Software Engineering and Services (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

January 16, 2023

Overview

- Software Engineering course covers the concepts of software construction with basic understanding of what is software engineering and difference between traditional process model and agile models.
- How agile methods have made software engineering to reduce the time to market of a software product.
- Course also deal with requirement engineering, architectural design, user interface design and also deals with some aspects how a software product is tested.
- It also covers some aspects of project management which deals which how a project is managed and how resources are used efficiently and how profit is gained.

Syllabus

- Software and software engineering: the nature of software, the unique nature of web apps, software engineering, the software process.
- The software process: process models, a generic process model, prescriptive process models: the waterfall model, incremental process models , evolutionary process models.
- Agile development: what is agility? agility and the cost of change, what is an agile process, extreme programming, other agile process models: Adaptive Software Development (ASD), scrum, Dynamic Systems Development Method (DSDM).

Syllabus

- Understanding Requirements: requirements engineering, establishing the groundwork, eliciting requirements, developing use cases, building the requirements model, negotiating requirements, validating requirements.
- Requirements modeling: scenarios, information, and analysis classes.
- Requirements modeling: flow, behavior, patterns, and webapps: requirements modeling strategies, flow-oriented Modeling, Creating a Behavioral Model, Patterns for Requirements Modeling, Requirements Modeling for WebApps.

Syllabus

- Design Concepts: design within the context of software engineering, the design process, design concepts, the design model.
- Architectural Design: software architecture, architectural genres, architectural styles, architectural design, assessing alternative architectural designs
- Component-Level Design: what is a component, designing class-based components, conducting component-level design, component-level design for webapps, designing traditional components, component-based development.
- User Interface Design: the golden rules, user interface analysis and design

Syllabus

- Software Testing Strategies: a strategic approach to software testing, strategic issues, test strategies for conventional software, test strategies for object-oriented software, test strategies for webapps, validation testing, system testing, the art of debugging.

- Testing Conventional Applications: software testing fundamentals, internal and external views of testing, white-box testing, basis path testing, control structure testing, black-box testing.

Syllabus

- Project Management Concepts: the management spectrum, people, product, process, project.
- Process And Project Metrics: metrics in the process & project domains, software measurement, metrics for quality, integrating metrics within the software process, metrics for small organizations, establishing a software metrics program.
- Estimation For Software Projects: observations on estimation, project planning process, software scope and reliability, resources, project estimation, decomposition techniques, empirical estimation models.

Details

- Course Code: DS-203
- Course Name: Software Engineering and Services
- (L-T-P): (3-0-2) 4

- Evaluation Criteria:
  - Project = 25%
  - Quiz/Viva = 10%
  - Mid Sem Exam = 25%
  - End Sem Exam = 40%

Reference Books

- Software Engineering - A Practitioners approach, Roger .G. Pressman, 7th Edition Tata McGrawhill
- Software Engineering: Theory and Practice by Shari Lawrence Pfleeger & Joanne M. Atlee, 4th edition, Pearson Education.
- Software Engineering, Ian Sommerville, 10th ed.Pearson 2018.
- An integral approach to software engineering, Pankaj Jalote, 3rd ed, Narosa Pblishing house

Software Engineering and Services (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

January 16, 2023

THE NATURE OF SOFTWARE

- Today, software takes on a dual role. It is a **product**, and at the same time, the vehicle for **delivering a product**.
- As a **product**, It delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware.
- Whether it resides within a mobile phone, a hand-held tablet, on the desktop, or within a mainframe computer, software is an **information transformer**—producing, managing, acquiring, modifying, displaying,
- or **transmitting information** that can be as simple as a single bit or as complex as a multimedia presentation derived from data acquired from dozens of independent sources.

THE NATURE OF SOFTWARE

- As the vehicle used to **deliver the product**, software acts as the basis for the **control of the computer** (operating systems), the **communication of information** (networks), and the **creation and control of other programs** (software tools and environments).
- Software delivers the most important product of our time— information. It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context.
- It manages business information to enhance competitiveness; it provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms
- The role of computer software has undergone significant change over the last half-century. Dramatic improvements in hardware performance, profound changes in computing architectures, vast increases in memory and storage capacity, and a wide variety of exotic input and output options have all precipitated more sophisticated and complex computer-

THE NATURE OF SOFTWARE

- Today, a huge software industry has become a dominant factor in the economies of the industrialized world.
- Teams of software specialists, each focusing on one part of the technology required to deliver a complex application, have replaced the lone programmer of an earlier era.
- And yet, the questions that were asked of the lone programmer are the same questions that are asked when modern computer-based systems are built

THE NATURE OF SOFTWARE

- Why does it take so long to get software finished?
- Why are development costs so high?
- Why can't we find all errors before we give the software to our customers?
- Why do we spend so much time and effort maintaining existing programs?
- Why do we continue to have difficulty in measuring progress as software is being developed and maintained?

Defining Software

- A textbook description of software might take the following form:
- Software is: (1) instructions (computer programs) that when executed provide desired features, function, and performance; (2) data structures that enable the programs to adequately manipulate information, and (3) descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.
- Frequently asked questions about software engineering. What is software?
- Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.

Defining Software

- What are the attributes of good software?
- Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
- What is software engineering?
- Software engineering is an engineering discipline that is concerned with all aspects of software production from initial conception to operation and maintenance.

Defining Software

- What are the fundamental software engineering activities?
- Software specification, software development, software validation and software evolution.
- What is the difference between software engineering and computer science?
- Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.

Defining Software

- What is the difference between software engineering and system engineering?
- System engineering is concerned with all aspects of computerbased systems development including hardware, software and process engineering. Software engineering is part of this more general process.
- What are the key challenges facing software engineering?
- Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
- What are the costs of software engineering?
- Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.

Defining Software

- What are the best software engineering techniques and methods?
- While all software projects have to be professionally managed and developed, different techniques are appropriate for different types of system. For example, games should always be developed using a series of prototypes whereas safety critical control systems require a complete and analyzable specification to be developed. There are no methods and techniques that are good for everything.
- What differences has the Internet made to software engineering?
- Not only has the Internet led to the development of massive, highly distributed, service-based systems, it has also supported the creation of an "app" industry for mobile devices which has changed the economics of software.

Software Product

- There are two kinds of software product:
- Generic products These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include apps for mobile devices, software for PCs such as databases, word processors, drawing packages, and project management tools. This kind of software also includes "vertical" applications designed for a specific market such as library information systems, accounting systems, or systems for maintaining dental records.
- Customized (or bespoke) software These are systems that are commissioned by and developed for a particular customer. A software contractor designs and implements the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process, and air traffic control systems.

Software Product

- The critical distinction between these types of software is that, in generic products, the organization that develops the software controls the software specification. This means that if they run into development problems, they can rethink what is to be developed.
- For custom products, the specification is developed and controlled by the organization that is buying the software. The software developers must work to that specification.
- However, the distinction between these system product types is becoming increasingly blurred. More and more systems are now being built with a generic product as a base, which is then adapted to suit the requirements of a customer. Enterprise Resource Planning (ERP) systems, such as systems from SAP and Oracle, are the best examples of this approach.

Software Application Domains

- System software— a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities) processes complex, but determinate, 3 information structures. Other systems applications (e.g., operating system components, drivers, networking software, telecommunications processors) process largely indeterminate data.

- Application software —stand-alone programs that solve a specific business need. Applications in this area process business or technical data in a way that facilitates business operations or management/technical decision making.

Software Application Domains

- Engineering/scientific software —a broad array of "number-crunching programs that range from astronomy to volcanology, from automotive stress analysis to orbital dynamics, and from computer-aided design to molecular biology, from genetic analysis to meteorology.

- Embedded software— resides within a product or system and is used to implement and control features and functions for the end user and for the system itself. Embedded software can perform limited and esoteric functions (e.g., key pad control for a microwave oven) or provide significant function and control capability (e.g., digital functions in an automobile such as fuel control, dashboard displays, and braking systems). making.

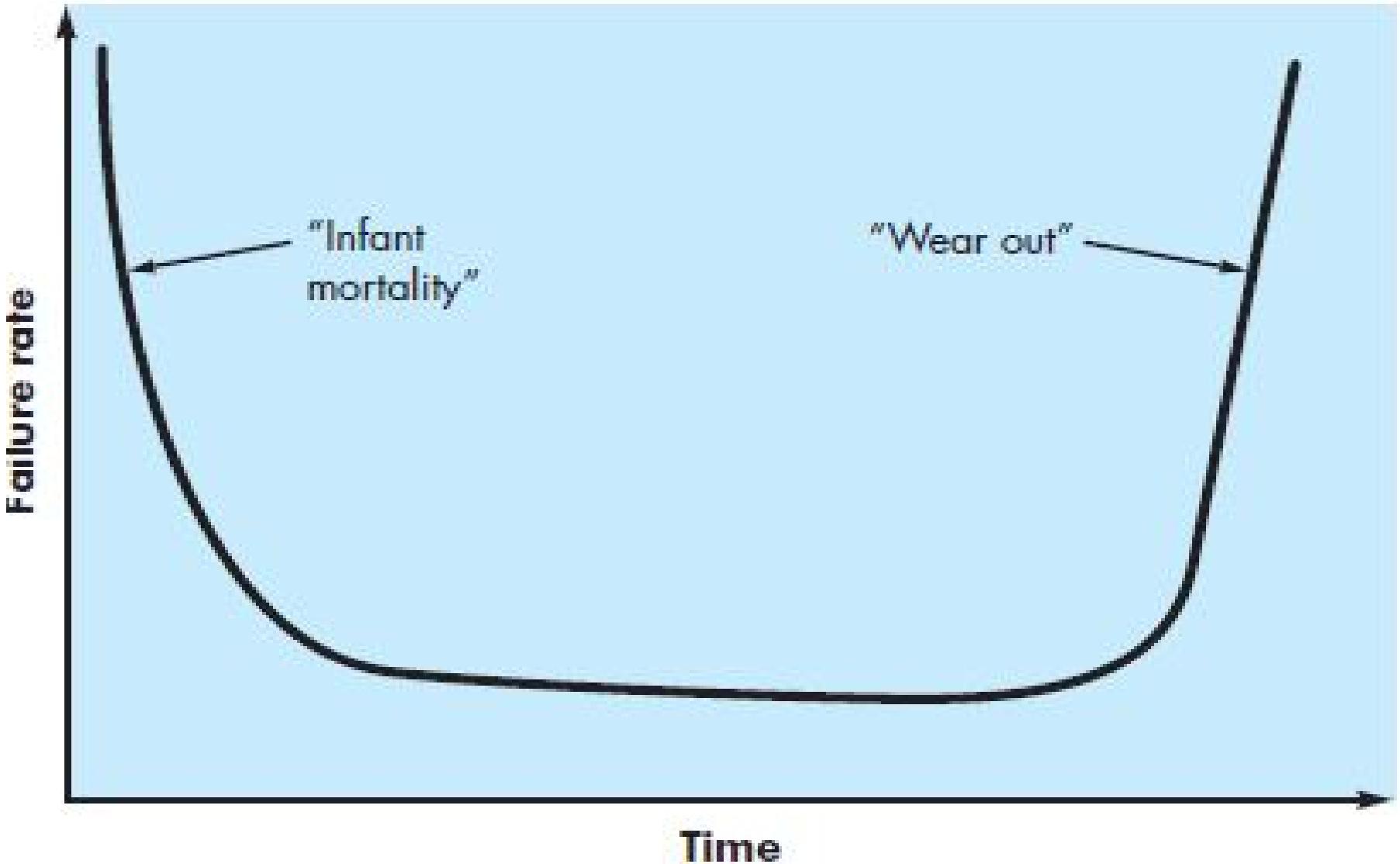
Software Application Domains

- Product-line software —designed to provide a specifi c capability for use by many different customers. Product-line software can focus on a limited and esoteric marketplace (e.g., inventory control products) or address mass consumer.

- Web/Mobile applications —this network-centric software category spans a wide array of applications and encompasses both browser-based apps and software that resides on mobile devices.

- Artificial intelligence software— makes use of nonnumerical algorithms to solve complex problems that are not amenable to computation or straightforward analysis. Applications within this area include robotics, expert systems, pattern recognition (image and voice), artifi cial neural networks, theorem proving, and game playing.

Failure curve for hardware



Failure curve for hardware

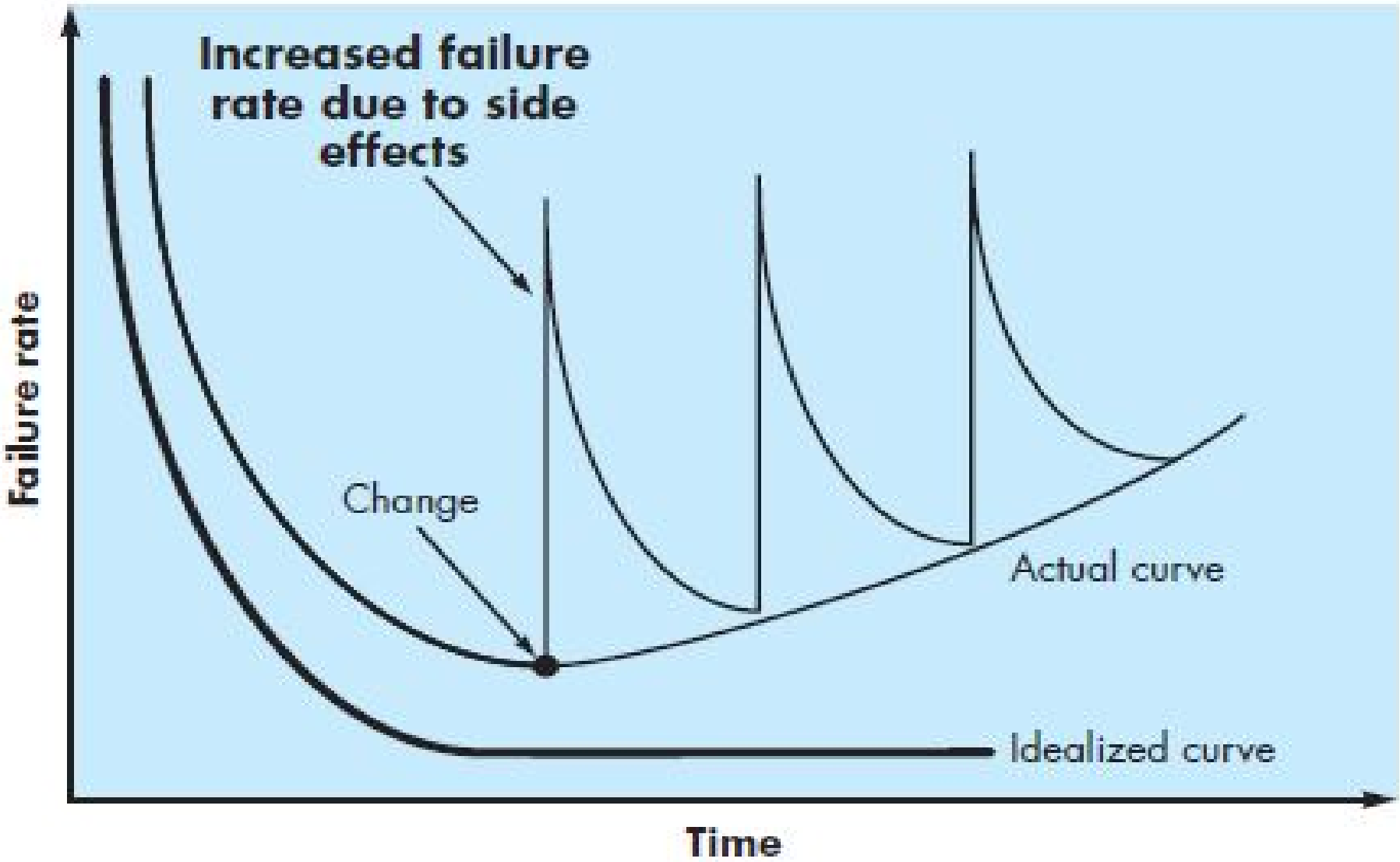
- Figure depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects)

- Defects are corrected and the failure rate drops to a steady-state level (hopefully, quite low) for some period of time.

- As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

- Software is not susceptible to the environmental maladies that cause hardware to wear out. However, the implication is clear—software doesn't wear out. But it does deteriorate!

Failure curves for software



Failure curves for software

- During its life, 2 software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve"
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.
- Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code.
- Therefore, the software maintenance tasks that accommodate requests for change involve considerably more complexity than hardware maintenance.

Software engineering

- Software engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.
- Engineering discipline Engineers make things work. They apply theories, methods, and tools where these are appropriate. However, they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods. Engineers also recognize that they must work within organizational and financial constraints, and they must look for solutions within these constraints.
- All aspects of software production Software engineering is not just concerned with the technical processes of software development. It also includes activities such as software project management and the development of tools, methods, and theories to support software development.

Software engineering

- Software engineering is important for two reasons:
- More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.
- It is usually cheaper, in the long run, to use software engineering methods and techniques for professional software systems rather than just write programs as a personal programming project. Failure to use software engineering method leads to higher costs for testing, quality assurance, and long-term maintenance.

Software engineering

- The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product.
- Four fundamental activities are common to all software processes.
- Software specification, where customers and engineers define the software that is to be produced and the constraints on its operation.

Software engineering

- Software development, where the software is designed and programmed.
- Software validation, where the software is checked to ensure that it is what the customer requires.
- Software evolution, where the software is modified to reflect changing customer and market requirements.

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

January 23, 2023

THE CHANGING NATURE OF SOFTWARE

- Four broad categories of software are evolving to dominate the industry.

- WebApps
- Mobile Applications
- Cloud Computing
- Product Line Software

WebApps

- In the early days of the World Wide Web (circa 1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics.
- As time passed, Web-based systems and applications (we refer to these collectively as WebApps) were born.
- Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications.

Mobile Applications

- The term app has evolved to connote software that has been specifically designed to reside on a mobile platform (e.g., iOS, Android, or Windows Mobile).
- It is important to recognize that there is a subtle distinction between mobile web applications and mobile apps.
- A mobile web application (WebApp) allows a mobile device to gain access to web-based content via a browser that has been specifically designed to accommodate the strengths and weaknesses of the mobile platform.

Mobile Applications

- A mobile app can gain direct access to the hardware characteristics of the device (e.g., accelerometer or GPS location) and then provide the local processing and storage capabilities noted earlier.
- As time passes, the distinction between mobile WebApps and mobile apps will blur as mobile browsers become more sophisticated and gain access to device level hardware and information

Cloud Computing

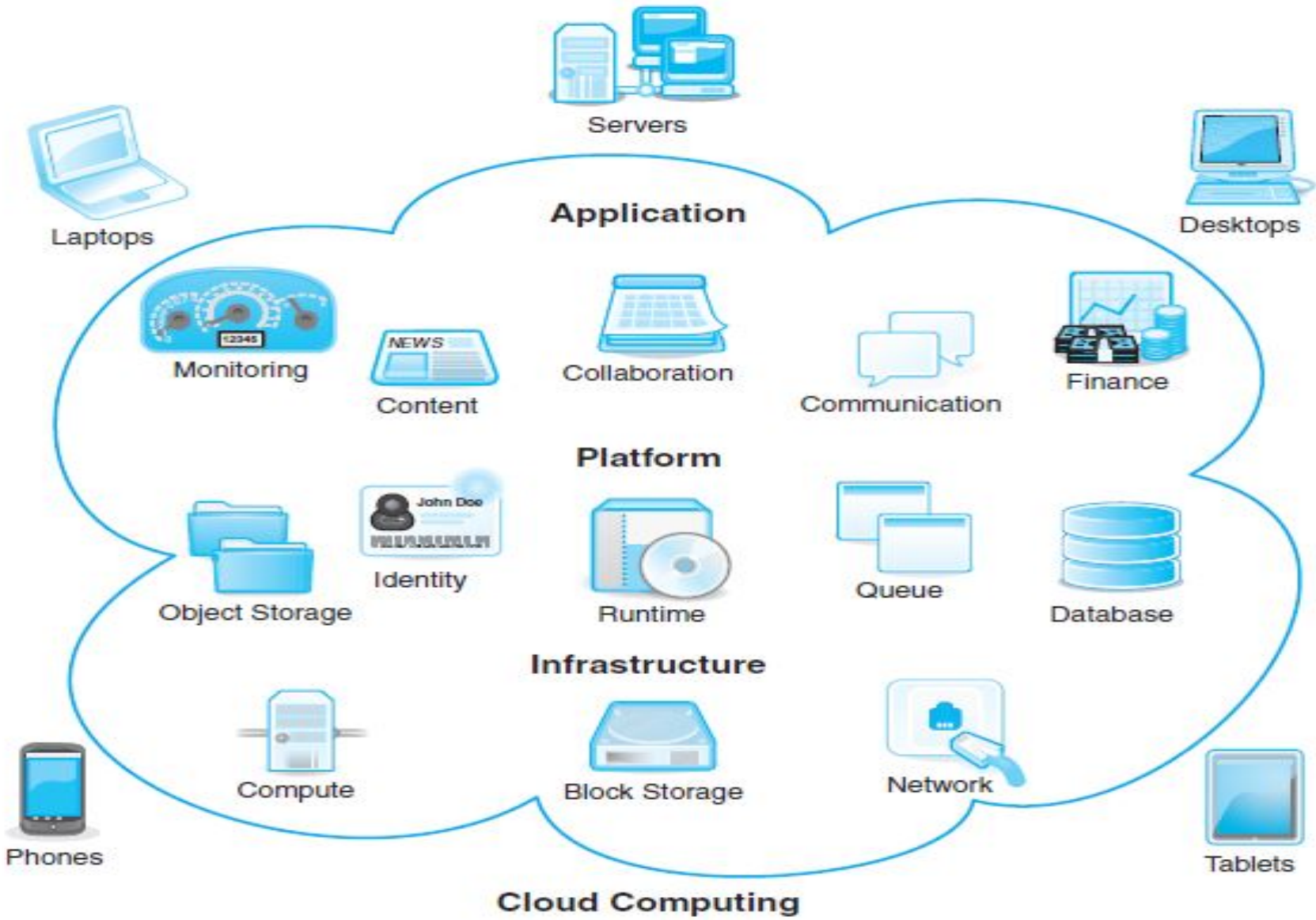


Figure 1. Cloud computing logical architecture

Cloud Computing

- Cloud computing encompasses an infrastructure or "ecosystem" that enables any user, anywhere, to use a computing device to share computing resources on a broad scale.
- Referring to the figure, computing devices reside outside the cloud and have access to a variety of resources within the cloud. These resources encompass applications, platforms, and infrastructure.
- In its simplest form, an external computing device accesses the cloud via a Web browser or analogous software.

Cloud Computing

- The implementation of cloud computing requires the development of an architecture that encompasses front-end and back-end services.
- The front-end includes the client (user) device and the application software (e.g., a browser) that allows the back-end to be accessed.
- The back-end includes servers and related computing resources, data storage systems (e.g., databases), server-resident applications, and administrative servers that use middleware to coordinate and monitor traffic by establishing a set of protocols for access to the cloud and its resident resources.

Product Line Software

- The Software Engineering Institute defines a software product line as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

- A software product line shares a set of assets that include requirements, architecture, design patterns, reusable components, test cases, and other software engineering work products.

SOFTWARE PROCESS STRUCTURE

- A Generic Process Model

- A process has been defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created. Each of these activities, actions, and tasks resides within a framework or model that defines their relationship with the process and with one another

- Generic process framework for software engineering defines five framework activities— **communication**, **planning**, **modeling**, **construction**, and **deployment**.

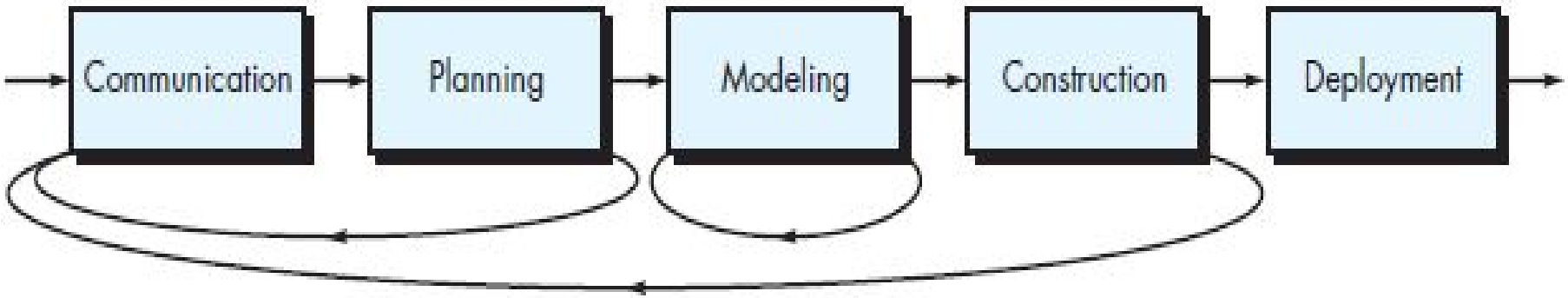
- In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others are applied throughout the process

Process

(a) Linear process flow

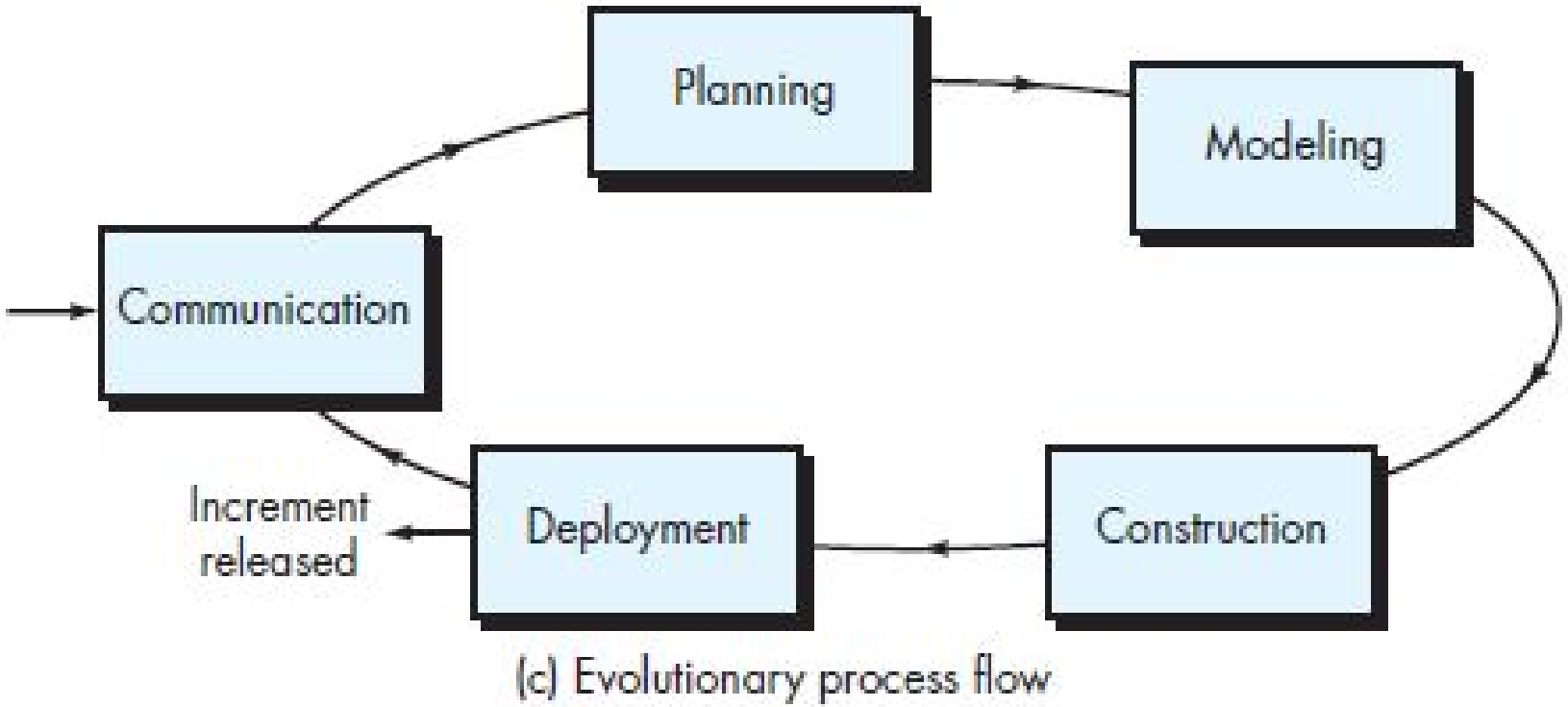
- A linear process flow executes each of the five framework activities in sequence, beginning with communication and culminating with deployment

Process



- An iterative process flow repeats one or more of the activities before proceeding to the next

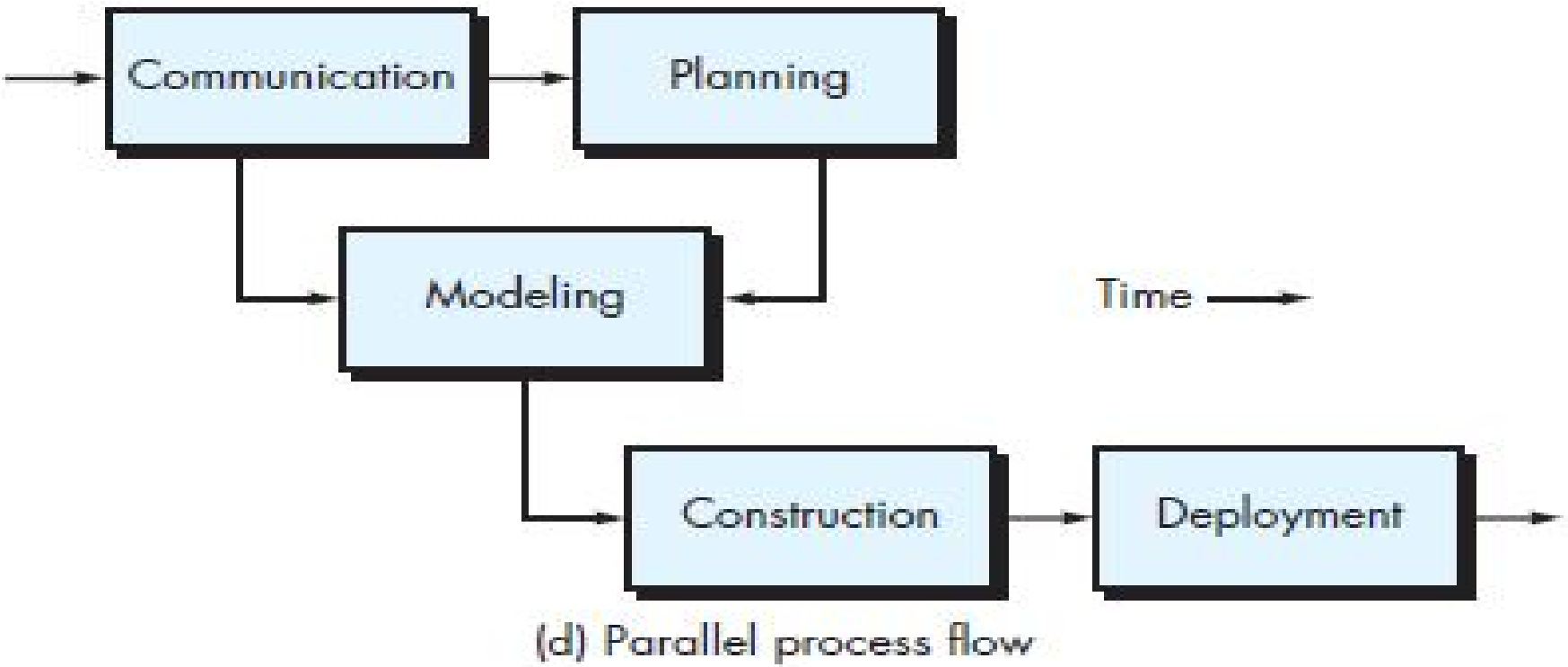
Process



(c) Evolutionary process flow

- An evolutionary process flow executes the activities in a "circular" manner. Each circuit through the five activities leads to a more complete version of the software

Process



(d) Parallel process flow

- A parallel process flow executes one or more activities in parallel with other activities

Software Processes

- A software process is a set of related activities that leads to the production of a software system.
- software process model (sometimes called a Software Development Life Cycle or SDLC model) is a simplified representation of a software process.
- A prescriptive process model strives for structure and order in software development. Activities and tasks occur sequentially with defined guidelines for progress.

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

January 23, 2023

The Process Model

- The general process models are as follows
- The waterfall model
- Incremental Process Models
- Evolutionary Process Models
- Concurrent Models

The waterfall model

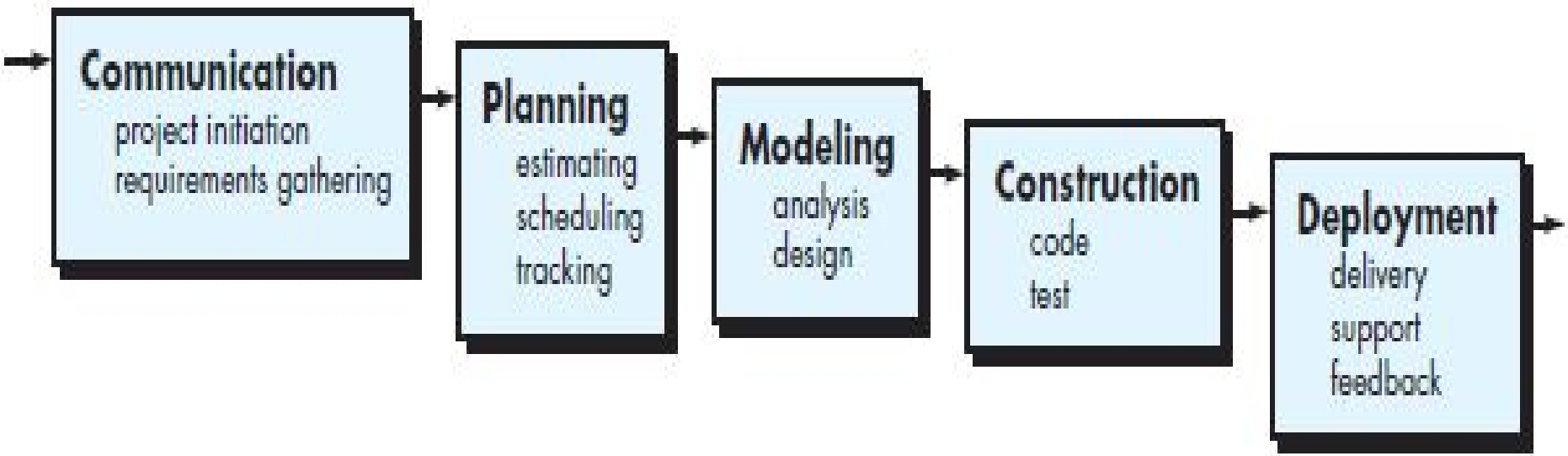
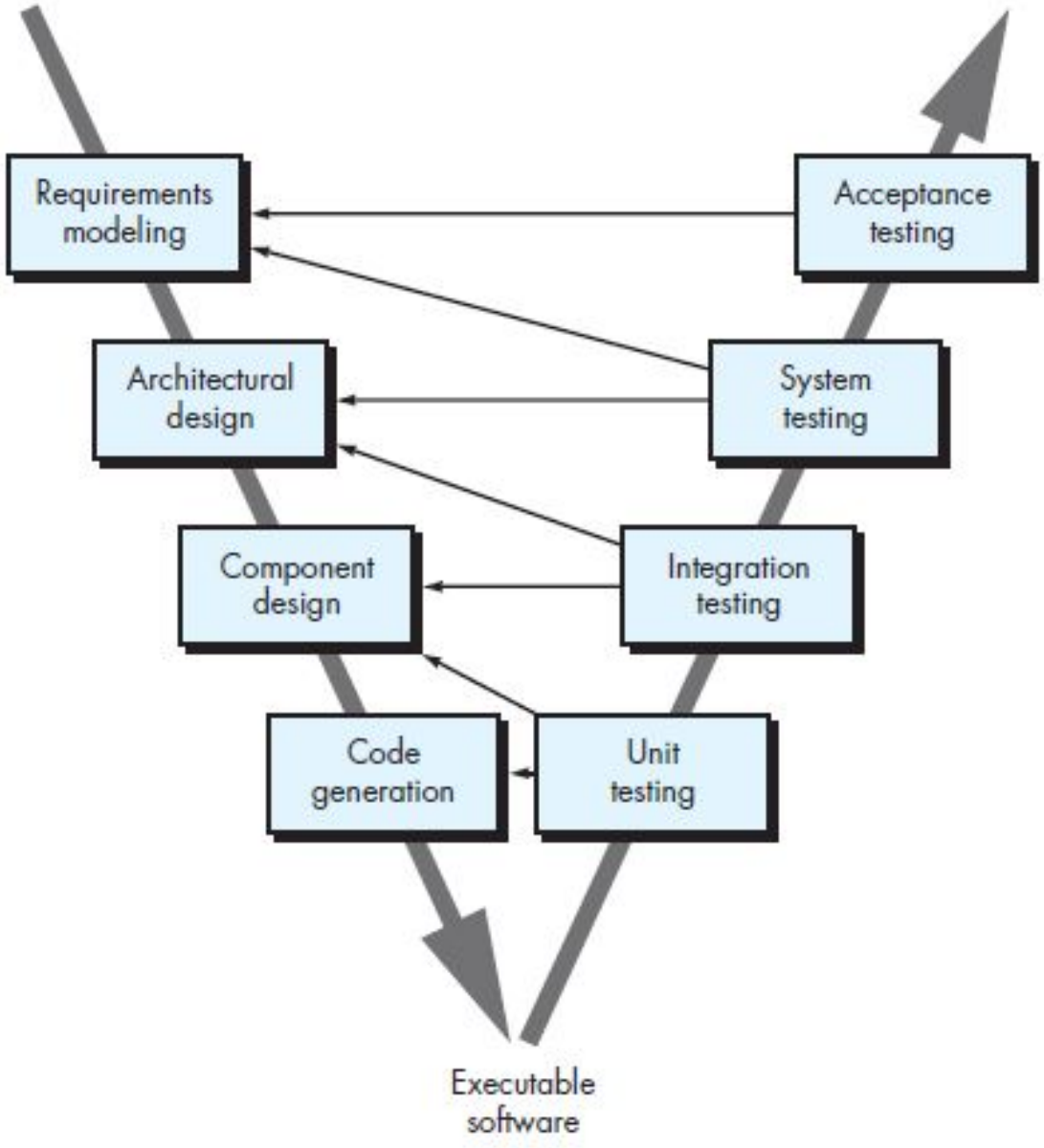


Figure 1. The waterfall model

The waterfall model

- The waterfall model, sometimes called the classic life cycle , suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in on-going support of the completed software
- A variation in the representation of the waterfall model is called the V-model.
- The V-model depicts the relationship of quality assurance actions to the actions associated with communication, modeling, and early construction activities.

The waterfall model



The waterfall model

- As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.
- Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moves down the left side.
- The waterfall model is the oldest paradigm for software engineering

The waterfall model

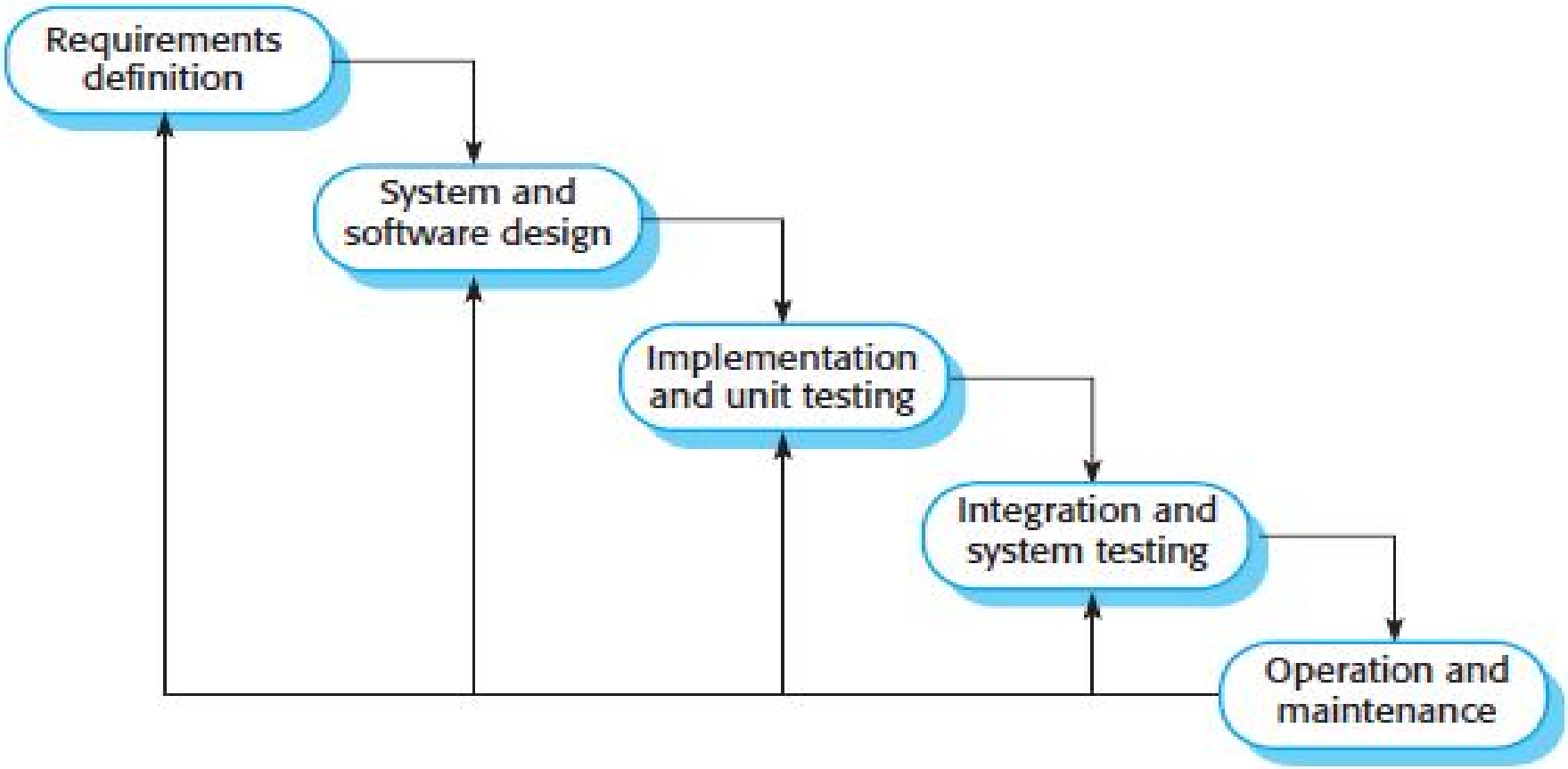


Figure 3. The waterfall model

The waterfall model

- The stages of the waterfall model directly reflect the fundamental software development activities:
- Requirements analysis and definition
- System and software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance

The waterfall model

- **Requirements analysis and definition** The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.

- **System and software design** The systems design process allocates the requirements to either hardware or software systems. It establishes an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.

- **Implementation and unit testing** During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.

The waterfall model

- **Integration and system testing** The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.

- **Operation and maintenance** Normally, this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors that were not discovered in earlier stages of the life cycle, improving the implementation of system units, and enhancing the system's services as new requirements are discovered.

The waterfall model

- The waterfall model is appropriate for some types of system
- Embedded systems where the software has to interface with hardware systems. Because of the inflexibility of hardware, it is not usually possible to delay decisions on the software's functionality until it is being implemented.
- Critical systems where there is a need for extensive safety and security analysis of the software specification and design. In these systems, the specification and design documents must be complete so that this analysis is possible.
- Large software systems that are part of broader engineering systems developed by several partner companies. The hardware in the systems may be developed using a similar model, and companies find it easier to use a common model for hardware and software

The waterfall model

- The problems that are sometimes encountered when the waterfall model is applied are:
- Real projects rarely follow the sequential flow that the model proposes. Although the linear model can accommodate iteration, it does so indirectly. As a result, changes can cause confusion as the project team proceeds.
- It is often difficult for the customer to state all requirements explicitly. The waterfall model requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

The waterfall model

- The customer must have patience. A working version of the program(s) will not be available until late in the project time span. A major blunder, if undetected until the working program is reviewed, can be disastrous.

- The waterfall model is not the right process model in situations where informal team communication is possible and software requirements change quickly. Iterative development and agile methods are better for these systems.

Incremental Process Model

- The incremental model combines the elements' linear and parallel process flows
- The incremental model applies linear sequences in a staggered fashion as calendar time progresses.
- Each linear sequence produces deliverable "increments" of the software.

Incremental Process Model

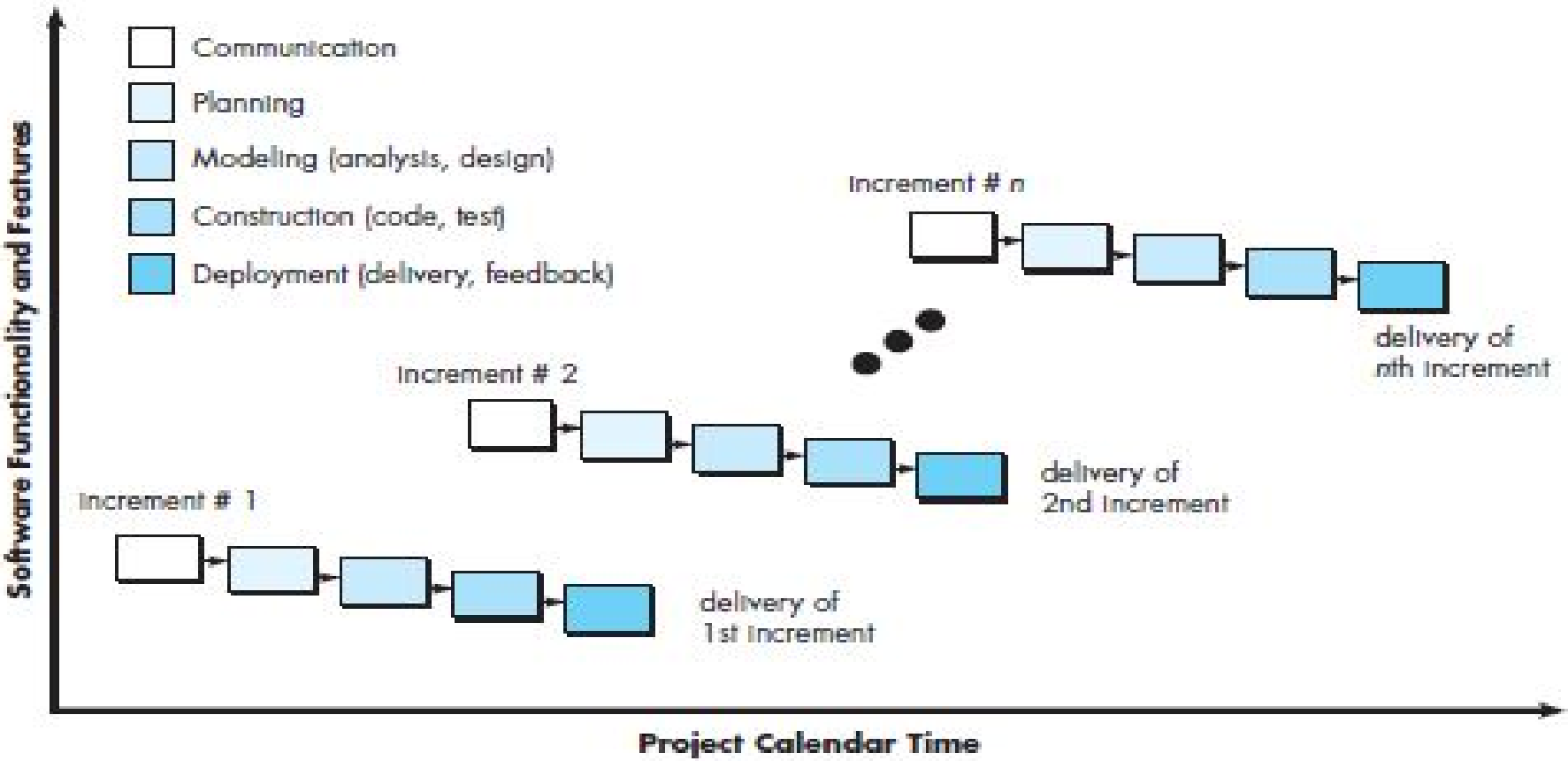


Figure 4. The incremental model

Incremental Process Models

- For example, word-processing software developed using the incremental paradigm might deliver basic fi le management, editing, and document production functions in the first increment
- More sophisticated editing and document production capabilities in the second increment
- Spelling and grammar checking in the third increment
- Advanced page layout capability in the fourth increment.

Incremental Process Models

- When an incremental model is used, the fi rst increment is often a core product. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered
- The core product is used by the customer (or undergoes detailed evaluation).
- As a result of use and/ or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality.
- This process is repeated following the delivery of each increment, until the complete product is produced.

Incremental Process Models

- Incremental development is based on the idea of developing an initial implementation, getting feedback from users and others
- Evolving the software through several versions until the required system has been developed
- Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities
- Incremental development in some form is now the most common approach for the development of application systems and software products.

The Incremental Model

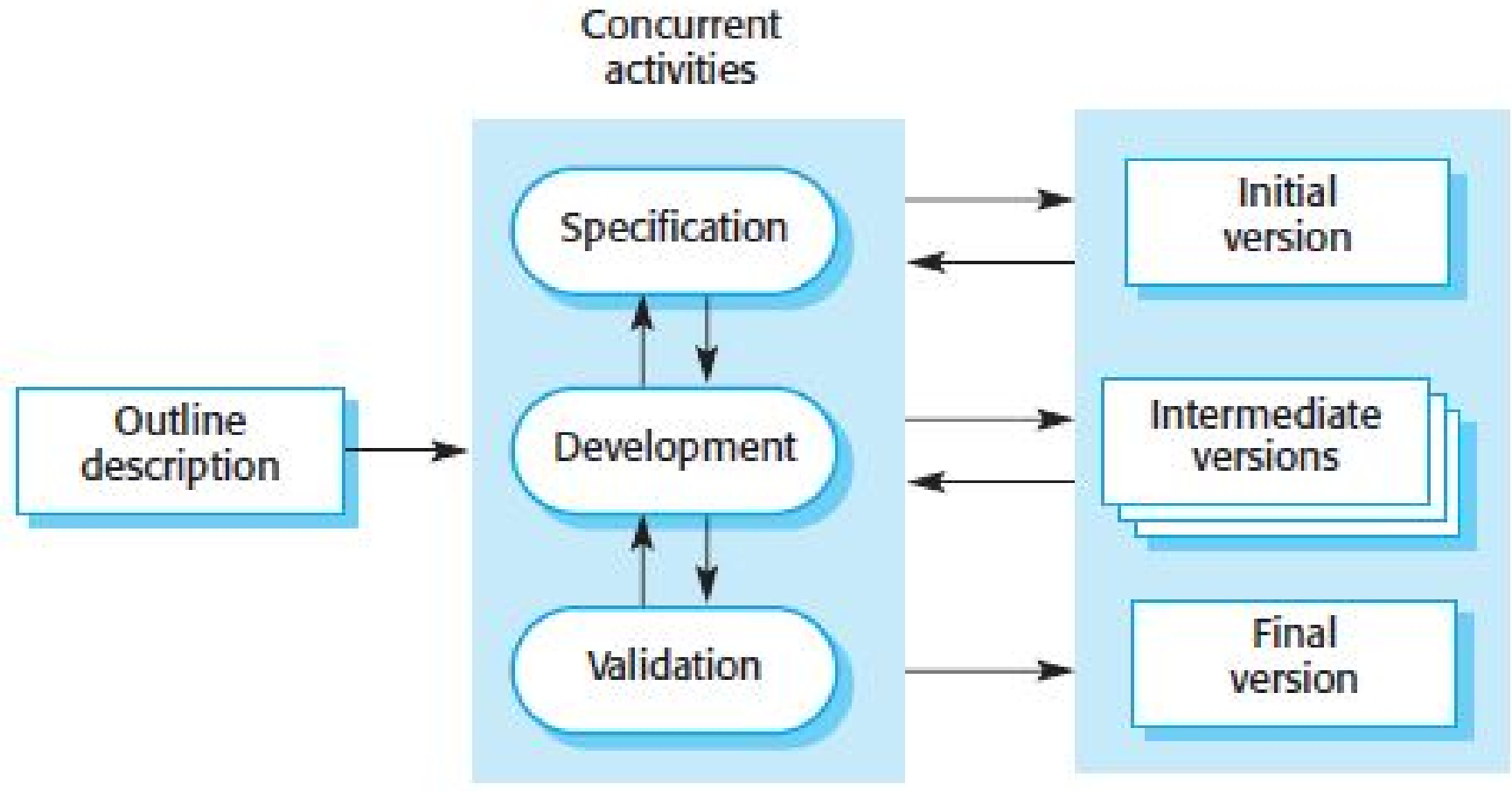


Figure 5. The incremental model

Incremental Process Models

- Incremental development has three major advantages over the waterfall model:

- The cost of implementing requirements changes is reduced. The amount of analysis and documentation that has to be redone is significantly less than is required with the waterfall model.

- It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how much has been implemented. Customers find it difficult to judge progress from software design documents.

- Early delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

Incremental Process Models

- From a management perspective, the incremental approach has two problems:
- The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost effective to produce documents that reflect every version of the system.
- System structure tends to degrade as new increments are added. Regular change leads to messy code as new functionality is added in whatever way is possible. It becomes increasingly difficult and costly to add new features to a system.

Evolutionary Process Models

- Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic
- Tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure
- A set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined.

Evolutionary Process Models

- In these and similar situations, you need a process model that has been explicitly designed to accommodate a product that grows and changes.

- Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software

Evolutionary Process Models

- Two common evolutionary process models.
- Prototyping Model
- The Spiral Model

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

February 6, 2023

Evolutionary Process Models

- Software, like all complex systems, evolves over a period of time. Business and product requirements often change as development proceeds, making a straight line path to an end product unrealistic
- Tight market deadlines make completion of a comprehensive software product impossible, but a limited version must be introduced to meet competitive or business pressure
- A set of core product or system requirements is well understood, but the details of product or system extensions have yet to be defined.

Evolutionary Process Models

- In these and similar situations, you need a process model that has been explicitly designed to accommodate a product that grows and changes.

- Evolutionary models are iterative. They are characterized in a manner that enables you to develop increasingly more complete versions of the software

Evolutionary Process Models

- Two common evolutionary process models.
- Prototyping Model
- The Spiral Model

Prototyping

- Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features.
- In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take.
- In these, and many other situations, a prototyping paradigm may offer the best approach.

The Prototyping model

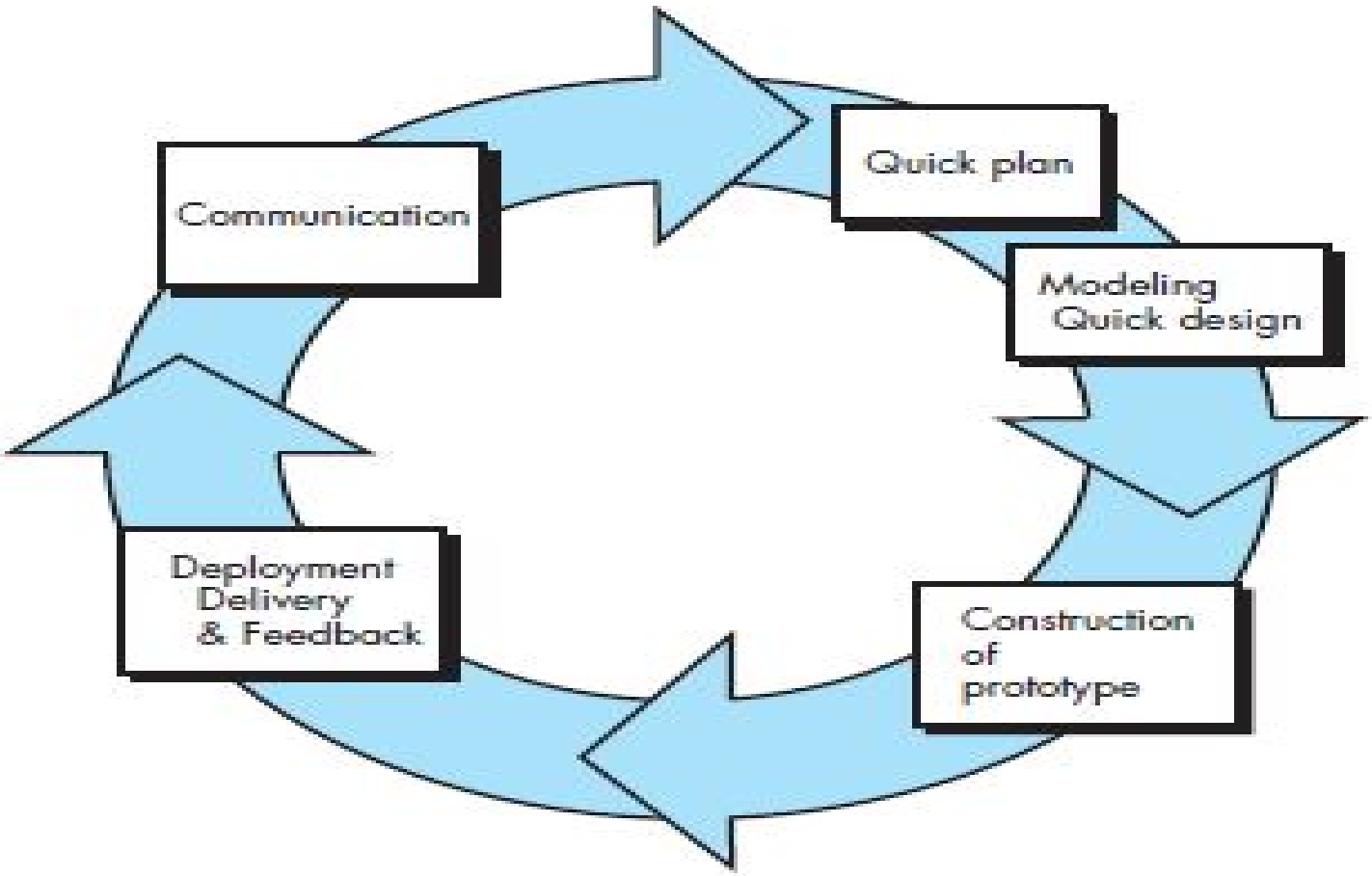


Figure 1. The Prototyping model

Prototyping

- You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.
- A prototyping iteration is planned quickly, and modeling (in the form of a “quick design”) occurs.
- A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).

Prototyping

- The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements.
- Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.
- Both stakeholders and software engineers like the prototyping paradigm.

Prototyping

- Users get a feel for the actual system, and developers get to build something immediately. Yet, prototyping can be problematic for the following reasons:

- Stakeholders see what appears to be a working version of the software, unaware that the prototype is held together haphazardly, unaware that in the rush to get it working you haven't considered overall software quality or long-term maintainability.

- An inefficient algorithm may be implemented simply to demonstrate capability. After a time, you may become comfortable with these choices and forget all the reasons why they were inappropriate. The less-than-ideal choice has now become an integral part of the system.

The Spiral Model

- The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- It provides the potential for rapid development of increasingly more complete versions of the software.
- The spiral development model is a risk -driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.

The Spiral Model

- The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.
- The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.

The Spiral model

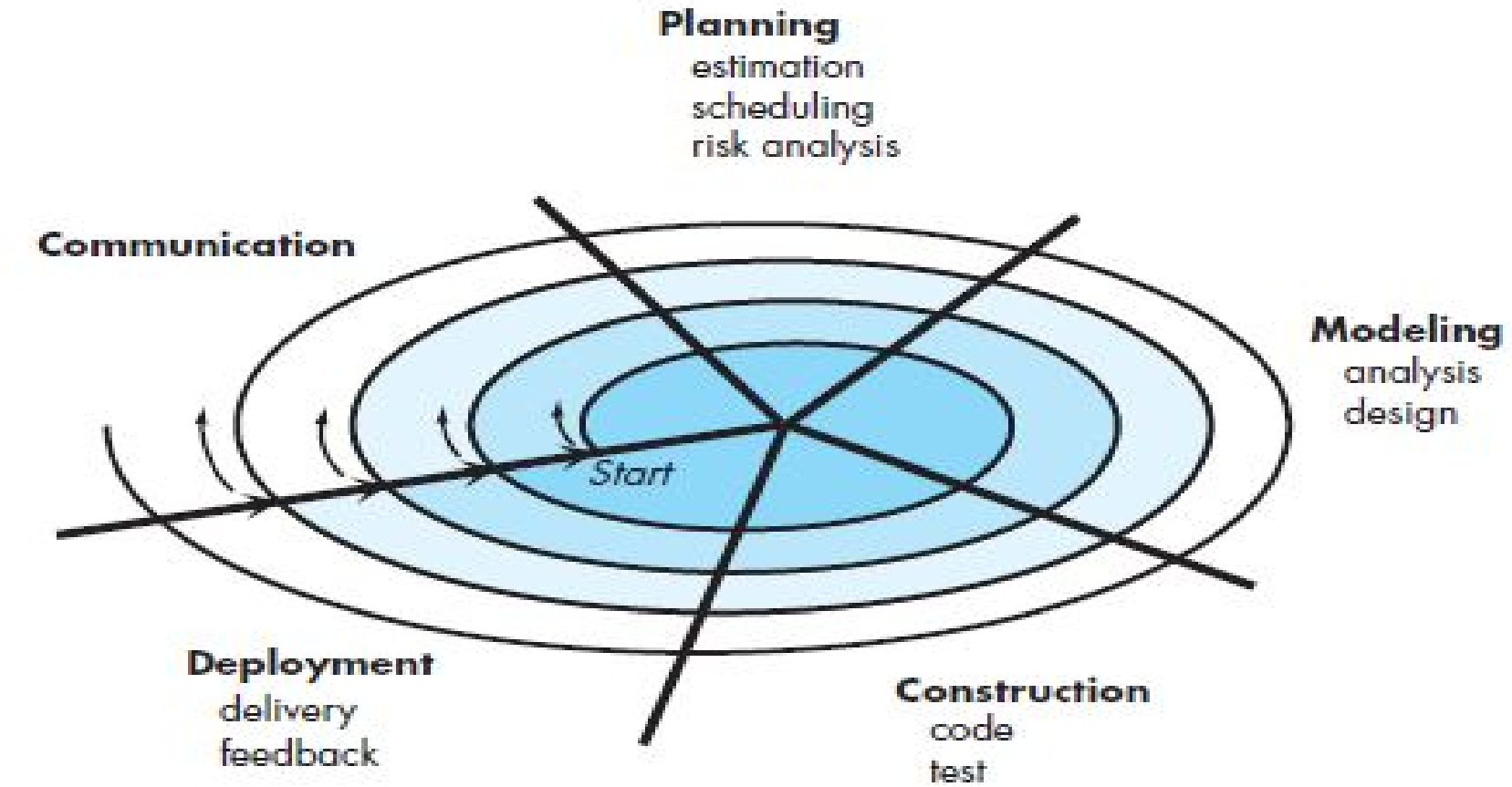


Figure 2. The Spiral model

The Spiral Model

- Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype.
- During later iterations, increasingly more complete versions of the engineered system are produced.
- A spiral model is divided into a set of framework activities defined by the software engineering team

The Spiral Model

- Each of the framework activities represent one segment of the spiral path illustrated in Figure
- As this evolutionary process begins, the software team performs activities that are implied by a circuit around the spiral in a clockwise direction, beginning at the center
- The first circuit around the spiral might result in the development of a product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

The Spiral Model

- Each pass through the planning region results in adjustments to the project plan.
- Cost and schedule are adjusted based on feedback derived from the customer after delivery
- The project manager adjusts the planned number of iterations required to complete the software.

AGILE Development

- What is Agile Development?
- Agile software engineering combines a philosophy and a set of development guidelines.
- The philosophy encourages customer satisfaction and early incremental delivery of software; small, highly motivated project teams; informal methods; minimal software engineering work products; and overall development simplicity.
- The development guidelines stress delivery over analysis and design (although these activities are not discouraged), and active and continuous communication between developers and customers.

AGILE Development

- Who does it
- Software engineers and other project stakeholders (managers, customers, end users) work together on an agile team—a team that is self-organizing and in control of its own destiny.
- An agile team fosters communication and collaboration among all who serve on it.

AGILE Development

- Why is it important
- The modern business environment that spawns computer-based systems and software products is fast-paced and ever-changing.
- Agile software engineering represents a reasonable alternative to conventional software engineering for certain classes of software and certain types of software projects.
- It has been demonstrated to deliver successful systems quickly.

AGILE Development

- What are the steps?
- Agile development might best be termed "software engineering lite."
- The basic framework activities— communication, planning, modeling, construction, and deployment.
- But they morph into a minimal task set that pushes the project team toward construction and delivery (some would argue that this is done at the expense of problem analysis and solution design).

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

February 6, 2023

Agility

- Agility has become today's buzzword when describing a modern software process.
- Everyone is agile. An agile team is a nimble team able to appropriately respond to changes. Change is what software development is very much about
- Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product.

Agility and the cost of change

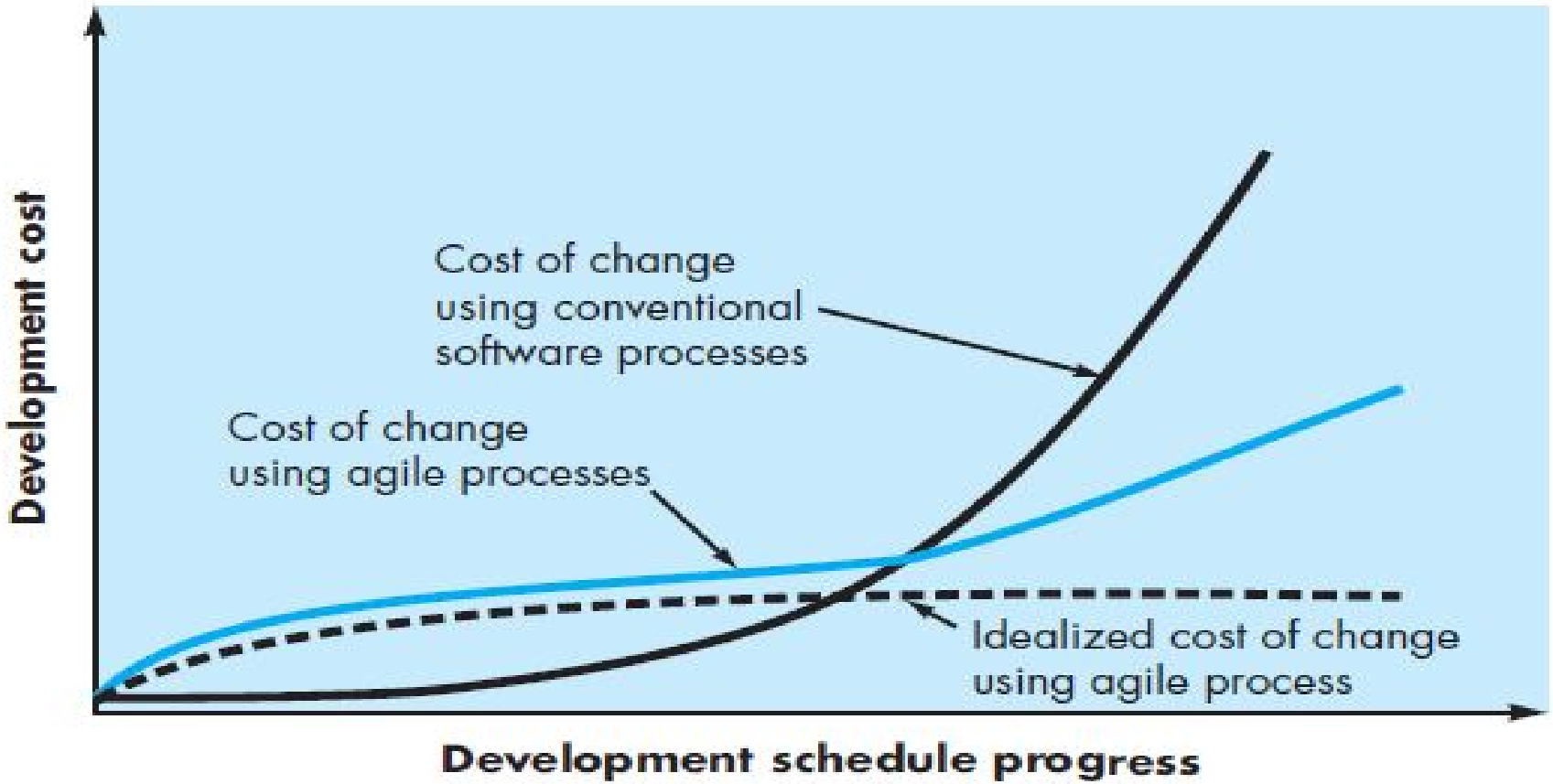


Figure 1. Change costs as a function of time in development

Agility and the cost of change

- The conventional wisdom in software development is that the cost of change increases nonlinearly as a project progresses (Figure 1, solid black curve).
- It is relatively easy to accommodate a change when a software team is gathering requirements (early in a project).
- A usage scenario might have to be modified, a list of functions may be extended, or a written specification can be edited.
- The change requires a modification to the architectural design of the software, the design and construction of three new components, modifications to another five components, the design of new tests.

What is the Agile Process

- Any agile software process is characterized in a manner that addresses a number of key assumptions
- It is difficult to predict in advance which software requirements will persist and which will change. It is equally difficult to predict how customer priorities will change as the project proceeds.
- For many types of software, design and construction are interleaved. That is, both activities should be performed in tandem so that design models are proven as they are created. It is difficult to predict how much design is necessary before construction is used to prove the design.
- Analysis, design, construction, and testing are not as predictable (from a planning point of view) as we might like.

Agility Principles

- The Agile Alliance defines 12 agility principles for those who want to achieve agility:
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Agility Principles

- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.

Agility Principles

- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity—the art of maximizing the amount of work not done—is essential.

Agility Principles

- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Extreme Programming

- Extreme Programming uses an object-oriented approach as its preferred development paradigm and encompasses a set of rules and practices that occur within the context of four framework activities
- Planning
- Design
- Coding
- Testing

Agility and the cost of change

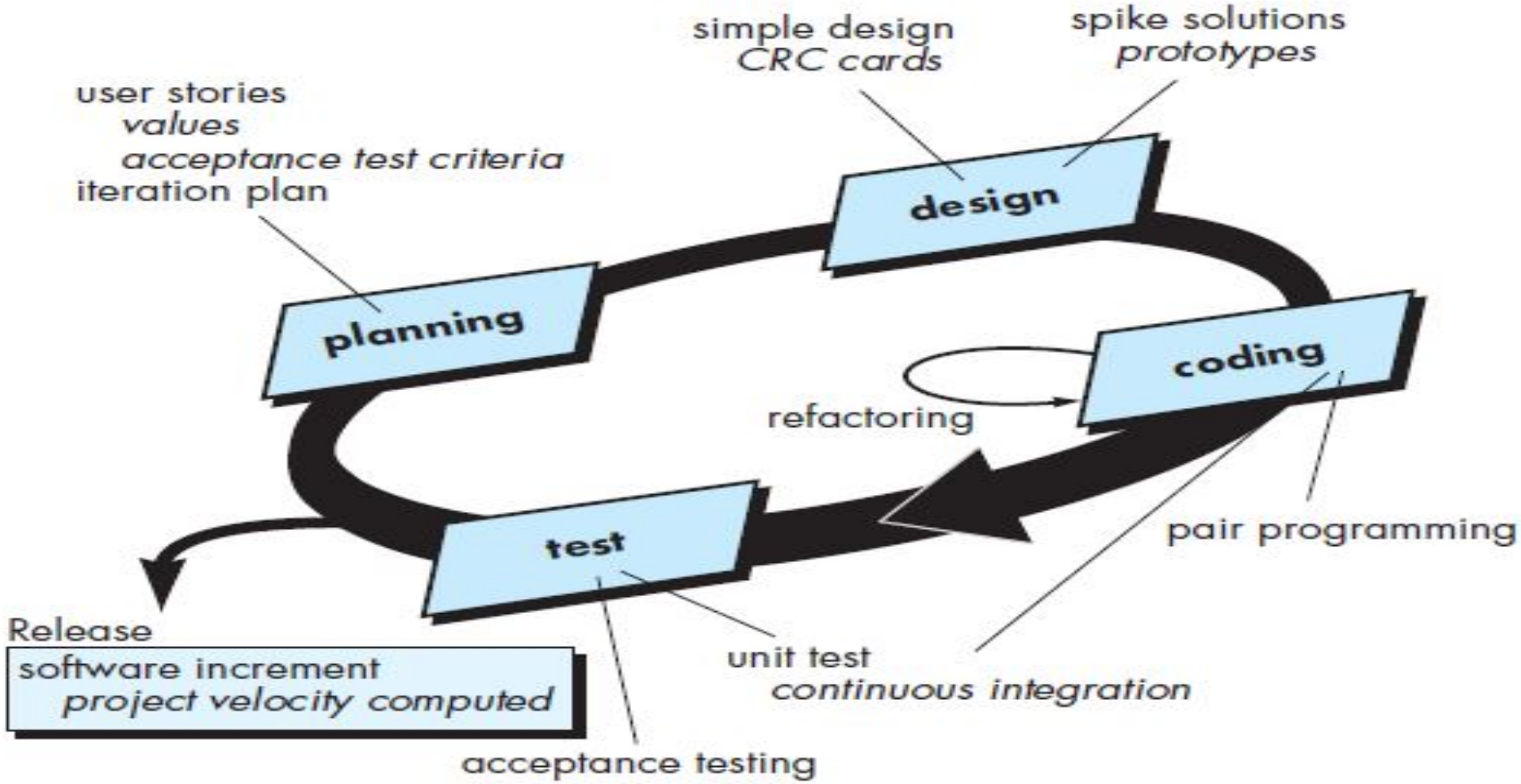


Figure 2. The Extreme Programming process

Extreme Programming

- **Planning.** The planning activity (also called the planning game) begins with listening —a requirements gathering activity that enables the technical members of the XP team to understand the business context for the software and to get a broad feel for required output and major features and functionality.

- Listening leads to the creation of a set of " stories " (also called user stories) that describe required output, features, and functionality for software to be built.

- Each story is written by the customer and is placed on an index card. The customer assigns a value to the story based on the overall business value of the feature or function.

- Members of the XP team then assess each story and assign a cost —measured in development weeks—to it.

Extreme Programming

- If the story is estimated to require more than three development weeks, the customer is asked to split the story into smaller stories and the assignment of value and cost occurs again. It is important to note that new stories can be written at any time.

- Customers and developers work together to decide how to group stories into the next release (the next software increment) to be developed by the XP team.

- After the first project release (also called a software increment) has been delivered, the XP team computes project velocity . Stated simply, project velocity is the number of customer stories implemented during the first release.

Extreme Programming

- **Design.** XP design rigorously follows the KIS (keep it simple) principle. A simple design is always preferred over a more complex representation.
- In addition, the design provides implementation guidance for a story as it is written—nothing less, nothing more.
- The design of extra functionality (because the developer assumes it will be required later) is discouraged.

Extreme Programming

- XP encourages the use of CRC cards as an effective mechanism for thinking about the software in an object-oriented context.
- CRC (class-responsibility-collaborator) cards identify and organize the object oriented classes that are relevant to the current software increment.
- The XP team conducts the design exercise using a process. The CRC cards are the only design work product produced as part of the XP process.

Extreme Programming

- **Coding.** After stories are developed and preliminary design work is done, the team does not move to code, but rather develops a series of unit tests that will exercise each of the stories that is to be included in the current release.
- Once the unit test has been created, the developer is better able to focus on what must be implemented to pass the test.
- Once the code is complete, it can be unit-tested immediately, thereby providing instantaneous feedback to the developers.

Extreme Programming

- A key concept during the coding activity (and one of the most talked-about aspects of XP) is pair programming
- XP recommends that two people work together at one computer workstation to create code for a story. This provides a mechanism for real-time problem solving and real-time quality assurance.
- It also keeps the developers focused on the problem at hand. In practice, each person takes on a slightly different role.

Extreme Programming

- For example, one person might think about the coding details of a particular portion of the design while the other ensures that coding standards (a required part of XP) are being followed or that the code for the story will satisfy the unit test that has been developed to validate the code against the story

- As pair programmers complete their work, the code they develop is integrated with the work of others. In some cases this is performed on a daily basis by an integration team.

- In other cases, the pair programmers have integration responsibility. This "continuous integration" strategy helps to avoid compatibility and interfacing problems and provides a "smoke testing" environment that helps to uncover errors early.

Extreme Programming

- **Testing.** The unit tests that are created should be implemented using a framework that enables them to be automated.
- This encourages a regression testing strategy whenever code is modified
- As the individual unit tests are organized into a "universal testing suite", integration and validation testing of the system can occur on a daily basis.

Extreme Programming

- "Fixing small problems every few hours takes less time than fixing huge problems just before the deadline."
- XP acceptance tests, also called customer tests, are specified by the customer. and focus on overall system features and functionality that are visible and reviewable by the customer.
- Acceptance tests are derived from user stories that have been implemented as part of a software release.

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

February 13, 2023

Other Agile Process Models

- The other agile process models are as follows
- Scrum
- Dynamic Systems Development Method (DSSM)
- Agile Modeling (AM)
- Agile Unified Process (AUP).

Scrum

- Scrum (the name is derived from an activity that occurs during a rugby match) is an agile software development method that was conceived by Jeff Sutherland and his development team
- Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the following framework activities:
- Requirements
- Analysis
- Evolution, and
- Delivery.

Scrum process flow

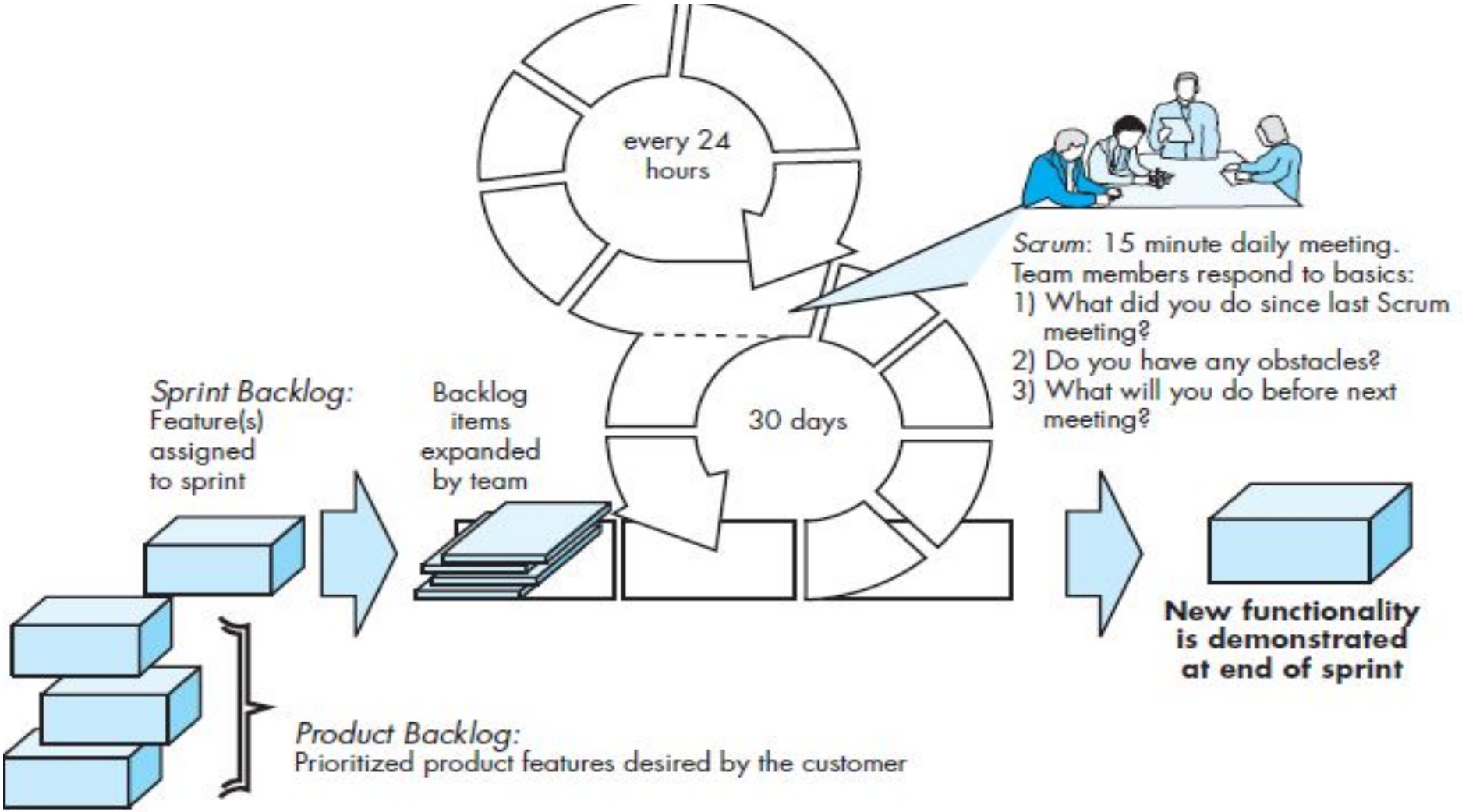


Figure 1. Scrum process flow

Scrum

- Within each framework activity, work tasks occur within a process pattern called a sprint.
- The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.
- Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality.

Scrum

- Each of these process patterns defines a set of development activities:
- **Backlog** —a prioritized list of project requirements or features that provide business value for the customer.
- Items can be added to the backlog at any time (this is how changes are introduced).
- The product manager assesses the backlog and updates priorities as required.

Scrum

- **Sprints** —consist of work units that are required to achieve a requirement defined in the backlog that must be fit into a predefined time-box (typically 30 days).
- Changes (e.g., backlog work items) are not introduced during the sprint.
- Hence, the sprint allows team members to work in a short-term, but stable environment.

Scrum

- **Scrum meetings** —are short (typically 15-minute) meetings held daily by the Scrum team.
- Three key questions are asked and answered by all team members
- What did you do since the last team meeting?
- What obstacles are you encountering?
- What do you plan to accomplish by the next team meeting?

Scrum

- A team leader, called a Scrum master, leads the meeting and assesses the responses from each person.
- The Scrum meeting helps the team to uncover potential problems as early as possible.
- Also, these daily meetings lead to "knowledge socialization" and thereby promote a self-organizing team structure.

Scrum

- **Demos** —deliver the software increment to the customer so that functionality that has been implemented can be demonstrated and evaluated by the customer.
- It is important to note that the demo may not contain all planned functionality, but rather those functions that can be delivered within the time-box that was established.

Dynamic Systems Development Method

- The Dynamic Systems Development Method (DSDM) is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment"

- The DSDM philosophy is borrowed from a modified version of the Pareto principle—80 percent of an application can be delivered in 20 percent of the time it would take to deliver the complete (100 percent) application.

Dynamic Systems Development Method

- DSDM is an iterative software process in which each iteration follows the 80 percent rule.
- That is, only enough work is required for each increment to facilitate movement to the next increment.
- The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.

Dynamic Systems Development Method

- The DSDM Consortium (www.dsdm.org) is a worldwide group of member companies that collectively take on the role of “keeper” of the method.
- The consortium has defined an agile process model, called the DSDM life cycle, that begins with a feasibility study that establishes basic business requirements and constraints and is followed by a business study that identifies functional and information requirements.

Dynamic Systems Development Method

- DSDM then defines three different iterative cycles:
- Functional model iteration
- Design and build iteration
- Implementation

Dynamic Systems Development Method

- **Functional model iteration**— produces a set of incremental prototypes that demonstrate functionality for the customer.
- The intent during this iterative cycle is to gather additional requirements by eliciting feedback from users as they exercise the prototype.

Dynamic Systems Development Method

- **Design and build iteration**— revisits prototypes built during the functional model iteration to ensure that each has been engineered in a manner that will enable it to provide operational business value for end users.

- In some cases, the functional model iteration and the design and build iteration occur concurrently.

Dynamic Systems Development Method

- **Implementation**— places the latest software increment (an "operationalized" prototype) into the operational environment.
- It should be noted that (1) the increment may not be 100 percent complete or (2) changes may be requested as the increment is put into place.
- In either case, DSDM development work continues by returning to the functional model iteration activity.
- DSDM can be combined with XP to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build software increments.

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

February 14, 2023

Agile Modeling

- Agile modeling adopts all of the values that are consistent with the agile manifesto.
- The agile modeling philosophy recognizes that an agile team must have the courage to make decisions that may cause it to reject a design and refactor.
- The team must also have the humility to recognize that technologists do not have all the answers and that business experts and other stakeholders should be respected and embraced.

Agile Modeling

- **Model with a purpose.** A developer who uses AM should have a specific goal (e.g., to communicate information to the customer or to help better understand some aspect of the software) in mind before creating the model. Once the goal for the model is identifi ed, the type of notation to be used and level of detail required will be more obvious.

- **Use multiple models.** There are many different models and notations that can be used to describe software. Only a small subset is essential for most projects. AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

Agile Modeling

- **Travel light.** As software engineering work proceeds, keep only those models that will provide long-term value.
- **Content is more important than representation.** Modeling should impart information to its intended audience.
- **Know the models and the tools you use to create them.** Understand the strengths and weaknesses of each model and the tools that are used to create it.
- **Adapt locally.** The modeling approach should be adapted to the needs of the agile team.

Agile Unified Process

- The Agile Unified Process (AUP) adopts a "serial in the large" and "iterative in the small" philosophy for building computer-based systems.

- By adopting the classic UP phased activities—inception, elaboration, construction, and transition—AUP provides a serial overlay (i.e., a linear sequence of software engineering activities) that enables a team to visualize the overall process fl ow for a software project.

- However, within each of the activities, the team iterates to achieve agility and to deliver meaningful software increments to end users as rapidly as possible.

Agile Unified Process

- Each AUP iteration addresses the following activities
- Modeling. UML representations of the business and problem domains are created.
- Implementation. Models are translated into source code.
- Testing. Like XP, the team designs and executes a series of tests to uncover errors and ensure that the source code meets its requirements
- Deployment. Deployment in this context focuses on the delivery of a software increment and the acquisition of feedback from end users.

Agile Unified Process

- Configuration and project management. In the context of AUP, configuration management addresses change management, risk management, and the control of any persistent work products that are produced by the team. Project management tracks and controls the progress of the team and coordinates team activities.

- Environment management. Environmental management coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

Requirements Engineering

- Designing and building computer software is challenging, creative, and just plain fun.
- In fact, building software is so compelling that many software developers want to jump right in before they have a clear understanding of what is needed.
- They argue that things will become clear as they build, that project stakeholders will be able to understand need only after examining early iterations of the software, that things change so rapidly that any attempt to understand requirements in detail is a waste of time, that the bottom line is producing a working program, and that all else is secondary

Requirements Engineering

- The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering.
- From a software process perspective, requirements engineering is a major software engineering action that begins during the communication activity and continues into the modeling activity.
- It must be adapted to the needs of the process, the project, the product, and the people doing the work.
- Requirements engineering builds a bridge to design and construction.

Requirements Engineering

- Requirements engineering encompasses seven distinct tasks:
- inception,
- elicitation,
- elaboration,
- negotiation,

Requirements Engineering

- specification,
- validation, and
- management

Inception

- How does a software project get started?
- Is there a single event that becomes the catalyst for a new computer-based system or product, or does the need evolve over time?
- There are no definitive answers to these questions.
- In some cases, a casual conversation is all that is needed to precipitate a major software engineering effort.

Inception

- But in general, most projects begin when a business need is identified or a potential new market or service is discovered.
- Stakeholders from the business community (e.g., business managers, marketing people, product managers) define a business case for the idea, try to identify the breadth and depth of the market, do a rough feasibility analysis, and identify a working description of the project's scope.
- All of this information is subject to change, but it is sufficient to precipitate discussions with the software engineering organization.
- At project inception, you establish a basic understanding of the problem, the people who want a solution, the nature of the solution that is desired, and the effectiveness of preliminary communication and collaboration between the other stakeholders and the software team.

Elicitation.

- It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are,
- what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product is to be used on a day-to-day basis.
- But it isn't simple— it's very hard.
- identify a number of problems that are encountered as elicitation occurs. Problems of scope occur when the boundary of the system is ill-defined or the customers and users specify unnecessary technical detail that may confuse, rather than clarify, overall system objectives.

Elicitation.

- Problems of understanding are encountered when customers and users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment,

- don't have a full understanding of the problem domain, have trouble communicating needs, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers and users, or specify requirements that are ambiguous or untestable.

- Problems of volatility occur when the requirements change over time. To help overcome these problems, you must approach the requirements-gathering activity in an organized manner.

Elaboration.

- The information obtained from the customer during inception and elicitation is expanded and refined during elaboration.
- This task focuses on developing a refined requirements model that identifies various aspects of software function, behavior, and information.
- Elaboration is driven by the creation and refinement of user scenarios that describe how the end user (and other actors) will interact with the system.

Elaboration.

- Each user scenario is parsed to extract analysis classes—business domain entities that are visible to the end user.
- The attributes of each analysis class are defined, and the services that are required by each class are identified.
- The relationships and collaboration between classes are identified, and a variety of supplementary diagrams are produced.

Negotiation.

- It isn't unusual for customers and users to ask for more than can be achieved, given limited business resources.
- It's also relatively common for different customers or users to propose conflicting requirements, arguing that their version is "essential for our special needs."
- You have to reconcile these conflicts through a process of negotiation.

Negotiation.

- Customers, users, and other stakeholders are asked to rank requirements and then discuss conflicts in priority.
- Using an iterative approach that prioritizes requirements, assesses their cost and risk, and addresses internal conflicts, requirements are eliminated, combined, and/or modified so that each party achieves some measure of satisfaction.

Specification.

- In the context of computer-based systems (and software), the term specification means different things to different people.
- A specification can be a written document, a set of graphical models, a formal mathematical model, a collection of usage scenarios, a prototype, or any combination of these.
- Some suggest that a "standard template" should be developed and used for a specification, arguing that this leads to requirements that are presented in a consistent and therefore more understandable manner.

Specification.

- However, it is sometimes necessary to remain flexible when a specification is to be developed.
- For large systems, a written document, combining natural language descriptions and graphical models may be the best approach.
- However, usage scenarios may be all that are required for smaller products or systems that reside within well-understood technical environments.

Validation.

- The work products produced as a consequence of requirements engineering are assessed for quality during a validation step.

- Requirements validation examines the specification to ensure that all software requirements have been stated unambiguously; that inconsistencies, omissions, and errors have been detected and corrected; and that the work products conform to the standards established for the process, the project, and the product.

Validation.

- The primary requirements validation mechanism is the technical review
- The review team that validates requirements includes software engineers, customers, users, and other stakeholders who examine the specification looking for errors in content or interpretation, areas where clarification may be required, missing information, inconsistencies (a major problem when large products or systems are engineered), conflicting requirements, or unrealistic (unachievable) requirements.

Validation.

- To illustrate some of the problems that occur during requirements validation, consider two seemingly innocuous requirements:
- The software should be user friendly
- The probability of a successful unauthorized database intrusion should be less than 0.0001.

Validation.

- The first requirement is too vague for developers to test or assess.
- What exactly does “user friendly” mean? To validate it, it must be quantified or qualified in some manner.

Validation.

- The second requirement has a quantitative element ("less than 0.0001"), but intrusion testing will be difficult and time consuming.
- Is this level of security even warranted for the application?
- Can other complementary requirements associated with security (e.g., password protection, specialized handshaking) replace the quantitative requirement noted?

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

February 15, 2023

Establishing the GroundWork

- In an ideal setting, stakeholders and software engineers work together on the same team. In such cases, requirements engineering is simply a matter of conducting meaningful conversations with colleagues who are well-known members of the team. But reality is often quite different.

- Customer(s) or end users may be located in a different city or country, may have only a vague idea of what is required, may have conflicting opinions about the system to be built, may have limited technical knowledge, and may have limited time to interact with the requirements engineer. None of these things are desirable, but all are fairly common, and you are often forced to work within the constraints imposed by this situation.

Establishing the GroundWork

- The discussion is about the steps required to establish the groundwork for an understanding of software requirements—to get the project started in a way that will keep it moving forward toward a successful solution.
- Identifying Stakeholders
- Recognizing Multiple Viewpoints
- Working toward Collaboration
- Asking the First Questions
- Nonfunctional Requirements
- Traceability

Identifying Stakeholders

- Sommerville define a stakeholder as "anyone who benefits in a direct or indirect way from the system which is being developed."
- We have already identified the usual suspects: business operations managers, product managers, marketing people, internal and external customers, end users, consultants, product engineers, software engineers, support and maintenance engineers, and others.

Identifying Stakeholders

- Each stakeholder has a different view of the system, achieves different benefits when the system is successfully developed, and is open to different risks if the development effort should fail.
- At inception, you should create a list of people who will contribute input as requirements are elicited.
- The initial list will grow as stakeholders are contacted because every stakeholder will be asked: "Whom else do you think I should talk to?"

Recognizing Multiple Viewpoints

- Because many different stakeholders exist, the requirements of the system will be explored from many different points of view.
- For example, the marketing group is interested in functions and features that will excite the potential market, making the new system easy to sell.
- Business managers are interested in a feature set that can be built within budget and that will be ready to meet defined market windows.

Recognizing Multiple Viewpoints

- End users may want features that are familiar to them and that are easy to learn and use.
- Software engineers may be concerned with functions that are invisible to nontechnical stakeholders but that enable an infrastructure that supports more marketable functions and features.
- Support engineers may focus on the maintainability of the software.

Recognizing Multiple Viewpoints

- Each of these constituencies (and others) will contribute information to the requirements engineering process.
- As information from multiple viewpoints is collected, emerging requirements may be inconsistent or may conflict with one another.
- You should categorize all stakeholder information (including inconsistent and conflicting requirements) in a way that will allow decision makers to choose an internally consistent set of requirements for the system.

Recognizing Multiple Viewpoints

- There are several things that can make it hard to elicit requirements for software that satisfies its users:
- project goals are unclear, stakeholders' priorities differ, people have unspoken assumptions, stakeholders interpret meanings differently, and requirements are stated in a way that makes them difficult to verify
- The goal of effective requirements engineering is to eliminate or at least reduce these problems.

Working toward Collaboration

- If five stakeholders are involved in a software project, you may have five (or more) different opinions about the proper set of requirements.
- Throughout earlier chapters, we have noted that customers (and other stakeholders) should collaborate among themselves (avoiding petty turf battles) and with software engineering practitioners if a successful system is to result.
- But how is this collaboration accomplished?

Working toward Collaboration

- The job of a requirements engineer is to identify areas of commonality (i.e., requirements on which all stakeholders agree) and areas of conflict or inconsistency (i.e., requirements that are desired by one stakeholder but conflict with the needs of another stakeholder).

- It is, of course, the latter category that presents a challenge.

Working toward Collaboration

- Collaboration does not necessarily mean that requirements are defined by committee.
- In many cases, stakeholders collaborate by providing their view of requirements, but a strong "project champion" (e.g., a business manager or a senior technologist) may make the final decision about which requirements make the cut.

Asking the First Questions

- Questions asked at the inception of the project should be "context free"
- The first set of context-free questions focuses on the customer and other stakeholders, the overall project goals and benefits. For example, you might ask:
- Who is behind the request for this work?
- Who will use the solution?
- What will be the economic benefit of a successful solution?
- Is there another source for the solution that you need?

Asking the First Questions

- The next set of questions enables you to gain a better understanding of the problem and allows the customer to voice his or her perceptions about a solution:
- How would you characterize "good" output that would be generated by a successful solution?
- What problem(s) will this solution address?
- Can you show me (or describe) the business environment in which the solution will be used?
- Will special performance issues or constraints affect the way the solution is approached?

Asking the First Questions

- The final set of questions focuses on the effectiveness of the communication activity itself.
- Are you the right person to answer these questions? Are your answers "official"?
- Are my questions relevant to the problem that you have?
- Am I asking too many questions?
- Can anyone else provide additional information?
- Should I be asking you anything else?

Nonfunctional Requirements

- A nonfunctional requirement (NFR) can be described as a quality attribute, a performance attribute, a security attribute, or a general constraint on a system.

- These are often not easy for stakeholders to articulate. Chung suggests that there is a lopsided emphasis on functionality of the software, yet the software may not be useful or usable without the necessary nonfunctional characteristics.

- Quality Function Deployment (QFD) attempts to translate unspoken customer needs or goals into system requirements. Nonfunctional requirements are often listed separately in a software requirements specification.

Nonfunctional Requirements

- As an adjunct to QFD, it is possible to define a two-phase approach that can assist a software team and other stakeholders in identifying nonfunctional requirements.
- During the first phase, a set of software engineering guidelines is established for the system to be built.
- A list of NFRs (e.g., requirements that address usability, testability, security or maintainability) is then developed. A simple table lists NFRs as column labels and software engineering guidelines as row labels .

Nonfunctional Requirements

- A relationship matrix compares each guideline to all others, helping the team to assess whether each pair of guidelines is complementary , overlapping , conflicting , or independent .

- In the second phase, the team prioritizes each nonfunctional requirement by creating a homogeneous set of nonfunctional requirements using a set of decision rules that establish which guidelines to implement and which to reject.

Traceability

- Traceability is a software engineering term that refers to documented links between software engineering work products (e.g., requirements and test cases).
- A traceability matrix allows a requirements engineer to represent the relationship between requirements and other software engineering work products.
- Rows of the traceability matrix are labeled using requirement names and columns can be labeled with the name of a software engineering work product (e.g., a design element or a test case).

Traceability

- A matrix cell is marked to indicate the presence of a link between the two.
- The traceability matrices can support a variety of engineering development activities. They can provide continuity for developers as a project moves from one project phase to another, regardless of the process model being used.
- Traceability matrices often can be used to ensure the engineering work products have taken all requirements into account.

Eliciting Requirements

- Requirements elicitation combines elements of problem solving, elaboration, negotiation, and specification
- In order to encourage a collaborative, team-oriented approach to requirements gathering, stakeholders work together to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.

Eliciting Requirements

- Collaborative Requirements Gathering
- Quality Function Deployment
- Usage Scenarios
- Elicitation Work Products
- Agile Requirements Elicitation
- Service-Oriented Methods

Collaborative Requirements Gathering

- Many different approaches to collaborative requirements gathering have been proposed.
- Each makes use of a slightly different scenario, but all apply some variation on the following basic guidelines:
- Meetings (either real or virtual) are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.

Collaborative Requirements Gathering

- An agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A “facilitator” (can be a customer, a developer, or an outsider) controls the meeting.
- A “definition mechanism” (can be work sheets, fl ip charts, or wall stickers or an electronic bulletin board, chat room, or virtual forum) is used.

Collaborative Requirements Gathering

- The goal is to identify the problem, propose elements of the solution, negotiate different approaches, and specify a preliminary set of solution requirements.
- A one- or two-page "product request" is generated during inception.
- A meeting place, time, and date are selected; a facilitator is chosen; and attendees from the software team and other stakeholder organizations are invited to participate.
- The product request is distributed to all attendees before the meeting date.

Quality Function Deployment

- Quality function deployment (QFD) is a quality management technique that translates the needs of the customer into technical requirements for software.
- QFD "concentrates on maximizing customer satisfaction from the software engineering process"
- To accomplish this, QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process.

Quality Function Deployment

- Within the context of QFD , normal requirements identify the objectives and goals that are stated for a product or system during meetings with the customer.
- If these requirements are present, the customer is satisfi ed. Expected requirements are implicit to the product or system and may be so fundamental that the customer does not explicitly state them.
- Their absence will be a cause for significant dissatisfaction. Exciting requirements go beyond the customer's expectations and prove to be very satisfying when present.

Quality Function Deployment

- Although QFD concepts can be applied across the entire software process specific QFD techniques are applicable to the requirements elicitation activity.
- QFD uses customer interviews and observation, surveys, and examination of historical data (e.g., problem reports) as raw data for the requirements gathering activity.
- These data are then translated into a table of requirements—called the customer voice table —that is reviewed with the customer and other stakeholders.
- A variety of diagrams, matrices, and evaluation methods are then used to extract expected requirements and to attempt to derive exciting requirements

Usage Scenarios

- As requirements are gathered, an overall vision of system functions and features begin to materialize.
- However, it is difficult to move into more technical software engineering activities until you understand how these functions and features will be used by different classes of end users.
- To accomplish this, developers and users can create a set of scenarios that identify a thread of usage for the system to be constructed.
- The scenarios, often called use cases, provide a description of how the system will be used.

Elicitation Work Products

- The work products produced as a consequence of requirements elicitation will vary depending on the size of the system or product to be built.
- For most systems, the work products include:
- (1) a statement of need and feasibility,
- (2) a bounded statement of scope for the system or product,

Elicitation Work Products

- (3) a list of customers, users, and other stakeholders who participated in requirements elicitation,
- (4) a description of the system's technical environment,
- (5) a list of requirements (preferably organized by function) and the domain constraints that applies to each,
- (6) a set of usage scenarios that provide insight into the use of the system or product under different operating conditions, and
- (7) any prototypes developed to better define requirements. Each of these work products is reviewed by all people who have participated in requirements elicitation.

Agile Requirements Elicitation

- Within the context of an agile process, requirements are elicited by asking all stakeholders to create user stories .
- Each user story describes a simple system requirement written from the user's perspective.
- User stories can be written on small note cards, making it easy for developers to select and manage a subset of requirements to implement for the next product increment.
- Proponents claim that using note cards written in the user's own language allows developers to shift their focus to communication with stakeholders on the selected requirements rather than their own agenda

Agile Requirements Elicitation

- Although the agile approach to requirements elicitation is attractive for many software teams, critics argue that a consideration of overall business goals and nonfunctional requirements is often lacking.

- In some cases, rework is required to accommodate performance and security issues. In addition, user stories may not provide a sufficient basis for system evolution over time

Service-Oriented Methods

- Service-oriented development views a system as an aggregation of services.
- A service can be "as simple as providing a single function, for example, a request/ response-based mechanism that provides a series of random numbers, or can be an aggregation of complex elements, such as the Web service API"

Service-Oriented Methods

- Requirements elicitation in service-oriented development focuses on the definition of services to be rendered by an application.
- Most service design methods emphasize understanding the customer, thinking creatively, and building solutions quickly
- To achieve these goals, requirements elicitation can include ethnographic studies, innovation workshops, and early low-fidelity prototypes.
- Techniques for eliciting requirements must also acquire information about the brand and the stakeholders' perceptions of it.

Negotiating Requirements

- In an ideal requirements engineering context, the inception, elicitation, and elaboration tasks determine customer requirements in sufficient detail to proceed to subsequent software engineering activities.
- Unfortunately, this rarely happens.
- In reality, you may have to enter into a negotiation with one or more stakeholders.
- The best negotiations strive for a “win-win” result.

Negotiating Requirements

- That is, stakeholders win by getting the system or product that satisfies the majority of their needs and you (as a member of the software team) win by working to realistic and achievable budgets and deadlines.
- A set of negotiation activities at the beginning of each software process iteration has to be defined. Rather than a single customer communication activity, the following activities are defined:
- Identification of the system or subsystem's key stakeholders.
- Determination of the stakeholders' "win conditions."
- Negotiation of the stakeholders' win conditions to reconcile them into a set of win-win conditions for all concerned (including the software team).

Validating Requirements

- As each element of the requirements model is created, it is examined for inconsistency, omissions, and ambiguity.
- The requirements represented by the model are prioritized by stakeholders and grouped within requirements packages that will be implemented as software increments.
- A review of the requirements model addresses the following questions:
- Is each requirement consistent with the overall objectives for the system or product?

Validating Requirements

- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?

Validating Requirements

- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function, and behavior of the system to be built?

Validating Requirements

- Has the requirements model been "partitioned" in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model? Have all patterns been properly validated? Are all patterns consistent with customer requirements?
- These and other questions should be asked and answered to ensure that the requirements model is an accurate reflection of stakeholder needs and that it provides a solid foundation for design.

Requirements Analysis

- Requirements analysis results in the specification of software's operational characteristics, indicates software's interface with other system elements, and establishes constraints that software must meet.

- Requirements analysis allows you (regardless of whether you're called a software engineer, an analyst, or a modeler) to elaborate on basic requirements established during the inception, elicitation, and negotiation tasks that are part of requirements engineering.

Requirements Analysis

- The requirements modeling action results in one or more of the following types of models:
- **Scenario-based models** of requirements from the point of view of various system "actors."
- **Class-oriented models** that represent object-oriented classes (attributes and operations) and the manner in which classes collaborate to achieve system requirements.

Requirements Analysis

- **Behavioral and patterns-based models** that depict how the software behaves as a consequence of external "events."
- **Data models** that depict the information domain for the problem.
- **Flow-oriented models** that represent the functional elements of the system and how they transform data as they move through the system.

Requirements Modeling

- The requirements model must achieve three primary objectives:
- (1) to describe what the customer requires,
- (2) to establish a basis for the creation of a software design, and
- (3) to define a set of requirements that can be validated once the software is built.

Requirements Modeling

- The analysis model bridges the gap between a system- level description and a software design
- System- level description that describes overall system or business functionality as it is achieved by applying software, hardware, data, human, and other system elements
- A software design that describes the software's application architecture, user interface, and component-level structure.
- (3) to define a set of requirements that can be validated once the software is built.

Requirements Modeling

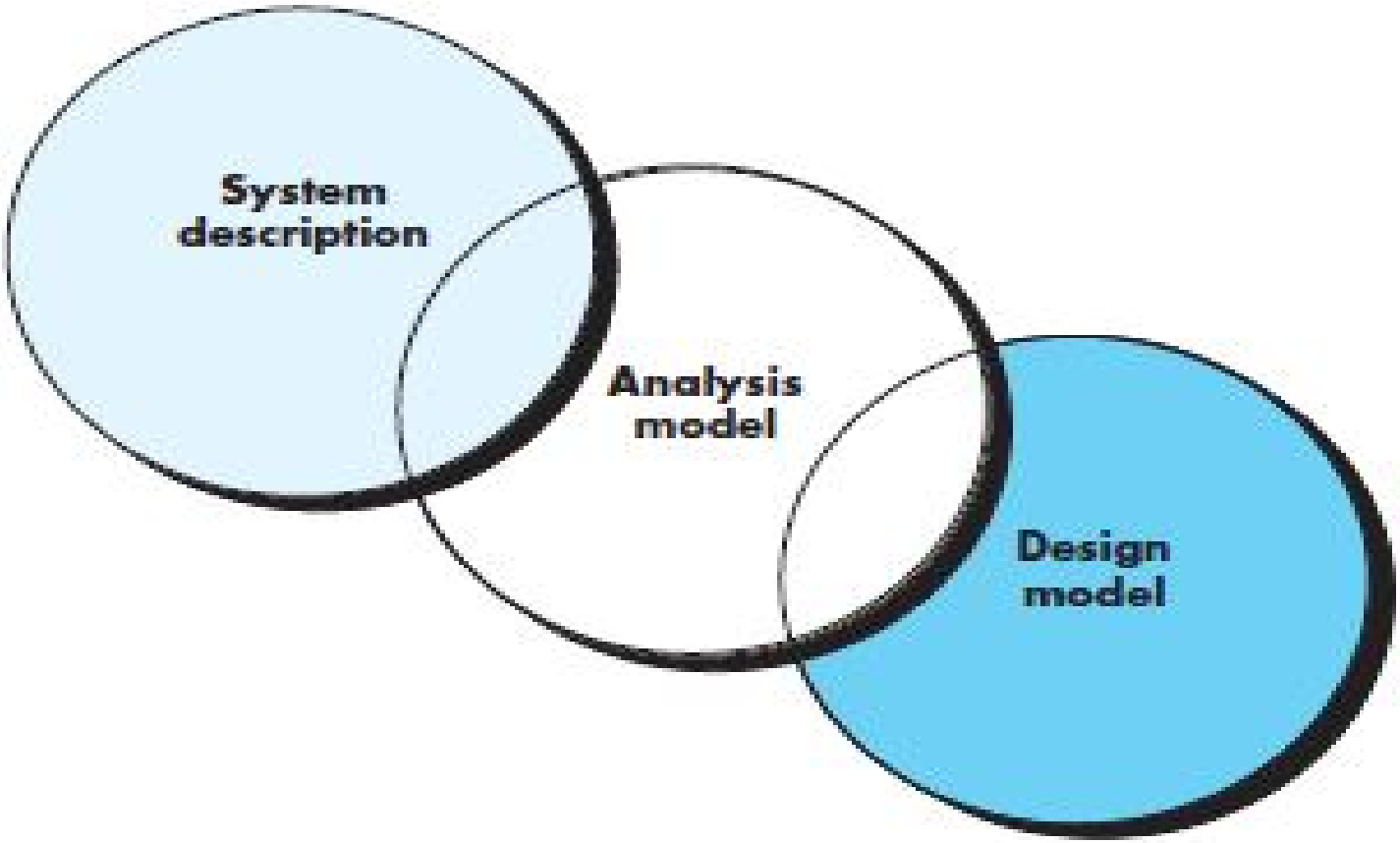


Figure 1. The requirements model as a bridge between the system description and the design model

Analysis Rules of Thumb

- A number of worthwhile rules of thumb that should be followed when creating the analysis model
- The model should focus on requirements that are visible within the problem or business domain. The level of abstraction should be relatively high.
- Each element of the requirements model should add to an overall understanding of software requirements and provide insight into the information domain, function, and behavior of the system.
- Delay consideration of infrastructure and other nonfunctional models until design. That is, a database may be required, but the classes necessary to implement it, the functions required to access it, and the behavior that will be exhibited as it is used should be considered only after problem domain analysis has been completed.

Analysis Rules of Thumb

- Minimize coupling throughout the system. It is important to represent relationships between classes and functions. However, if the level of "interconnectedness" is extremely high, efforts should be made to reduce it.

- Be certain that the requirements model provides value to all stakeholders. Each constituency has its own use for the model. For example, business stakeholders should use the model to validate requirements; designers should use the model as a basis for design; QA people should use the model to help plan acceptance tests.

- Keep the model as simple as it can be. Don't add additional diagrams when they add no new information. Don't use complex notational forms when a simple list will do.

Creating A Behavioral Model

- It is time to make a transition to the dynamic behavior of the system or product.
- To accomplish this, you can represent the behavior of the system as a function of specific events and time.
- The behavioral model indicates how software will respond to external events or stimuli.

Creating A Behavioral Model

- To create the model, you should perform the following steps:
- (1) evaluate all use cases to fully understand the sequence of interaction within the system,
- (2) identify events that drive the interaction sequence and understand how these events relate to specific objects,

Creating A Behavioral Model

- (3) create a sequence for each use case,
- (4) build a state diagram for the system, and
- (5) review the behavioral model to verify accuracy and consistency. Each of these steps is discussed in the sections that follow.

Patterns of Requirement Modeling

- Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered.
- In some cases, the domain knowledge is applied to a new problem within the same application domain.
- In other cases, the domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.

Patterns of Requirement Modeling

- Once the pattern has been discovered, it is documented by describing "explicitly the general problem to which the pattern is applicable, the prescribed solution, assumptions and constraints of using the pattern in practice, and often some other information about the pattern, such as the motivation and driving forces for using the pattern, discussion of the pattern's advantages and disadvantages, and references to some known examples of using that pattern in practical applications"

Discovering Analysis Patterns

- The requirements model comprises a wide variety of elements: scenario based (use cases), class-based (objects and classes), and behavioral (events and states).
- Each of these elements represents the problem from a different perspective, and each provides an opportunity to discover patterns that may occur throughout an application domain, or by analogy, across different application domains.
- The most basic element in the description of a requirements model is the use case.

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

February 27, 2023

Interaction Modeling- Use Case Models

- Actors - An actor is direct external user of the system
- An object or set of objects that communicates directly with the system but that is not part of the system.
- Each actor represent those objects that behave in particular way toward the system
- For example, our project management system involves various types of users, including project managers, resource managers, human resources, and system administrators. These users are all actors.

Interaction Modeling- Use Case Models

- An actor is external to a system, interacts with the system, may be a human user or another system, and has goals and responsibilities to satisfy in interacting with the system.
- Actors address the question of who and what interacts with a system.
- An actor is shown as a "stick figure" icon

Interaction Modeling- Use Case Models

- Various actors associated with the project management system
- A project manager - Responsible for ensuring that a project delivers a quality product within specified time and cost, and within specified resource constraints
- A resource manager - Responsible for ensuring that trained and skilled human resources are available for projects
- A human resource - Responsible for ensuring that worker skills are maintained, and that quality work is completed for a project.
- A system administrator - Responsible for ensuring that a project management system is available for a project

Interaction Modeling- Use Case Models

- For example, the actors for the vending machine are
- A vending machine is an automated machine that provides items such as snacks, beverages, tickets, etc. to consumers

Interaction Modeling- Use Case Models

- For example, the actors for the vending machine are
- A vending machine is an automated machine that provides items such as snacks, beverages, tickets, etc. to consumers
- customer and

Interaction Modeling- Use Case Models

- For example, the actors for the vending machine are
- A vending machine is an automated machine that provides items such as snacks, beverages, tickets, etc. to consumers
- customer and
- repair technician.

Interaction Modeling- Use Case Models

- For a travel agency system, actors might include
- A travel agency is a private retailer or public service that provides travel and tourism-related services to the general public

Interaction Modeling- Use Case Models

- For a travel agency system, actors might include
- A travel agency is a private retailer or public service that provides travel and tourism-related services to the general public
- traveler

Interaction Modeling- Use Case Models

- For a travel agency system, actors might include
- A travel agency is a private retailer or public service that provides travel and tourism-related services to the general public
- traveler
- agent and

Interaction Modeling- Use Case Models

- For a travel agency system, actors might include
- A travel agency is a private retailer or public service that provides travel and tourism-related services to the general public
- traveler
- agent and
- airline

Interaction Modeling- Use Case Models

- For a travel agency system, actors might include
- A travel agency is a private retailer or public service that provides travel and tourism-related services to the general public

Interaction Modeling- Use Case Models

- For a travel agency system, actors might include
- A travel agency is a private retailer or public service that provides travel and tourism-related services to the general public
- traveler

Interaction Modeling- Use Case Models

- For a travel agency system, actors might include
- A travel agency is a private retailer or public service that provides travel and tourism-related services to the general public
- traveler
- agent and

Interaction Modeling- Use Case Models

- For a travel agency system, actors might include
- A travel agency is a private retailer or public service that provides travel and tourism-related services to the general public
- traveler
- agent and
- airline

Interaction Modeling- Use Case Models

- For a computer database system, actors might include

Interaction Modeling- Use Case Models

- For a computer database system, actors might include
- user and

Interaction Modeling- Use Case Models

- For a computer database system, actors might include
- user and
- administrator
- Actors can be persons, devices, and other systems — anything that interacts directly with the system

Interaction Modeling- Use Case Models

- An object can be bound to multiple actors if it has different facets to its behavior
- For example, the objects Mary, Frank, and Paul may be customers of a vending machine.
- Paul may also be a repair technician for the vending machine.
- An actor has a single well-defined purpose.

Interaction Modeling- Use Case Models

- In contrast, objects and classes often combine many different purposes.
- An actor represents a particular facet of objects in its interaction with a system.
- The same actor can represent objects of different classes that interact similarly toward a system
- For example, even though many different individual persons use a vending machine, their behavior toward the vending-machine can all be summarized by the actors customer and repair technician.
- Each actor represents a coherent set of capabilities for its objects.

Interaction Modeling- Use Case Models

- Modeling the actors helps to define a system by identifying the objects within the system and those on its boundary.
- An actor is directly connected to the system an indirectly connected object is not an actor and should not be included as part of the system model.
- Any interactions with an indirectly connected object must pass through the actors
- For example, the dispatcher of repair technicians from a service bureau is not an actor of a vending machine—only the repair technician interacts directly with the machine.

Interaction Modeling- Use Case Models

- Modeling the actors helps to define a system by identifying the objects within the system and those on its boundary.
- An actor is directly connected to the system an indirectly connected object is not an actor and should not be included as part of the system model.
- Any interactions with an indirectly connected object must pass through the actors
- For example, the dispatcher of repair technicians from a service bureau is not an actor of a vending machine—only the repair technician interacts directly with the machine.

Interaction Modeling- Use Case Models

- If it is necessary to model the interactions among such indirect objects, then a model should be constructed of the environment itself as a larger system.

- For example, it might be useful to build a model of a repair service that includes dispatchers, repair technicians, and vending machines as actors, but that is a different model from the vending machine model.

Interaction Modeling- Use Case Models

- The various interactions of actors with a system are quantized into use cases.
- A use case is a coherent piece of functionality that a system can provide by interacting with actors
- For example, a customer actor can buy a beverage from a vending machine.
- The customer inserts money into the machine, makes a selection, and ultimately receives a beverage
- Similarly, a repair technician can perform scheduled maintenance on a vending machine.

Interaction Modeling- Use Case Models

- Each use case involves one or more action as well as the system itself.
- The use case buy a beverage involves the customer actor and the use case perform scheduled maintenance involves the repair technician actor.
- In a telephone system, the use case make a call involves two actors, a caller and a receiver.

Interaction Modeling- Use Case Models

- The actors need not all be persons.
- The use-case make a trade on an online stock broker involves a customer actor and a stock exchange actor.
- The stock broker system needs to communicate with both actors to execute a trade.

Interaction Modeling- Use Case Models

- A usecase involves a sequence of messages among the system and its actor.
- For example in the buy a beverage use case, the customer first inserts a coin and the vending machine displays the the amount deposited.
- This can be repeated several times.
- Then the customer pushes a button to indicate a selection; the vending machine dispenses the beverage and issue change, if necessary.

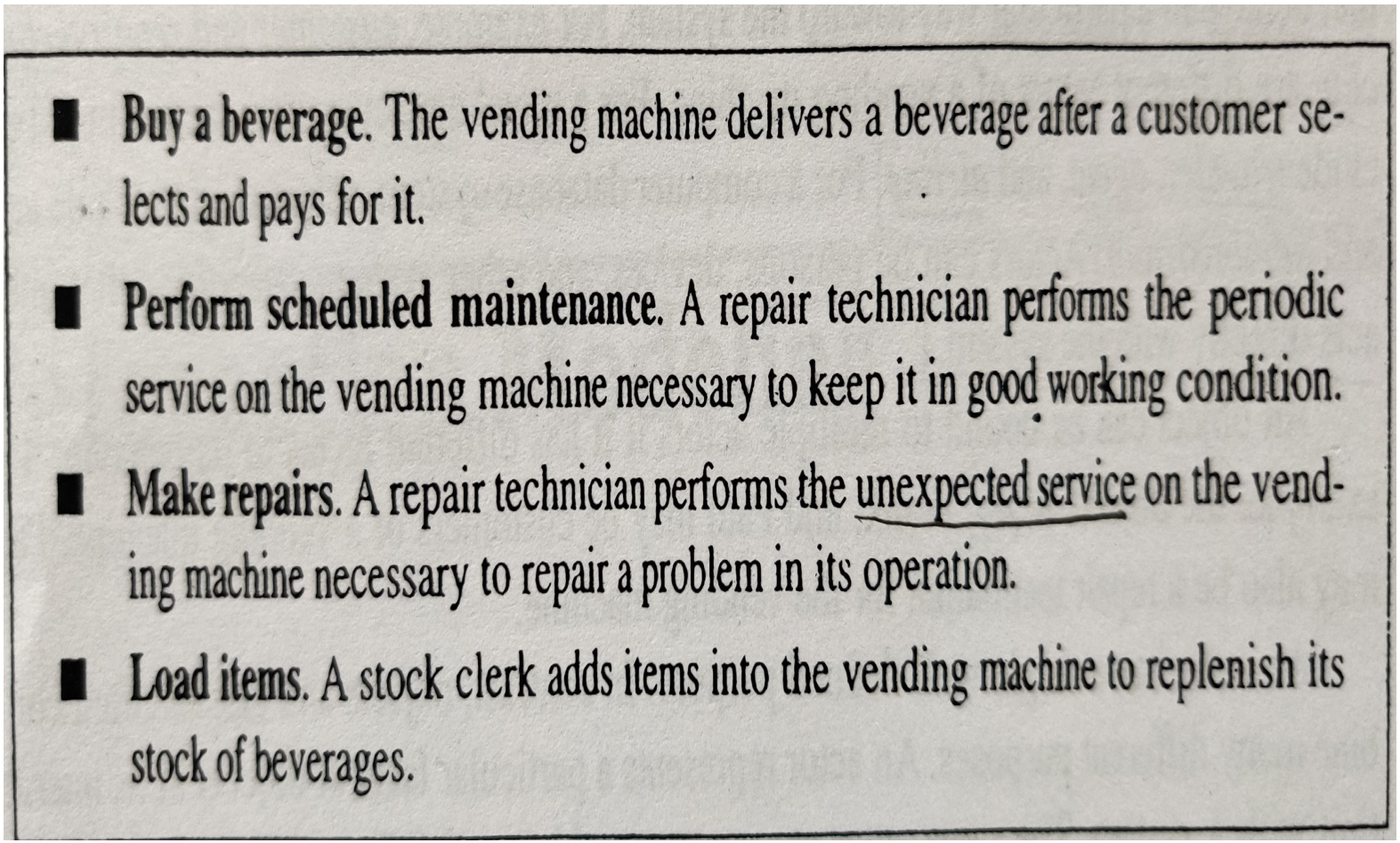
Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

February 21, 2022

Interaction Modeling- Use Case Models



- **Buy a beverage.** The vending machine delivers a beverage after a customer selects and pays for it.
- **Perform scheduled maintenance.** A repair technician performs the periodic service on the vending machine necessary to keep it in good working condition.
- **Make repairs.** A repair technician performs the unexpected service on the vending machine necessary to repair a problem in its operation.
- **Load items.** A stock clerk adds items into the vending machine to replenish its stock of beverages.

Figure 1. Usecase Summaries for a Vending Machine

Interaction Modeling- Use Case Models

- Some use cases have a fixed sequence of messages.
- More often, however, the message sequence may have some variations.
- For example, a customer-can deposit a variable number of coins in the buy a beverage use case. Depending on the money inserted and the item selected, the machine may, or may not, return change.
- You can represent such variability by showing several examples of distinct behavior sequences.

Interaction Modeling- Use Case Models

- Typically you should first define a mainline behavior sequence, then define optional subsequences, repetitions, and other variations.
- Error conditions are also part of a use case.
- For example, if the customer selects a beverage whose supply is exhausted, the vending machine displays a warning message.
- Similarly, the vending transaction can be cancelled.

Interaction Modeling- Use Case Models

- A use case brings together all of the behavior relevant to a slice of system functionality. This includes normal mainline behavior, variations on normal behavior, exception conditions, error conditions, and cancellations of a request.
- Figure explains the buy a beverage use case in detail.
- Grouping normal and abnormal behavior under a single use case helps to ensure that all the consequences of an interaction are considered together.

Interaction Modeling- Use Case Models

Use Case: Buy a beverage

Summary: The vending machine delivers a beverage after a customer selects and pays for it.

Actors: Customer

Preconditions: The machine is waiting for money to be inserted.

Description: The machine starts in the waiting state in which it displays the message "Enter coins." A customer inserts coins into the machine. The machine displays the total value of money entered and lights up the buttons for the items that can be purchased for the money inserted. The customer pushes a button. The machine dispenses the corresponding item and makes change, if the cost of the item is less than the money inserted.

Exceptions:

Canceled: If the customer presses the cancel button before an item has been selected, the customer's money is returned and the machine resets to the waiting state.

Out of stock: If the customer presses a button for an out-of-stock item, the message "That item is out of stock" is displayed. The machine continues to accept coins or a selection.

Insufficient money: If the customer presses a button for an item that costs more than the money inserted, the message "You must insert $*nn.nn* more for that item" is displayed, where *nn.nn* is the amount of additional money needed. The machine continues to accept coins or a selection.

No change: If the customer has inserted enough money to buy the item but the machine cannot make the correct change, the message "Cannot make correct change" is displayed and the machine continues to accept coins or a selection.

Postconditions: The machine is waiting for money to be inserted.

Figure 2. Use case Description: A use case brings together all of the behavior relevant to a slice of system functionality

Use Case Diagrams

- A system involves a set of use cases and a set of actors.
- Each use case represents a slice of the functionality the system provides.
- The set of use cases shows the complete functionality of the system at some level of detail.
- Similarly, each actor represents one kind of object for which the system can perform behavior.
- The set of actors represents the complete set of objects that the system can serve.

Use Case Diagrams

- The Unified Modeling Language (UML) has a graphical notation for summarizing use cases and figure shows an example.
- A rectangle contains the use cases for a system with the actors listed on the outside.
- The name of the system may be written near a side of the rectangle.
- A name within an ellipse donates a use case.
- A "stick man" icon denotes an actor, with the name being placed below or adjacent to the icon.

Use Case Diagrams

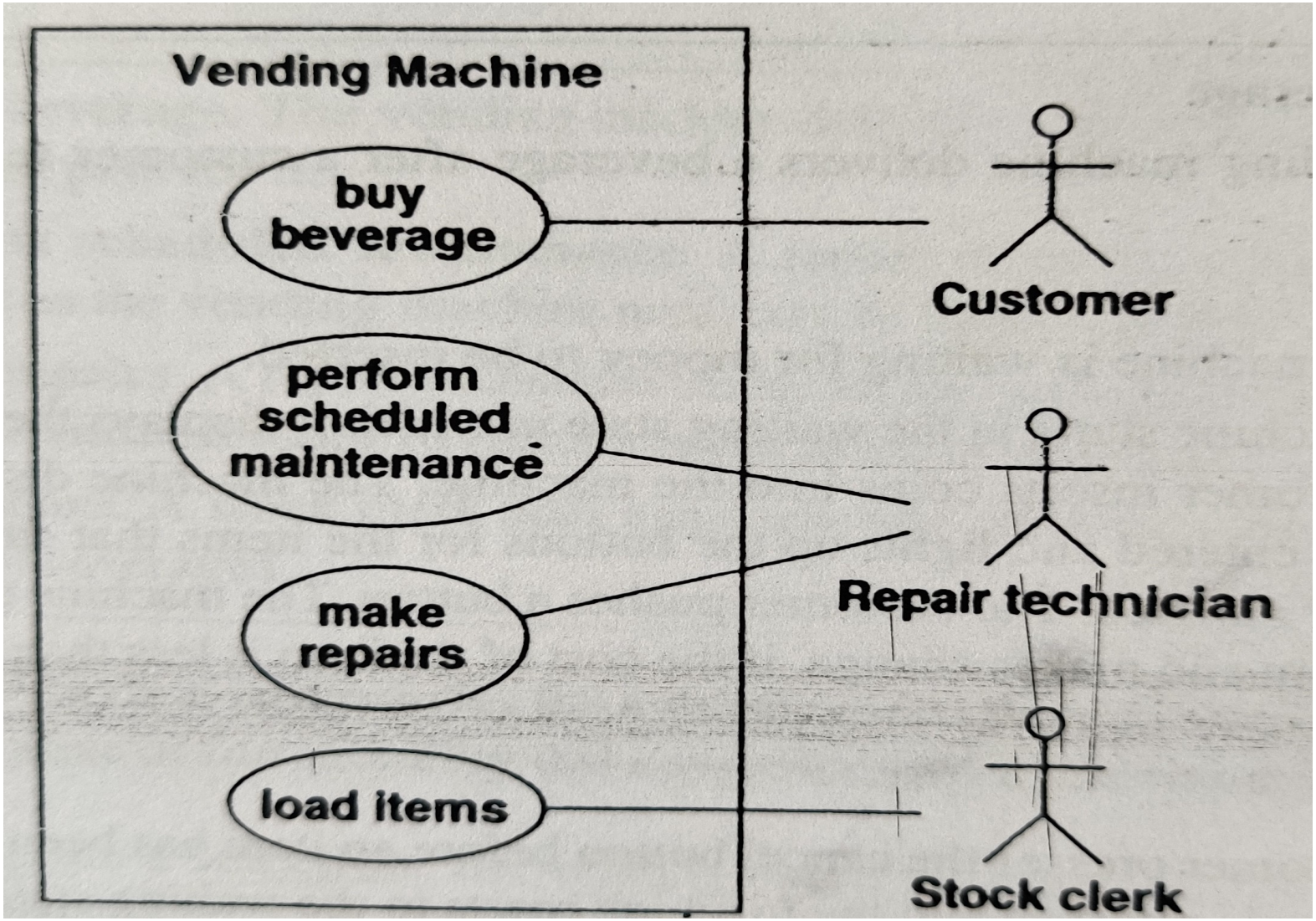


Figure 3. Use case Diagram for vending machine. A system involves a set of use cases and set of actors

Use Case Diagrams

- Solid lines connect use cases to participating actors.
- In the figure, the actor, Repair technician participates in two use cases, the others in one each.
- Multiple actors can participate in a use case even though the example has only one actor per use case.

Software Engineering (DS 203)

Dr. Manjunath K Vanahalli

Department of Data Science and Artifical Intelligence
Indian Institute of Information Technology Dharwad, India

February 21, 2022

Guidelines for Use Case Models

- Use cases identify the functionality of a system and organize it according to the perspective of users.
- Here are some guidelines for constructing use case models. variations.
- **First determine the system boundary**. It is impossible to identify use cases or actors if the system boundary is unclear.
- **Ensure that actors are focused**. Each actor should have a single, coherent purpose.

Guidelines for Use Case Models

- If a real-world object embodies multiple purposes, capture them with separate actors.
- For example, the owner of a personal computer may install software, set up a database, and send email variations.
- These functions differ greatly in their impact on the computer system and the potential for system damage.
- They might be broken into three actors: system administrator, database administrator, and computer user

Guidelines for Use Case Models

- Each use case must provide value to users. A use case should represent a complete transaction that provides value to users and should not be defined too narrowly.
- For example, dial a telephone number is not a good use case for a telephone system.
- It does not represent a complete transaction of value by itself; it is merely part of the use case make telephone call.
- The latter use case involves placing the call, talking, and terminating the call.

Guidelines for Use Case Models

- **Relate use cases and actors.**
- Every use case should have at least one actor, and every actor should participate in at least, one use case.
- A use case may involve several actors, and an actor may participate in several use cases.

Guidelines for Use Case Models

- **Remember that use cases are informal.**
- It is important not to be obsessed by formalism in specifying use cases.
- They are not intended as a formal mechanism but as a way to identify and organize system functionality from a user-centered point of view.
- It is acceptable if use cases are a bit loose at first

Sequence Models

- The sequence model elaborates the themes of use cases.
- There are two kinds of sequence models: scenarios and a more structured format called sequence diagrams.

Scenarios

- A scenario is a sequence of events that occurs during one particular execution of a system, such as for a use case.
- There are two kinds of sequence models: scenarios and a more structured format called sequence diagrams.
- The scope of a scenario can vary; it may include all events in the system, or it may include only those events impinging on or generated by certain objects.
- A scenario can be the historical record of executing an actual system or a thought experiment of executing a proposed system.

Scenarios

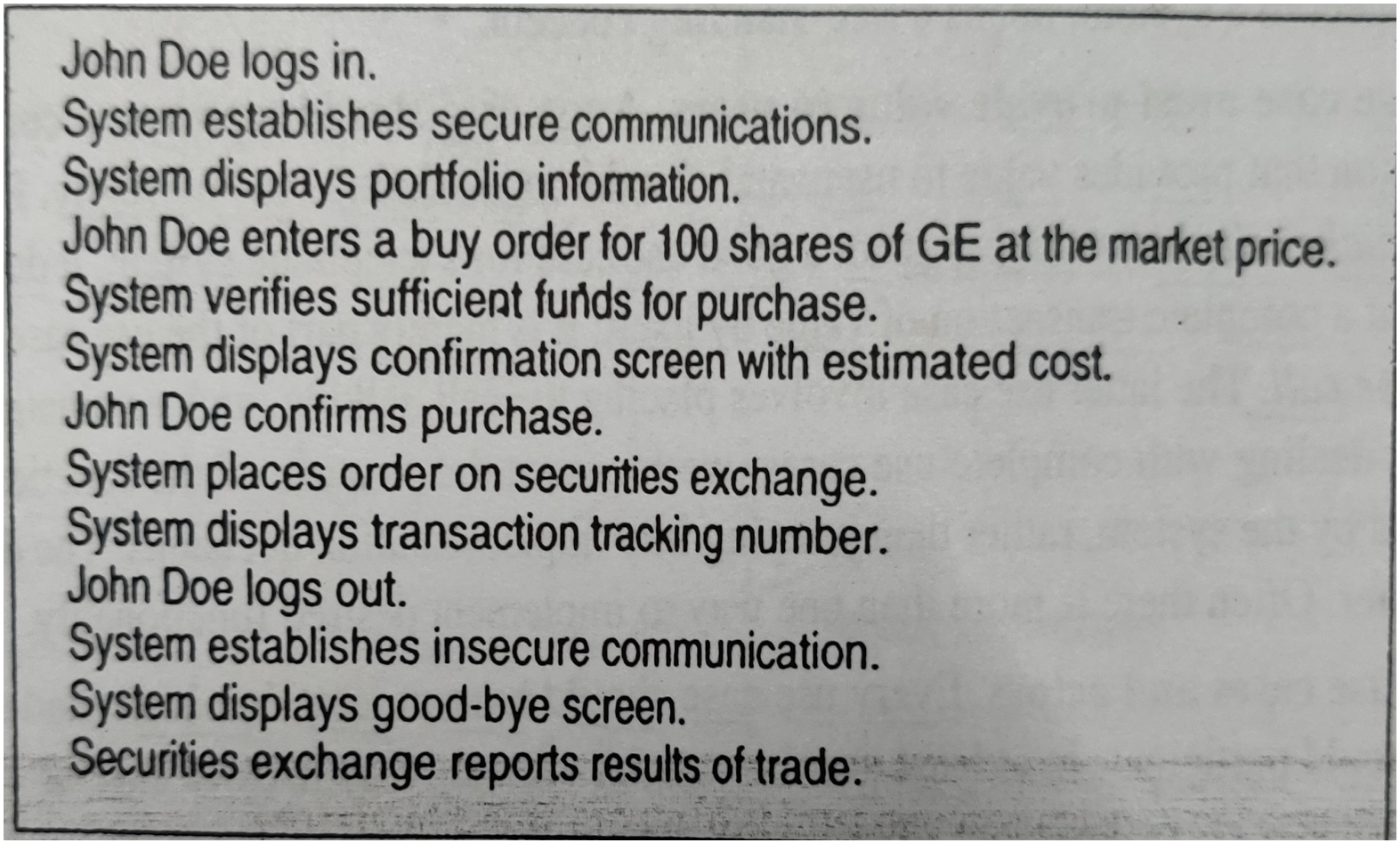


Figure 1. Scenario for session with an online stock broker: A scenario is as sequence of events that occurs during one particular execution of a system.

Scenarios

- A scenario can be displayed as a list of text statements as Figure illustrates.
- In this example, John Doe logs on with an online stock broker system, places an order for GE stock, and then logs off.
- Sometime later, after the order is executed, the securities exchange reports the results of the trade to the broker system.
- John Doe will see the results on the next login, but that is not part of this scenario.

Scenarios

- A scenario contains messages between objects as well as activities performed by objects.
- Each message transmits information from one object to another.
- The first step of writing a scenario is to identify the objects exchanging messages.
- Then you must determine the sender and receiver of each message, as well as the sequence of the messages.
- Finally, you can add activities for internal computations as scenarios are reduced to code.

Sequence Diagram

- A text format is convenient for writing, but it does not clearly show the sender and receiver of each 'message, especially if there are more than two objects.
- A sequence diagram shows the participants in an interaction and the sequence of messages among them.
- A sequence diagram shows the interaction of a system with its actors to perform all or part of a use case.

Sequence Diagram

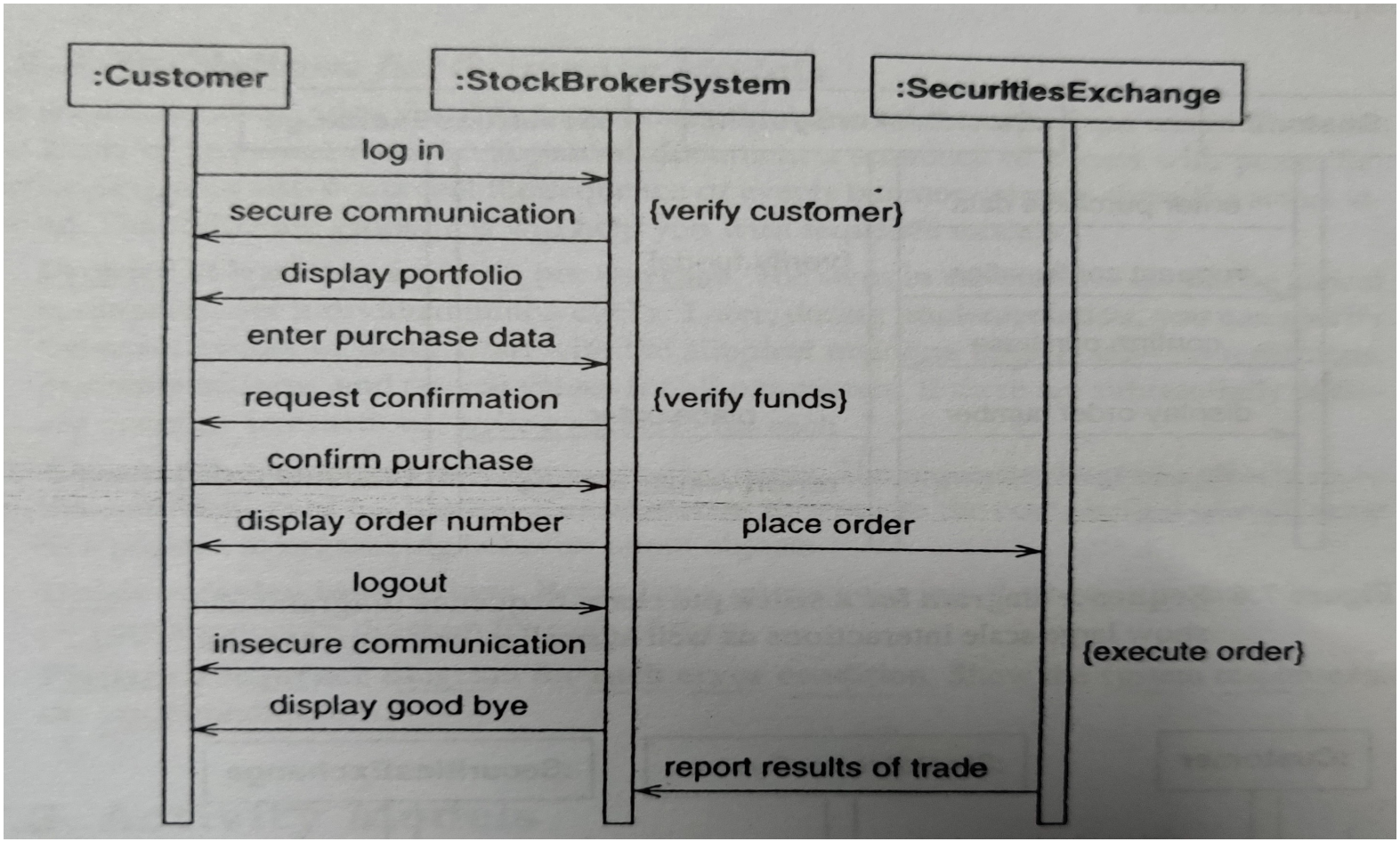


Figure 2. Sequence Diagram for a session with an online stock broker. A sequence diagram shows the participants in an interaction and a sequence of message among them

Sequence Diagram

- Figure shows a sequence diagram corresponding to the previous stock broker scenario.
- Each actor as well as the system is represented by a vertical line called a lifeline and each message by a horizontal arrow from the sender to the receiver.
- Time proceeds from top to bottom, but the spacing is irrelevant; the diagram shows only the sequence of messages, not their exact timing.
- Note that sequence diagrams can show concurrent signals—stock broker system sends messages to customer and securities exchange concurrently and signals between participants need not alternate stock broker system sends secure communication followed by display portfolio.

Sequence Diagram

- Each use case requires one or more sequence diagrams to describe its behavior.
- Each sequence diagram shows a particular behavior sequence of the use case.
- It is best to show a specific portion of a use case and not attempt to be too general.
- Sequence diagrams can show large-scale interactions, such as an entire session with the stock broker system, but often such interactions contain many independent tasks that can be combined in various ways.

Sequence Diagram

- Rather than repeating information, you can draw a separate sequence diagram for each task.
- For example, Figure 3 and Figure 4 show an order to purchase a stock and a request for a quote on a stock.
- These and various other tasks (not shown) would fit within an entire stock trading session.

Sequence Diagram

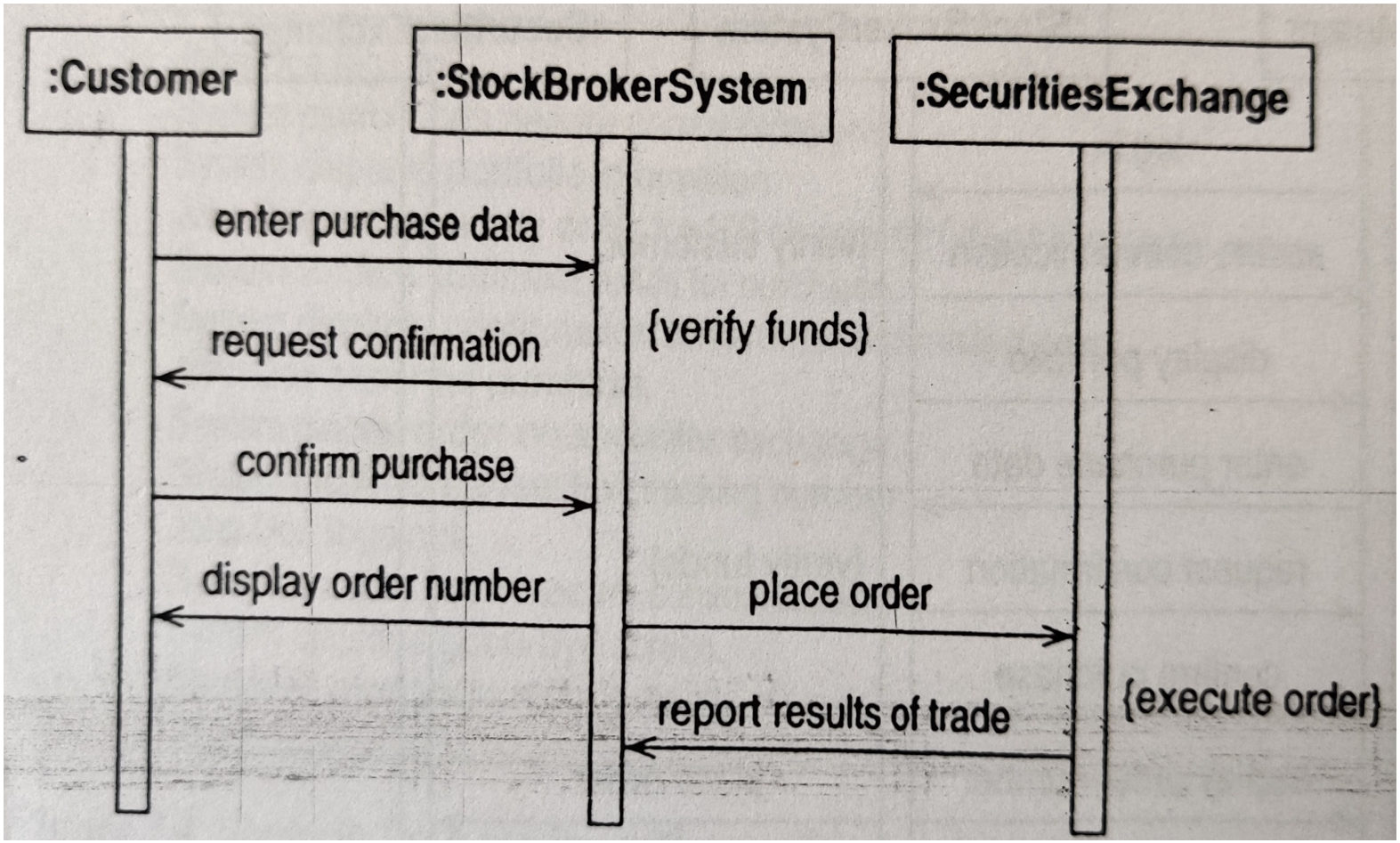


Figure 3. Sequence Diagram for a stock purchase. Sequence diagram can show large-scale interactions as well as smaller, constitute tasks

Sequence Diagram

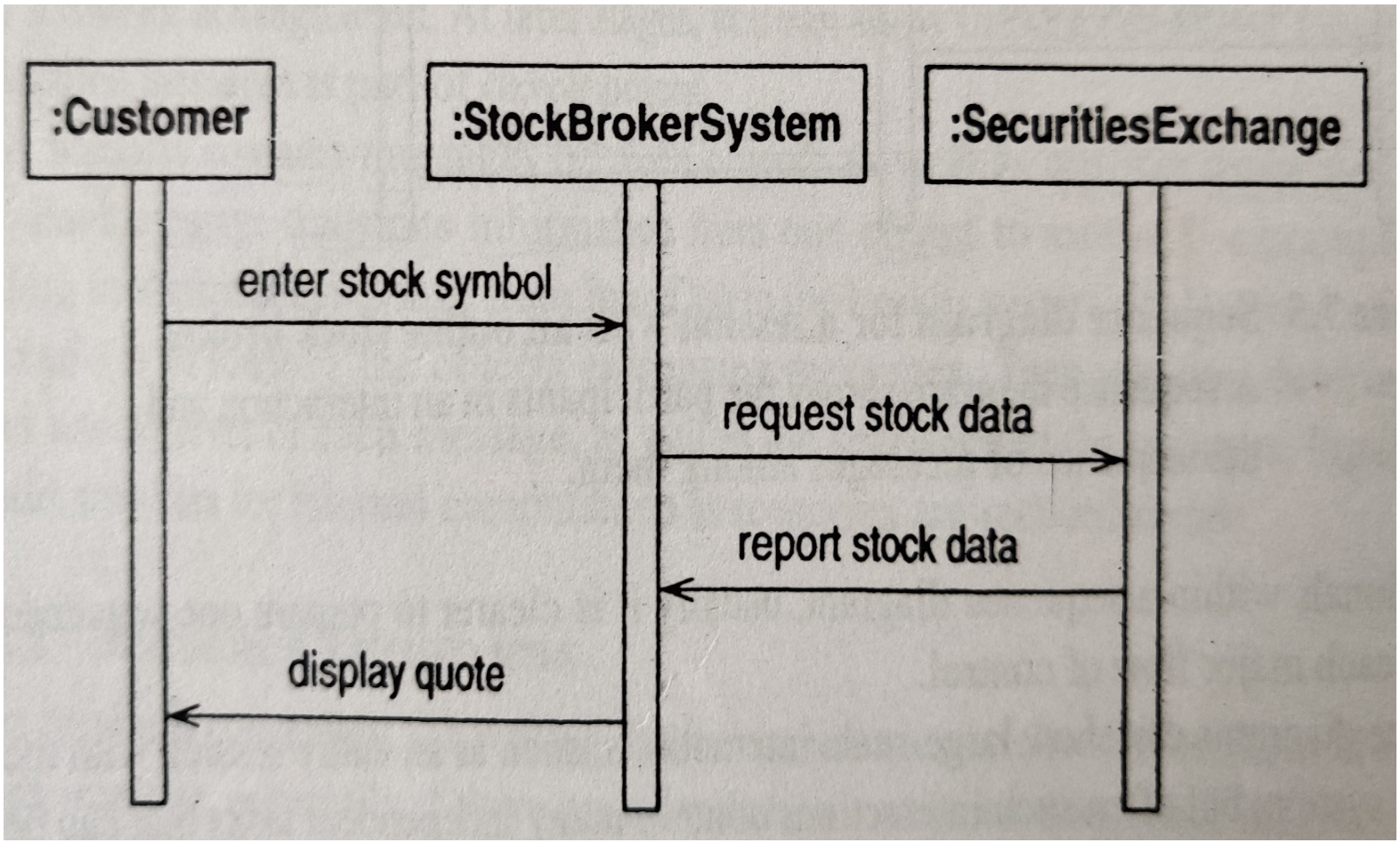


Figure 4. Sequence Diagram for a stock quote

Sequence Diagram

- You should also prepare a sequence diagram for each exception condition within the use case.
- For example, Figure 4 shows a variation in which the customer does not have sufficient funds to place the order.
- In this example, the customer cancels the order.

Sequence Diagram

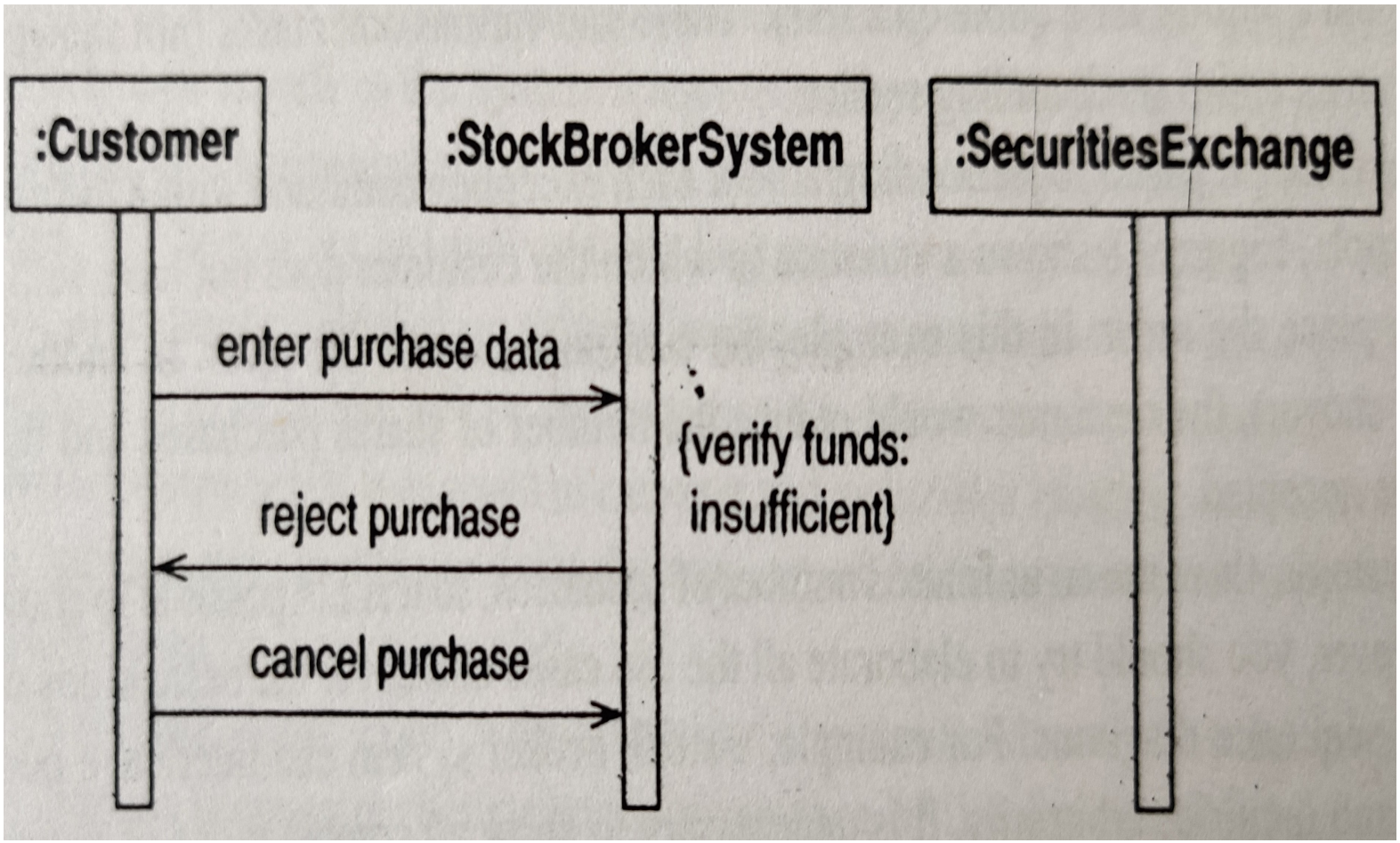


Figure 5. Sequence Diagram for a stock purchase that fails

Sequence Diagram

- In another variation (not shown), the customer would reduce the number of shares purchased and the order would be accepted.
- In most systems, there are an unlimited number of scenarios, so it is not possible to show them all.
- However, you should try to elaborate all the use cases and cover the basic kinds of behavior with sequence diagrams.
- For example, a stock broker system can interleave purchases, sales, and inquiries arbitrarily. It is unnecessary to show all combinations of activities, once the basic pattern is established.

Guidelines for Sequence Models

- The sequence model adds detail and elaborates the informal themes of use cases.
- There are two kinds of sequence models.
- Scenarios document a sequence of events in the form of spoken or written language.
- Sequence diagrams also document the sequence of events but more clearly show the actors involved.

Guidelines for Sequence Models

- The following guidelines will help you with sequence models.
- **Prepare at least one scenario per use case**:
- The steps in the scenario should be logical commands, not individual button clicks.
- Later, during implementation, you can specify the exact syntax of input.
- Start with the simplest mainline interaction-no repetitions, one main activity, and typical values for all parameters.

Guidelines for Sequence Models

- **Abstract the scenarios into sequence diagrams:**
- The sequence diagrams clearly show the contribution of each actor.
- It is important to separate the contribution of each actor as a event to organizing behavior about objects.

Guidelines for Sequence Models

- **Divide complex interactions:**
- Break large interactions into their constituent task and prepare a sequence diagram for each of them
- **Prepare a sequence diagram for each error condition:**
- Show the system response to the error condition.