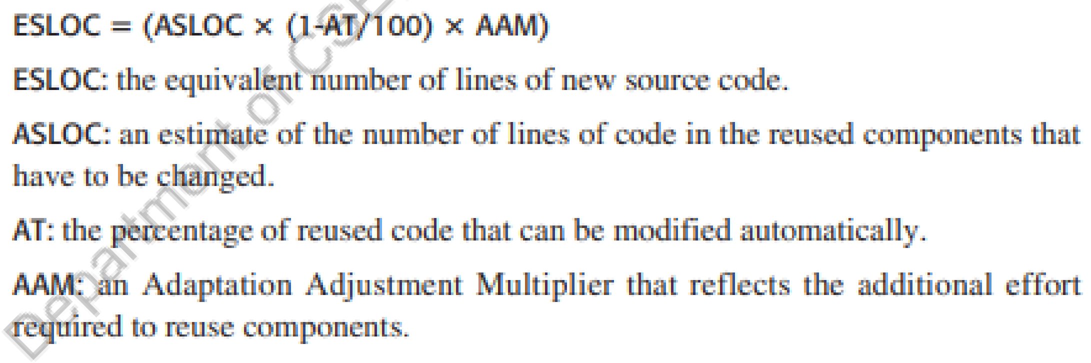
- Black-box code is code that can be reused without understanding the code or making changes to it.
- Examples of black-box code are components that are automatically generated from UML models or application libraries such as graphics libraries.
- It is assumed that the development effort for blackbox code is zero.
- Its size is not taken into account in the overall effort computation.

White-box code

- White-box code is reusable code that has to be adapted to integrate it with new code or other reused components.
- Development effort is required for reuse because the code has to be understood and modified before it can work correctly in the system.
- White-box code could be automatically generated code that needs manual changes or additions. Alternatively, it can be reused components from other systems that have to be modified in the system that is being developed.
- Three factors contribute to the effort involved in reusing white-box code components:
    1. The effort involved in assessing whether or not a component could be reused in a system that is being developed.
    2. The effort required to understand the code that is being reused.
    3. The effort required to modify the reused code to adapt it and integrate it with the system being developed.

The reuse model

- The development effort in the reuse model is calculated using the COCOMO early design model and is based on the total number of lines of code in the system.
- The code size includes new code developed for components that are not reused plus an additional factor that allows for the effort involved in reusing and integrating existing code.
- This additional factor is called ESLOC, the equivalent number of lines of new source code.
- That is, you express the reuse effort as the effort that would be involved in developing some additional source code.
- The formula used to calculate the source code equivalence is:



$$ESLOC = (ASLOC \times (1 - AT/100) \times AAM)$$

ESLOC: the equivalent number of lines of new source code.

ASLOC: an estimate of the number of lines of code in the reused components that have to be changed.

AT: the percentage of reused code that can be modified automatically.

AAM: an Adaptation Adjustment Multiplier that reflects the additional effort required to reuse components.

The reuse model

- In some cases, the adjustments required to reuse code are syntactic and can be implemented by an automated tool.
- These do not involve significant effort, so you should estimate what fraction of the changes made to reused code can be automated (AT).
- This reduces the total number of lines of code that have to be adapted.
- The Adaptation Adjustment Multiplier (AAM) adjusts the estimate to reflect the additional effort required to reuse code. The COCOMO model documentation discusses in detail how AAM should be calculated.
- Simplistically, AAM is the sum of three components:

1. An assessment factor (referred to as AA) that represents the effort involved in deciding whether or not to reuse components. AA varies from 0 to 8 depending on the amount of time you need to spend looking for and assessing potential candidates for reuse.

2. An understanding component (referred to as SU) that represents the costs of understanding the code to be reused and the familiarity of the engineer with the code that is being reused. SU ranges from 50 for complex, unstructured code to 10 for well-written, object-oriented code.

3. An adaptation component (referred to as AAF) that represents the costs of making changes to the reused code. These include design, code, and integration changes.

Once you have calculated a value for ESLOC, you apply the standard estimation formula to calculate the total effort required, where the Size parameter = ESLOC. Therefore, the formula to estimate the reuse effort is:

$$\text{Effort} = A \times \text{ESLOC}^{B} \times M$$

where A, B, and M have the same values as used in the early design model.

The post-architecture level

- The post-architecture model is the most detailed of the COCOMO II models.
- It is used when you have an initial architectural design for the system.
- The starting point for estimates produced at the post-architecture level is the same basic formula used in the early design estimates:

$$PM = A \times Size^{B} \times M$$

- By this stage in the process, you should be able to make a more accurate estimate of the project size, as you know how the system will be decomposed into subsystems and components.
- You make this estimate of the overall code size by adding three code size estimates:

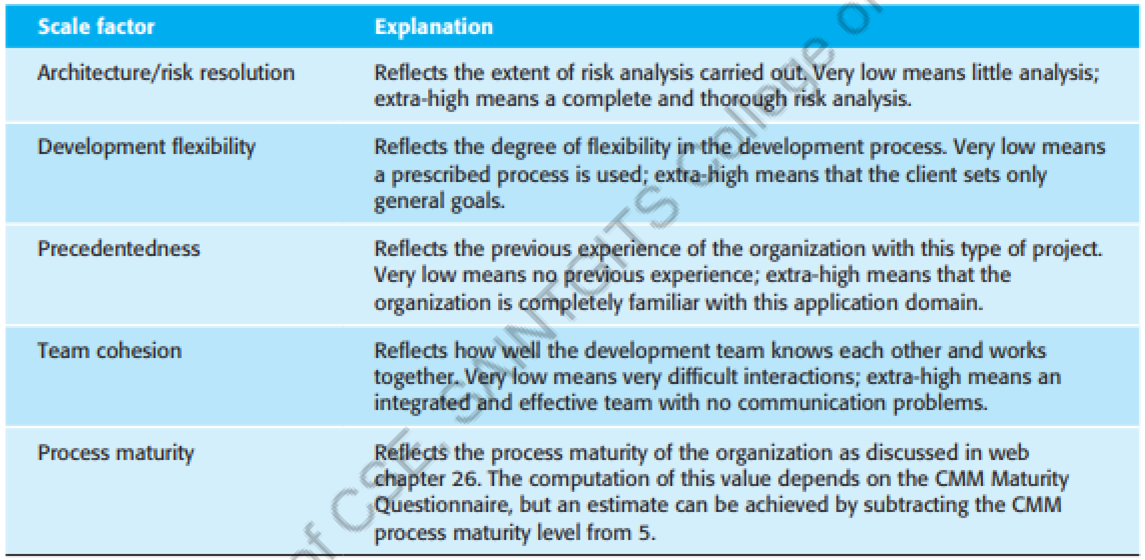
1. An estimate of the total number of lines of new code to be developed (SLOC).
2. An estimate of the reuse costs based on an equivalent number of source lines of code (ESLOC), calculated using the reuse model.
3. An estimate of the number of lines of code that may be changed because of changes to the system requirements.

❖ The final component in the estimate—the number of lines of modified code— reflects the fact that software requirements always change. This leads to rework and development of extra code, which you have to take into account.

❖ Of course there will often be even more uncertainty in this figure than in the estimates of new code to be developed. The exponent term (B) in the effort computation formula is related to the levels of project complexity. As projects become more complex, the effects of increasing system size become more significant. The value of the exponent B is based on five factors, as shown in Figure 40(next slide).

❖ These factors are rated on a sixpoint scale from 0 to 5, where 0 means “extra high” and 5 means “very low.” To calculate B, you add the ratings, divide them by 100, and add the result to 1.01 to get the exponent that should be used

The post-architecture level



Scale factor	Explanation
Architecture/risk resolution	Reflects the extent of risk analysis carried out. Very low means little analysis; extra-high means a complete and thorough risk analysis.
Development flexibility	Reflects the degree of flexibility in the development process. Very low means a prescribed process is used; extra-high means that the client sets only general goals.
Precedentedness	Reflects the previous experience of the organization with this type of project. Very low means no previous experience; extra-high means that the organization is completely familiar with this application domain.
Team cohesion	Reflects how well the development team knows each other and works together. Very low means very difficult interactions; extra-high means an integrated and effective team with no communication problems.
Process maturity	Reflects the process maturity of the organization as discussed in web chapter 26. The computation of this value depends on the CMM Maturity Questionnaire, but an estimate can be achieved by subtracting the CMM process maturity level from 5.

The post-architecture level

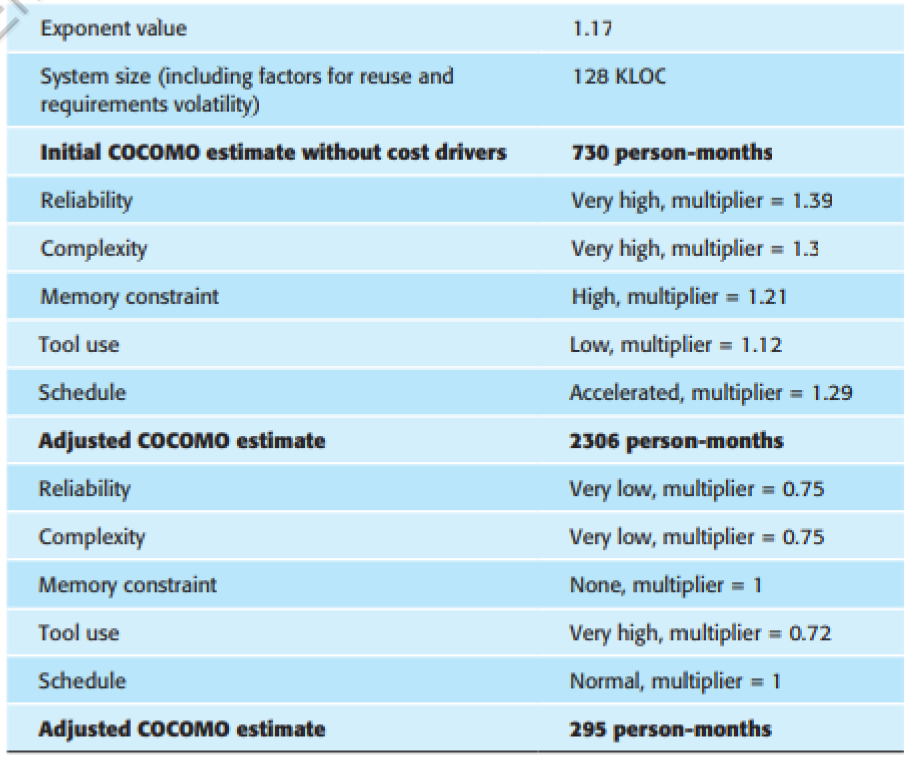
- Example:
- Imagine that an organization is taking on a project in a domain in which it has little previous experience. The project client has not defined the process to be used or allowed time in the project schedule for significant risk analysis. A new development team must be put together to implement this system. The organization has recently put in place a process improvement program and has been rated as a Level 2 organization according to the SEI capability assessment. These characteristics lead to estimates of the ratings used in exponent calculation as follows:

1. Precedentedness, rated low (4). This is a new project for the organization.
2. Development flexibility, rated very high (1). There is no client involvement in the development process, so there are few externally imposed changes.
3. Architecture/risk resolution, rated very low (5). There has been no risk analysis carried out.
4. Team cohesion, rated nominal (3). This is a new team, so there is no information available on cohesion.
5. Process maturity, rated nominal (3). Some process control is in place.

The sum of these values is 16. You then calculate the final value of the exponent by dividing this sum by 100 and adding 0.01 to the result. The adjusted value of B is therefore 1.17. The overall effort estimate is refined using an extensive set of 17 product, process, and organizational attributes (see breakout box) rather than the seven attributes used in the early design model. You can estimate values for these attributes because you have more information about the software itself, its non-functional requirements, the development team, and the development process.

The post-architecture level

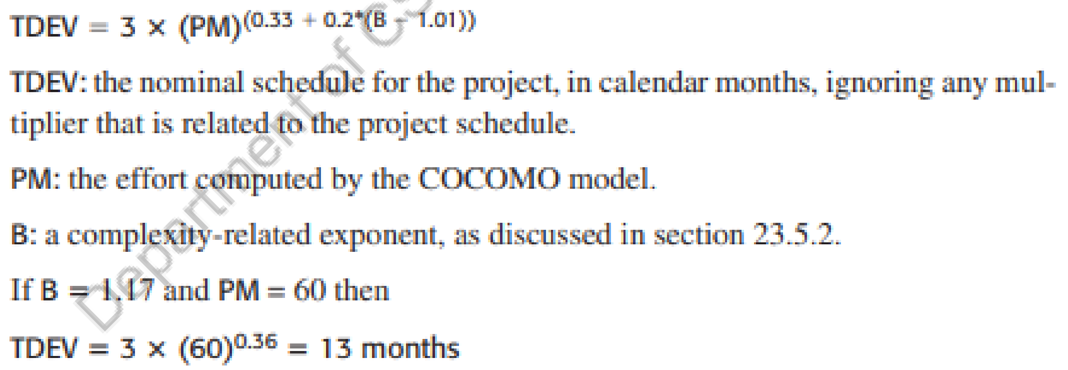
- Figure shows how the cost driver attributes influence effort estimates.
- Assume that the exponent value is 1.17 as discussed in the above example.
- Reliability (RELY), complexity (CPLX), storage (STOR), tools (TOOL), and schedule (SCED) are the key cost drivers in the project.
- All of the other cost drivers have a nominal value of 1, so they do not affect the effort computation.
- In the figure 23.13, maximum and minimum values have assigned to the key cost drivers to show how they influence the effort estimate. The values used are those from the COCOMO II reference manual .
- You can see that high values for the cost drivers lead an effort estimate that is more than three times the initial estimate, whereas low values reduce the estimate to about one third of the original.
- This highlights the significant differences between different types of project and the difficulties of transferring experience from one application domain to another.



Exponent value	1.17
System size (including factors for reuse and requirements volatility)	128 KLOC
Initial COCOMO estimate without cost drivers	**730 person-months**
Reliability	Very high, multiplier = 1.39
Complexity	Very high, multiplier = 1.3
Memory constraint	High, multiplier = 1.21
Tool use	Low, multiplier = 1.12
Schedule	Accelerated, multiplier = 1.29
Adjusted COCOMO estimate	**2306 person-months**
Reliability	Very low, multiplier = 0.75
Complexity	Very low, multiplier = 0.75
Memory constraint	None, multiplier = 1
Tool use	Very high, multiplier = 0.72
Schedule	Normal, multiplier = 1
Adjusted COCOMO estimate	**295 person-months**

Project duration and staffing

- Project managers must estimate how long the software will take to develop and when staff will be needed to work on the project.
- Increasingly, organizations are demanding shorter development schedules so that their products can be brought to market before their competitor’s.
- The COCOMO model includes a formula to estimate the calendar time required to complete a project.



$\text{TDEV} = 3 \times (\text{PM})^{(0.33 + 0.2*(B - 1.01))}$

TDEV: the nominal schedule for the project, in calendar months, ignoring any multiplier that is related to the project schedule.

PM: the effort computed by the COCOMO model.

B: a complexity-related exponent, as discussed in section 23.5.2.

If B = 1.17 and PM = 60 then

$\text{TDEV} = 3 \times (60)^{0.36} = 13$ months

Project duration and staffing

- The nominal project schedule predicted by the COCOMO model does not necessarily correspond with the schedule required by the software customer.
- You may have to deliver the software earlier or (more rarely) later than the date suggested by the nominal schedule.
- If the schedule is to be compressed (i.e., software is to be developed more quickly), this increases the effort required for the project.
- This is taken into account by the SCED multiplier in the effort estimation computation.
- Assume that a project estimated TDEV as 13 months, as suggested above, but the actual schedule required was 10 months. This represents a schedule compression of approximately 25%. Using the values for the SCED multiplier, we see that the effort multiplier for this level of schedule compression is 1.43. Therefore, the actual effort that will be required if this accelerated schedule is to be met is almost 50% more than the effort required to deliver the software according to the nominal schedule.

Project duration and staffing

- There is a complex relationship between the number of people working on a project, the effort that will be devoted to the project and the project delivery schedule.
- If four people can complete a project in 13 months (i.e., 52 person-months of effort), then you might think that by adding one more person, you could complete the work in 11 months (55 person-months of effort).
- However, the COCOMO model suggests that you will, in fact, need six people to finish the work in 11 months (66 person-months of effort).
- The reason for this is that **adding people to a project reduces the productivity of existing team members**.
- As the project team increases in size, team members spend more time communicating and defining interfaces between the parts of the system developed by other people.
- **Doubling the number of staff (for example) therefore does not mean that the duration of the project will be halved**.
- Consequently, when you add an extra person, the actual increment of effort added is less than one person as others become less productive. I
- f the development team is large, adding more people to a project sometimes increases rather than reduces the development schedule because of the overall effect on productivity.
- You cannot simply estimate the number of people required for a project team by dividing the total effort by the required project schedule.
- **Usually, a small number of people are needed at the start of a project to carry out the initial design. The team then builds up to a peak during the development and testing of the system, and then declines in size as the system is prepared for deployment**.
- As a project manager, you should therefore avoid adding too many staff to a project early in its lifetime.

MODULE 4.3

Software Project Management - Risk management, Managing people, Teamwork. Project Planning, Software pricing, Plan-driven development, Project scheduling, Agile planning. Estimation techniques, COCOMO cost modeling. **Configuration management, Version management, System building, Change management, Release management**, Agile software management - SCRUM framework. Kanban methodology and lean approaches.

Configuration management

- Configuration management (CM) is concerned with the policies, processes, and tools for managing changing software systems.
- You need to manage evolving systems because it is easy to lose track of what changes and component versions have been incorporated into each system version.
- Versions implement proposals for change, corrections of faults, and adaptations for different hardware and operating systems.
- Several versions may be under development and in use at the same time.
- If you don’t have effective configuration management procedures in place, you may waste effort modifying the wrong version of a system, delivering the wrong version of a system to customers, or forgetting where the software source code for a particular version of the system or component is stored.
- Configuration management is useful for individual projects as it is easy for one person to forget what changes have been made.
- It is essential for team projects where several developers are working at the same time on a software system.
- The configuration management system provides team members with access to the system being developed and manages the changes that they make to the code

Configuration management

- The configuration management of a software system product involves four closely related activities (Figure 1):

1. Version control: This involves keeping track of the multiple versions of system components and ensuring that changes made to components by different developers do not interfere with each other.
2. System building: This is the process of assembling program components, data, and libraries, then compiling and linking these to create an executable system.
3. Change management: This involves keeping track of requests for changes to delivered software from customers and developers, working out the costs and impact of making these changes, and deciding if and when the changes should be implemented.
4. Release management: This involves preparing software for external release and keeping track of the system versions that have been released for customer use.

Configuration management

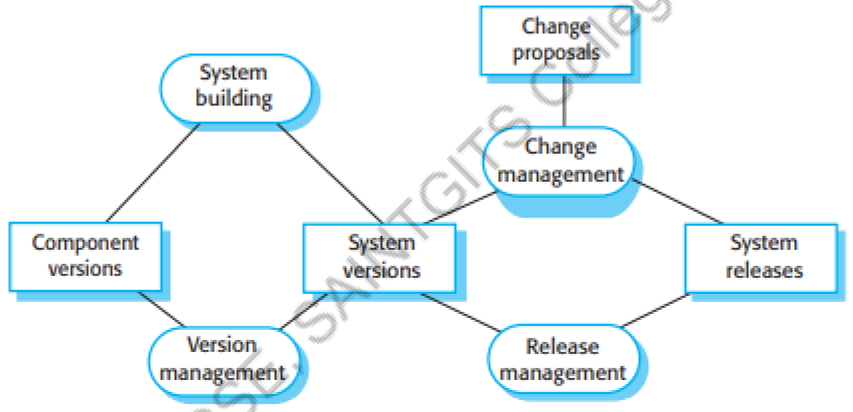


Figure 1: Configuration management activities

Configuration management

- Because of the large volume of information to be managed and the relationships between configuration items, tool support is essential for configuration management.
- Configuration management tools are used to store versions of system components, build systems from these components, track the releases of system versions to customers, and keep track of change proposals.
- CM tools range from simple tools that support a single configuration management task, such as bug tracking, to integrated environments that support all configuration management activities.
- Agile development, where components and systems are changed several times a day, is impossible without using CM tools.
- The definitive versions of components are held in a shared project repository, and developers copy them into their own workspace.
- They make changes to the code and then use system-building tools to create a new system on their own computer for testing.
- Once they are happy with the changes made, they return the modified components to the project repository. This makes the modified components available to other team members.

Configuration management

- The development of a software product or custom software system takes place in three distinct phases:

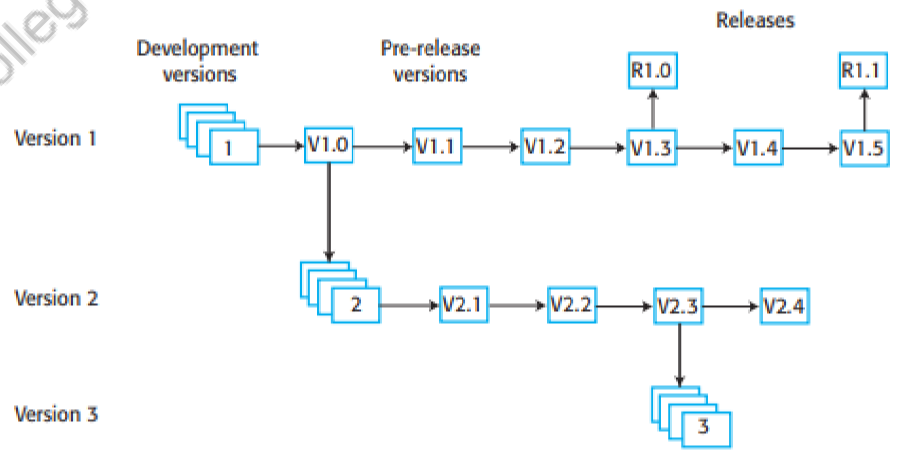
1. A **development phase** where the development team is responsible for managing the software configuration and new functionality is being added to the software. The development team decides on the changes to be made to the system.
2. A **system testing phase** where a version of the system is released internally for testing. This may be the responsibility of a quality management team or an individual or group within the development team. At this stage, no new functionality is added to the system. The changes made at this stage are bug fixes, performance improvements, and security vulnerability repairs. There may be some customer involvement as beta testers during this phase.
3. A **release phase** where the software is released to customers for use. After the release has been distributed, customers may submit bug reports and change requests. New versions of the released system may be developed to repair bugs and vulnerabilities and to include new features suggested by customers.

Configuration management

- For large systems, there is never just one “working” version of a system; there are always several versions of the system at different stages of development.
- Several teams may be involved in the development of different system versions. Figure 2 shows situations where three versions of a system are being developed:

1. Version 1.5 of the system has been developed to repair bug fixes and improve the performance of the first release of the system. It is the basis of the second system release (R1.1).
2. Version 2.4 is being tested with a view to it becoming release 2.0 of the system. No new features are being added at this stage.
3. Version 3 is a development system where new features are being added in response to change requests from customers and the development team. This will eventually be released as release 3.0.

These different versions have many common components as well as components or component versions that are unique to that system version. The CM system keeps track of the components that are part of each version and includes them as required in the system build



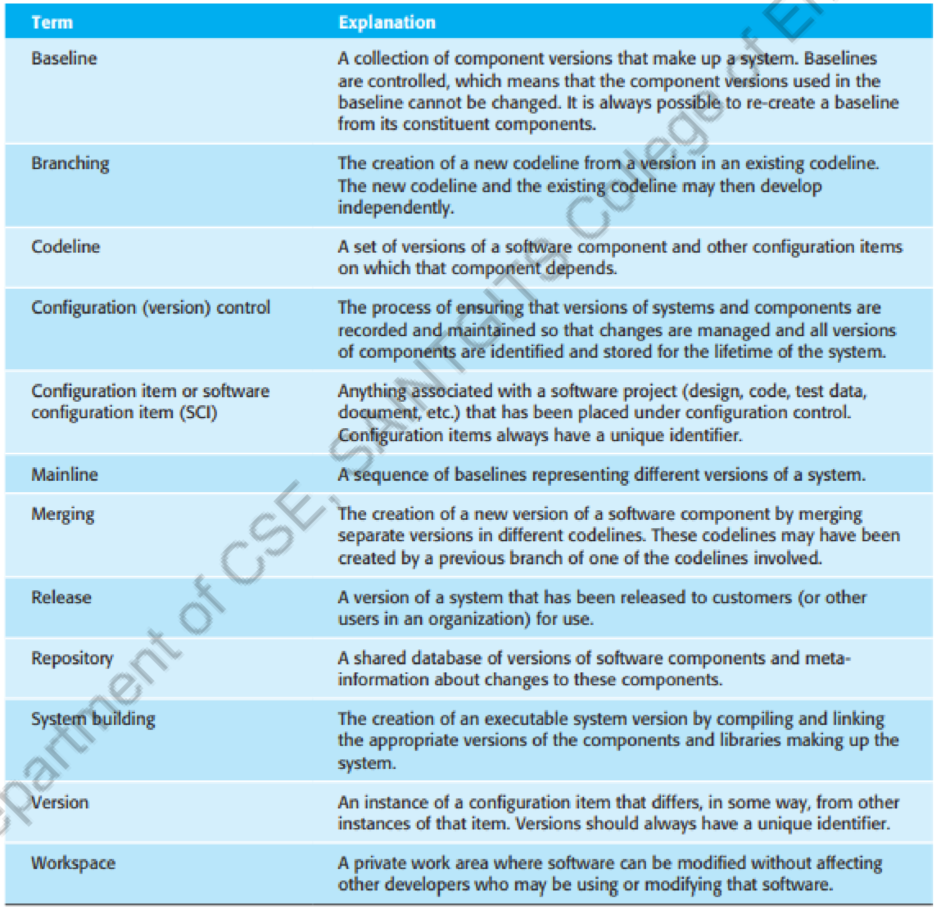
Configuration management

- In large software projects, configuration management is sometimes part of software quality management.
- The **quality manager** is responsible for both quality management and configuration management.
- When a pre-release version of the software is ready, the development team hands it over to the **quality management team**.
- The QM team checks that the system quality is acceptable. If so, it then becomes a controlled system, which means that all changes to the system have to be agreed on and recorded before they are implemented.
- Many **specialized terms** are used in configuration management. Unfortunately, these are not standardized.
- Military software systems were the first systems in which software CM was used, so the terminology for these systems reflected the processes and terminology used in hardware configuration management.
- Commercial systems developers did not know about military procedures or terminology and so often invented their own terms.
- Agile methods have also devised new terminology in order to distinguish the agile approach from traditional CM methods

Configuration management

- The definition and use of configuration management standards are essential for quality certification in both ISO 9000 and the SEI’s capability maturity model.
- CM standards in a company may be based on generic standards such as IEEE 828-2012, an IEEE standard for configuration management.
- These standards focus on CM processes and the documents produced during the CM process (IEEE 2012).
- Using the external standards as a starting point, companies may then develop more detailed, company-specific standards that are tailored to their specific needs.
- However, agile methods rarely use these standards because of the documentation overhead involved.

CM Terminology



Term	Explanation
Baseline	A collection of component versions that make up a system. Baselines are controlled, which means that the component versions used in the baseline cannot be changed. It is always possible to re-create a baseline from its constituent components.
Branching	The creation of a new codeline from a version in an existing codeline. The new codeline and the existing codeline may then develop independently.
Codeline	A set of versions of a software component and other configuration items on which that component depends.
Configuration (version) control	The process of ensuring that versions of systems and components are recorded and maintained so that changes are managed and all versions of components are identified and stored for the lifetime of the system.
Configuration item or software configuration item (SCI)	Anything associated with a software project (design, code, test data, document, etc.) that has been placed under configuration control. Configuration items always have a unique identifier.
Mainline	A sequence of baselines representing different versions of a system.
Merging	The creation of a new version of a software component by merging separate versions in different codelines. These codelines may have been created by a previous branch of one of the codelines involved.
Release	A version of a system that has been released to customers (or other users in an organization) for use.
Repository	A shared database of versions of software components and meta-information about changes to these components.
System building	The creation of an executable system version by compiling and linking the appropriate versions of the components and libraries making up the system.
Version	An instance of a configuration item that differs, in some way, from other instances of that item. Versions should always have a unique identifier.
Workspace	A private work area where software can be modified without affecting other developers who may be using or modifying that software.

Version Management

Version management

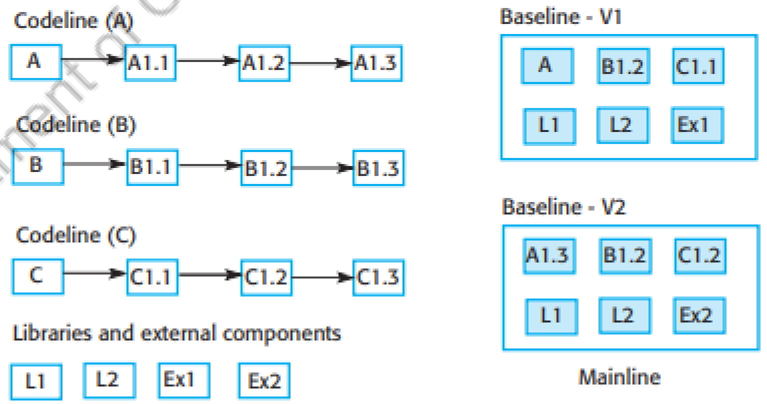
- Version management is the **process of keeping track of different versions of software components and the systems in which these components are used.**
- It also involves ensuring that changes made by different developers to these versions do not interfere with each other.
- In other words, version management is the process of managing **codelines and baselines.**

Version management

- A **codeline** is a sequence of versions of source code, with later versions in the sequence derived from earlier versions.
- Codelines normally apply to components of systems so that there are different versions of each component.
- A **baseline** is a definition of a specific system.
- The baseline specifies the component versions that are included in the system and identifies the libraries used, configuration files, and other system information.

Version management

- In the figure different baselines use different versions of the components from each codeline.
- In the diagram, boxes representing components are shaded in the baseline definition to indicate that these are actually references to components in a codeline.
- The mainline is a sequence of system versions developed from an original baseline



Version management

- Baselines may be specified using a configuration language in which you define what components should be included in a specific version of a system.
- It is possible to explicitly specify an individual component version (X.1.2, say) or simply to specify the component identifier (X).
- If you simply include the component identifier in the configuration description, the most recent version of the component should be used.
- Baselines are important because you often have to re-create an individual version of a system.

Version management

- **Version control (VC) systems** identify, store, and control access to the different versions of components.
- There are **two types of modern version control system**:

1. **Centralized systems**, where a single master repository maintains all versions of the software components that are being developed. Subversion is a widely used example of a centralized VC system.
2. **Distributed systems**, where multiple versions of the component repository exist at the same time. Git, is a widely used example of a distributed VC system.

Version management

- **Centralized and distributed VC systems** provide comparable functionality but implement this functionality in different ways. **Key features** of these systems include:

1. Version and release identification: Managed versions of a component are assigned unique identifiers when they are submitted to the system. These identifiers allow different versions of the same component to be managed, without changing the component name. Versions may also be assigned attributes, with the set of attributes used to uniquely identify each version.
2. Change history recording: The VC system keeps records of the changes that have been made to create a new version of a component from an earlier version.
3. Independent development: Different developers may be working on the same component at the same time. The version control system keeps track of components that have been checked out for editing and ensures that changes made to a component by different developers do not interfere.
4. Project support: A version control system may support the development of several projects, which share components. It is usually possible to check in and check out all of the files associated with a project rather than having to work with one file or directory at a time.
5. Storage management: Rather than maintain separate copies of all versions of a component, the version control system may use efficient mechanisms to ensure that duplicate copies of identical files are not maintained. Where there are only small differences between files, the VC system may store these differences rather than maintain multiple copies of files. A specific version may be automatically re-created by applying the differences to a master version

Version management

- Most software development is a team activity, so several team members often work on the same component at the same time.
- It's important to avoid situations where changes interfere with each other.
- The project repository maintains the "master" version of all components, which is used to create baselines for system building. When modifying components, developers copy (check-out) these from the repository into their workspace and work on these copies.
- When they have completed their changes, the changed components are returned (checked-in) to the repository.
- However, **centralized and distributed VC systems support independent development of shared components in different ways.**
- In **centralized** systems (FIGURE 3**)**, developers check out components or directories of components from the project repository into their private workspace and work on these copies in their private workspace.
- When their changes are complete, they check-in the components back to the repository. This creates a new component version that may then be shared.
- If two or more people are working on a component at the same time, each must check out the component from the repository.
- If a component has been checked out, the version control system warns other users wanting to check out that component that it has been checked out by someone else.
- The system will also ensure that when the modified components are checked in, the different versions are assigned different version identifiers and are stored separately.

Version management

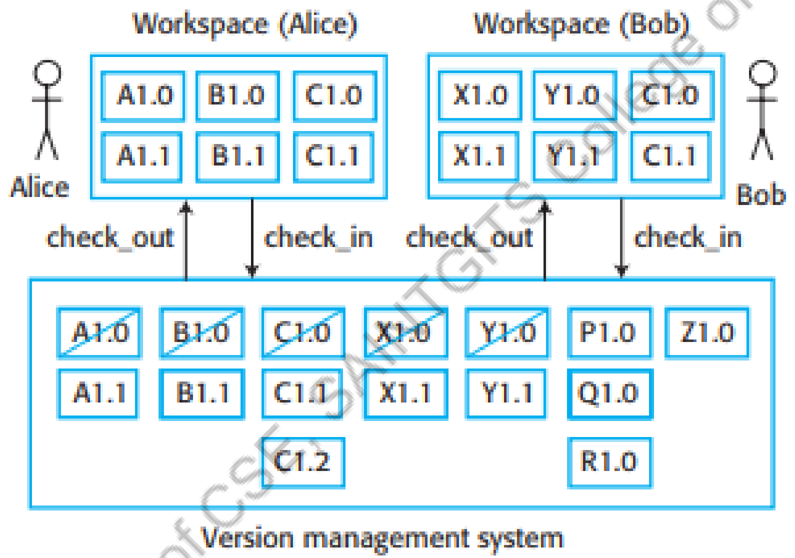


Fig 3: Check-in and check-out from a centralized version repository

Version management

- In a **distributed VC system**, such as Git, a different approach is used.
- A “master” repository is created on a server that maintains the code produced by the development team.
- Instead of simply checking out the files that they need, a developer creates a clone of the project repository that is downloaded and installed on his or her computer.
- Developers work on the files required and maintain the new versions on their private repository on their own computer.
- When they have finished making changes, they “commit” these changes and update their private server repository.
- They may then “push” these changes to the project repository or tell the integration manager that changed versions are available.
- He or she may then “pull” these files to the project repository (see Figure 4). In this example, both Bob and Alice have cloned the project repository and have updated files.
- They have not yet pushed these back to the project repository.

Version management

- This model of development has a number of advantages:

1. It provides a backup mechanism for the repository. If the repository is corrupted, work can continue and the project repository can be restored from local copies.
2. It allows for offline working so that developers can commit changes if they do not have a network connection.
3. Project support is the default way of working. Developers can compile and test the entire system on their local machines and test the changes they have made.

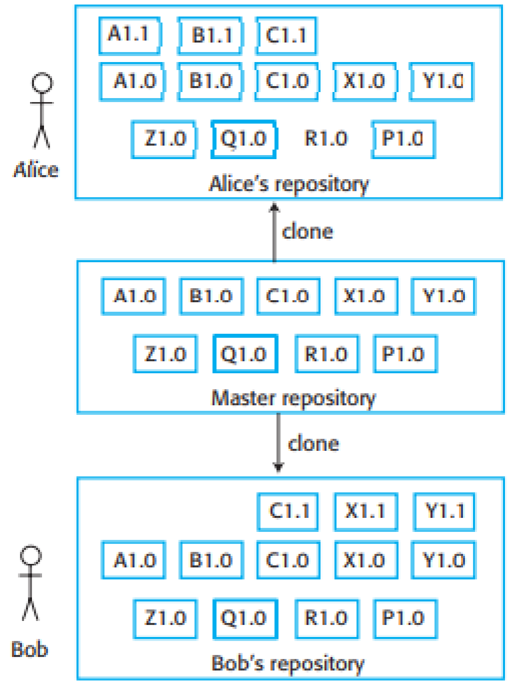


Figure 4: Repository cloning

Version management

- Distributed version control is essential for open-source development where several people may be working simultaneously on the same system without any central coordination.
- There is no way for the open-source system "manager" to know when changes will be made.
- In this case, as well as a private repository on their own computer, developers also maintain a public server repository to which they push new versions of components that they have changed.
- It is then up to the open-source system "manager" to decide when to pull these changes into the definitive system.
- This organization is shown in figure 5.

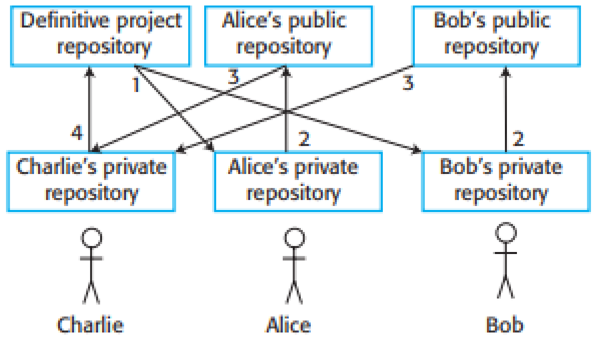

Definitive project repository
Alice's public repository
Bob's public repository
Charlie's private repository
Alice's private repository
Bob's private repository
1
2
3
4
Charlie
Alice
Bob

Figure 5

Version management

- A consequence of the independent development of the same component is that **codelines may branch**.
- Rather than a linear sequence of versions that reflect changes to the component over time, there may be **several independent sequences**, as shown in Figure 6.
- This is normal in system development, where different developers work independently on different versions of the source code and change it in different ways.
- It is generally recommended when working on a system that a new branch should be created so that changes do not accidentally break a working system.
- At some stage, it may be necessary to **merge codeline branches** to create a new version of a component that includes all changes that have been made.
- This is also shown in Figure 6, where component versions 2.1.2 and 2.3 are merged to create version 2.4.
- If the changes made involve completely different parts of the code, the component versions may be merged automatically by the version control system by combining the code changes.
- This is the normal mode of operation when new features have been added.
- These code changes are merged into the master copy of the system. However, the changes made by different developers sometimes overlap.
- The changes may be incompatible and interfere with each other. In this case, a developer has to check for clashes and make changes to the components to resolve the incompatibilities between the different versions.

Version management

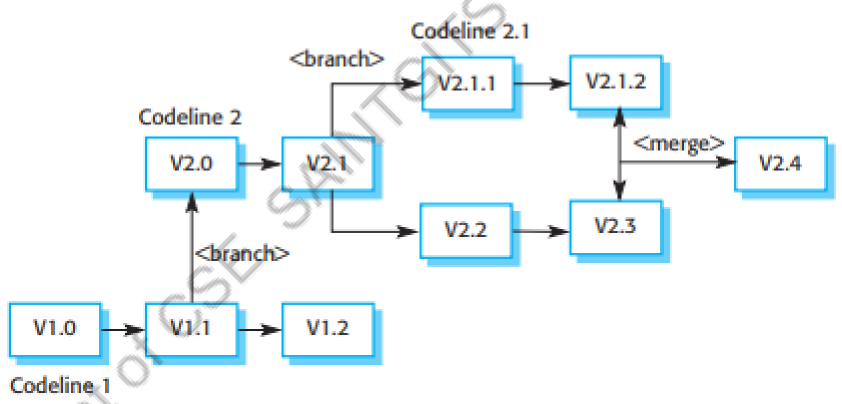
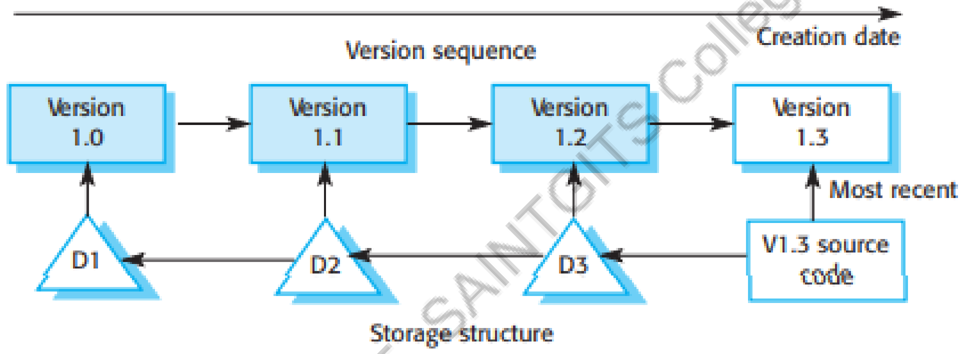


Figure 6: Branching and Merging

Version management

- When version control systems were first developed, **storage management** was one of their most important functions. Disk space was expensive, and it was important to minimize the disk space used by the different copies of components.
- Instead of keeping a complete copy of each version, the **system stores a list of differences (deltas) between one version and another.**
- By applying these to a master version (usually the most recent version), a target version can be re-created. This is illustrated in Figure 7.
- When a new version is created, the system simply stores a delta, a list of differences, between the new version and the older version used to create that new version.
- In Figure 7, the shaded boxes represent earlier versions of a component that are automatically re-created from the most recent component version.
- Deltas are usually stored as lists of changed lines, and, by applying these automatically, one version of a component can be created from another.
- As the most recent version of a component will most likely be the one used, most systems store that version in full. The deltas then define how to re-create earlier system versions.
- One of the problems with **a delta-based** approach to storage management is that it can take a long time to apply all of the deltas.
- As disk storage is now relatively cheap, Git uses an alternative, faster approach. Git does not use deltas but applies a standard **compression algorithm** to stored files and their associated meta-information. It does not store duplicate copies of files.
- Retrieving a file simply involves decompressing it, with no need to apply a chain of operations.
- Git also uses the notion of packfiles where several smaller files are combined into an indexed single file. This reduces the overhead associated with lots of small files. Deltas are used within packfiles to further reduce their size



Storage management using deltas

System building

- System building is the process of creating a complete, executable system by compiling and linking the system components, external libraries, configuration files, and other information.
- System-building tools and version control tools must be integrated as the build process takes component versions from the repository managed by the version control system.
- System building involves assembling a large amount of information about the software and its operating environment.
- Therefore, it always makes sense to use an automated build tool to create a system build (Figure 8).
- source code files that are involved in the build are not enough. You may have to link these with externally provided libraries, data files (such as a file of error messages), and configuration files that define the target installation.
- You may have to specify the versions of the compiler and other software tools that are to be used in the build. Ideally, you should be able to build a complete system with a single command or mouse click.

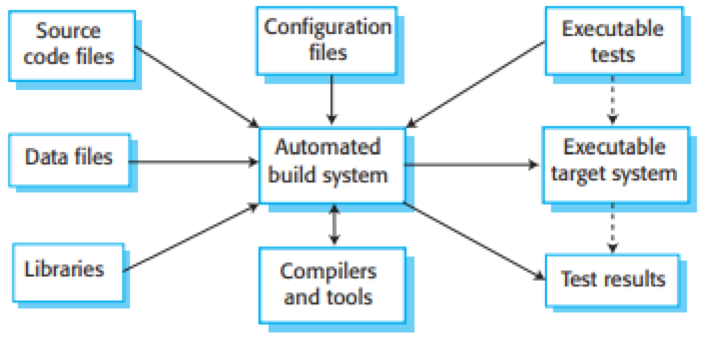


Figure 8: System building

System building

- Tools for system integration and building include some or all of the following features:

1. Build script generation: The build system should analyze the program that is being built, identify dependent components, and automatically generate a build script (configuration file). The system should also support the manual creation and editing of build scripts.
2. Version control system integration: The build system should check out the required versions of components from the version control system.
3. Minimal recompilation: The build system should work out what source code needs to be recompiled and set up compilations if required.
4. Executable system creation: The build system should link the compiled object code files with each other and with other required files, such as libraries and configuration files, to create an executable system.
5. Test automation: Some build systems can automatically run automated tests using test automation tools such as JUnit. These check that the build has not been "broken" by changes.
6. Reporting: The build system should provide reports about the success or failure of the build and the tests that have been run.
7. Documentation generation: The build system may be able to generate release notes about the build and system help pages.

System building

- The build script is a definition of the system to be built.
- It includes information about components and their dependencies, and the versions of tools used to compile and link the system.
- The configuration language used to define the build script includes constructs to describe the system components to be included in the build and their dependencies.

System building

- Building is a complex process, which is potentially error-prone, as three different system platforms may be involved (Figure 9):

1. **The development system**, which includes development tools such as compilers and source code editors. Developers check out code from the version control system into a private workspace before making changes to the system. They may wish to build a version of a system for testing in their development environment before committing changes that they have made to the version control system.
2. The **build server**, which is used to build definitive, executable versions of the system. This server maintains the definitive versions of a system. All of the system developers check in code to the version control system on the build server for system building.
3. The **target environment**, which is the platform on which the system executes. This may be the same type of computer that is used for the development and build systems. However, for real-time and embedded systems, the target environment is often smaller and simpler than the development environment (e.g., a cell phone). For large systems, the target environment may include databases and other application systems that cannot be installed on development machines. In these situations, it is not possible to build and test the system on the development computer or on the build server

System building

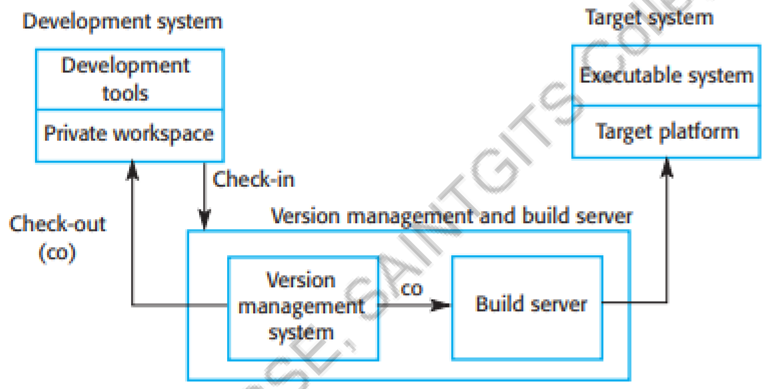


Figure 9: Development, build, and target platforms

System building

- Agile methods recommend that very frequent system builds should be carried out, with automated testing used to discover software problems. Frequent builds are part of a process of continuous integration as shown in Figure 10.
- In keeping with the agile methods notion of making many small changes, **continuous integration** involves rebuilding the mainline frequently, after small source code changes have been made.
- The **steps in continuous integration** are:

1. Extract the mainline system from the VC system into the developer's private workspace.
2. Build the system and run automated tests to ensure that the built system passes all tests. If not, the build is broken, and you should inform whoever checked in the last baseline system. He or she is responsible for repairing the problem.
3. Make the changes to the system components.
4. Build the system in a private workspace and rerun system tests. If the tests fail, continue editing.
5. Once the system has passed its tests, check it into the build system server but do not commit it as a new system baseline in the VC system.
6. Build the system on the build server and run the tests. Alternatively, if you are using Git, you can pull recent changes from the server to your private workspace. You need to do this in case others have modified components since you checked out the system. If this is the case, check out the components that have failed and edit these so that tests pass on your private workspace.
7. If the system passes its tests on the build system, then commit the changes you have made as a new baseline in the system mainline.

System building

- Tools such as Jenkins are used to support continuous integration.
- These tools can be set up to build a system as soon as a developer has completed a repository update.
- The advantage of continuous integration is that it allows problems caused by the interactions between different developers to be discovered and repaired as soon as possible.
- The most recent system in the mainline is the definitive working system.
- However, although continuous integration is a good idea, it is not always possible to implement this approach to system building:
    1. If the system is very large, it may take a long time to build and test, especially if integration with other application systems is involved. It may be impractical to build the system being developed several times per day.
    2. If the development platform is different from the target platform, it may not be possible to run system tests in the developer’s private workspace. There may be differences in hardware, operating system, or installed software. Therefore, more time is required for testing the system.

System building

- For large systems or for systems where the execution platform is not the same as the development platform, continuous integration is usually impossible. In those circumstances, frequent system building is supported using a **daily build system**:

1. The development organization sets a delivery time (say 2 p.m.) for system components. If developers have new versions of the components that they are writing, they must deliver them by that time. Components may be incomplete but should provide some basic functionality that can be tested.
2. A new version of the system is built from these components by compiling and linking them to form a complete system.
3. This system is then delivered to the testing team, which carries out a set of predefined system tests.
4. Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

The advantages of using frequent builds of software are that the chances of finding problems stemming from component interactions early in the process are increased. Frequent building encourages thorough unit testing of components.

Frequent building encourages thorough unit testing of components.

System building

- As compilation is a computationally intensive process, tools to support system building may be designed to minimize the amount of compilation that is required. They do this by checking if a compiled version of a component is available. If so, there is no need to recompile that component.
- Therefore, there has to be a way of unambiguously linking the source code of a component with its equivalent object code.
- This linking is accomplished by associating a unique signature with each file where a source code component is stored.
- The corresponding object code, which has been compiled from the source code, has a related signature.
- The signature identifies each source code version and is changed when the source code is edited. By comparing the signatures on the source and object code files, it is possible to decide if the source code component was used to generate the object code component

System building

- Two types of signature may be used.(figure 10)

1. Modification timestamps:

- The signature on the source code file is the time and date when that file was modified.
- If the source code file of a component has been modified after the related object code file, then the system assumes that recompilation to create a new object code file is necessary.
- [For example, say components Comp.java and Comp.class have modification signatures of 17:03:05:02:14:2014 and 16:58:43:02:14:2014, respectively. This means that the Java code was modified at 3 minutes and 5 seconds past 5 on the 14th of February 2014 and the compiled version was modified at 58 minutes and 43 seconds past 4 on the 14th of February 2014. In this case, the system would automatically recompile Comp.java because the compiled version has an earlier modification date than the most recent version of the component.]

2. Source code checksums

- The signature on the source code file is a checksum calculated from data in the file.
- A checksum function calculates a unique number using the source text as input.
- If you change the source code (even by one character), this will generate a different checksum. You can therefore be confident that source code files with different checksums are actually different.
- The checksum is assigned to the source code just before compilation and uniquely identifies the source file.
- The build system then tags the generated object code file with the checksum signature.
- If there is no object code file with the same signature as the source code file to be included in a system, then recompilation of the source code is necessary

System building

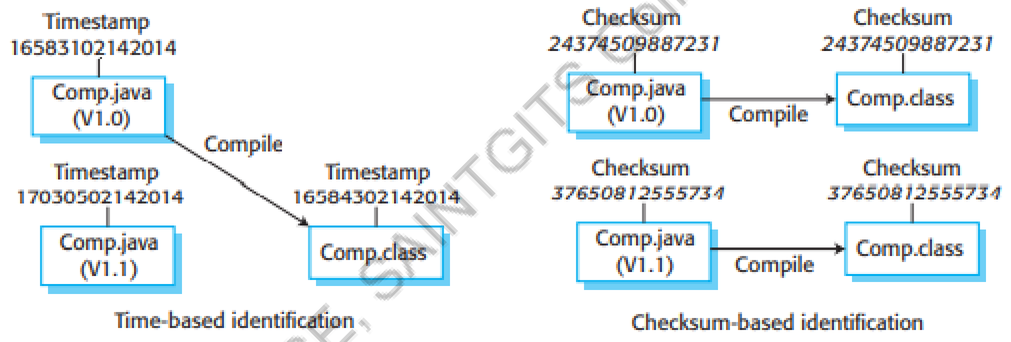


Figure 10: Linking source and object code

System building

- As object code files are not normally versioned, the first approach(modification timestamps) means that only the most recently compiled object code file is maintained in the system.
- This is normally related to the source code file by name; that is, it has the same name as the source code file but with a different suffix. Therefore, the source file Comp.Java may generate the object file Comp.class.
- Because source and object files are linked by name, it is not usually possible to build different versions of a source code component into the same directory at the same time.
- The compiler would generate object files with the same name, so only the most recently compiled version would be available.
- The checksum approach has the advantage of allowing many different versions of the object code of a component to be maintained at the same time.
- The signature rather than the filename is the link between source and object code. The source code and object code files have the same signature. Therefore, when you recompile a component, it does not overwrite the object code, as would normally be the case when the timestamp is used.
- Rather, it generates a new object code file and tags it with the source code signature. Parallel compilation is possible, and different versions of a component may be compiled at the same time.

Change management

- Change is a fact of life for large software systems. Organizational needs and requirements change during the lifetime of a system, bugs have to be repaired, and systems have to adapt to changes in their environment.
- To ensure that the changes are applied to the system in a controlled way, you need a set of tool-supported, change management processes.
- **Change management is intended to ensure that the evolution of the system is controlled and that the most urgent and cost-effective changes are prioritized.**
- **Change management is the process of analyzing the costs and benefits of proposed changes, approving those changes that are cost-effective, and tracking which components in the system have been changed.**
- Figure 11 is a model of a change management process that shows the main change management activities. This process should come into effect when the software is handed over for release to customers or for deployment within an organization

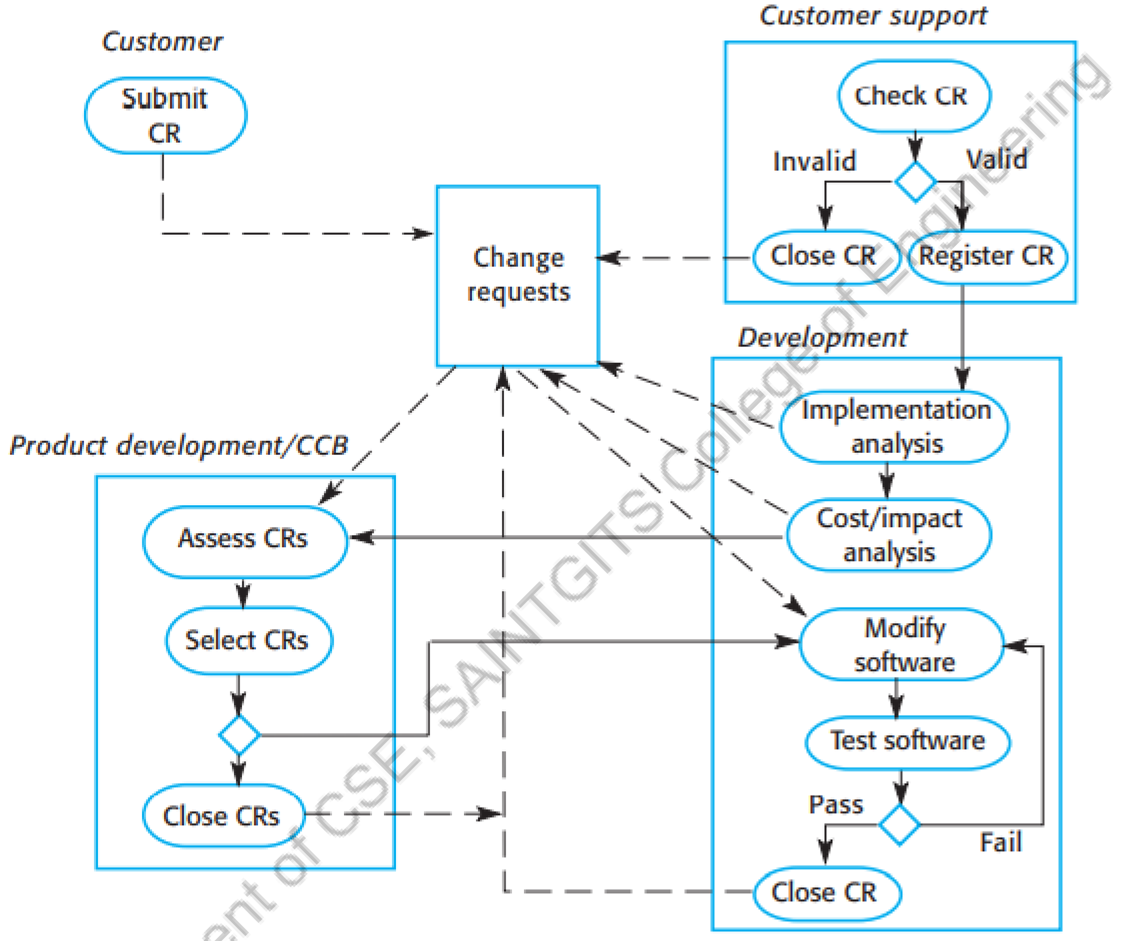


Figure 11: The change management process

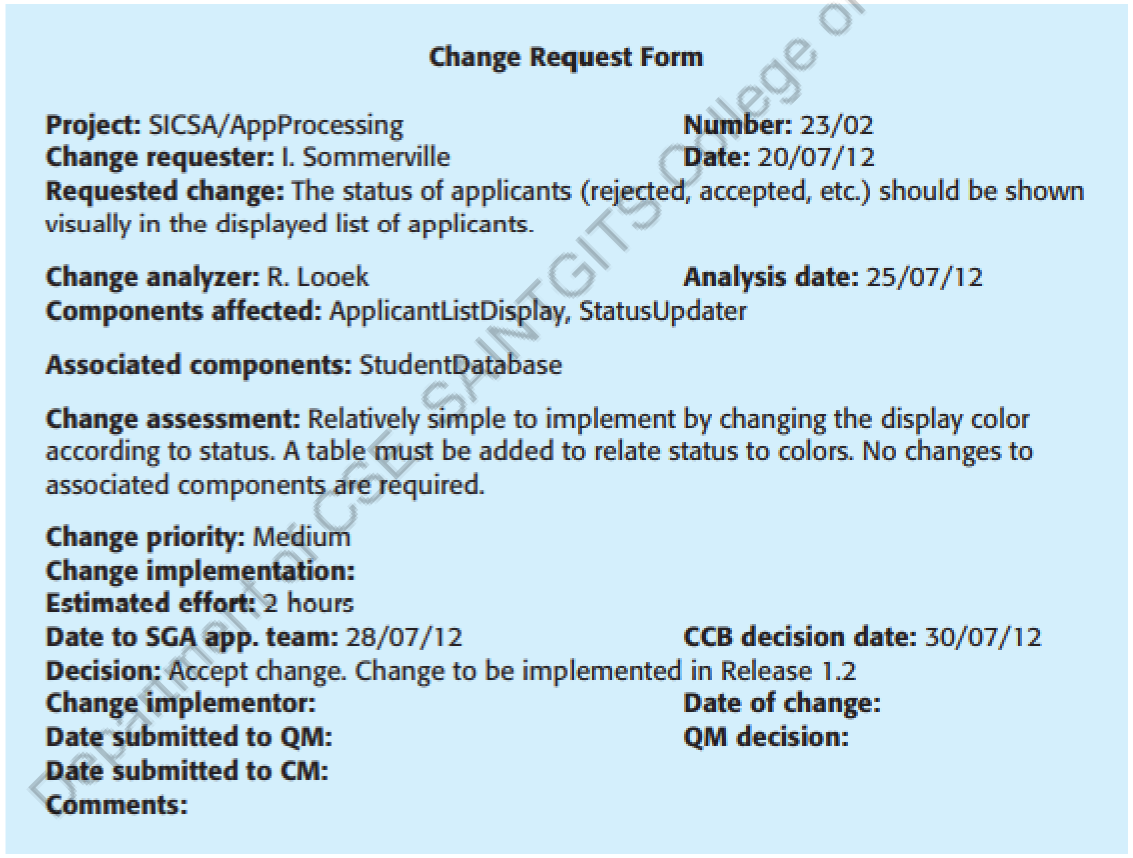
Change management

- [Many variants of this process are in use depending on whether the software is a custom system, a product line, or an off-the-shelf product. The size of the company also makes a difference—small companies use a less formal process than large companies that are working with corporate or government customers.]
- All change management processes should include some way of checking, costing, and approving changes.
- Tools to support change management may be relatively simple issue or bug tracking systems or software that is integrated with a configuration management package for large-scale systems, such as Rational Clearcase.
- Issue tracking systems allow anyone to report a bug or make a suggestion for a system change, and they keep track of how the development team has responded to the issues.
- More complex systems are built around a process model of the change management process. They automate the entire process of handling change requests from the initial customer proposal to final change approval and change submission to the development team.

Change management

- **The change management process is initiated when a system stakeholder completes and submits a change request describing the change required to the system.**
- This could be a bug report, where the symptoms of the bug are described, or a request for additional functionality to be added to the system.
- Change requests may be submitted using a **change request form (CRF).**
- Stakeholders may be system owners and users, beta testers, developers, or the marketing department of a company.
- Electronic change request forms record information that is shared between all groups involved in change management.
- As the change request is processed, information is added to the CRF to record decisions made at each stage of the process.
- At any time, it therefore represents a snapshot of the state of the change request.
- In addition to recording the change required, the CRF records the recommendations regarding the change, the estimated costs of the change, and the dates when the change was requested, approved, implemented, and validated.
- The CRF may also include a section where a developer outlines how the change may be implemented. Again, the degree of formality in the CRF varies depending on the size and type of organization that is developing the system.

Change management


Change Request Form

Project: SICSA/AppProcessing **Number:** 23/02
Change requester: I. Sommerville **Date:** 20/07/12
Requested change: The status of applicants (rejected, accepted, etc.) should be shown visually in the displayed list of applicants.

Change analyzer: R. Looek **Analysis date:** 25/07/12
Components affected: ApplicantListDisplay, StatusUpdater

Associated components: StudentDatabase

Change assessment: Relatively simple to implement by changing the display color according to status. A table must be added to relate status to colors. No changes to associated components are required.

Change priority: Medium
Change implementation:
Estimated effort: 2 hours
Date to SGA app. team: 28/07/12 **CCB decision date:** 30/07/12
Decision: Accept change. Change to be implemented in Release 1.2
Change implementor: **Date of change:**
Date submitted to QM: **QM decision:**
Date submitted to CM:
Comments:

Change management

- System developers decide how to implement the change and estimate the time required to complete the change implementation.
- **After a change request has been submitted, it is checked to ensure that it is valid**.
- The checker may be from a customer or application support team or, for internal requests, may be a member of the development team. **The change request may be rejected at this stage**.
- If the change request is a bug report, the bug may have already been reported and repaired.
- Sometimes, what people believe to be problems are actually misunderstandings of what the system is expected to do.
- On occasions, people request features that have already been implemented but that they don't know about.
- **If any of these features are true(ie. the change is not valid), the issue is closed and the form is updated with the reason for closure**.
- **If it is a valid change request, it is then logged as an outstanding request for subsequent analysis**.
- **For valid change requests, the next stage of the process is change assessment and costing**.
- This function is usually the responsibility of the development or maintenance team as they can work out what is involved in implementing the change.

Change management

- The impact of the change on the rest of the system must be checked. To do this, you have to identify all of the components affected by the change.
- If making the change means that further changes elsewhere in the system are needed, this will obviously increase the cost of change implementation.
- Next, the required changes to the system modules are assessed.
- Finally, the cost of making the change is estimated, taking into account the costs of changing related components
- Following this analysis, **a separate group decides if it is cost-effective for the business to make the change to the software**.
- For military and government systems, this group is often called the **change control board (CCB**).
- In industry, it may be called something like a "**product development group**" responsible for making decisions about how a software system should evolve.
- This group should review and approve all change requests, unless the changes simply involve correcting minor errors on screen displays, web pages, or documents.
- These small requests should be passed to the development team for immediate implementation. The CCB or product development group considers the impact of the change from a strategic and organizational rather than a technical point of view.
- It decides whether the change in question is economically justified, and it prioritizes accepted changes for implementation.
- **Accepted changes are passed back to the development group**; **rejected change requests are closed and no further action is taken**.

Change management

- The factors that influence the decision on whether or not to implement a change include:

1. The consequences of not making the change: When assessing a change request, you have to consider what will happen if the change is not implemented.[If the change is associated with a reported system failure, the seriousness of that failure has to be taken into account. If the system failure causes the system to crash, this is very serious, and failure to make the change may disrupt the operational use of the system. On the other hand, if the failure has a minor effect, such as incorrect colors on a display, then it is not important to fix the problem quickly. The change should therefore have a low priority.]
2. The benefits of the change: Will the change benefit many users of the system, or will it only benefit the change proposer?
3. The number of users affected by the change: If only a few users are affected, then the change may be assigned a low priority. In fact, making the change may be inadvisable if it means that the majority of system users have to adapt to it.
4. The costs of making the change If making the change affects many system components (hence increasing the chances of introducing new bugs) and/or takes a lot of time to implement, then the change may be rejected.
5. The product release cycle If a new version of the software has just been released to customers, it may make sense to delay implementation of the change until the next planned release

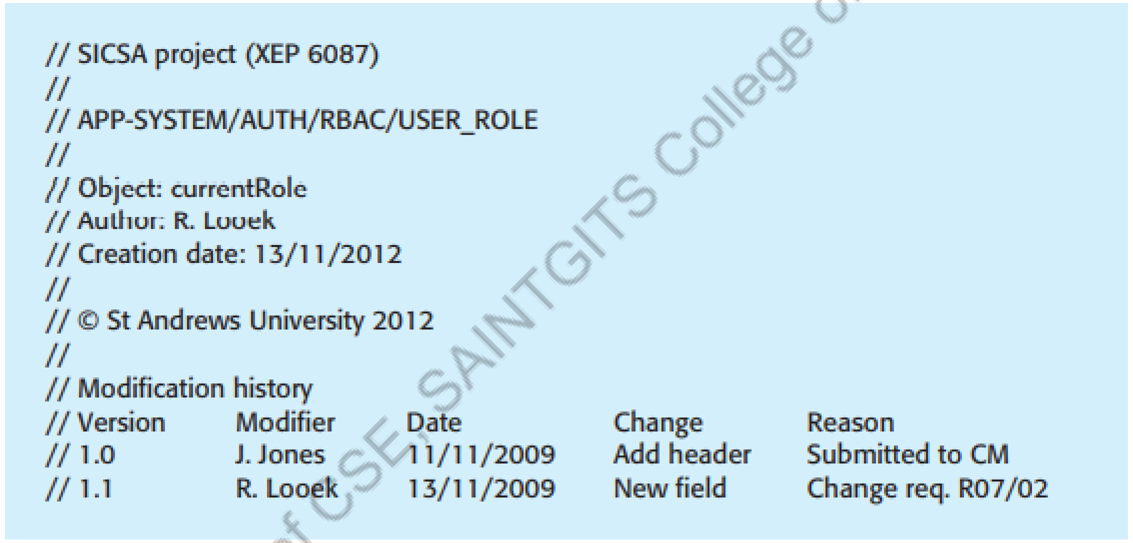
Change management

- **Change management for software products (e.g., a CAD system product), rather than custom systems specifically developed for a certain customer, are handled in a different way.**
- In software products, the customer is not directly involved in decisions about system evolution, so the relevance of the change to the customer’s business is not an issue.
- Change requests for these products come from the customer support team, the company marketing team, and the developers themselves. These requests may reflect suggestions and feedback from customers or analyses of what is offered by competing products.
- The customer support team may submit change requests associated with bugs that have been discovered and reported by customers after the software has been released.
- Customers may use a web page or email to report bugs. A bug management team then checks that the bug reports are valid and translates them into formal system change requests.
- Marketing staff may meet with customers and investigate competitive products.
- They may suggest changes that should be included to make it easier to sell a new version of a system to new and existing customers.
- The system developers themselves may have some good ideas about new features that can be added to the system.

Change management

- During development, when new versions of the system are created through daily (or more frequent) system builds, there is no need for a formal change management process.
- Problems and requested changes are recorded in an issue tracking system and discussed in daily meetings.
- Changes that only affect individual components are passed directly to the system developer, who either accepts them or makes a case for why they are not required.
- However, an independent authority, such as the system architect, should assess and prioritize changes that cut across system modules that have been produced by different development teams.
- In some agile methods, customers are directly involved in deciding whether a change should be implemented. When they propose a change to the system requirements, they work with the team to assess the impact of that change and then decide whether the change should take priority over the features planned for the next increment of the system.
- However, changes that involve software improvement are left to the discretion of the programmers working on the system.
- Refactoring, where the software is continually improved, is not seen as an overhead but as a necessary part of the development process.
- **As the development team changes software components, they should maintain a record of the changes made to each component. This is sometimes called the derivation history of a component**.
- A good way to keep the derivation history is in a **standardized comment at the beginning of the component source** code (Figure 12). This comment should reference the change request that triggered the software change. These comments can be processed by scripts that scan all components for the derivation histories and then generate component change reports.
- For documents, records of changes incorporated in each version are usually maintained in a separate page at the front of the document.

Change management



```
// SICSA project (XEP 6087)
//
// APP-SYSTEM/AUTH/RBAC/USER_ROLE
//
// Object: currentRole
// Author: R. Looek
// Creation date: 13/11/2012
//
// © St Andrews University 2012
//
// Modification history
// Version     Modifier    Date          Change       Reason
// 1.0         J. Jones    11/11/2009    Add header   Submitted to CM
// 1.1         R. Looek    13/11/2009    New field    Change req. R07/02
```

Figure 12: Derivation History

Release management

- A system release is a version of a software system that is distributed to customers.
- For mass-market software, it is usually possible to identify two types of release: **major releases**, which deliver significant new functionality, and **minor releases**, which repair bugs and fix customer problems that have been reported.
- A software product release is not just the executable code of the system.
- The release may also include:

■ configuration files defining how the release should be configured for particular installations;

■ data files, such as files of error messages in different languages, that are needed for successful system operation;

■ an installation program that is used to help install the system on target hardware;

■ electronic and paper documentation describing the system;

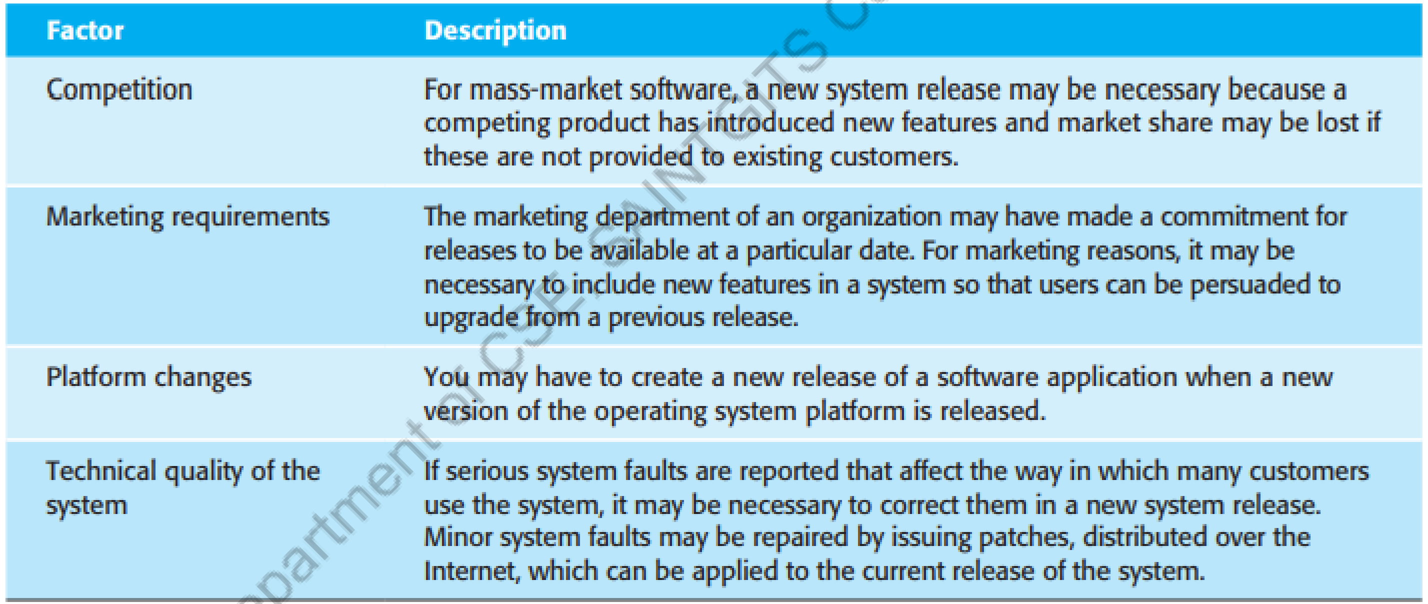
■ packaging and associated publicity that have been designed for that release

Release management

- Preparing and distributing a **system release for mass-market products is an expensive process**.
- In addition to the technical work involved in creating a release distribution, advertising and publicity material have to be prepared.
- Marketing strategies may have to be designed to convince customers to buy the new release of the system.
- Careful thought must be given to **release timing**.
- If releases are too frequent or require hardware upgrades, customers may not move to the new release, especially if they have to pay for it.
- If system releases are infrequent, market share may be lost as customers move to alternative systems.

Release management

- The various technical and organizational factors that you should take into account when deciding on when to release a new version of a software product are shown in Figure 13.



Factor	Description
Competition	For mass-market software, a new system release may be necessary because a competing product has introduced new features and market share may be lost if these are not provided to existing customers.
Marketing requirements	The marketing department of an organization may have made a commitment for releases to be available at a particular date. For marketing reasons, it may be necessary to include new features in a system so that users can be persuaded to upgrade from a previous release.
Platform changes	You may have to create a new release of a software application when a new version of the operating system platform is released.
Technical quality of the system	If serious system faults are reported that affect the way in which many customers use the system, it may be necessary to correct them in a new system release. Minor system faults may be repaired by issuing patches, distributed over the Internet, which can be applied to the current release of the system.

Figure 13: Factors influencing system release planning

Release management

- **Release creation is the process of creating the collection of files and documentation that include all components of the system release**.
- This process involves several steps:

1. The executable code of the programs and all associated data files must be identified in the version control system and tagged with the release identifier.
2. Configuration descriptions may have to be written for different hardware and operating systems.
3. Updated instructions may have to be written for customers who need to configure their own systems.
4. Scripts for the installation program may have to be written.
5. Web pages have to be created describing the release, with links to system documentation.
6. Finally, when all information is available, an executable master image of the software must be prepared and handed over for distribution to customers or sales outlets.

Release management

- For custom software or software product lines, the complexity of the system release management process depends on the number of system customers.
- Special releases of the system may have to be produced for each customer.
- Individual customers may be running several different releases of the system at the same time on different hardware.
- Where the software is part of a complex system of systems, different variants of the individual systems may have to be created.
- A software company may have to manage tens or even hundreds of different releases of their software.
- Their configuration management systems and processes have to be designed to provide information about which customers have which releases of the system and the relationship between releases and system versions.
- In the event of a problem with a delivered system, you have to be able to recover all of the component versions used in that specific system

Release management

- When a system release is produced, it must be documented to ensure that it can be re-created exactly in the future.
- This is particularly important for customized, long-lifetime embedded systems, such as military systems and those that control complex machines. These systems may have a long lifetime—30 years in some cases.
- Customers may use a single release of these systems for many years and may require specific changes to that release long after it has been superseded.
- To document a release, you have to record the specific versions of the source code components that were used to create the executable code.
- You must keep copies of the source code files, corresponding executables, and all data and configuration files.
- It may be necessary to keep copies of older operating systems and other support software because they may still be in operational use.

Release management

- You should also record the versions of the operating system, libraries, compilers, and other tools used to build the software.
- These tools may be required in order to build exactly the same system at some later date.
- Accordingly, you may have to store copies of the platform software and the tools used to create the system in the version control system, along with the source code of the target system.

Release management

- When planning the installation of new system releases, you cannot assume that customers will always install new system releases. Some system users may be happy an existing system and may not consider it worthwhile to absorb the cost of changing to a new release.
- New releases of the system cannot, therefore, rely on the installation of previous releases.
- One benefit of delivering software as a service (SaaS) is that it avoids all of these problems.
- It simplifies both release management and system installation for customers.
- The software developer is responsible for replacing the existing release of a system with a new release, which is made available to all customers at the same time.
- However, this approach requires that all servers running the services be updated at the same time. To support server updates, specialized distribution management tools such as Puppet have been developed for “pushing” new software to servers.

Kanban Methodology and Lean Approaches

Module 4

KANBAN

- KAN is a word for “card” in Japanese.
- BAN is a word for “signal”.
- KANBAN means signal card.
- Each card contains a small amount of work, say a story to develop and some clarifying testable examples.
- The kanban method consists of principles, practices, kanban boards and kanban cards.
- **What Is Kanban?**
- Kanban is a project management methodology that gives project managers full transparency into the task management process..
- The kanban method was originally created as a lean manufacturing tool to maximize production efficiency.

Kanban Principles

- Start with what you’re doing now. Don’t make changes to your process immediately, but use kanban for your current workflow.
- Changes occur organically over time and shouldn’t be rushed.
- Evolutionary change is incremental, not radical, so as not to give teams cause for alarm or resistance.
- Respect current roles and responsibilities, and allow teams to collaboratively identify and implement any changes.
- Encourage leadership from everyone to help keep the mandate of continuous change for maximizing improvements.

KANBAN vs SCRUM

- Kanban is a project management method that helps visualize tasks, while Scrum is a method that provides structure to the team and schedule.
- Kanban and Scrum are project management methodologies that complete project tasks in small increments and emphasize continuous improvement. But the processes they use to achieve those ends are different.
  - While Kanban is centered around visualizing tasks and continuous flow, Scrum is more about implementing timelines for each delivery cycle and assigning set roles.

KANBAN

KANBAN BOARD

- The heart of the Kanban method is the Kanban board—physical or digital—in which phases of the project are divided into columns.
- The kanban board represents the overall project and is usually broken up into three columns: to do, in progress and done.

KANBAN CARD

- Each kanban card is filled with information related to that task, such as its name and a short description.
- Task will be assigned to the team member, who is responsible for executing the task by the deadline.

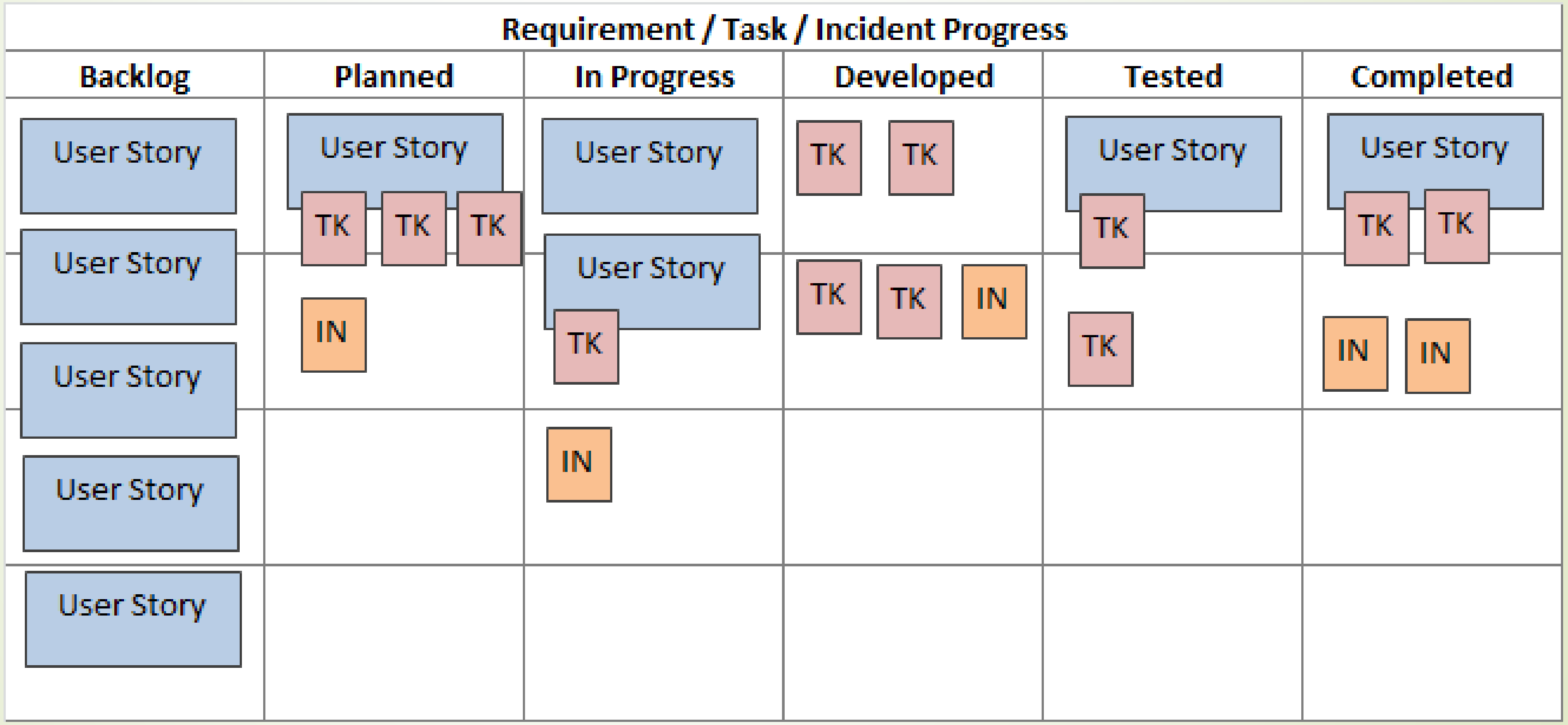
KANBAN COLUMNS

- Columns reside on the board are a way to break up the different stages in the project workflow.
- Tasks progress from one column to the next, until the task is completed, so its an indicator of current status of a task .

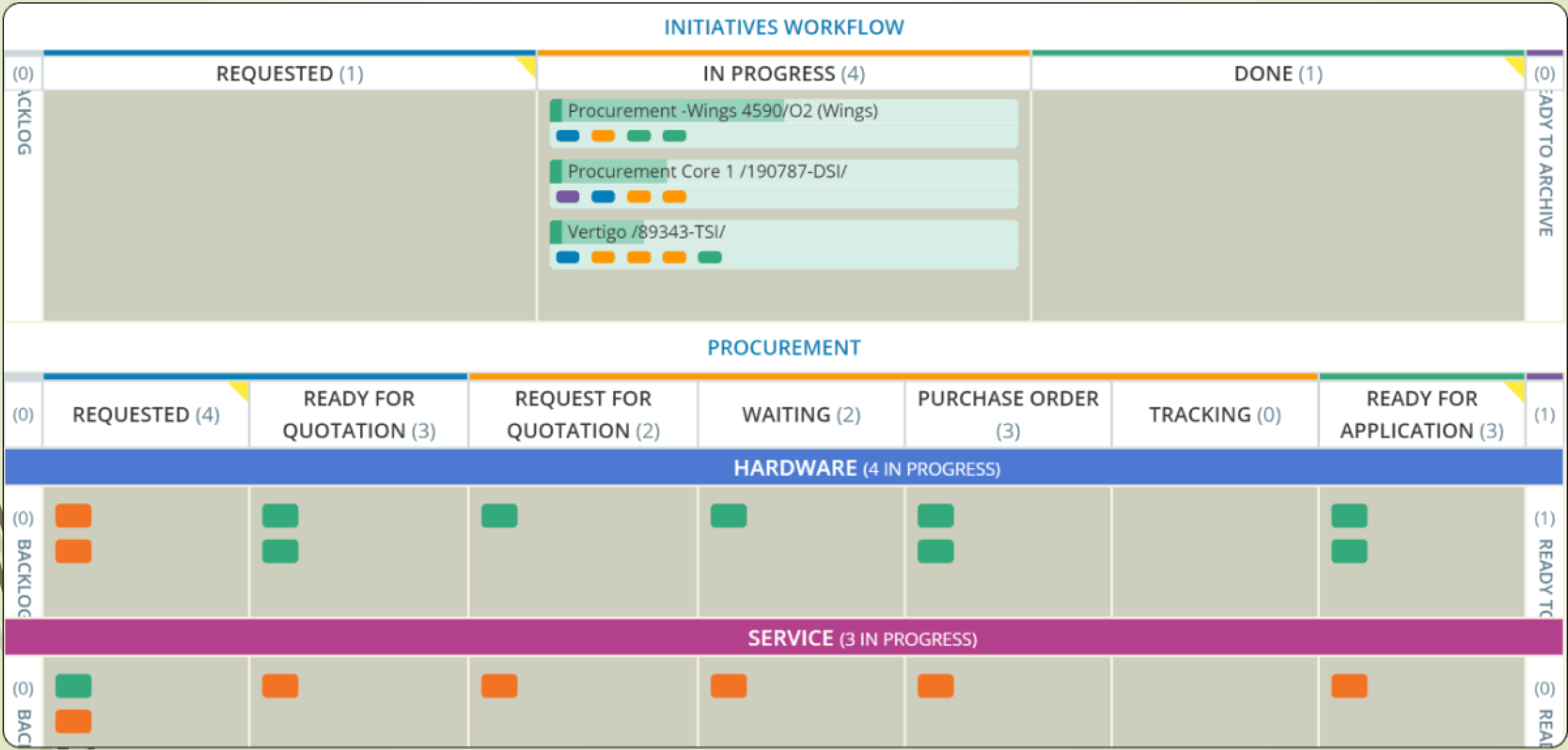
KANBAN SWIMLANES

- The key component that will enable you to visualize the whole process on a single board and distinguish the different parts that actually run simultaneously.
- Horizontal lanes help to separate different work items, activities, teams, services, etc.

Kanban Board/ Card/Columns



Example of KANBAN swimlanes



INITIATIVES WORKFLOW

(0) | REQUESTED (1) | IN PROGRESS (4) | DONE (1) | (0)

Procurement -Wings 4590/O2 (Wings)

Procurement Core 1 /190787-DSI/

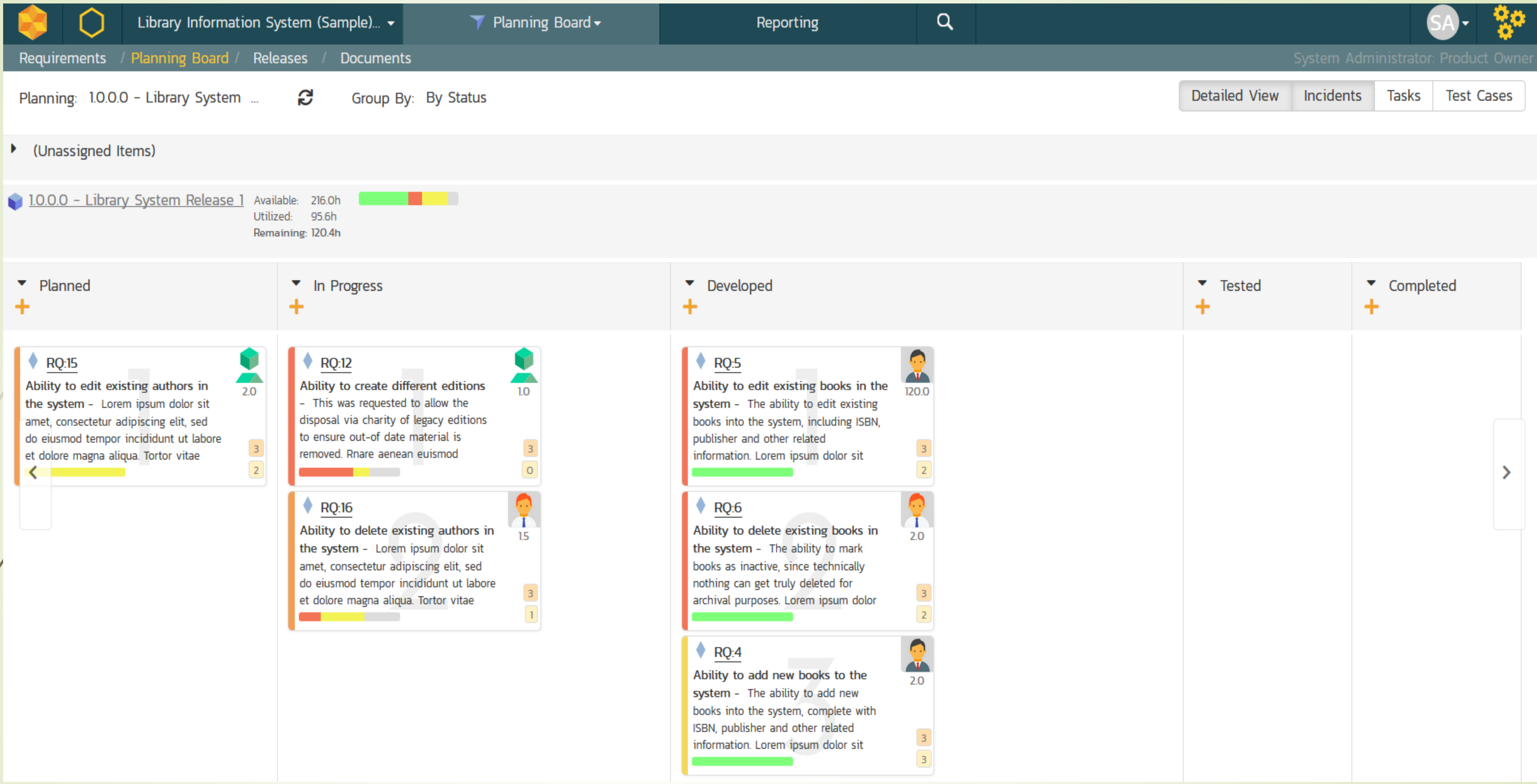
Vertigo /89343-TSI/

PROCUREMENT

(0) | REQUESTED (4) | READY FOR QUOTATION (3) | REQUEST FOR QUOTATION (2) | WAITING (2) | PURCHASE ORDER (3) | TRACKING (0) | READY FOR APPLICATION (3) | (1)

HARDWARE (4 IN PROGRESS)

SERVICE (3 IN PROGRESS)


Library Information System (Sample)... | Planning Board | Reporting

Requirements / Planning Board / Releases / Documents

System Administrator: Product Owner

Planning: 1.0.0.0 - Library System ... Group By: By Status

Detailed View | Incidents | Tasks | Test Cases

(Unassigned Items)

1.0.0.0 - Library System Release 1
Available: 216.0h
Utilized: 95.6h
Remaining: 120.4h

Planned	In Progress	Developed	Tested	Completed
RQ:15 Ability to edit existing authors in the system - Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Tortor vitae (2.0)	RQ:12 Ability to create different editions - This was requested to allow the disposal via charity of legacy editions to ensure out-of date material is removed. Rnare aenean euismod (1.0)	RQ:5 Ability to edit existing books in the system - The ability to edit existing books into the system, including ISBN, publisher and other related information. Lorem ipsum dolor sit (120.0)		
	RQ:16 Ability to delete existing authors in the system - Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Tortor vitae (1.5)	RQ:6 Ability to delete existing books in the system - The ability to mark books as inactive, since technically nothing can get truly deleted for archival purposes. Lorem ipsum dolor (2.0)		
		RQ:4 Ability to add new books to the system - The ability to add new books into the system, complete with ISBN, publisher and other related information. Lorem ipsum dolor sit (2.0)		

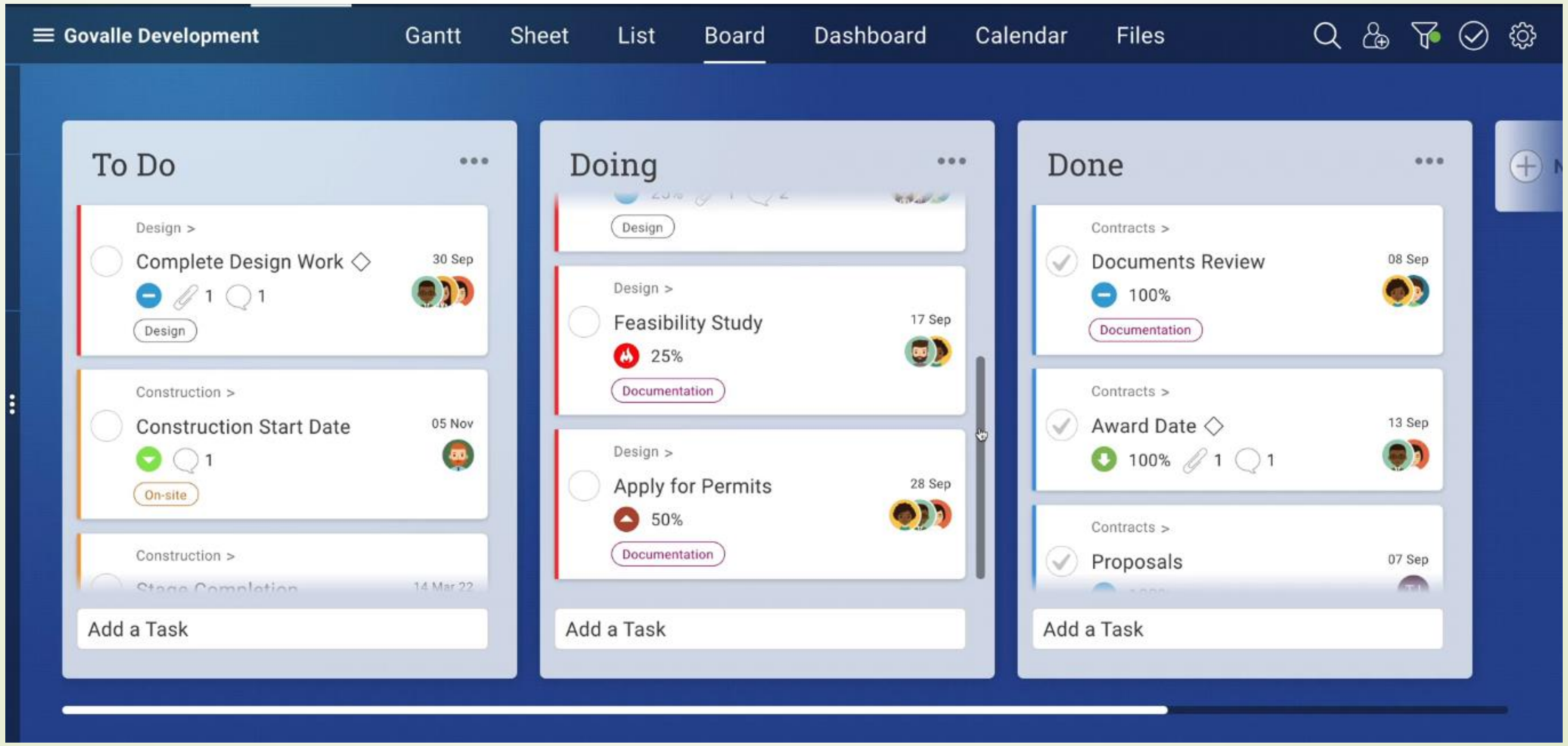
When KANBAN is used in software development, it uses the stages in the software development lifecycle (SDLC) to represent the different stages in the manufacturing process.

Benefits

- Kanban increases transparency in a project by visually clarifying what tasks need to be completed and where tasks are piling up. This visual aid makes it easier to delegate resources where they need to go, reducing inefficiencies.
    - Keeps tasks organized
    - Create customized workflows
    - Share boards for collaboration
    - Track production of tasks in real-time
- Kanban uses principles from both Agile and Lean.

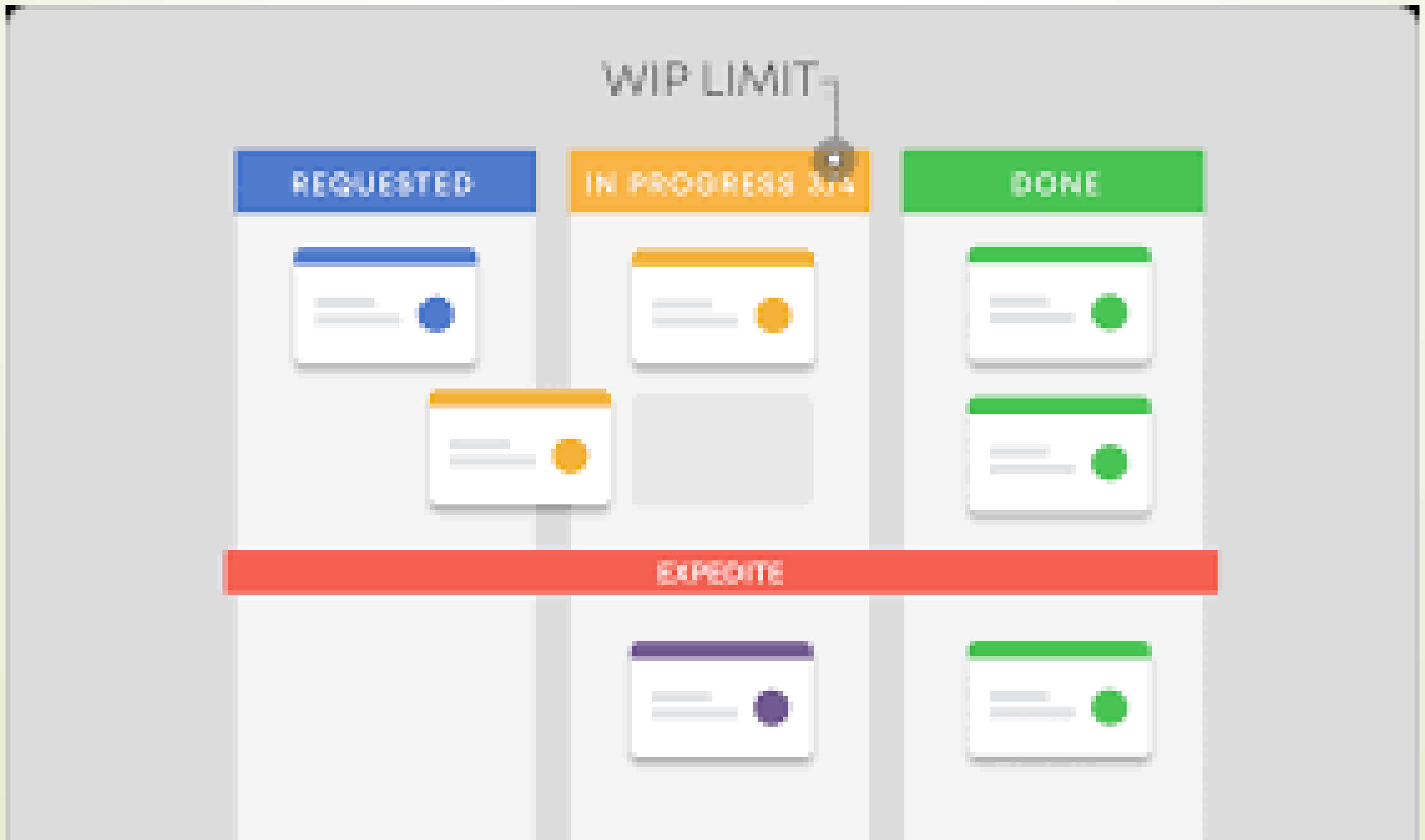
Key concepts in KANBAN

- **Definition of Workflow (DoW):** The DoW defines key parts of the Kanban workflow, such as what units are moving through the board, what "started" or "finished" means, and how long it should take for an item to progress through the columns.

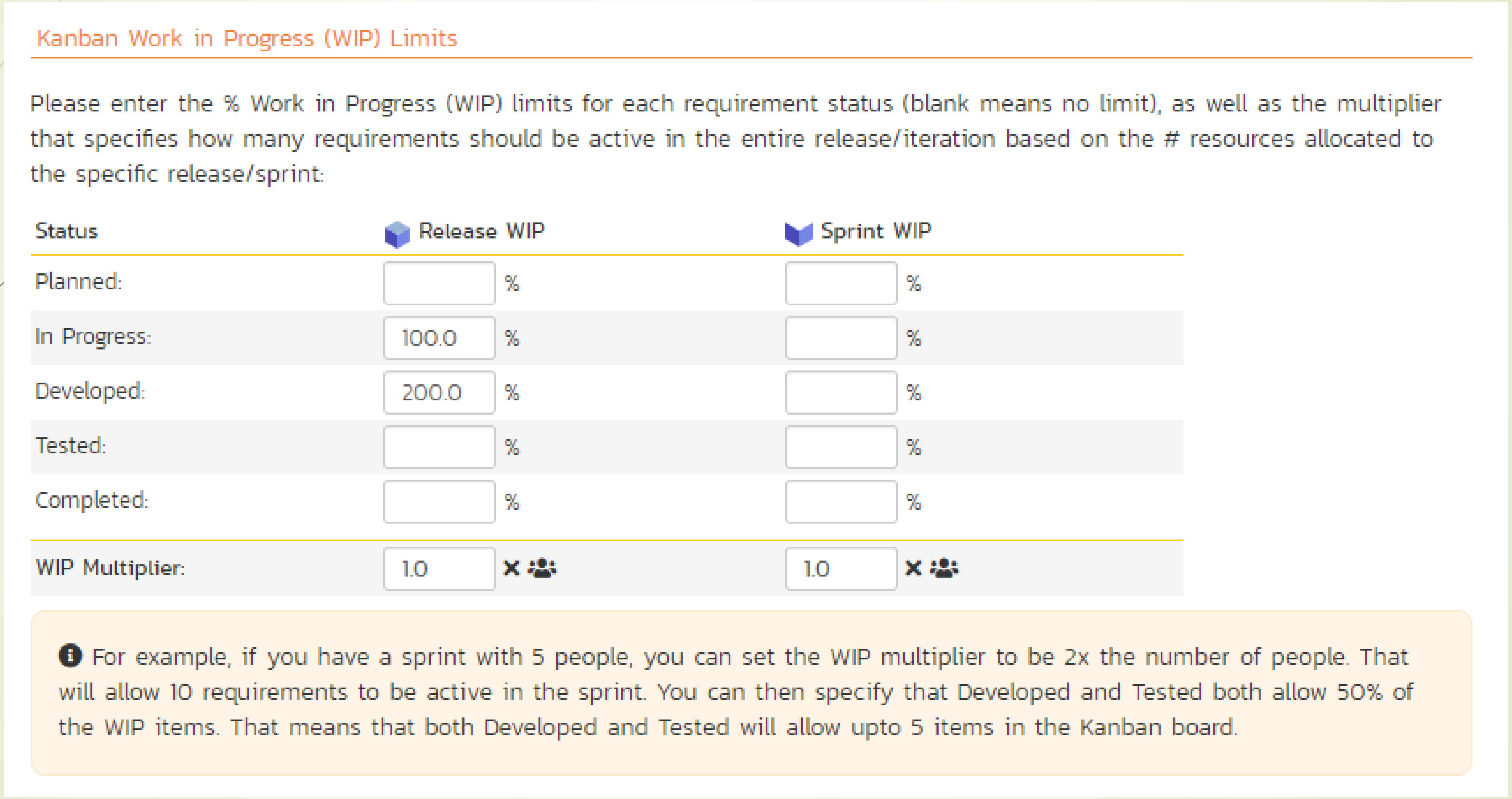


Key concepts in KANBAN

- **Work in progress (WIP) limits:** Teams can set WIP limits in a column, groups of columns, or the entire board. This means a column with a WIP limit of five can't have more than five cards in it at a time. If there are five, the team must tackle the tasks in that column before new ones can be moved in. WIP limits can help surface bottlenecks in the production process.



Setting WIP Limits



Kanban Work in Progress (WIP) Limits

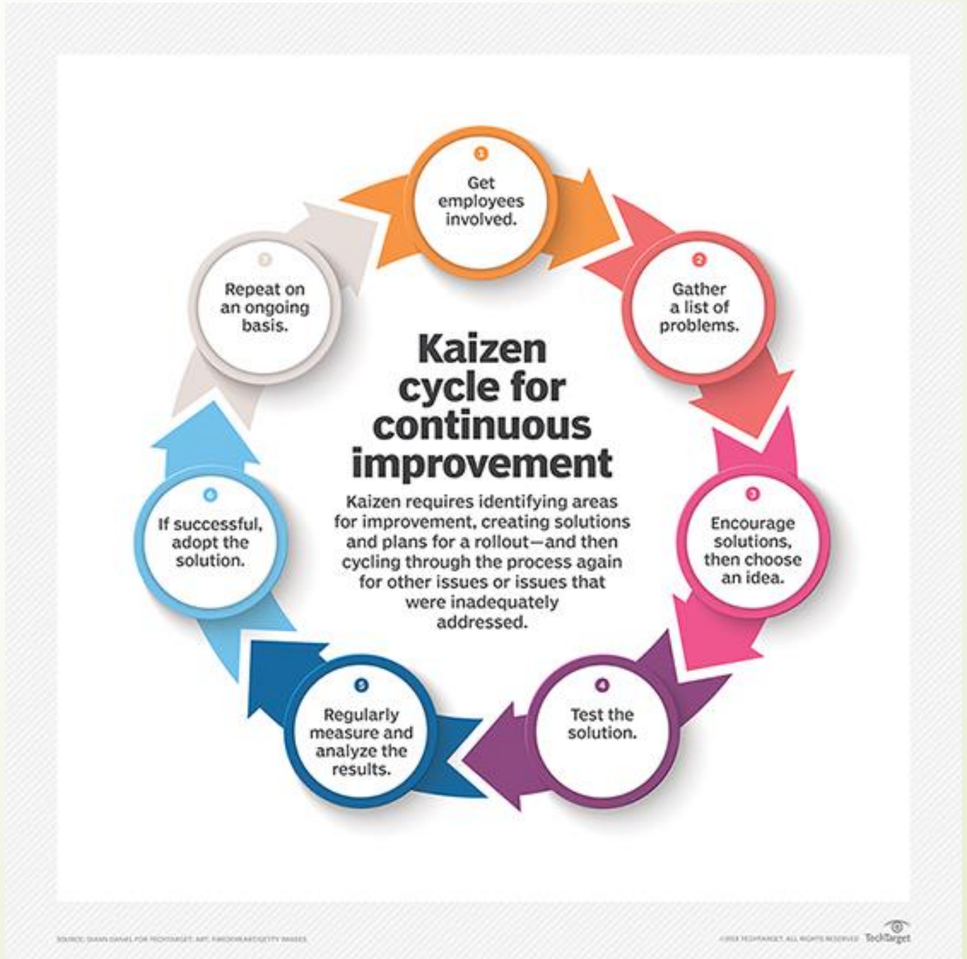
Please enter the % Work in Progress (WIP) limits for each requirement status (blank means no limit), as well as the multiplier that specifies how many requirements should be active in the entire release/iteration based on the # resources allocated to the specific release/sprint:

Status	Release WIP	Sprint WIP
Planned:	%	%
In Progress:	100.0 %	%
Developed:	200.0 %	%
Tested:	%	%
Completed:	%	%
WIP Multiplier:	1.0 ×	1.0 ×

For example, if you have a sprint with 5 people, you can set the WIP multiplier to be 2x the number of people. That will allow 10 requirements to be active in the sprint. You can then specify that Developed and Tested both allow 50% of the WIP items. That means that both Developed and Tested will allow upto 5 items in the Kanban board.

Key concepts in KANBAN

- **Kaizen :** Meaning "improvement" in Japanese, kaizen encourages a mindset to continually better the process. This encourages all team members to share their insights and work to improve the team, not just managers.



Kaizen cycle for continuous improvement
Kaizen requires identifying areas for improvement, creating solutions and plans for a rollout—and then cycling through the process again for other issues or issues that were inadequately addressed.
1 Get employees involved.
2 Gather a list of problems.
3 Encourage solutions, then choose an idea.
4 Test the solution.
5 Regularly measure and analyze the results.
6 If successful, adopt the solution.
7 Repeat on an ongoing basis.
TechTarget

Kanban vs. Scrum: Which should I choose?

- Kanban and Scrum each have their separate strengths. But putting Kanban against Scrum is a false dilemma; you can easily use both in your work to maximize the benefits.

When to use Kanban:

- Kanban is to improve visibility, foster a culture of continuous improvement, and increase productivity.
- Kanban can fit in with processes that already exist—including Scrum. If you don't want to overhaul your entire work process but are hoping to gain the benefits that an Agile process can bring, Kanban can be a good way to start.

When to use Scrum:

- Scrum has been linked to higher productivity, faster delivery, lower costs, and higher quality. Many project managers also see Scrum as an effective method to tackle complex projects, or projects that might see frequent change.
- Scrum can make sense to use if you're in an industry that sees frequent change, or if your project might need space to adapt to feedback. This might include industries that have frequent technology updates, or projects creating new products.

Scrumban: choosing both

- Scrumban is a hybrid method that combines both Kanban and Scrum. Scrumban uses the processes of Scrum and the visualization tools of Kanban. Scrumban can be a good way for teams familiar with either Scrum or Kanban to incorporate the other into their process.

Kanban vs. Scrum: Similarities and differences

- The similarities and differences between Kanban and Scrum can be summarized as follows:
- Kanban and Scrum are both methodologies that allow projects to adapt to change, encourage engagement by all team members, have short development cycles, and increase transparency.
- Kanban is a methodology centered around visualizing tasks, while Scrum is a methodology that structures workflow and team culture to deliver projects in short timelines.
- Kanban delivers tasks continuously until the project is finished, while Scrum delivers chunks of deliverables in one- to four-week periods.

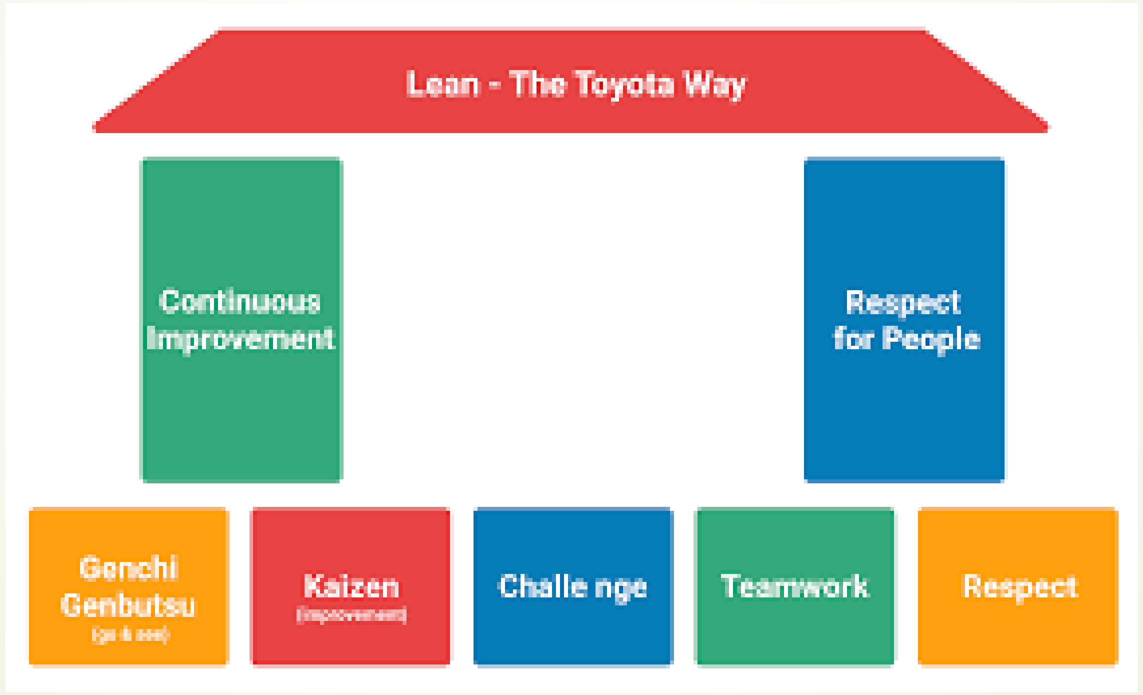
Methodology	Kanban	Scrum
Roles	No defined roles	Scrum master, product owner, and development team
Delivery cycle	Continuous	Sprint cycle lasts one to four weeks
Change policy	Can be incorporated any time	Generally not made during sprint
Artifacts	Kanban board	Product backlog, sprint backlog, product increments
Tools	Jira Software, Kanbanize, SwiftKanban, Trello, Asana	Jira Software, Axosoft, VivifyScrum, Targetprocess
Key concepts or pillars	Effective, efficient, predictable	Transparency, adaptation, inspection

Kanban vs. Agile

- **Agile** is a set of project management principles that encourage an adaptive and iterative way of approaching project management.
- Agile is an overarching philosophy, and not a set of tools or processes.
- It emphasizes flexibility over following a plan, and is often used for projects that are met frequently with change.
- **Kanban**, on the other hand, is an Agile methodology. This means it offers the specific tools and processes to implement Agile.
- It exhibits many principles characteristic of Agile, including the capacity to adapt to changes, and transparency across the team.

LEAN

- Lean development is the application of Lean principles to software development.
- Lean principles got their start in manufacturing, as a way to optimize the production line to minimize waste and maximize value to the customer. These two goals are also relevant to software development, which also:
  - Follows a repeatable process
  - Requires particular quality standards
  - Relies on the collaboration of a group of specialized workers
- *Manufacturing* deals with the production of physical goods, while the value being created in *software development* is created within the mind of the developer.


Lean - The Toyota Way
Continuous Improvement
Respect for People
Genchi Genbutsu (go & see)
Kaizen (improvement)
Challe nge
Teamwork
Respect

7 Lean Development Principles

- The seven Lean principles are:
  - Eliminate waste
  - Build quality in
  - Create knowledge
  - Defer commitment
  - Deliver fast
  - Respect people
  - Optimize the whole

Eliminate waste

One of the key elements of practicing Lean is to **eliminate anything that does not add value to the customer**

- **Unnecessary code or functionality**: Delays time to customer, slows down feedback loops
- **Starting more than can be completed**: Adds unnecessary complexity to the system, results in context-switching, handoff delays, and other impediments to flow
- **Delay in the software development process**: Delays time to customer, slows down feedback loops
- **Unclear or constantly changing requirements**: Results in rework, frustration, quality issues, lack of focus
- **Bureaucracy**: Delays speed
- **Slow or ineffective communication**: Results in delays, frustrations, and poor communication to stakeholders which can impact IT's reputation in the organization
- **Partially done work**: Does not add value to the customer or allow team to learn from work
- **Defects and quality issues**: Results in rework, abandoned work, and poor customer satisfaction
- **Task switching:** Results in poor work quality, delays, communication breakdowns, and low team morale

Build quality in

- Every team wants to build quality into their work.
- In Lean development, quality is everyone’s job, not just that of the quality analyst.
- Lean development tools for building quality in:
  - **Pair programming**: Avoid quality issues by combining the skills and experience of two developers instead of one
  - **Test-driven development**: Writing criteria for code before writing the code to ensure it meets business requirements
  - **Incremental development and constant feedback**
  - **Minimize wait states**: Reduce context switching, knowledge gaps, and lack of focus
  - **Automation**: Automate any tedious, manual process or any process prone to human error

Defer commitment

- Defer Commitment does not mean that teams should be flaky or irresponsible about their decision making.
- To defer commitment means to:
  - **Not plan (in excessive detail) for months in advance**
  - **Not commit to ideas or projects without a full understanding of the business requirements**
  - **Constantly be collecting and analyzing information regarding any important decisions**
- This Lean principle encourages team to demonstrate responsibility by keeping their options open and continuously collecting information, rather than making decisions without the necessary data.

Respect for people

- It applies to every aspect of the way Lean teams operate, from how they communicate, handle conflict, hire and on board new team members, deal with process improvement, and more.
- Lean development teams can encourage respect for people by:
  - **Communicating proactively and effectively**
  - **Encouraging healthy conflict**
  - **Surfacing any work-related issues as a team.**
  - **Empowering each other to do their best work**

Deliver fast

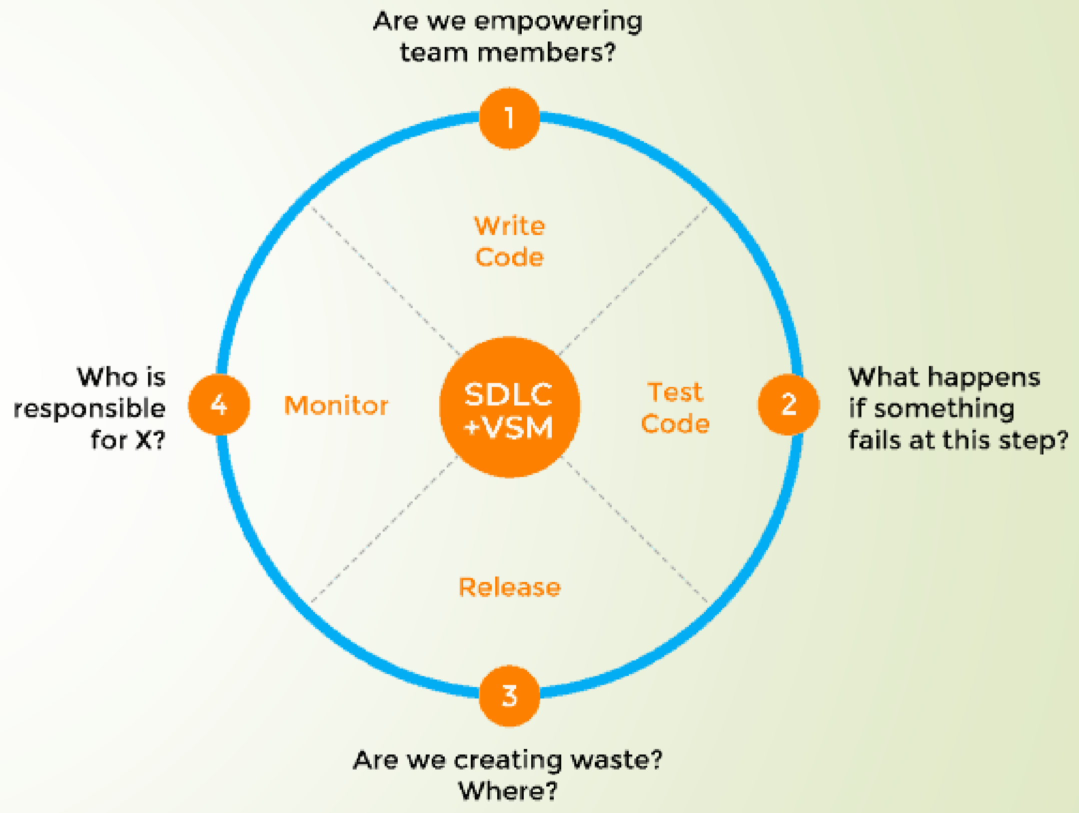
- Every team wants to deliver fast, to put value into the hands of the customer as quickly as possible.
- Common culprits that slows down are
  - Thinking too far in advance about future requirements
  - Blockers that aren't responded to with urgency
  - Over-engineering solutions and business requirements
- **Lean development** is based on this concept:
  - Build a simple solution, put it in front of customers, enhance incrementally based on customer feedback.

Optimize the whole

- **Suboptimization** is a serious issue in software development. Mary and Tom Poppendieck describe two vicious cycles into which Lean development teams often fall.
- The first is releasing sloppy code for the sake of speed.
  - Developers release code that may or may not meet quality requirements. This increases the complexity of the code base, resulting in more defects.
- The second is an issue with testing.
  - When testers are overloaded, it creates a long cycle time between when developers write code and when testers are able to give feedback on it. This means that developers continue writing code that may or may not be defective, resulting in more defects and therefore requiring more testing.

Optimize the whole

- Optimising the whole is an antidote to sub optimization with a better understanding of capacity and the downstream impact of work.
- It's based on the idea that every business represents a **value stream** –
  - the sequence of activities required to design, produce, and deliver a product or service to customers.
  - If our goal is to deliver as much value to our customers as quickly as possible, then we have to optimize our value streams to be able to do just that.
  - To understand how to optimize our value streams, first we have to properly identify them.


Are we empowering team members?
1
Write Code
Who is responsible for X?
4
Monitor
SDLC +VSM
Test Code
2
What happens if something fails at this step?
Release
3
Are we creating waste? Where?

References

- https://www.projectmanager.com/kanban
- https://www.coursera.org/articles/kanban-vs-scrum
- https://www.planview.com/resources/articles/lkdc-principles-lean-development/
- What is Agile Kanban Methodology? Learn the Methods & Tools (inflectra.com)
- What Are Kanban Swimlanes and How to Use Them? (kanbanize.com)
- https://www.plutora.com/blog/lean-software-development