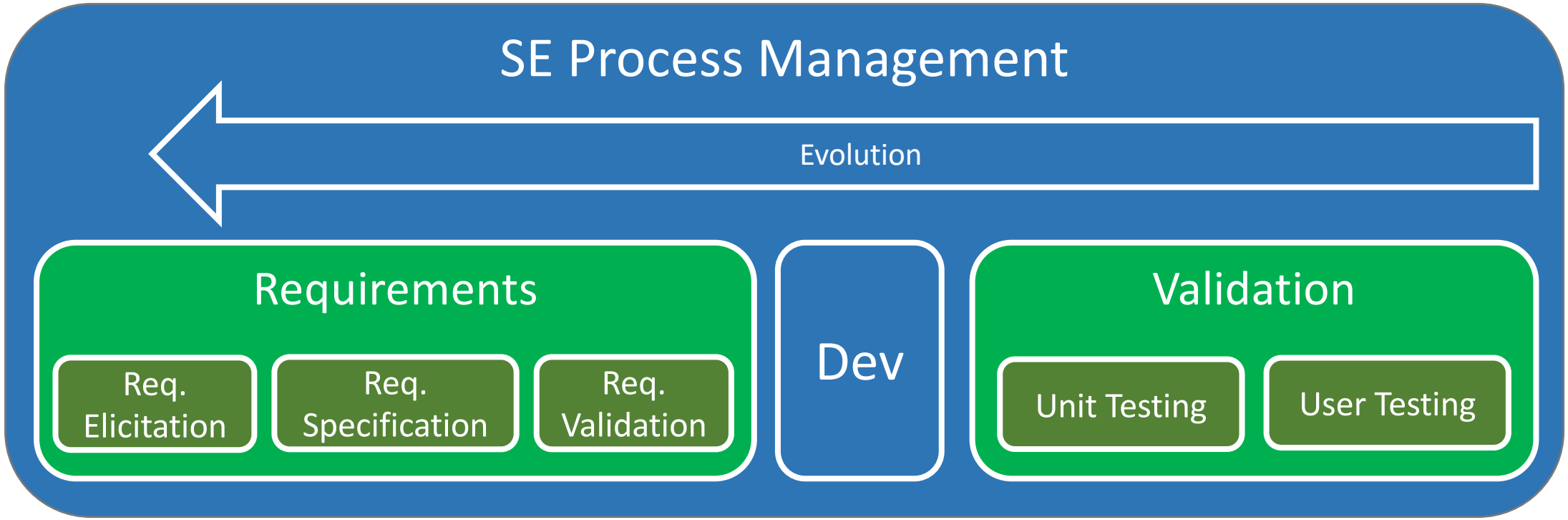
Software Engineering COMP1035

Revision

Lecture 01 - 18



Revision

- Requirements: Bryan
- Validation: Tianxiang

Part 1: Requirements

BRYAN

Back to Start

- Why do we have Software Engineering methodologies?
  - The Software Crisis – software was bad and worse, behind schedule.

- Why is having a Software Engineering approach important?
  - Cost and Quality!
  - Systematic approach, appropriate tools

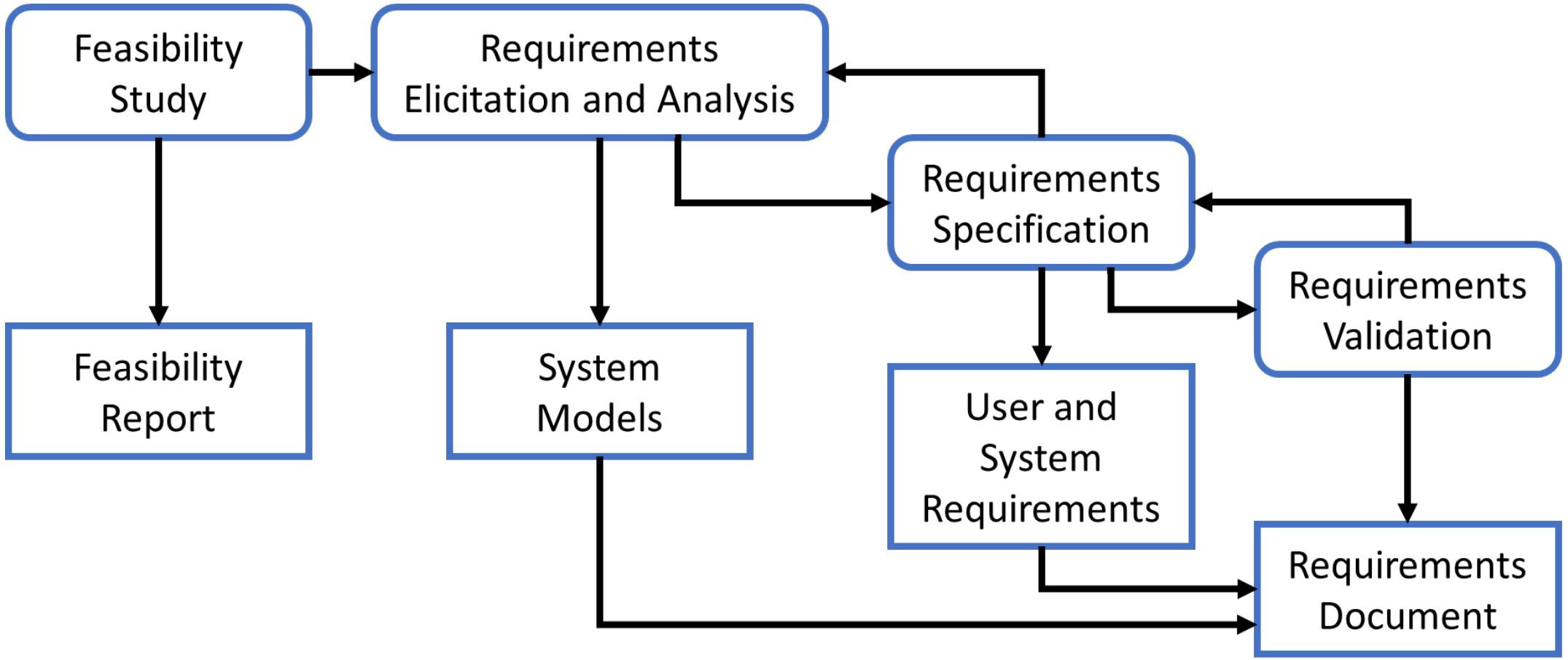
- What are the four main aspects of Software Engineering?
  - Requirements & Specifications, Development, Validation and Evolution.

Back to Start

- What SE methodologies exist?
  - Waterfall – needs ”perfect requirements”.
  - V-Model – not account for revision or refactoring.
  - Iterative Model – too much doing, not enough planning.
  - Agile – Challenge for large projects even for experienced Agile manager.

- What are problems cheapest and most expensive to fix?
  - Requirement Time vs Maintenance.

Requirements Engineering



Requirements Engineering

- What's the difference?
  - User Requirements: Software must provide (stakeholder).
  - System Requirements: What software must do to meet user requirements.

- Three major **phases**:
  - Requirements Elicitation & Analysis
  - Requirements Specifications
  - Requirements Validation

Requirements Engineering

- What's the difference between Functional & Non-Functional Requirements?

	Functional requirements	Nonfunctional requirements
Objective	Describe what the product does	Describe how the product works
End result	Define product features	Define product properties
Focus	Focus on user requirements	Focus on user expectations
Essentiality	They are mandatory	They are not mandatory but desirable
Origin type	Usually defined by the user	Usually defined by developers or other tech experts
Testing	Component, API, UI testing, etc. Tested before nonfunctional testing	Performance, usability, security testing, etc. Tested after functional testing
Types	Authentication, authorization levels, data processing, reporting, etc.	Usability, reliability, scalability, performance, etc.

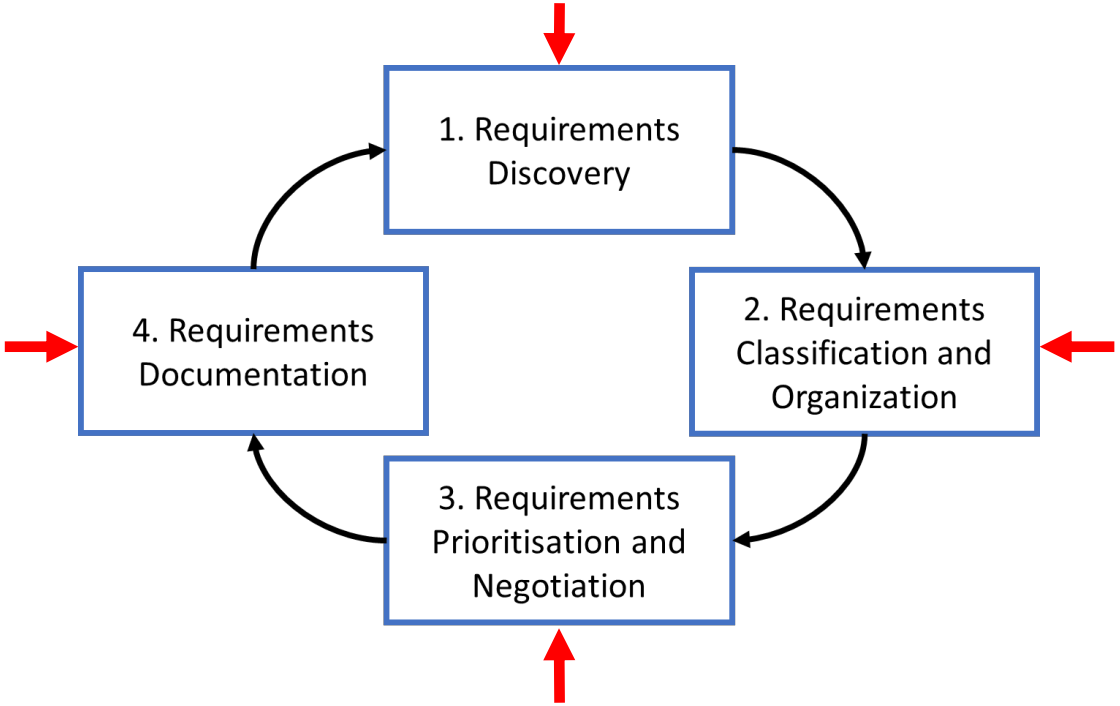
Requirements Elicitation

- Requirements elicitation involves the **process of investigating and identifying the needs of a system from users, customers, and other stakeholders**.
  - This practice is also known as "requirement gathering".

- Stands out as **one of the most challenging, error-prone, and communication-intensive** aspects of software development.

Requirements Elicitation and Analysis Process

Involve **identifying requirements from stakeholders** and collating documentation during this process.

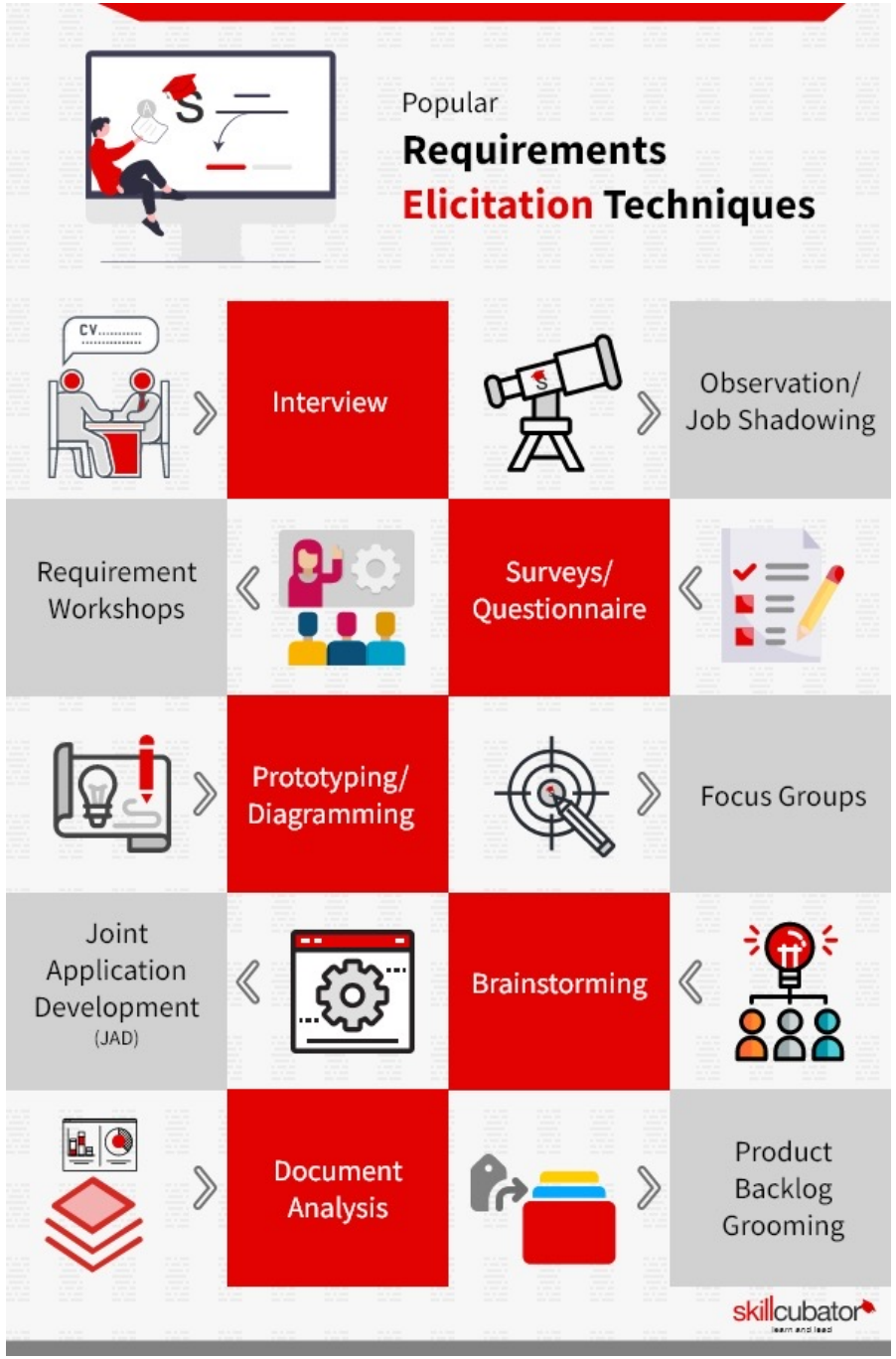


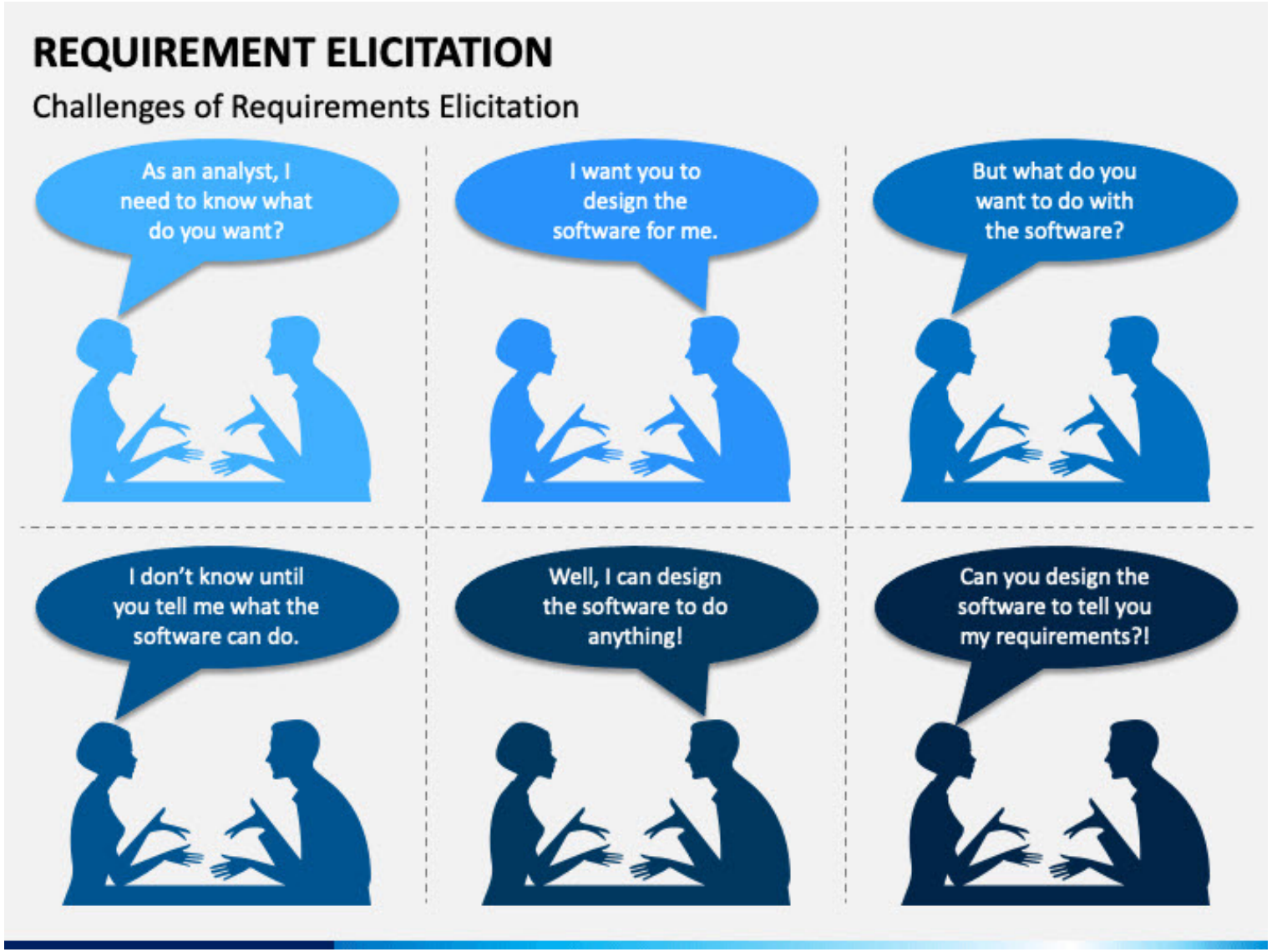
Initial draft of the software **requirements document** may be designed during this stage, or the requirements may be informally maintained on platforms like whiteboards, wikis, or shared spaces.

The initially **unorganized set of requirements is grouped** based on their relevance and organized into coherent categories.

Focuses on **prioritizing requirements and resolving conflicts** among multiple stakeholders – to address disparities and reach consensus on compromise requirements.

How to Engage?





Requirements Specifications

- How can you model requirements? Users, tasks, etc.?
  - PPT, UML, Codes?

- What can you use UML for?
  - Shows complex decision process using different types of UML?
    - Use Case Diagram / Persona / Scenarios / User Stories
    - Context Diagram
    - Activity Diagram
    - Sequence Diagram
    - Class Diagram
    - State Diagram.

How to Choose Different Diagrams?

In the initial steps of requirements:

- What is or is not part of the system? -- **Context Models**: *Show how a system that is being modeled is positioned in an environment with other systems and processes. Help to define the **boundaries** of the system to be developed*.
- How are the systems used? -- **Activity Diagrams**: *Used to model the processing of data, where each activity represents one process step*.

Next, how to model the interactions between users, internal and external systems?

- **Use Case Diagrams** are drafts: *Describe interactions between a system and external actors*.
- **Sequence Diagrams** are used to explain the specific use case: *Add more information to these interactions by showing interactions between system objects*.

Most importantly, how to design the system architecture?

- Static view of the software system can be modeled as **Class diagrams**: *Define the static structure of classes in a system and their associations*.
- Dynamic view can be modeled as **Sequence Diagrams** if the system are driven by data.
- If the systems are driven by events, it can be modeled as **State Diagrams**: *Used to model a system's behavior in respond to internal or external events*.

Requirements Specifications

- Is the **process of writing down the user and system requirements** in a requirements document.

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. UML (unified modeling language) use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want, and they are reluctant to accept it as a system contract. (I discuss this approach, in Chapter 10, which covers system dependability.)

Software Requirements Document

- Translation from anecdotal user wishes to a formal document.
- Usually, **a big set of nested lists, with unique IDs**.
- Diagrams produced go with them (and cross reference those IDs).
- Lists should be **categorised** – e.g., importance, risk etc.
- Usually define the “acceptance testing” at the end of the project.

Structure of a Requirements Document

Sample SRS document based on an IEEE standard for SRS (IEEE 1998).

Chapter	Description
Preface	This defines the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This describes the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This defines the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter presents a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These provide detailed, specific information that is related to the application being developed—for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Requirements Validation

- Process of **checking that requirements define the system that the customer really wants**, both internal and external.
- There is a formal review process you can go through to review the requirements:
  - **Validity** checks (are the areas of functionality identified as necessary?).
  - **Consistency** checks (do requirements conflict with one another?).
  - **Completeness** checks (does it specify a coherent system or only parts of it?).
  - **Realism** checks (can requirements actually be implemented?).
  - **Verifiability** checks (can requirements be tested?).
- Resolve requirements conflict!
- May lead to requirements change and management:
  - Formal **process for making change proposals** to system requirements.

Prototyping

- Why do we create prototypes?
  - Specifications are hard to understand – bad for conveying overall idea.
  - Prototype is a way of show how all specifications work together – easy to show people and for discussion.

- Can be used in a software development process to help anticipate changes:
  - **Requirement engineering process** – help with elicitation and validation of system requirements.
  - **System design process** - used to explore software solutions and in the development of a user interface for the system.

Prototyping

Low Fidelity

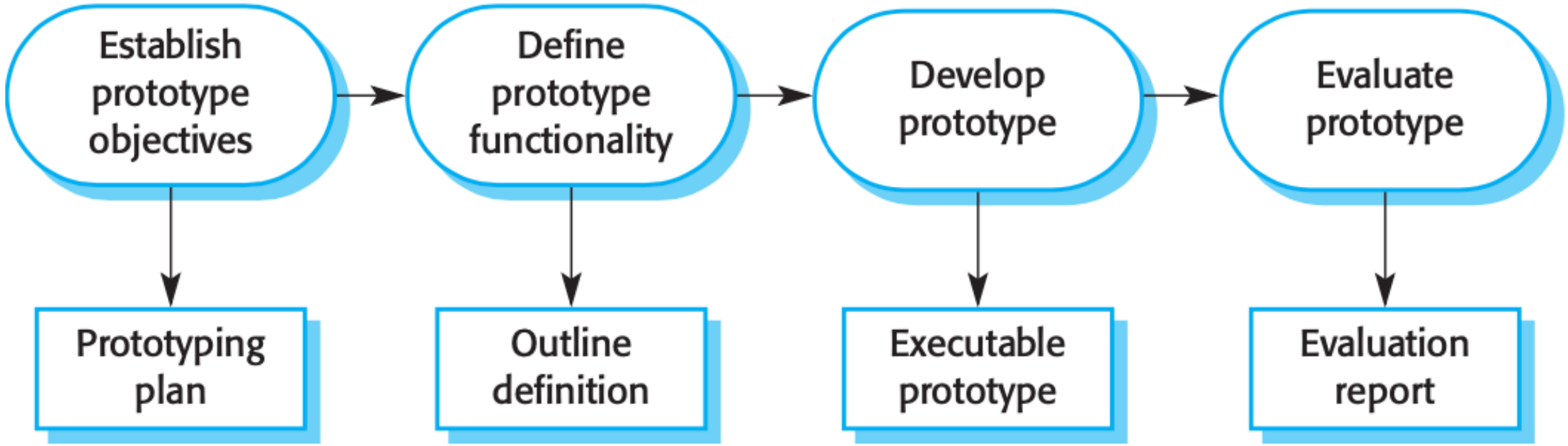
- E.g., sketches/'Paper' prototypes.
- Focused on underlying ideas.
- Key functionality, content etc.
- Produced quickly.
- **Thrown away**.
- Generates many possible ideas.
- **Help client acceptance**.

High Fidelity

- Built in software for automation.
- Similar style to final product.
- Accurate detail is important.
- Finalize chosen ideas.
- Still thrown away.
- Used in realistic studies.
- Helps client acceptance.

Defining Prototype Goals

Fig: Prototype Development



Wireframe vs Mockup vs Code Prototype

	Wireframe	Mockup	Prototype
What	A **quick sketch** to convey the high-level concept of new product functionality	A realistic **visual design** that resembles what the new product functionality will look like	**Interactive simulation** of new product functionality
Purpose	To gain consensus and collect internal feedback on how new functionality will work	To facilitate more detailed critiques of visual elements and functionality so changes can be made	To collect feedback by user testing the real experience
Design fidelity	Low	Middle	High
Included elements	The format and structure of content	Additional visual elements like logos, colors, and icons	Final interactive elements and navigation
Time invested	Low	Medium	High
Creator	PM or UX Designer	UX Designer	UX Designer

Prototyping Risks

1. **Investing too much time/energy on high-fidelity prototypes**.
   - When a low fidelity one would let you test the point.

2. Adhoc prototyping code is **re-used** in the real system.
   - But the code wasn't produced to a professional standard.

3. **Prototying is used instead of, rather than alongside, documentation**.
   - Which may then be insufficient for software maintenance phase.

4. Prototypes might be approved by the **wrong stakeholders**.
   - E.g., managers rather than the end users.

SET/SEM

- Please spend 5 minutes to complete the SET/SEM.
  - https://bluecastle-cn-surveys.nottingham.ac.uk

Part 2: Validation

TIANXIANG

Test-Driven Development

- Testing Methods
  - White Box Testing, Black Box Testing
- Testing Processes
  - Automation, manual
- Test Driven Development

The Three Rules of TDD

- You are not allowed to write any production code unless it is to make a failing unit test pass. (**Create a unit test that fails.**)
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures. (**Write the unit tests that are complied and sufficiently good enough.**)
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test. (**Write the production codes that pass the unit tests.**)

Testing Metrics

Code Coverage - to verify the extent to which the code has been executed.

Test Coverage – to monitor the number of tests that have been executed.

Code Coverage (Statement Coverage)

- Ensure that each code statement is executed once.

Source code

```
Prints (int a, int b) {                          ------------  Prin
tsum is a function
    int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }                                            -----------   End o
f the source code
```

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Senario1 : a=3,b=9

Number of executed statements 5.

Number of total statements 7.

Statement coverage = 5/7.

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Senario2 : a=-3,b=-9

Number of executed statements 6.

Number of total statements 7.

Statement coverage = 6/7.

Senario1 & Senario2

Statement coverage = 7/7=100%.

Code Coverage (Condition Coverage)

- Ensure that each Boolean sub-expression is executed once.

Source code

```
Prints (int a, int b) {                          ------------  Prin
tsum is a function
    int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }                                            -----------   End o
f the source code
```

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Senario1 : a=3,b=9

The first condition result>0 is fulfilled.

Condition coverage = 1/2.

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

Senario2 : a=-3,b=-9

The second condition result<0 is fulfilled.

Condition coverage = 1/2.

Senario1 & Senario2

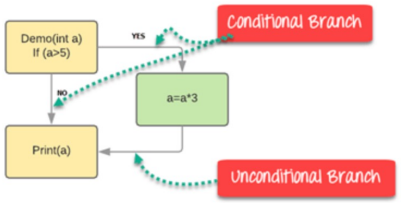
Condition coverage = 2/2=100%.

Code Coverage (Path Coverage)

- Ensure that each branch of each control structure is executed once.

Source code

```
Demo(int a) {
     If (a> 5)
        a=a*3
     Print (a)
    }
```



Senario1 : a=6

The condition (a>5) is fulfilled.

path coverage = 2/3=67%.

Senario2 : a=3.

The condition (a>5) is not met.

path coverage = 1/3=33%.

Senario1 & Senario2

Path coverage = 3/3=100%.

Testing Coverage

- Simple case:
  - If there are 100 test cases and 30 of those are executed, test coverage = 30/100

- Associated with requirement:
  - If there are 10 requirements and 100 tests created – when these 100 tests target all of the 10 requirements and don’t leave out any, test coverage = 10/10
  - When only 80 of the created tests are executed and target only 6 of the requirements, test coverage = 6/10
  - When only 90 tests relating to 8 requirements are executed and the rest of them are not, test coverage = 8/10

Comparisons of Different Testing

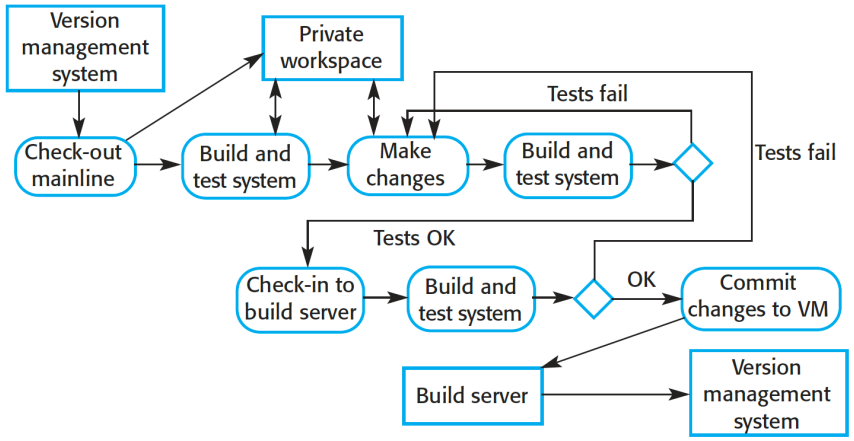
	Unit Testing	**(Subsystems) Integration Testing**	**(Full System) Release Testing**	**(Customer) Acceptance Testing**
What to test	Individual pieces (e.g., classes)	Combinations of pieces (subsystems)	Full system	Full system
Input	Functional specification of unit (unit test case)	Functional specification of subsystem	Full functional and non-functional specifications	User requirements
Output	Pass/Fail	Bug reports,	1) Validation reports 2) Sign off for acceptance tests	1) Acceptance reports 2) Sign off for end of contract
By who	Developer	Development team	QA team (Testing team)	QA team (Testing team) and customer
Frequency	Many per day	Preiodically e.g., end of a sprint	Prior to showing client	Prior to use
Difficulty	Easy	Difficult	Difficult	Usually, Difficult
Speed	Very fast (few seconds or less)	Relatively slow (few minutes or more)	Slow	Usually, Slow

Configuration Management

- Configuration Management (CM): track and control changes in the software.
  1. **Version Control**: keep track of the multiple versions of system components and ensure that changes made to components by different developers do not interfere with each other.
  2. **System Building**: assemble program components, data, and libraries, then compile and link these to create an **executable system**.
  3. **Change Management**: keep track of requests for changes to delivered software from customers and developers, working out the costs and impact, and deciding if and when the changes should be implemented.
  4. **Release Management**: Prepare software for external release and keep track of the system versions that have been released for customer use.

Continuous Integration: Workflow

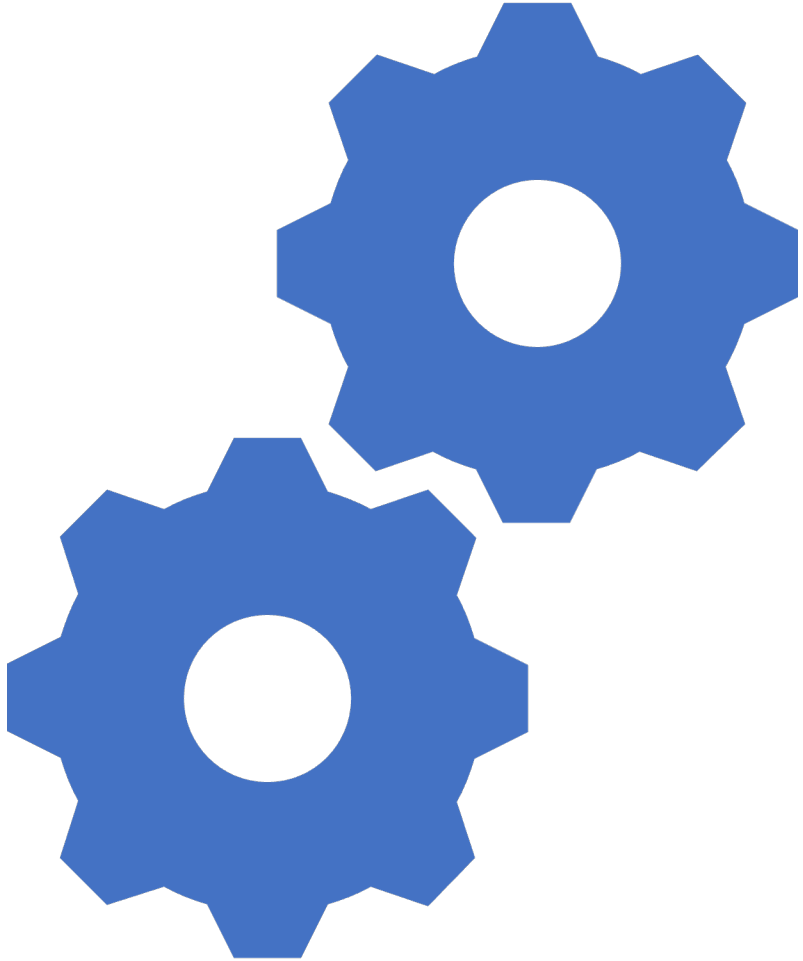
Figure 25.12 Continuous integration



- **Check out the mainline system** from the version management system into the developer's private workspace.
- **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. They are responsible for repairing the problem.
- **Make the changes** to the system components.
- **Build the system** in the **private workspace** and **re-run system tests**. If the tests fail, continue editing.
- Once the system has passed its tests, **check it into the build system** but **do not commit it** as a new system baseline.
- **Build the system** on the **build server** and run the tests. 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.
- 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.

Maintenance

- Maintenance involves:
  - **Correcting** errors which were not discovered in earlier stage of the life cycle
  - **Improving** the implementation of system units
  - **Enhancing** the system's services as new requirements are discovered
- Most of the lifetime of software engineering is the maintenance phase.



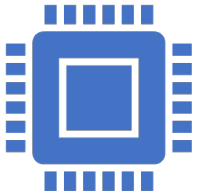
THREE Types of Change

Fault Repairs

to fix coding errors

usually cheap to fix

do not involve much redesign

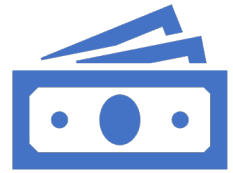


Environmental Adaption

e.g., updates for new OS

bit more expensive to fix (many changes)

but do not involve much redesign



Functionality Addition

to meet business changes

much more expensive

often involves redesign

Process/Project Management

- Deliver good quality product
- Use good reliable methods
- Reducing as much risk as possible
- Plan / budget / manage a project based on these

Quality Assurance

- Quality Assurance is a lot more than release/acceptance testing
- Quality Assurance - is about planning for high quality goals
- Quality Assurance - is about everyone aiming for high quality
- Quality Assurance - is about inspecting for quality at each stage
- Quality Assurance - is about taking / learning from metrics of software quality - and process quality

Agile Methods

1. Our highest priority is to satisfy the customer through **early and continuous delivery** of valuable software.
2. Welcome changing requirements, even late in development. Agile processes **harness change for the customer's competitive advantage**.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a **preference to the shorter timescale**.
4. **Businessmen and developers must work together daily** throughout the project.
5. Build projects around motivated individuals. Give them the **environment and support they need and** trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.

Agile Methods

7. **Working software is the primary measure of progress**.
8. Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. **Continuous attention** to technical excellence and good design enhances agility.
10. **Simplicity** — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs **emerge from self-organizing** teams.
12. At regular intervals, **the team reflects on how to become more effective**, then tunes and adjusts its behavior accordingly.

Scrum vs Kanban vs XP

- All three framework are designed to uphold the Agile Principles
  - they just have different focuses
- Scrum is about optimising by time and delivery
- Kanban is about speed of delivery
- XP is about team effectiveness

Agile vs Traditional – Comparison

Traditional Methods	Agile Methods
Waterfall, V-Model	Spiral, Iterative Models
Requirements Tables	User Stories, Personas
Detailed UML/Specs	Rapid Prototyping
Integration Testing Phase	TDD, Paired Coding
Formal User Testing	Frequent Client Interaction
Separate Skill Teams	Integrated skills teams

Waterfall Pros

- Forward & backward **planning and implementation is easy**.
- The waterfall model is simple to use and easy to understand. It does not require any special training for project managers or employees. It has an **easy learning curve**.
- Being rigid in nature, it is **easy to manage** the waterfall cycle. Each phase has fixed deliverables and a review process.
- **Less complexity** as the phases does not overlap.
- Works **well for small projects** where we have fixed and crystal-clear requirements.
- Processes and results are **well-documented**.
- It is **easy to measure the progress** as the start and endpoints of each phase are predetermined.
- There are **no financial surprises**. Once the requirements are fixed, the final cost can be calculated before starting the development.

Waterfall Cons

- As all the requirements must be clearly known before starting the development, it **delays the project**.
- **Requires extensive research** into the user needs.
- Low flexibility makes it **difficult to accommodate any such changes**, especially when the product needs to be re-engineered to a large extent.
- **Slow delivery times**. The customer is not able to see the product until it is fully completed.
- The **client is not informed** well about the health of the project.
- **High risk and uncertainty** are involved in the waterfall model as there is too much room for issues to remain unnoticed until the project comes near to completion.
- It is difficult to **measure the progress** within each phase.

Agile Pros

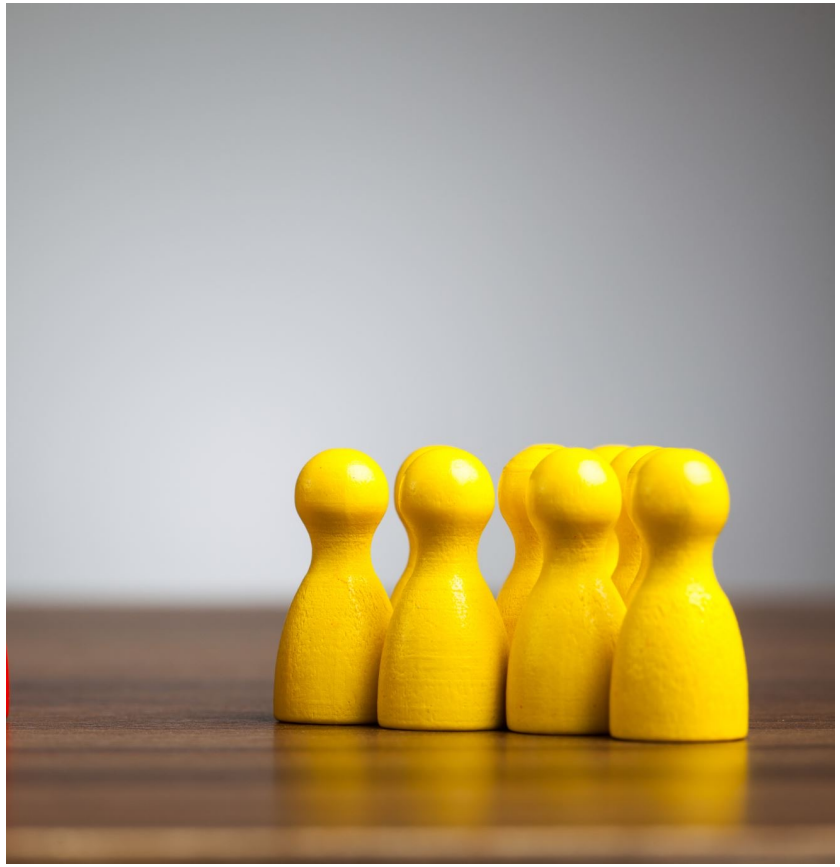
- **Great adaptability.** Agile is very flexible in dealing with the changes in customer needs and priorities.
- Allows to **constantly refine and re-prioritize the overall product backlog.**
- Agile methodology offers a great degree of **stakeholder engagement**.
- Fixed schedule sprints of one to four weeks allow for **early and predictable delivery**.
- Valuable **customer feedback** is gained **early** in the project and changes to the product can be made as required.
- By producing **frequent builds**, any misalignment with the customer requirements can also be detected and fixed early.
- Agile **promotes teamwork**.
- In an agile project, there is room for **continuous improvement**.

Agile Cons

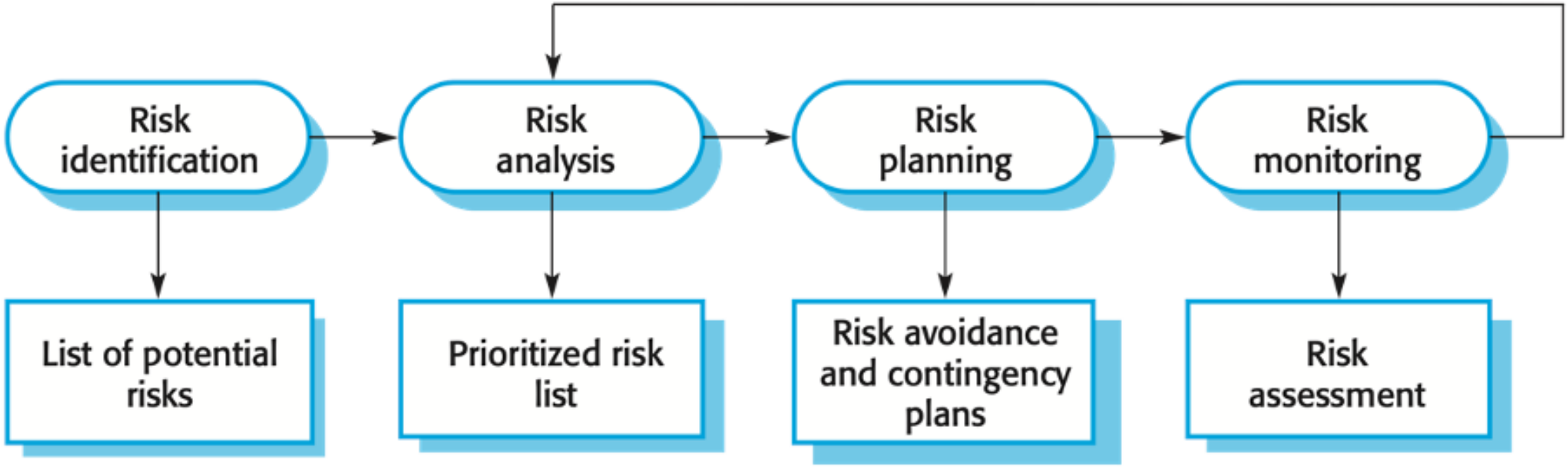
- It is often seen that Agile teams have a **tendency to neglect documentation**.
- If the initial architecture and design are weak, then **frequent refactoring** is required.
- When compared to the waterfall, Agile is **difficult to practice**. The team members must be well versed in Agile concepts.
- Due to time-boxed delivery and frequent re-prioritization, there are chances for a few features to not get delivered in the allocated timeline. This can lead to **additional sprints and additional costs**.

Risks, Opportunities, and Problems

- A **risk** is the probability of unwanted consequences of an event and decision
- An **opportunity** is the probability of exceeding expectations. A risk is the probability of failing to meet expectations
- A **risk** is **not** a problem. A risk is a potential problem over which we have some choices.
- Causes of Risks:
  - Uncertainty in time
  - Uncertainty in control
  - Uncertainty in information
- There is no software project without risks…



Risk Management Workflow



Risk Planning

- You should choose a strategy for the risks
- **Avoidance strategies** - actions taken to reduce the risk happening
- **Minimization strategies** - reducing the impact if it happens
- **Contingency plans** - what you will do/change if it happens
- Avoidance is always the best strategy, if possible.
  - it's often worth taking pre-emptive actions to avoid risks.

Project Planning

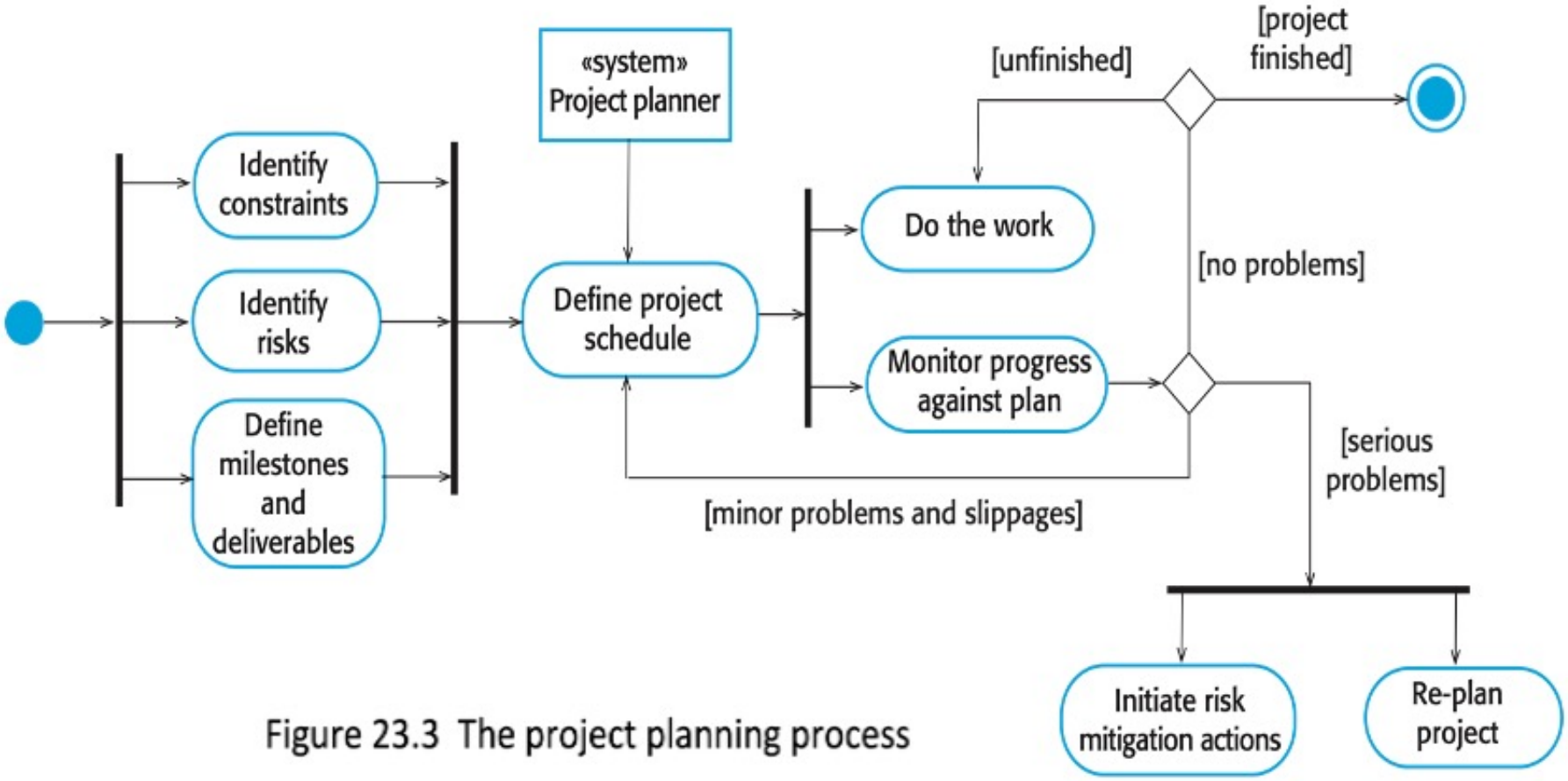
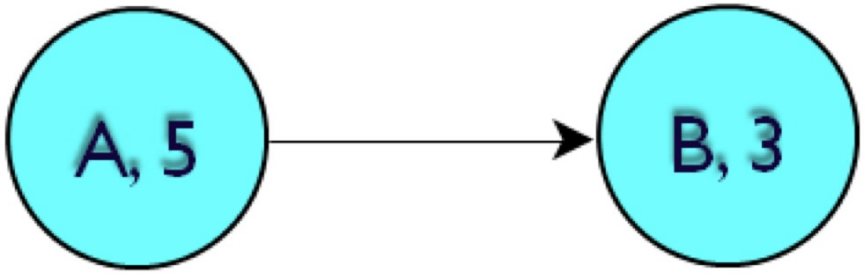


Figure 23.3 The project planning process

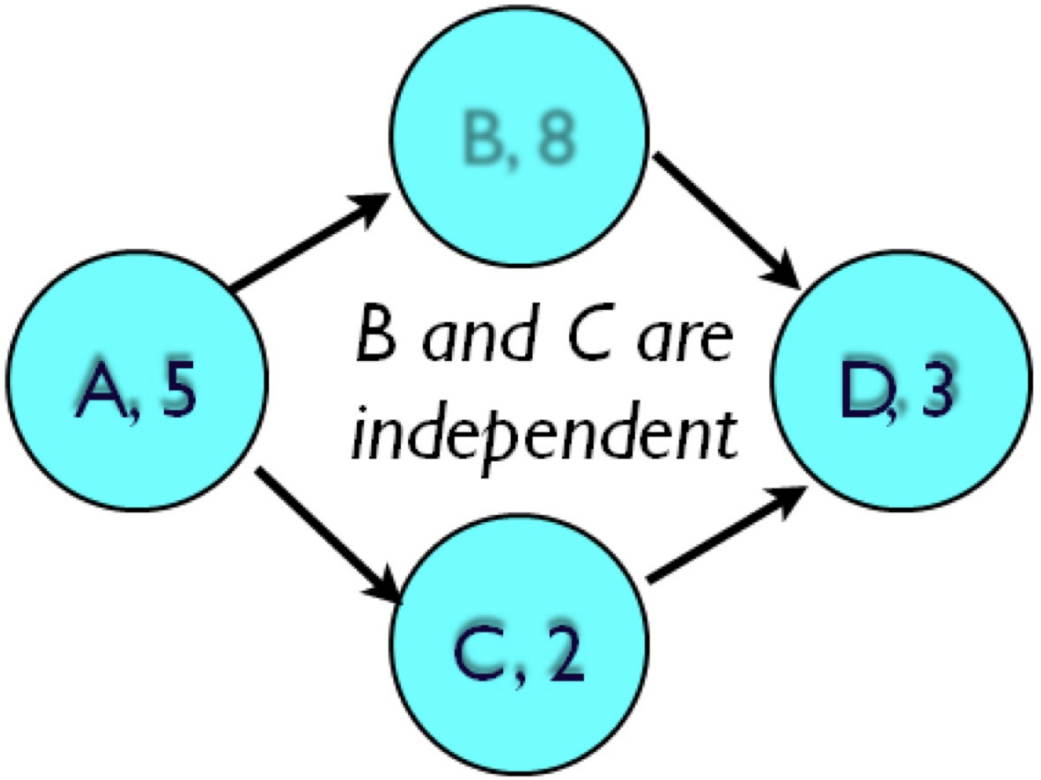
Planning Diagrams

- PERT (1958) - tasks and dependencies
- Critical Path Method (1960) - risk analysis
- Gantt Chart (1910) - adding time to the tasks/dependencies
- Staff Allocation charts - who can do the tasks/dependencies

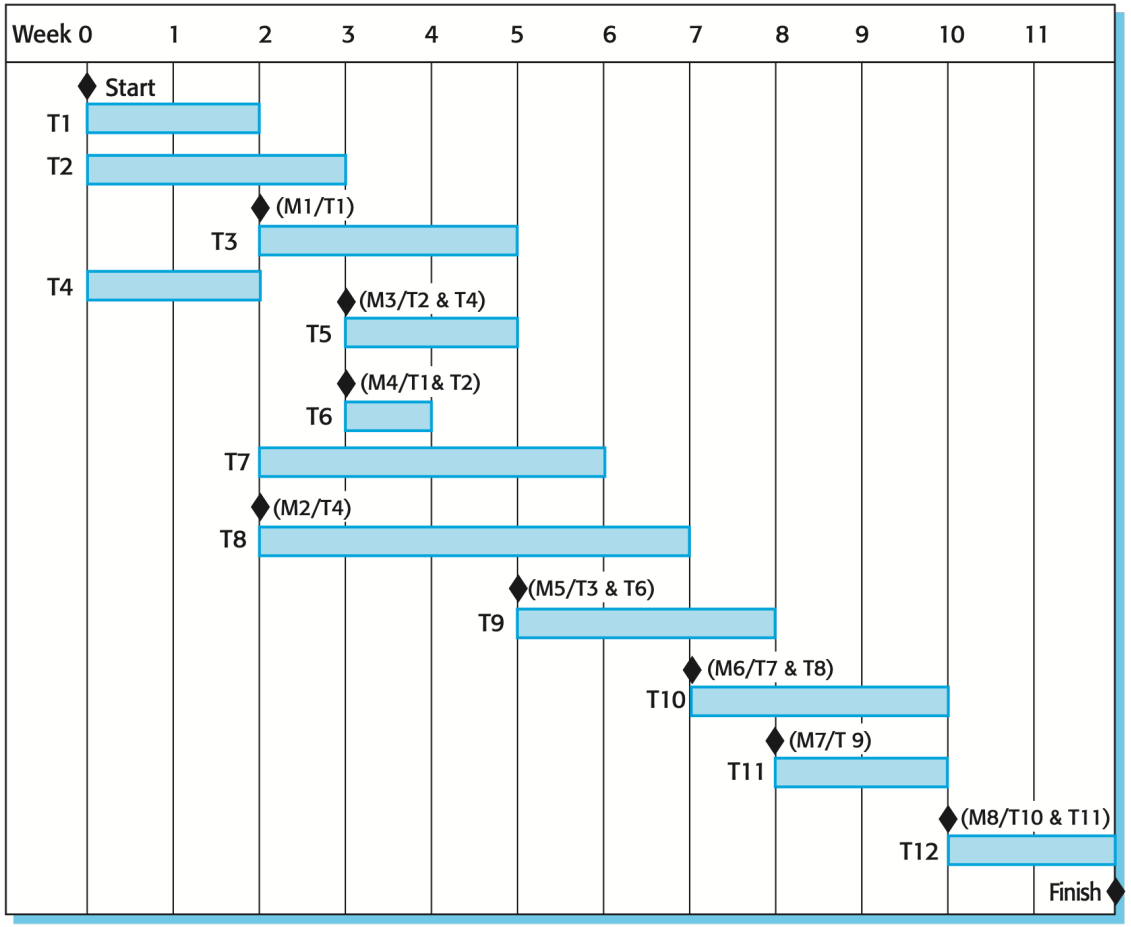
PERT Charts



Can't do B until A



Gantt Charts



Staff-Allocation Charts

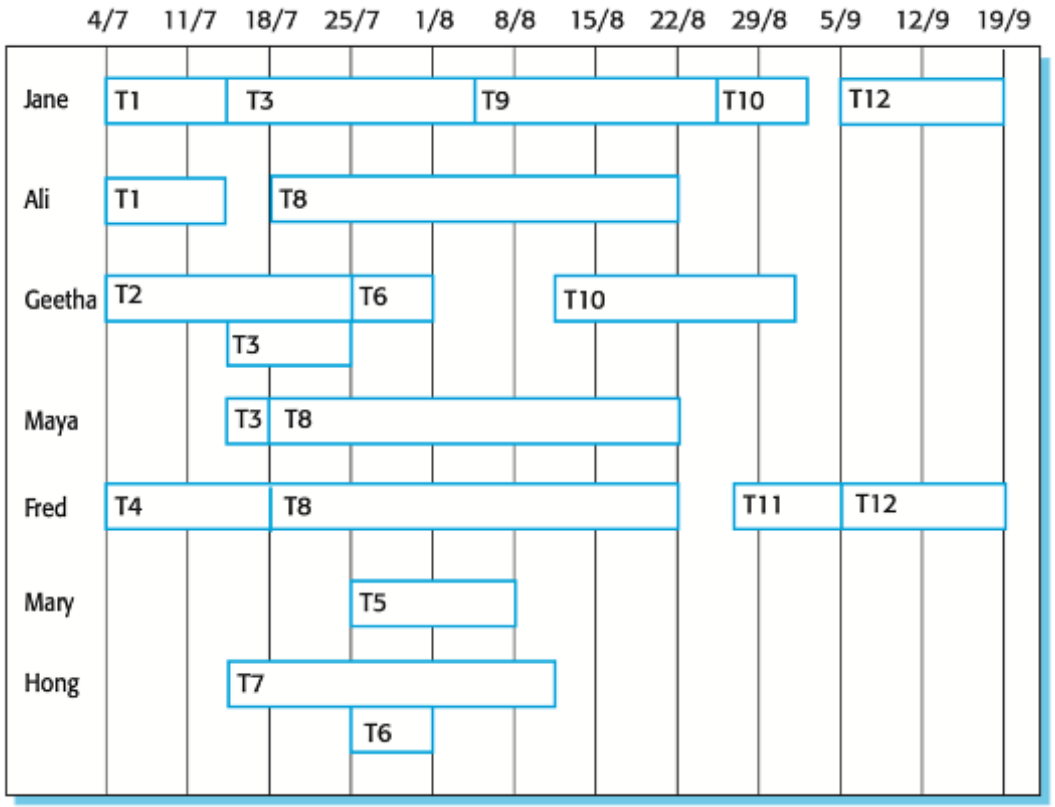


Figure 23.7 Staff allocation chart

Cost Estimation

- Experience-based techniques
  - based on experience of prior similar projects
  - based on managers familiarity with work involved

- Algorithmic Cost modelling
  - mathematical calculations based on
    - estimated amount of effort
    - experience-based constants
    - e.g., COCOMO II
    - although often as much as 25%~400% wrong

Reference – Textbook

- Chapter 1
- Chapter 2 – 2.1, 2.2
- Chapter 3
- Chapter 4 – 4.1, 4.2, 4.3, 4.4
- Chapter 5 – 5.1, 5.2, 5.3, 5.4
- Chapter 7 – 7.1, 7.2, 7.3
- Chapter 8
- Chapter 9
- Chapter 22 – 22.1
- Chapter 23 – 23.1, 23.2, 23.3


THANK
YOU