Course Name : Software Engineering

BITS Pilani
Pilani Campus

T V Rao
Work-Integrated Learning Programme

What is Software?

1) instructions (programs) that when executed provide desired function and performance
2) data structures that enable the programs to adequately manipulate information
3) documents that describe the operation and use of the programs

- A ***logical*** rather than ***physical*** entity

Several Roles of Software

- Software can be a product and a vehicle for delivering a product
  - Product
    - Delivers computing potential
    - Produces, manages, acquires, modifies, display, or transmits information
  - Vehicle
    - Supports or directly provides system functionality
    - Controls other programs (e.g., operating systems)
    - Effects communications (e.g., networking software)
    - Helps build other software (e.g., software tools)
- Software bridges gap between the way a bare machine thinks and the way humans think

Software Products

innovate achieve lead

- **Generic products**
  - Stand-alone systems which are produced by a development organization and sold on the open market to any customer
  - The specification of what the software should do is owned by the software developer and decisions on software change are made by the developer.
    - Examples – PC software such as word processor, project management tools; CAD software; software for specific markets such as appointments systems for dentists.
- **Bespoke (customized) products**
  - Systems which are commissioned by a specific customer and developed specially by some Contractor
  - The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.
    - Examples –e-commerce application of an organization, embedded software for anti-lock braking systems.
- **Most software expenditure is on generic products but most development effort is on bespoke systems**

Wide-ranging Software Applications

- System software
  - –Compiler, Operating System…
- Application software
  - –Railway reservation…
- Engineering/scientific Software
  - –Astronomy, automated manufacturing…
- Embedded software
  - –Braking system, Mobile phone…
- Product-line software
  - –Word processor, personal finance application…
- Web applications
  - –Social networks…
- AI software
  - –Pattern recognition, neural networks…

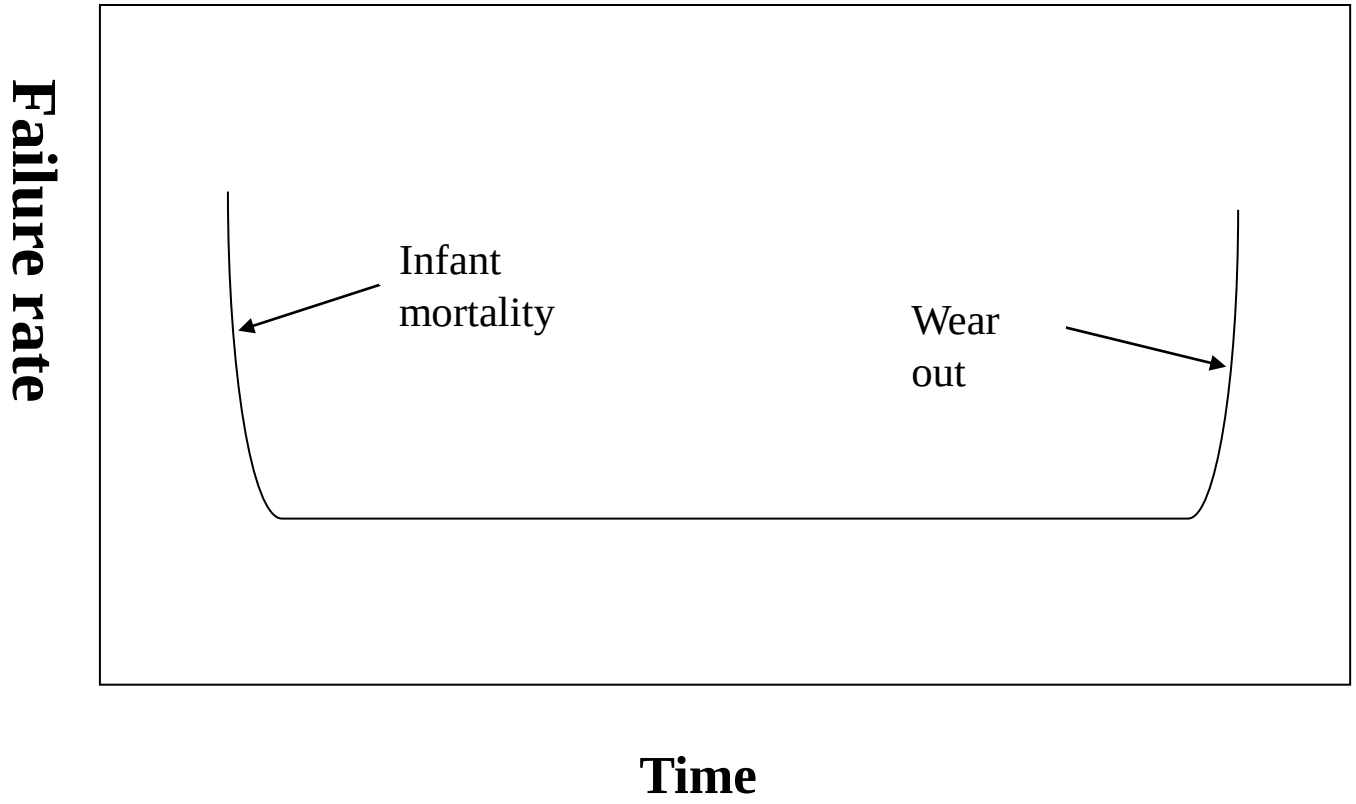
Software Characteristics

innovate achieve lead

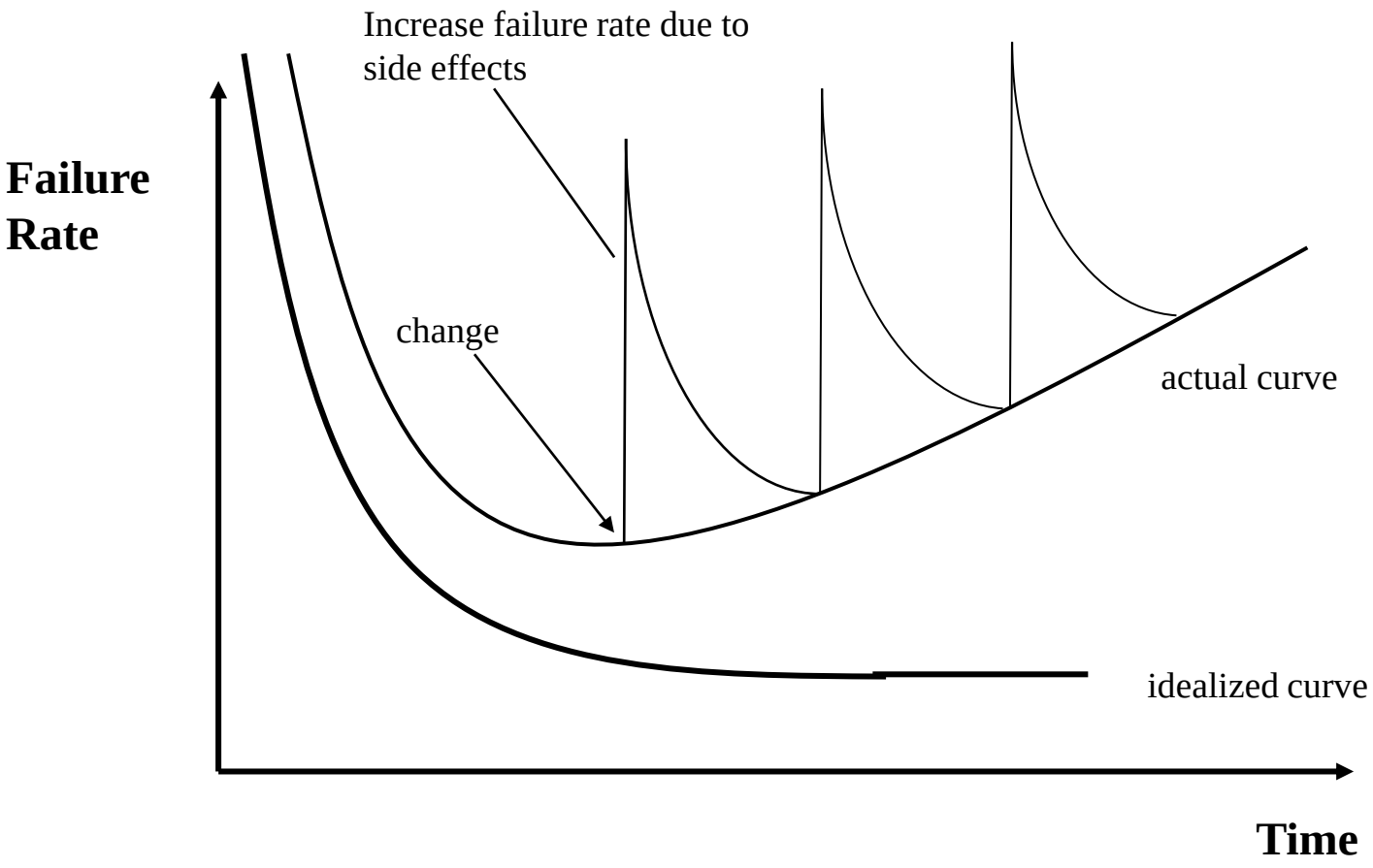
- Software is developed or engineered....Not manufactured in the classical sense
- Software does NOT “wear out”

  –Aging of software is very unlike material objects (hardware)
- Industry is moving towards component-based construction, but most of the software effort is for the custom-building

Failure ("Bathtub") Curve for Hardware



Software Deterioration



Evolving the Legacy Software

- (Adaptive) Must be adapted to meet the needs of new computing environments or more modern systems, databases, or networks
- (Perfective) Must be enhanced to implement new business requirements , improve performance etc.
- (Corrective) Must be changed because of errors found in the specification, design, or implementation

(Note: These are three major reasons for software maintenance, along with Preventive maintenence)

Characteristics of Contemporary Apps

(WebApps)

Network intensiveness. A WebApp resides on a network and must serve the needs of a diverse community of clients.

Concurrency. A large number of users may access the WebApp at one time.

Unpredictable load. The number of users of the WebApp may vary by orders of magnitude from day to day.

Performance. If a WebApp user must wait too long (for access, for server-side processing, for client-side formatting and display), he or she may decide to go elsewhere.

Availability. Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a “24/7/365” basis.

Data driven. The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end-user.

Characteristics of Contemporary Apps (Cont)

(WebApps)

Content sensitive. The quality and aesthetic nature of content remains an important determinant of the quality of a WebApp

Continuous evolution. Unlike conventional application software that evolves over a series of planned, chronologically-spaced releases, Web applications evolve continuously

Immediacy. Although *immediacy—the compelling need to get* software to market quickly—is a characteristic of many application domains, WebApps often exhibit a time to market that can be a matter of a few days or weeks

Security. Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end-users who may access the application

Aesthetics. An undeniable part of the appeal of a WebApp is its look and feel

Unchanging Questions

Some Questions About Software Haven't Changed Over the Decades

- 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?

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
    - Using appropriate theories and methods to solve problems bearing in mind organizational and financial constraints.
  - All aspects of software production
    - Not just technical process of development. Also project management and the development of tools, methods etc. to support software production.

FAQs on Software Engineering

Question	Answer
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.
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.
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.

Attributes of good software

innovate achieve lead

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilisation, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

What Lies Ahead?

Some trends that will exacerbate Hardware-software-human factors integration problems are

- ***Complex, multi-owner systems of systems.*** Current collections of incompatible, separately-developed systems will cause numerous challenges.
- ***Emergent requirements*****.** Demands for most appropriate user interfaces and collaboration modes for a complex human-intensive system.
- ***Rapid change*****.** Specifying current-point-in-time snapshot requirements on a cost-competitive contract generally leads to a big design up front, and a point-solution architecture that is hard to adapt to new developments.
- ***Reused components.*** Reuse-based development has major bottom-up development implications, and is incompatible with pure top-down requirements-first approaches.
- ***High assurance of qualities.*** Future systems will need higher assurance levels of such qualities as safety, security, reliability/availability/maintainability, performance, adaptability, interoperability, usability, and scalability.

- Barry Boehm and Jo Ann Lane, University of Southern California, 2007

Concern about Software Engineering:

Do we stand on quicksand or the shoulders of giants?

- Have you ever found that a new method or practice is just the re-branding and regurgitation of old?
- Do you find every new idea about software development seems to be at the expense and in aggressive competition with everything that has gone before?
- Does it seem to you that following that latest software development trend has become more important than producing great software?
- Have you noticed how in their hurry to forge ahead people seem to throw away the good with the bad? It is as though they have no solid knowledge to stand upon.
- Many teams carelessly discard expensive process and tool investments, almost before they have even tried them.
- Every time someone changes their job they have to learn a new approach before they can get on with the real task at hand. People cannot learn from experience as they are forever starting over.

- Ivar Jacobson, Ian Spencer "Why we need a theory for Software Engineering", 2009

Thank You…

Credits

- Software Engineering 7/ed by Roger Pressman – Reference

Course Name : Software Engineering

BITS Pilani
Pilani Campus

T V Rao

Software Engineering Definitions

Course Name : Software Engineering

BITS Pilani
Pilani Campus

T V Rao

Software Engineering Definitions

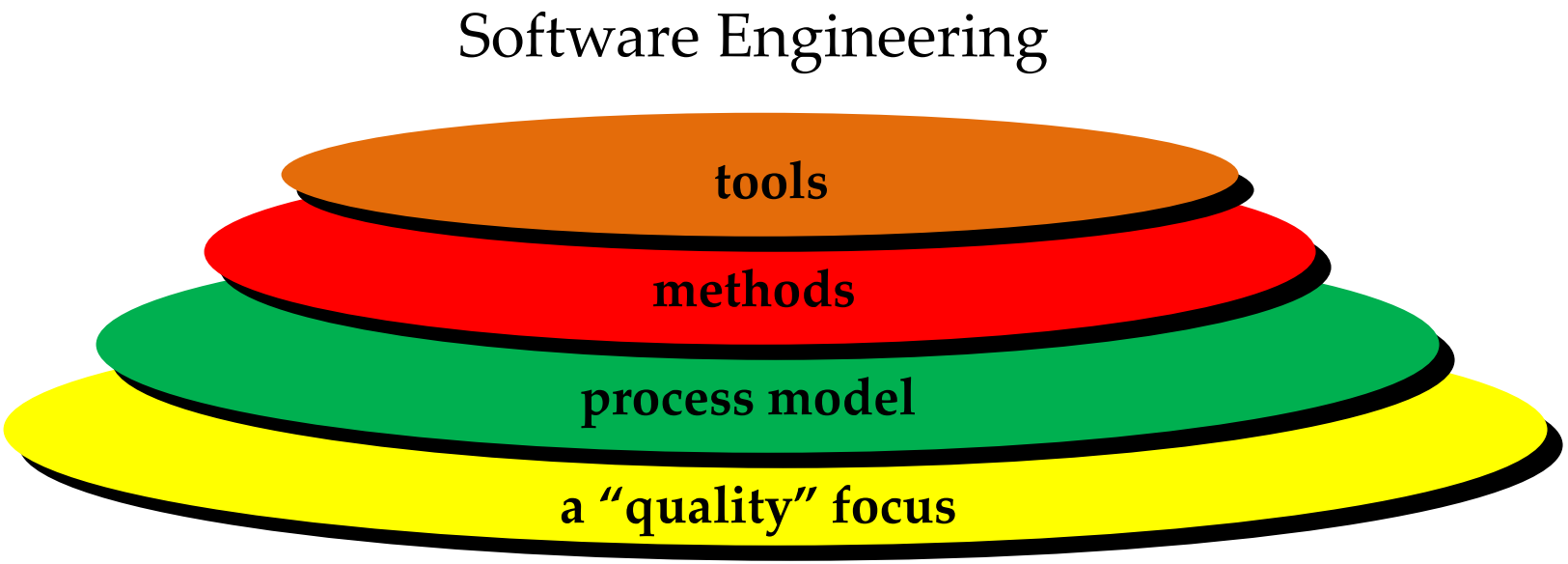
Software Engineering - Definitions

innovate achieve lead

- (1969 – Fritz Bauer) Software engineering is the establishment and use of *sound engineering principles* in order to obtain *economically* software that is *reliable* and works *efficiently* on *real machines*
  - Some may want to have technical aspects, customer satisfaction, timeliness, measurements, process included in the definition

- (IEEE) The application of a *systematic, disciplined, quantifiable* approach to the *development, operation, and maintenance* of software; that is, the application of engineering to software
  - Some may consider *adaptability and agility* more important than *systematic, disciplined, quantifiable approach*

A Layered Technology



Process, Methods, and Tools

- Process
  - Provides the glue that holds the layers together; enables rational and timely development; provides a framework for effective delivery of technology; forms the basis for management; provides the context for technical methods, work products, milestones, quality measures, and change management
- Methods
  - Provide the technical "how to" for building software; rely on a set of basic principles; encompass a broad array of tasks; include modeling activities
- Tools
  - Provide automated or semi-automated support for the process and methods (i.e., CASE tools)

A Process Framework

Software Process

Process framework

Framework activity 1

Framework activity n

Umbrella Activities

Process framework

Framework activities

- work tasks
- work products
- milestones & deliverables
- QA checkpoints

Umbrella Activities

A Process Framework

Process framework

Framework activities
work tasks
work products
milestones &
deliverables
QA checkpoints

Umbrella Activities

Process framework

Modeling activity

Software Engineering action: Analysis

work tasks: requirements gathering,
elaboration, negotiation,
specification, validation

work products: analysis model and/or
requirements specification

milestones & deliverables

QA checkpoints

Software Engineering action: Design

work tasks: data design, architectural,
interface design,
component design

work products: design model
and/or design specification

Umbrella Activities

Framework Activities

- Communication
  - Involves communication among the customer and other stake holders; encompasses requirements gathering
- Planning
  - Establishes a plan for software engineering work; addresses technical tasks, resources, work products, and work schedule
- Modeling (Analyze, Design)
  - Encompasses the creation of models to better understand the requirements and the design
- Construction (Code, Test)
  - Combines code generation and testing to uncover errors
- Deployment
  - Involves delivery of software to the customer for evaluation and feedback

Umbrella Activities

- Software project tracking and control
  - Assess progress against the plan
- Software quality assurance
  - Activities required to ensure quality
- Software configuration management
  - Manage effects of change
- Technical Reviews
  - Uncover errors before going to next activity
- Formal technical reviews
  - Assess work products to uncover errors

Umbrella Activities (contd)

- Risk management
  - Assess risks that may affect quality
- Measurement – process, project, product
- Reusability management (component reuse)
- Work product preparation and production
  - Models, documents, logs, forms, lists…

etc.

How Process Models Differ?

While all Process Models take same framework and umbrella activities, they differ with regard to

- Overall flow of activities, actions, and tasks and the interdependencies among them
- Degree to which actions and tasks are defined within each framework activity
- Degree to which work products are identified and required
- Manner in which quality assurance activities are applied
- Manner in which project tracking and control activities are applied
- Overall degree of detail and rigor with which the process is described
- Degree to which customer and other stakeholders are involved in the project
- Level of autonomy given to the software team
- Degree to which team organization and roles are prescribed

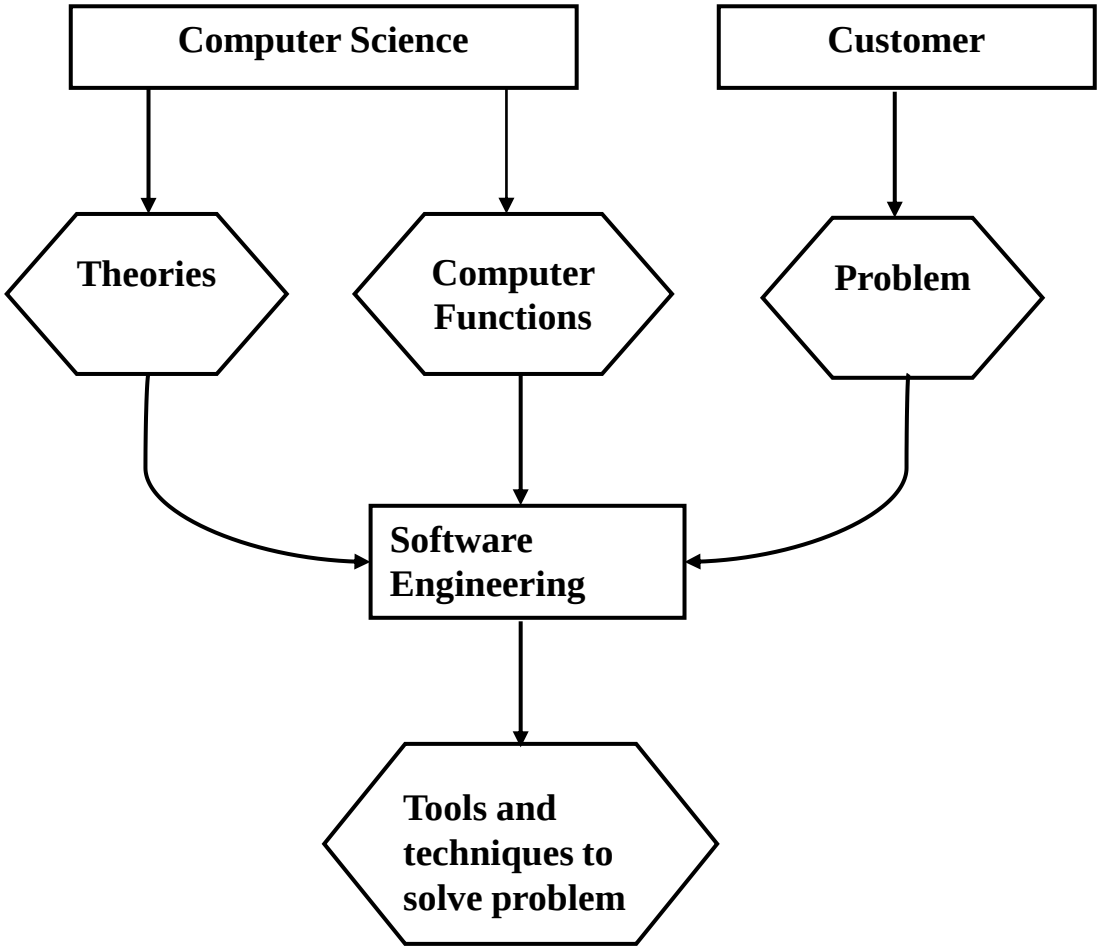
Process Patterns

- A proven solution to a problem
  - Describes process-related problem
  - Environment in which it was encountered
- A template to describe solution
  - Ambler proposed a template for describing a process pattern
- Can be defined at any level of abstraction
  - Complete process model (e.g. Prototyping)
  - Frame work activity (e.g. Planning)
  - An action within framework activity (e.g. Estimation)

Process Patterns – Ambler's Template

- Pattern Name
- Forces
  - Describes environment of the problem
- Type
  - Phase (e.g. Prototyping), Stage (e.g. Planning), or Task (e.g. Estimation)
- Initial Context
- Problem
- Solution
- Resulting Context
- Related Patterns
- Known uses & examples

Software Engineering Context



Seven Core Principles for Software Engineering (David Hooker – 1996)

1) Remember the reason that the software exists
   - The software should provide value to its users and satisfy the requirements
2) Keep it simple, stupid (KISS)
   - All design and implementation should be as simple as possible
3) Maintain the vision of the project
   - A clear vision is essential to the project's success
4) Others will consume what you produce
   - Always specify, design, and implement knowing that someone else will later have to understand and modify what you did
5) Be open to the future
   - Never design yourself into a corner; build software that can be easily changed and adapted
6) Plan ahead for software reuse
   - Reuse of software reduces the long-term cost and increases the value of the program and the reusable components
7) Think, then act
   - Placing clear, complete thought before action will almost always produce better results

Software Myths - Management

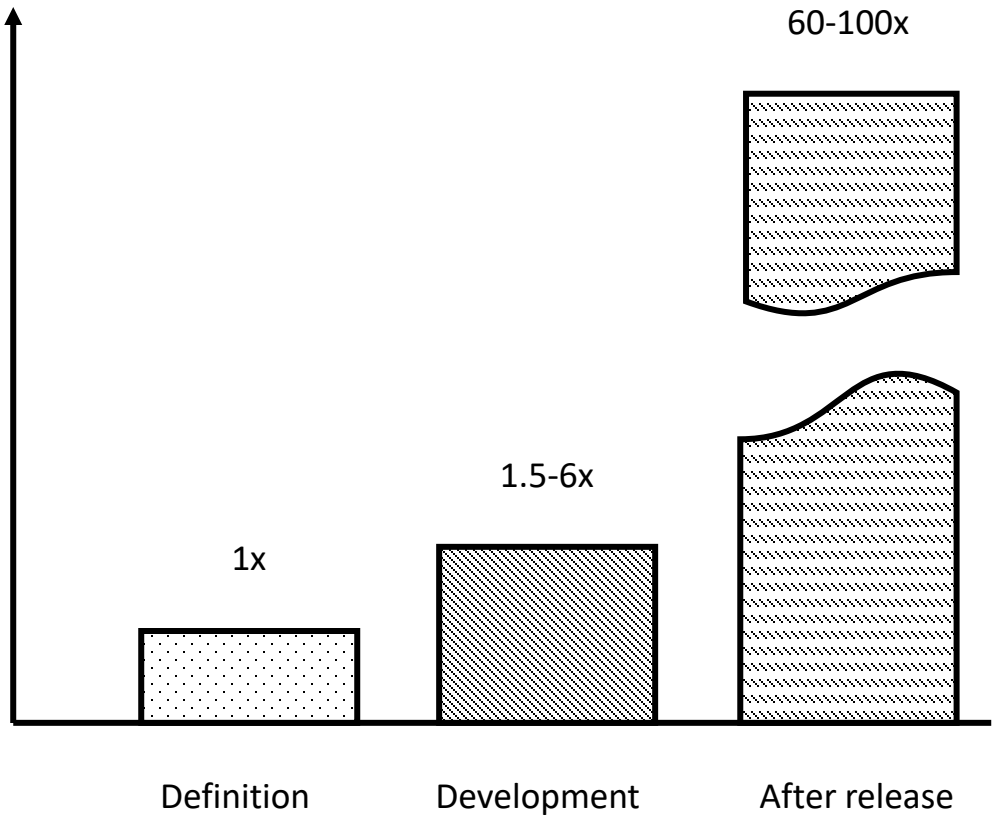
innovate achieve lead

- "We already have a book that is full of standards and procedures for building software. Won't that provide my people with everything they need to know?"
  - Not used, not up to date, not complete, not focused on quality, time, and money
- "If we get behind, we can add more programmers and catch up"
  - Adding people to a late software project makes it later
  - Training time, increased communication lines
- "If I decide to outsource the software project to a third party, I can just relax and let that firm build it"
  - Software projects need to be controlled and managed

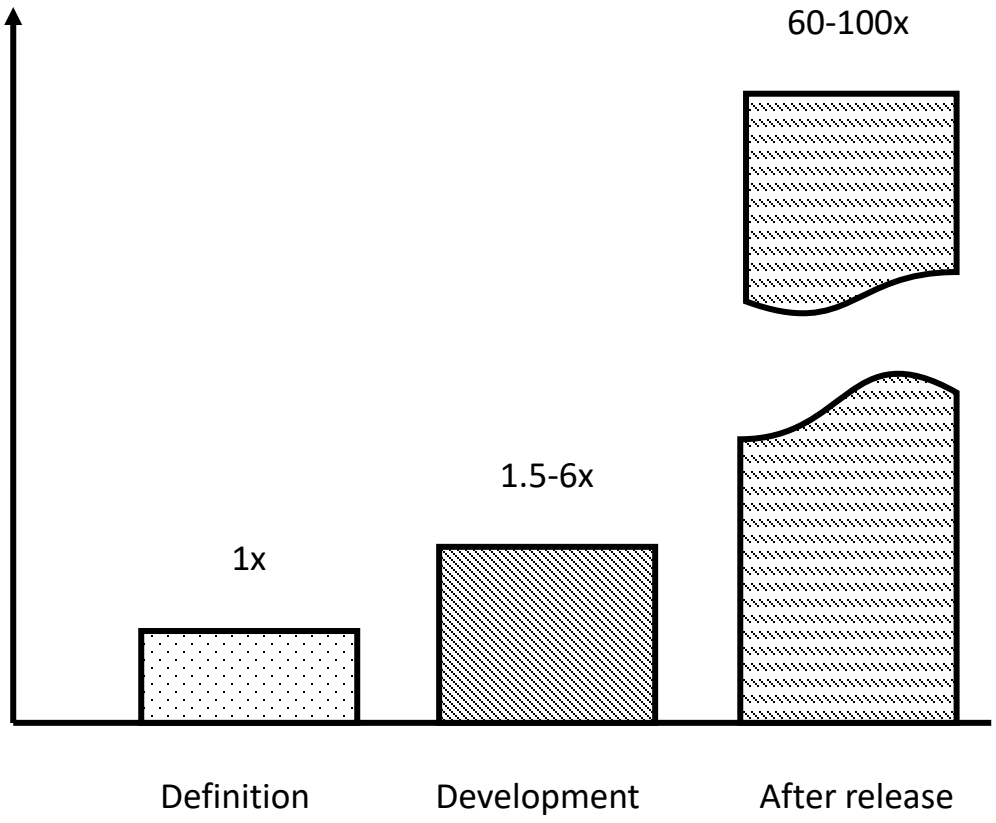
Software Myths - Customer

- "A general statement of objectives is sufficient to begin writing programs – we can fill in the details later"
  - Ambiguous statement of objectives spells disaster
- "Project requirements continually change, but change can be easily accommodated because software is flexible"
  - Impact of change depends on where and when it occurs in the software life cycle (requirements analysis, design, code, test)

The Cost of Change



The Cost of Change



Software Myths - Practitioner

- "Once we write the program and get it to work, our job is done"
  - 60% to 80% of all effort expended on software occurs after it is delivered
- "Until I get the program running, I have no way of assessing its quality
  - Formal technical reviews of requirements analysis documents, design documents, and source code (more effective than actual testing)
- "The only deliverable work product for a successful project is the working program"
  - Software, documentation, test drivers, test results
- "Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down"
  - Creates quality, not documents; quality reduces rework and provides software on time and within the budget

Thank You…

Credits

- Software Engineering 7/ed by Roger Pressman – Reference

Course Name : Software Engineering

BITS Pilani
Pilani Campus

T V Rao

Prescriptive Process Flows

A process defines who does what, when and how in order to achieve a preset goal

- Jacobson et al [99]

Prescriptive and Agile Processes

- Prescriptive processes are processes where all of the process activities are planned in advance and progress is measured against this plan.
- In agile processes, planning is incremental and it is easier to change the process to reflect changing customer requirements.
- In practice, most practical processes may include elements of both plan-driven and agile approaches.

- *There are NO right or wrong software processes.*

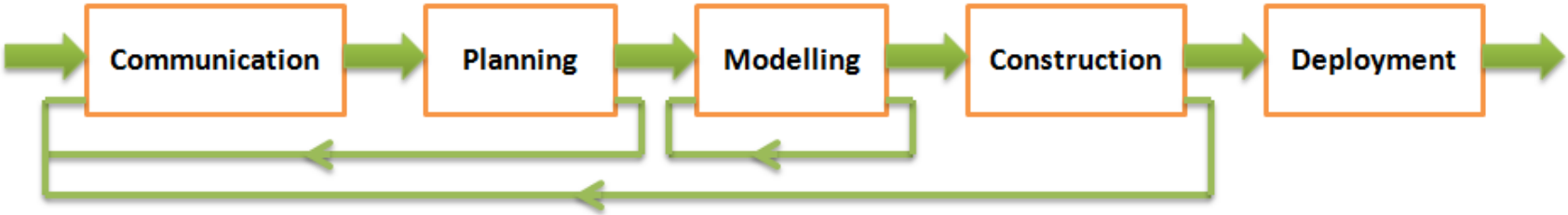
Prescriptive Process Model

- Defines a distinct set of activities, actions, tasks, milestones, and work products that are required to engineer high-quality software

- The activities may be linear, incremental, or evolutionary

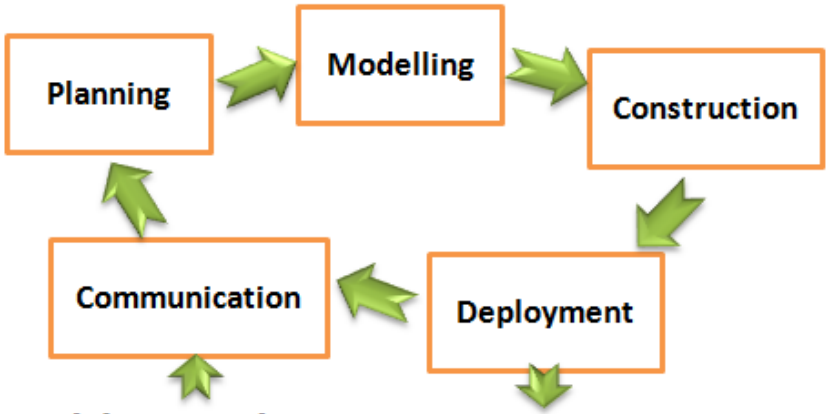
Process Models

(a) Linear Process Flow



(b) Iterative Process Flow

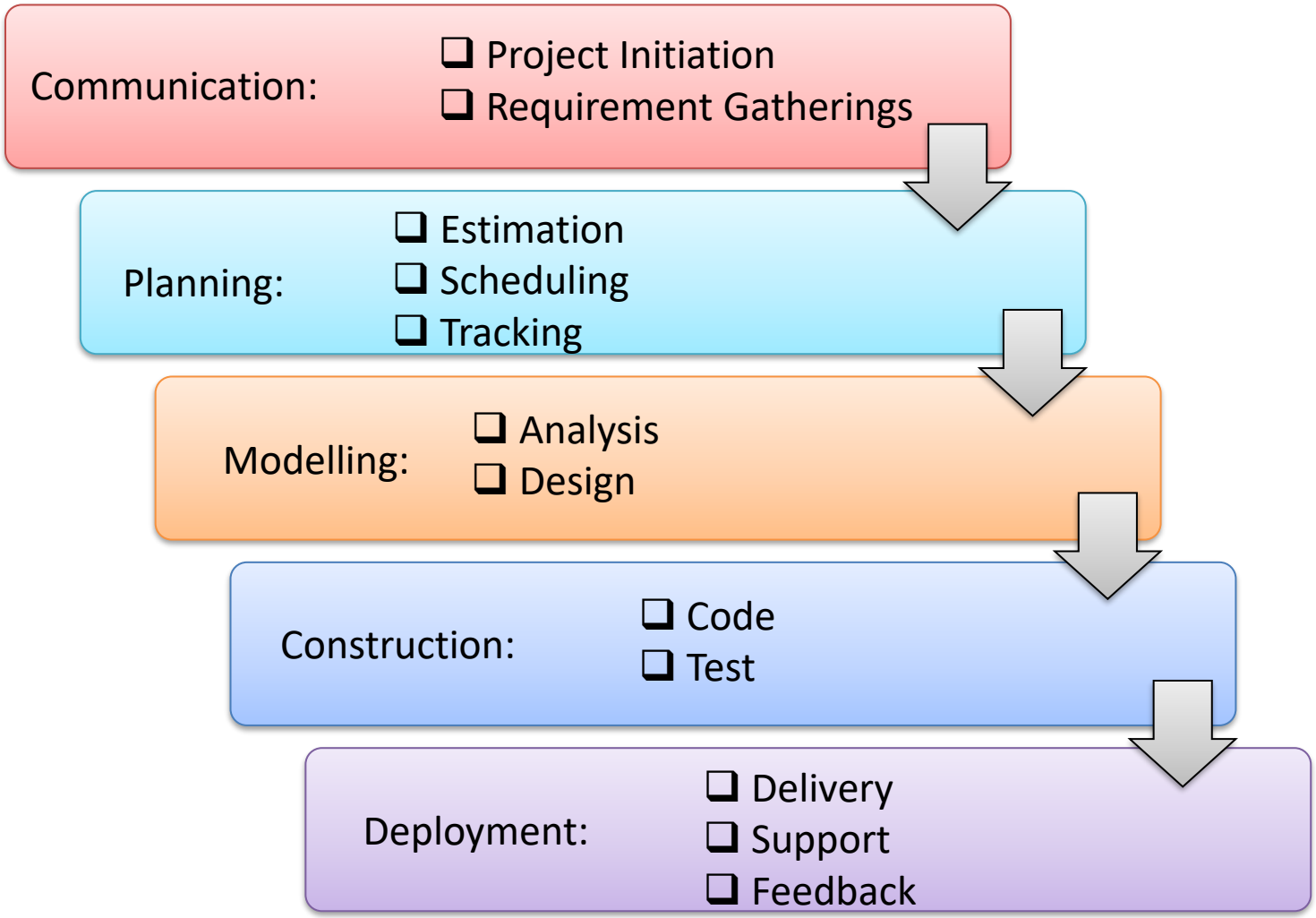
An important variation among process models comes from flow of activities



(c) Evolutionary Process Flow

Waterfall Model

(Diagram)



Critique of the waterfall model

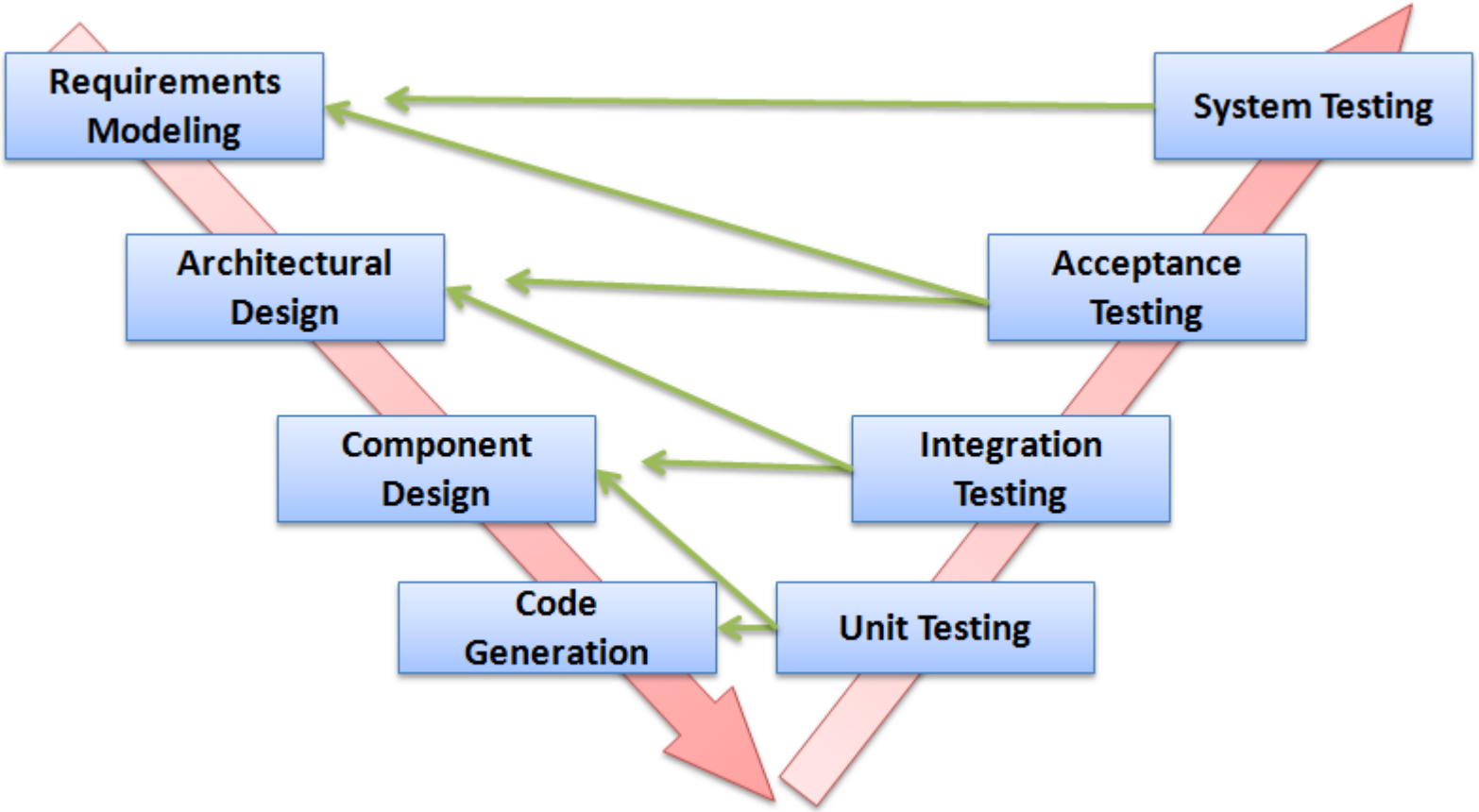
- The model implies that you should attempt to complete a given stage before moving on to the next stage
  - Does not account for the fact that requirements constantly change.
  - It also means that customers can not use anything until the entire system is complete.
- The model makes no allowances for prototyping.
- Assumes understanding of problem and full requirements early on
- It implies that you can get the requirements right by simply writing them down and reviewing them.
- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.

Critique of Waterfall Model

continued

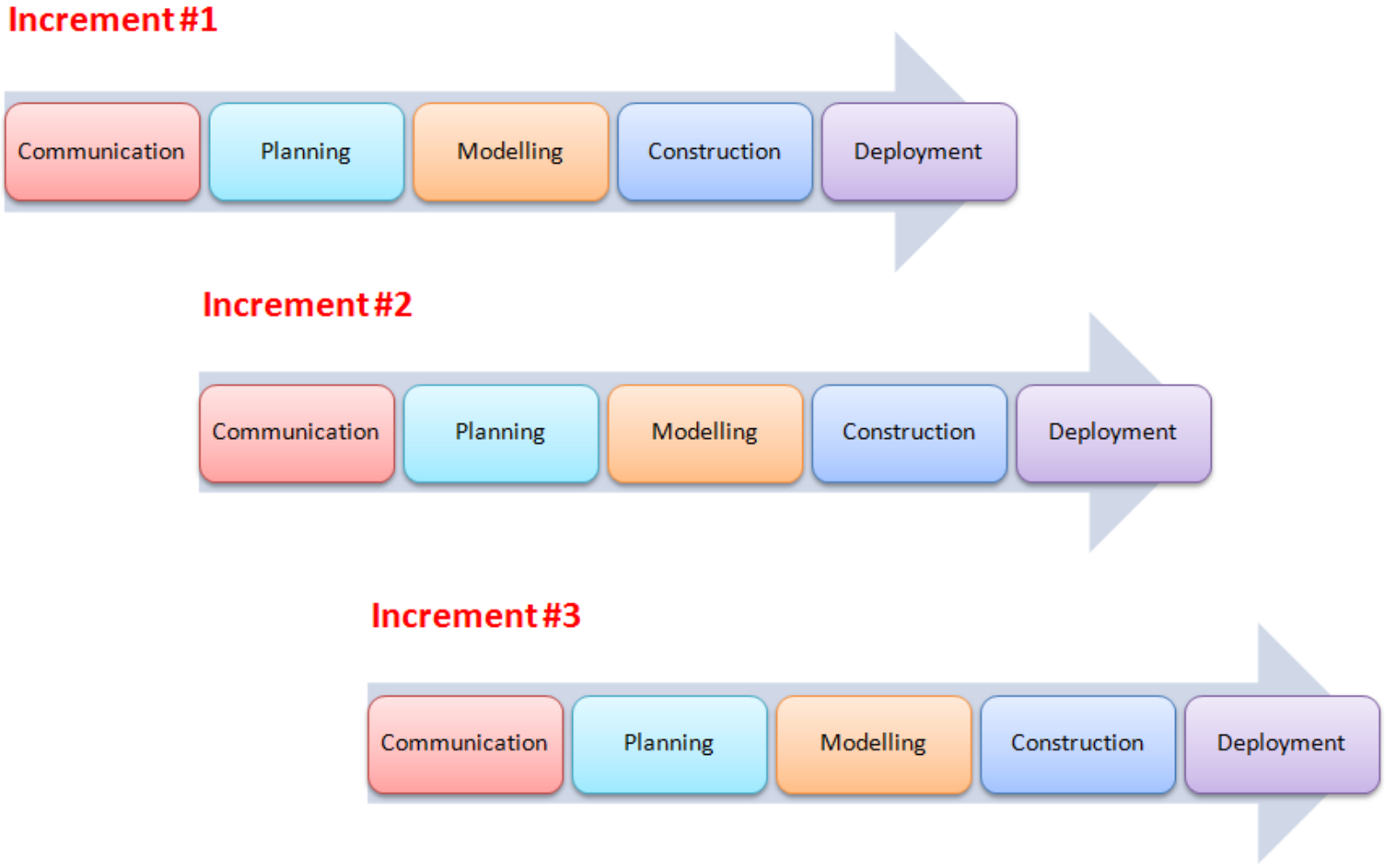
- Follows systematic approach to development
- The model implies that once the product is finished, everything else is maintenance.
- Assumes patience from customer
- Surprises at the end are very expensive
- Some teams sit idle for other teams to finish
- Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.

V-Model


Requirements Modeling
System Testing
Architectural Design
Acceptance Testing
Component Design
Integration Testing
Code Generation
Unit Testing

Incremental Model

(Diagram)



The Incremental Model

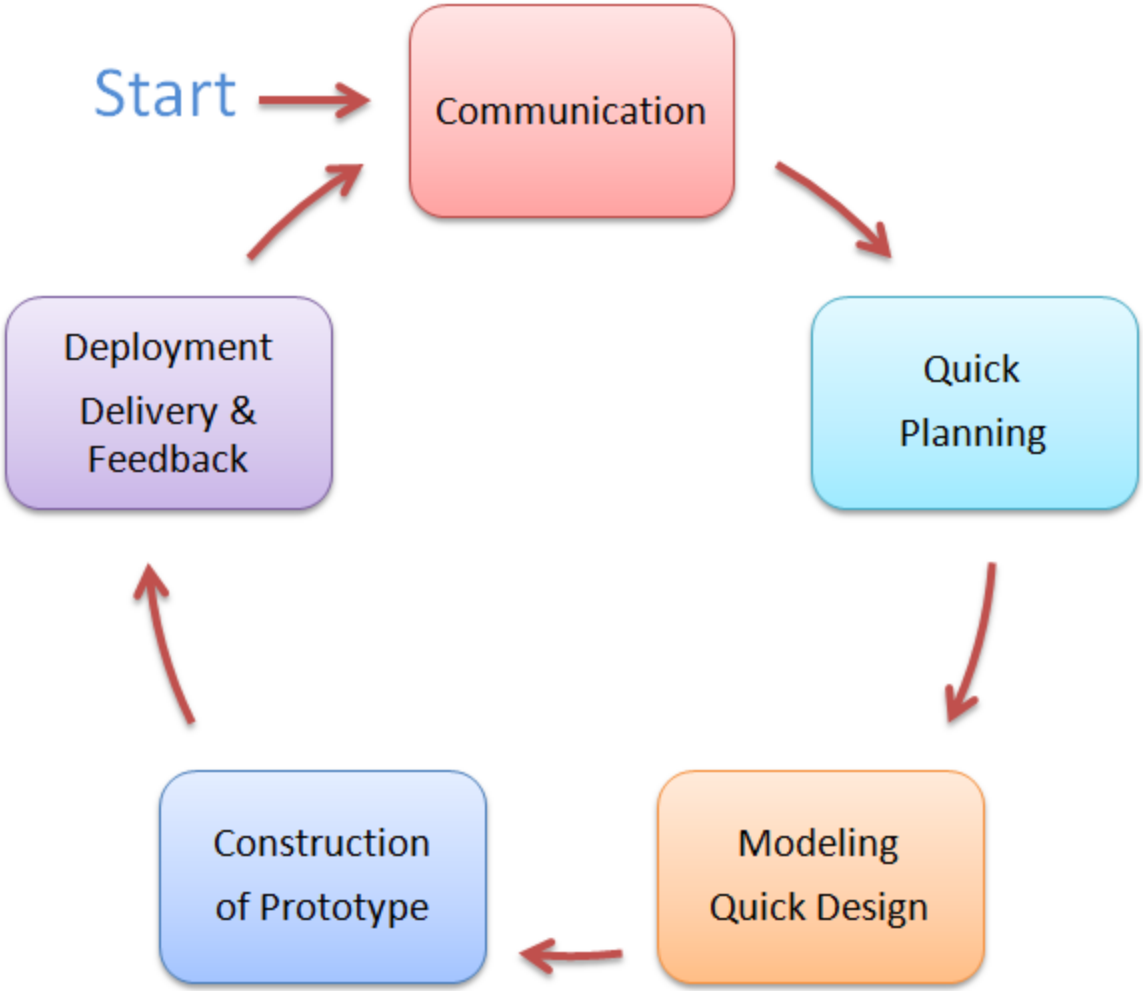
- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- First Increment is often core product
  - Includes basic requirement
  - Many supplementary features (known & unknown) remain undelivered
- First Increment is used or evaluated
- A plan of next increment is prepared
  - Modifications of the first increment
  - Additional features of the first increment
- It is particularly useful when enough staffing is not available for the whole project
- Increment can be planned to manage technical risks

The Incremental Model

- User requirements are prioritised and the highest priority requirements are included in early increments.
- Once the development of an increment is started, the requirements are frozen though requirements for later increments can continue to evolve.
- Customer value can be delivered with each increment so system functionality is available earlier.
- Early increments act as a prototype to help elicit requirements for later increments.
- Lower risk of overall project failure.
- The highest priority system services tend to receive the most testing.

Prototyping Model

(Diagram)



Prototyping Model

- Follows an evolutionary and iterative approach
- Used when requirements are not well understood
- Serves as a mechanism for identifying software requirements
- Focuses on those aspects of the software that are visible to the customer/user
- Feedback is used to refine the prototype

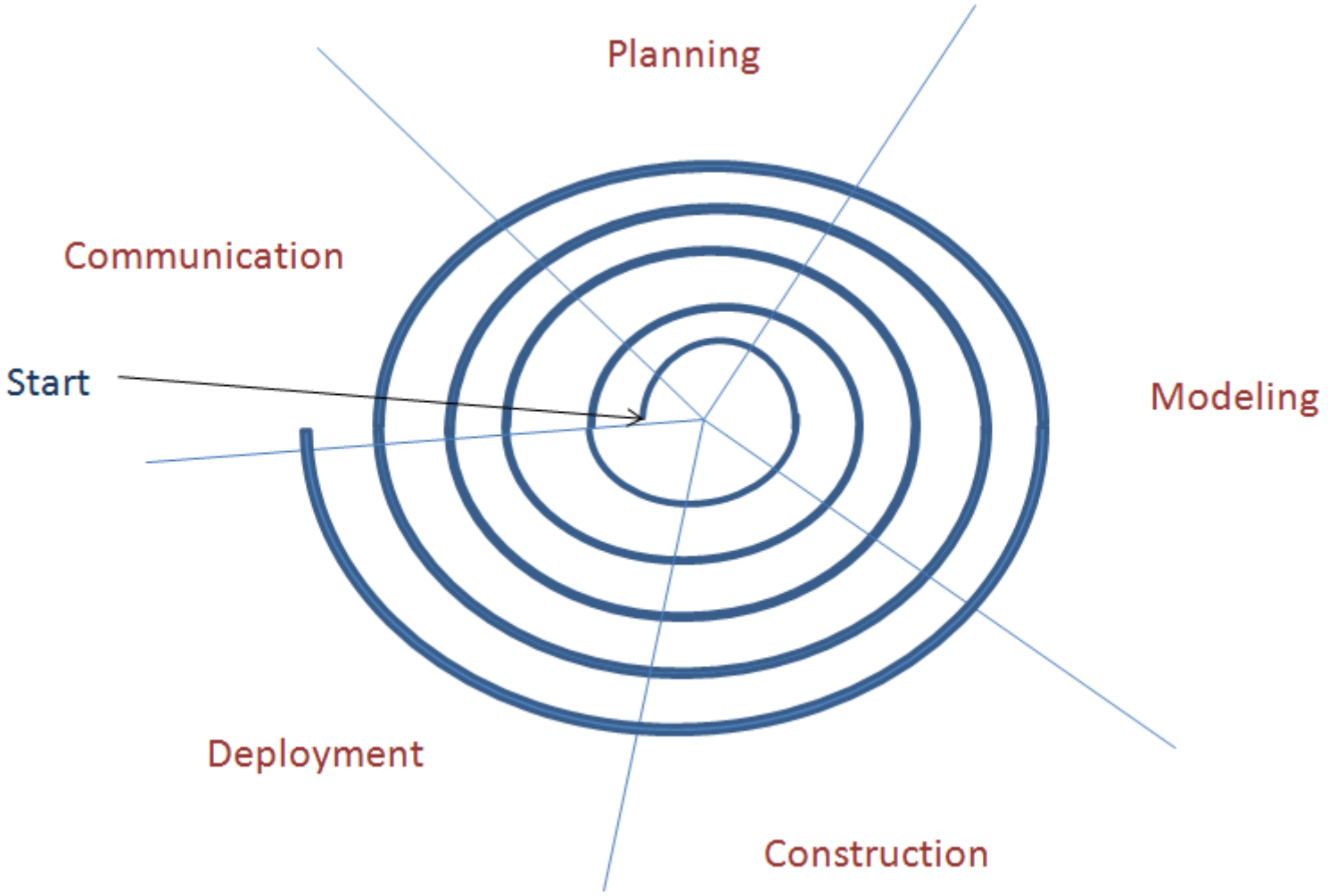
Prototyping Model

(Potential Problems)

- The customer sees a "working version" of the software, wants to stop all development and then buy the prototype after a "few fixes" are made
- Developers often make implementation compromises to get the software running quickly (e.g., language choice, user interface, operating system choice, inefficient algorithms)
- Important Considerations:
  - Define the rules up front on the final disposition of the prototype before it is built
  - In most circumstances, plan to discard the prototype and engineer the actual production software with a goal toward quality

Spiral Model

(Diagram)



Spiral Model

- Proposed by Dr. Barry Boehm in 1988 while working at TRW
- Follows an evolutionary approach
- Used when requirements are not well understood and risks are high
- Inner spirals focus on identifying software requirements and project risks; may also incorporate prototyping
- Outer spirals take on a classical waterfall approach after requirements have been defined, but permit iterative growth of the software
- Operates as a risk-driven model…a go/no-go decision occurs after each complete spiral in order to react to risk determinations
- Requires considerable expertise in risk assessment
- Serves as a realistic model for large-scale software development

General Weaknesses of Evolutionary Process Models

As per Nogueira et al,

1) Evolutionary models pose a problem to project planning because of the uncertain number of iterations required to construct the product
2) Evolutionary software processes do not establish the maximum speed of the evolution
   - If too fast, the process will fall into chaos
   - If too slow, productivity could be affected
3) Software processes should focus first on flexibility and extensibility, and second on high quality
   - We should prioritize the speed of the development over zero defects
   - Extending the development in order to reach higher quality could result in late delivery

Thank You…

Credits

- Software Engineering 7/ed by Roger Pressman – Reference

Course Name : Software Engineering

BITS Pilani
Pilani Campus

T V Rao

Prescriptive Process Variants

Process Variants

- There are three principal types of flow for prescriptive process models
  - Linear
  - Incremental
  - Iterative
- However, there are several variants of process models based on emphasis on work products, process feature, or product feature

Component-based Development Model

- Software Team has to perform following additional steps
  - Available component-based products are researched and evaluated for the application domain in question
  - Component integration issues are considered
  - A software architecture is designed to accommodate the components
  - Components are integrated into the architecture
  - Comprehensive testing is conducted to ensure proper functionality
- Relies on a robust component library
- Capitalizes on software reuse, which leads to documented savings in project cost and time

Formal Methods Model

- Encompasses a set of activities that leads to formal mathematical specification of computer software
- Enables a software engineer to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation
- Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily through mathematical analysis
- Offers the promise of defect-free software
- Used often when building safety-critical systems

Formal Methods Model

- Development of formal methods is currently quite time-consuming and expensive
- Because few software developers have the necessary background to apply formal methods, extensive training is required
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers

Aspect-Oriented Software Development

Provides a process and methodological approach for defining, specifying, designing, and constructing *aspects* such as

- user interfaces,
- security,
- memory management

that impact many parts of the system being developed

Wisdom from Watt S Humphrey

- Some simple processes can improve quality when a software is being engineered by
  - An individual (Personal Software Process)
  - A small Team (Team Software Process)

Personal Software Process (PSP)

innovate achieve lead

- Recommends five framework activities:
  - Planning
  - High-level design
  - High-level design review
  - Development
  - Postmortem
- Stresses the need for each software engineer to identify errors early and as important, to understand the types of errors

Team Software Process (TSP)

innovate achieve lead

- Each project is “launched” using a “script” that defines the tasks to be accomplished
- Teams are self-directed
- Measurement is encouraged
- Measures are analyzed with the intent of improving the team process

Team Software Process (TSP)

innovate achieve lead

- Recommends five framework activities for TSP:
  - Project Launch
  - High-level design
  - Implementation
  - Integration and Testing
  - Postmortem

The Unified Process

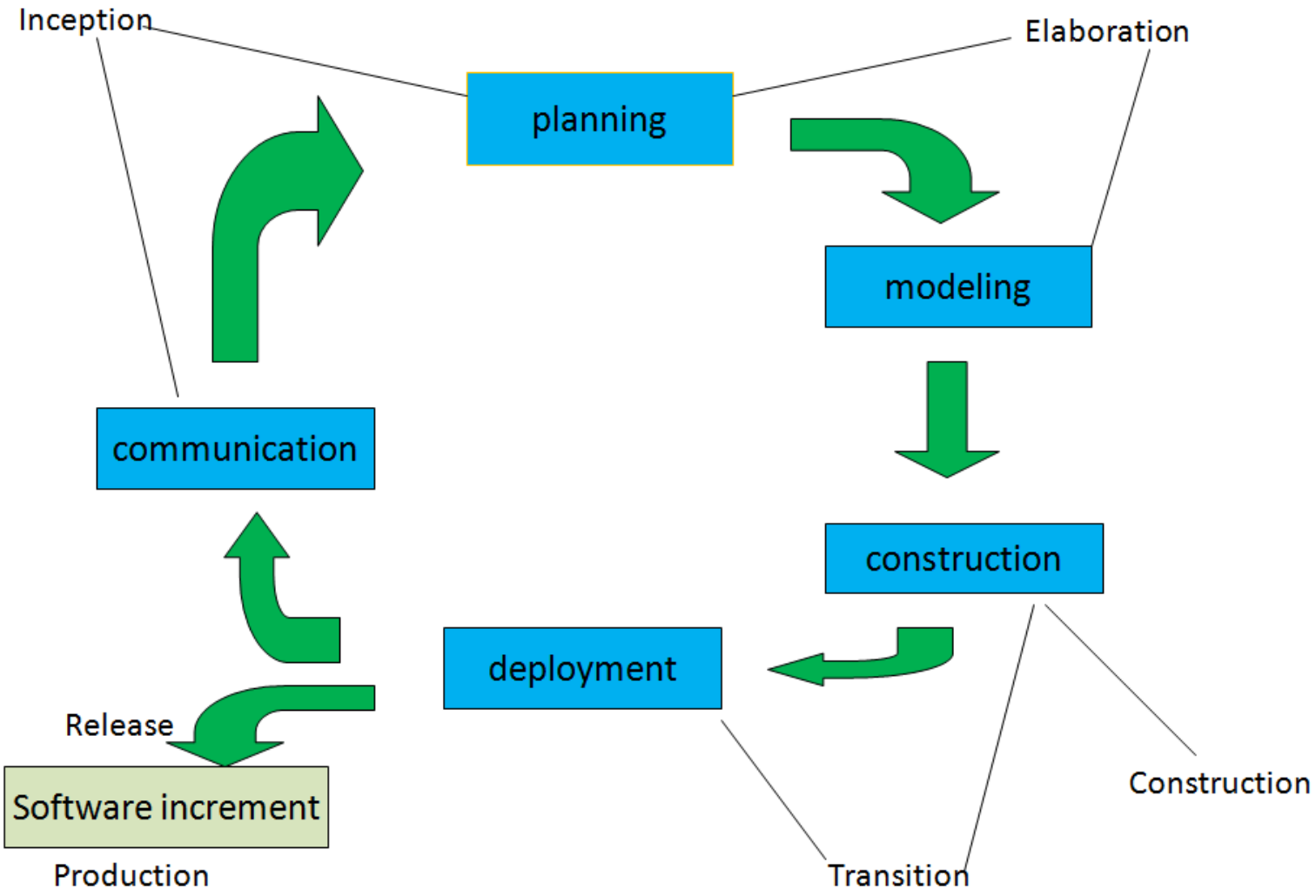
Background

- Started during the late 1980's and early 1990s when object-oriented languages were gaining wide-spread use
- Many object-oriented analysis and design methods were proposed; three top authors were Grady Booch, Ivar Jacobson, and James Rumbaugh
- They eventually worked together on a unified method for modeling, called the Unified Modeling Language (UML)
  - UML is a robust notation for the modeling and development of object-oriented systems
  - UML became an industry standard in 1997
  - However, UML does not provide the process framework, only the necessary technology for object-oriented development

Background (continued)

- Booch, Jacobson, and Rumbaugh later developed the unified process, which is a framework for object-oriented software engineering using UML
  - Draws on the best features and characteristics of conventional software process models
  - Emphasizes the important role of software architecture
  - Consists of a process flow that is iterative and incremental, thereby providing an evolutionary feel
- Consists of five phases: inception, elaboration, construction, transition, and production

Phases of the Unified Process



Inception Phase

- Encompasses both customer communication and planning activities of the generic process
- Business requirements for the software are identified
- A rough architecture for the system is proposed
- A plan is created for an incremental, iterative development
- Fundamental business requirements are described through preliminary use cases
  - A use case describes a sequence of actions that are performed by a user

Elaboration Phase

- Encompasses both the planning and modelling activities of the generic process
- Refines and expands the preliminary use cases
- Expands the architectural representation to include five views
  - Use-case model
  - Analysis model
  - Design model
  - Implementation model
  - Deployment model
- Often results in an executable architectural baseline that represents a first cut executable system
- The baseline demonstrates the viability of the architecture but does not provide all features and functions required to use the system

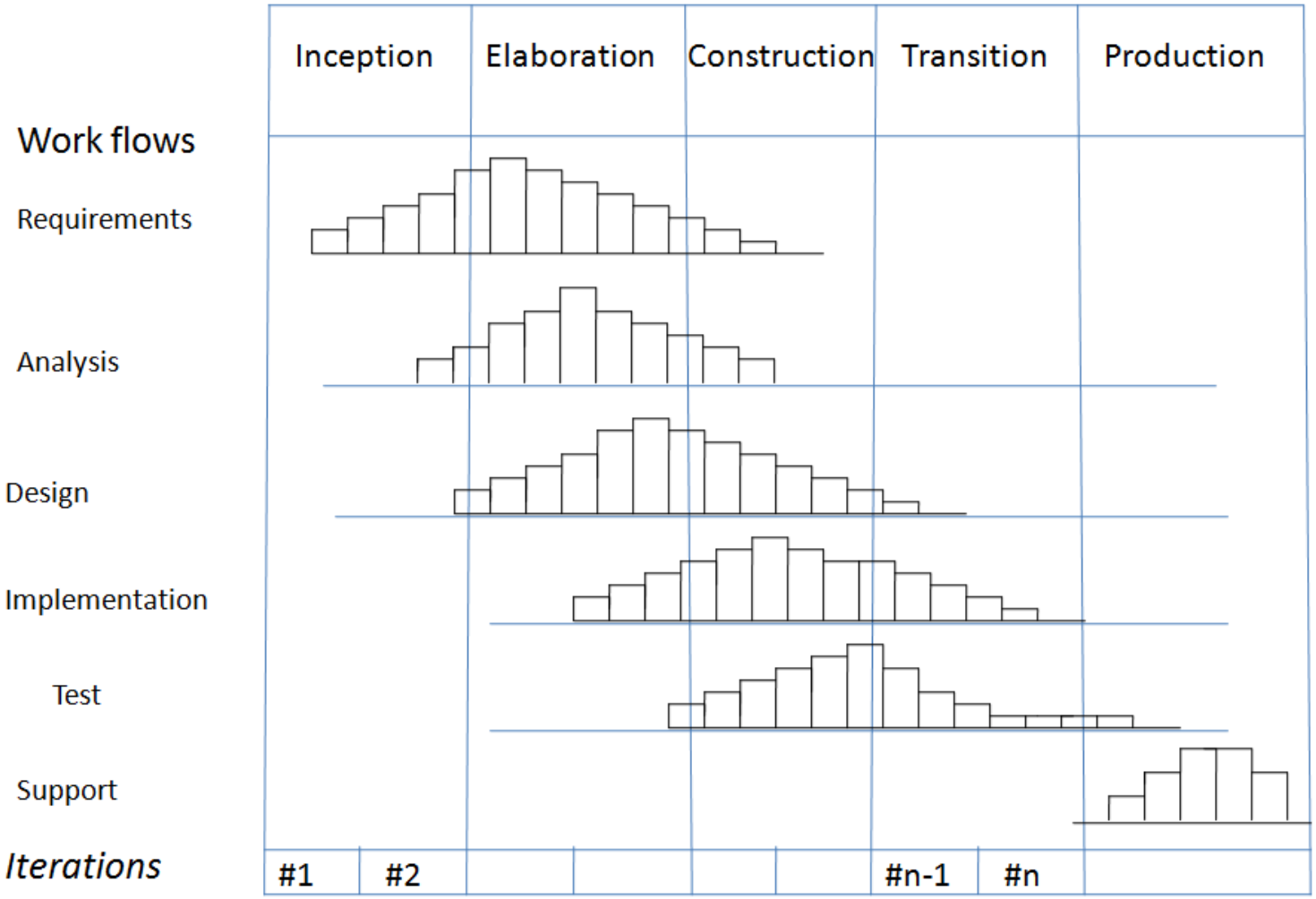
Construction Phase

- Encompasses the construction activity of the generic process
- Uses the architectural model from the elaboration phase as input
- Develops or acquires the software components that make each use-case operational
- Analysis and design models from the previous phase are completed to reflect the final version of the increment
- Use cases are used to derive a set of acceptance tests that are executed prior to the next phase

Transition Phase

- Encompasses the last part of the construction activity and the first part of the deployment activity of the generic process
- Software is given to end users for beta testing and user feedback reports on defects and necessary changes
- The software teams create necessary support documentation (user manuals, trouble-shooting guides, installation procedures)
- At the conclusion of this phase, the software increment becomes a usable software release

UP Phases


Inception
Elaboration
Construction
Transition
Production
Work flows
Requirements
Analysis
Design
Implementation
Test
Support
Iterations
#1
#2
#n-1
#n

Production Phase

- Encompasses the last part of the deployment activity of the generic process
- On-going use of the software is monitored
- Support for the operating environment (infrastructure) is provided
- Defect reports and requests for changes are submitted and evaluated

UP Work Products

Inception phases

Vision document

Initial use case model
Initial project glossary
Initial business case
Initial risk assessment
Project plan, phase
And iteration
Business model,
If necessary .

One or more
prototypes

Elaboration phase

Use case model
Supplementary
requirements including
non functional
Analysis Model
Software architecture
Description .
Executable
architectural
prototype.

Preliminary design
Model.
Revised risk list.
Project plan including
Iteration plan.
Adapted work flows
Milestones
Technical work
products.
Preliminary user
manual.

Construction phase

Design model
Software
components.
Integrated software
increment.

Test plan and
procedure
Test cases

Support
documentation
User manuals
Installation manuals
Description of
current increment

Transition phase

Delivered software
increment
Beta test reports

General user
Feedback.

Thank You…

Credits

- Software Engineering 7/ed by Roger Pressman – Reference

Course Name : Software Engineering

BITS Pilani
Pilani Campus

T V Rao

Concept of Agility

A View on Software Process

innovate achieve lead

- Because software, like all capital, is embodied knowledge, and because that knowledge is initially dispersed, tacit, latent, and incomplete in large measure, software development is a social learning process. The process is a dialogue in which the knowledge that must become software is brought together and embodied in the software.

-Howard Baetjer (economist)

Rapid software development

- Rapid development and delivery is now often the most important requirement for software systems
  - Businesses operate in a fast –changing requirement and it is practically impossible to produce a set of stable software requirements
  - Software has to evolve quickly to reflect changing business needs.
- Rapid software development
  - Specification, design and implementation are inter-leaved
  - System is developed as a series of versions with stakeholders involved in version evaluation
  - User interfaces are often developed using an IDE and graphical toolset.

Agile methods

- Dissatisfaction with the overheads involved in software design methods of the 1980s and 1990s led to the creation of agile methods. These methods:
  - Focus on the code rather than the design
  - Are based on an iterative approach to software development
  - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.

- The aim of agile methods is to reduce overheads in the software process (e.g. by limiting documentation) and to be able to respond quickly to changing requirements without excessive rework.

The Manifesto for Agile Software Development

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more."

Kent Beck et al

What is “Agility”?

- Effective (rapid and adaptive) response to change
- Effective communication among all stakeholders
- Drawing the customer onto the team
- Organizing a team so that it is in control of the work performed

Yielding …

- Rapid, incremental delivery of software

An Agile Process

innovate achieve lead

- Is driven by customer descriptions of what is required (scenarios)
- Recognizes that plans are short-lived
- Develops software iteratively with a heavy emphasis on construction activities
- Delivers multiple ‘software increments’
- Adapts as changes occur

Plan-driven versus Agile Development

- Plan-driven development
  - A plan-driven approach to software engineering is based around separate development stages with the outputs to be produced at each of these stages planned in advance.
  - Not necessarily waterfall model – plan-driven, incremental development is possible
  - Iteration occurs within activities.
- Agile development
  - Specification, design, implementation and testing are inter-leaved and the outputs from the development process are decided through a process of negotiation during the software development process.

Agility Principles

- 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.
- 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.

Agility Principles (contd)

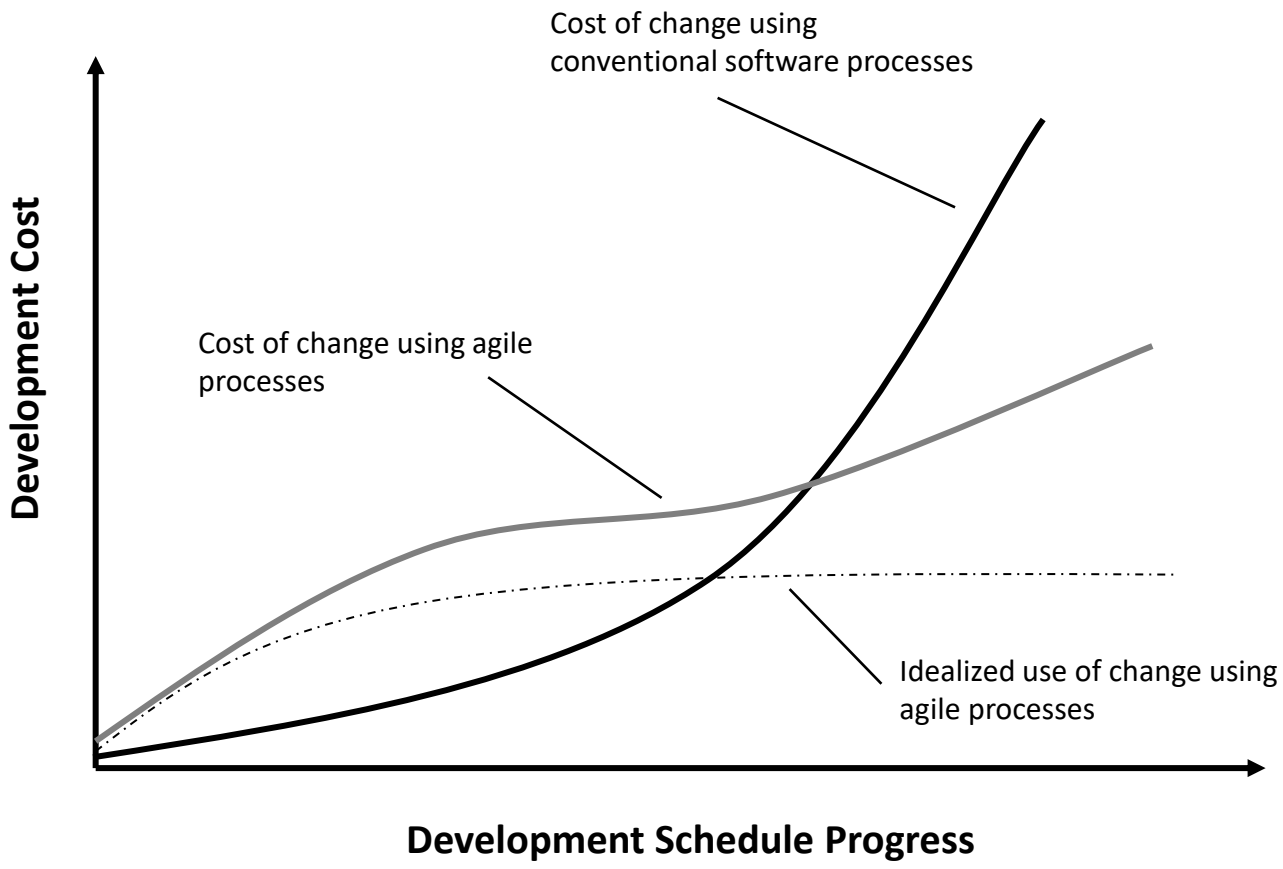
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity – the art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

Agility Human Factors

innovate achieve lead

- The process molds to the needs of the people and team, not the other way around.
- Some key traits must exist among the people on an agile team
  - Competence
  - Common focus
  - Collaboration
  - Decision-making ability
  - Fuzzy problem-solving ability
  - Mutual trust and respect
  - Self-organization

Cost of Change



An Agile Concept - Time Boxing

innovate achieve lead

... teams are managing the triple constraints that face any organisation - time, quality, scope. When using a fixed duration, we are telling everyone involved, 'time is urgent and we are going to include as much as we can within this time framework.' Since quality cannot be compromised, the only variable is scope. 'Time boxing' creates a sense of urgency and criticality for the entire organization

—Mark P. Dangelo, Author: *Innovative relevance*

Agile Modeling

- Originally proposed by Scott Ambler
- Suggests a set of agile modeling principles
  - Model with a purpose
  - Use multiple models
  - Travel light
  - Content is more important than representation
  - Know the models and the tools you use to create them
  - Adapt locally

innovate achieve lead

Thank You…

Credits

- Software Engineering 7/ed by Roger Pressman – Reference

innovate achieve lead

Course Name : Software Engineering

BITS Pilani
Pilani Campus

T V Rao
Agile Process Models

Extreme Programming (XP)

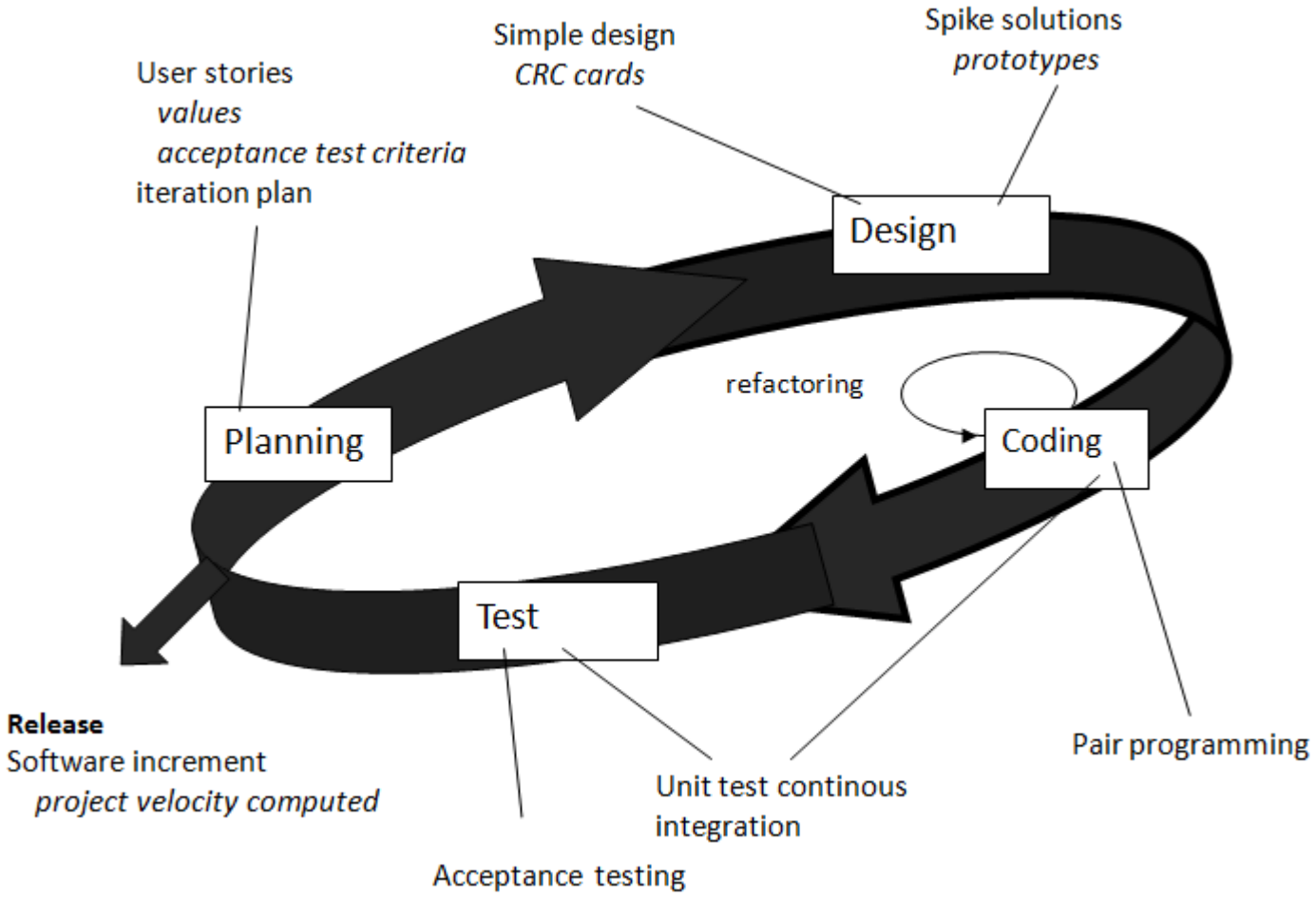
innovate achieve lead

- The most widely used agile process, originally proposed by Kent Beck
- XP Planning
  - Begins with the creation of "*user stories*"
  - Agile team assesses each story and assigns a *cost*
  - Stories are grouped to form a *deliverable increment*
  - A *commitment* is made on delivery date
  - After the first increment "*project velocity*" is used to help define subsequent delivery dates for other increments

Extreme Programming (XP)

- XP Design
  - Follows the *KIS principle*
  - Encourage the use of *CRC cards*
  - For difficult design problems, suggests the creation of "*spike solutions*"—a design prototype
  - Encourages "*refactoring*"—an iterative refinement of the internal program design
- XP Coding
  - Recommends the *construction of a unit test* for a store *before* coding commences
  - Encourages "*pair programming*"
- XP Testing
  - All *unit tests are executed daily*
  - "*Acceptance tests*" are defined by the customer and executed to assess customer visible functionality

Extreme Programming (XP)



Refactoring

- Programming team look for possible software improvements and make these improvements even where there is no immediate need for them.
- This improves the understandability of the software and so reduces the need for documentation.
- Changes are easier to make because the code is well-structured and clear.
- However, some changes requires architecture refactoring and this is much more expensive.

Examples of refactoring

- Re-organization of a class hierarchy to remove duplicate code.
- Tidying up and renaming attributes and methods to make them easier to understand.
- The replacement of inline code with calls to methods that have been included in a program library.

Pair programming

- In pair programming, programmers sit together at the same workstation to develop the software.
- Pairs are created dynamically so that all team members work with each other during the development process.
- The sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave.
- It serves as an informal review process as each line of code is looked at by more than 1 person.
- Pair programming is not necessarily inefficient and there is evidence that a pair working together is more efficient than 2 programmers working separately.

Advantages of Pair programming

- It supports the idea of collective ownership and responsibility for the system.
  - Individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
- It acts as an informal review process because each line of code is looked at by at least two people.
- It helps support refactoring, which is a process of software improvement.
  - Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

Industrial XP

- Incorporates six new practices to ensure that XP works for significant projects in large organization.
- Incorporates six new practices
  - Readiness Assessment
    - Ascertain environment, team, culture
  - Project Community
    - Right people – team becomes community
  - Project Chartering
    - Appropriate business justification within org
  - Test-driven management
    - State of the project as per measurable destinations
  - Retrospectives
    - Specialized technical review after delivery of increment
  - Continuous learning

Criticism of XP

Concerns expressed by critics w.r.t. XP include

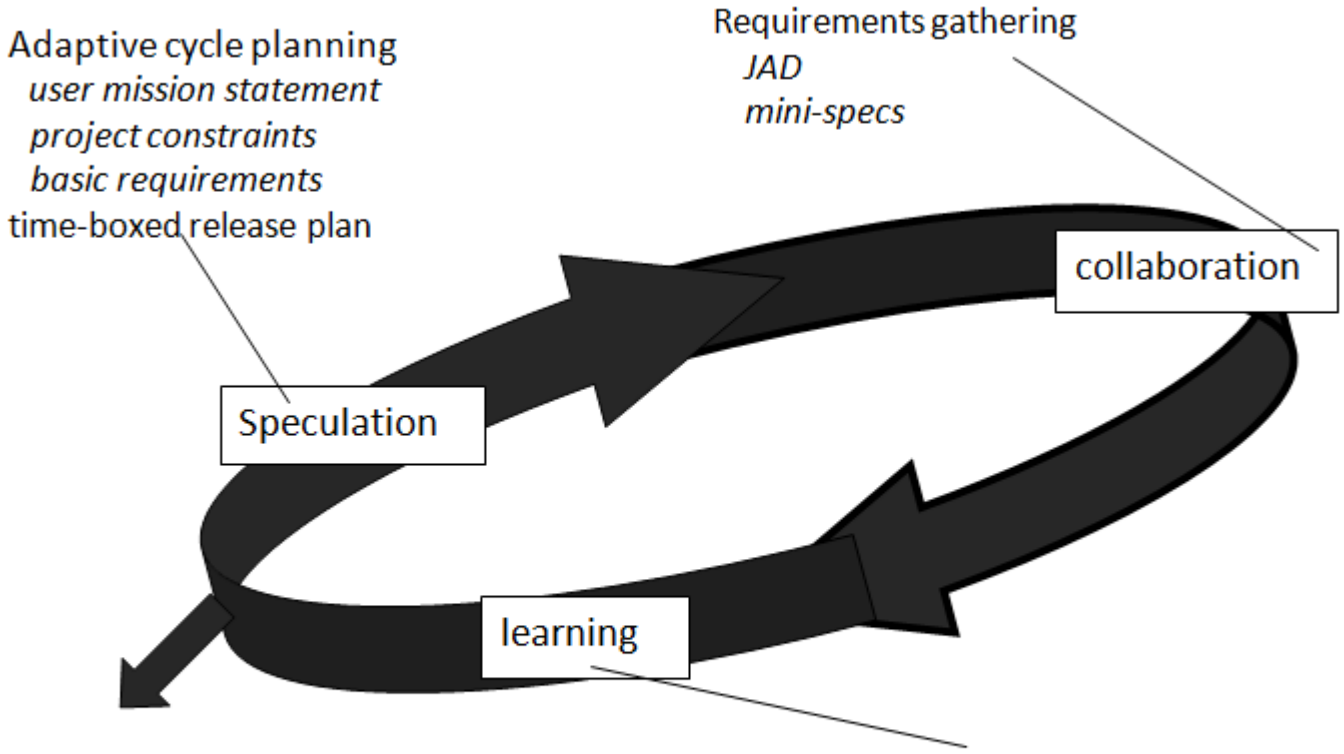
- Requirements volatility
  - -rework may be unmanageable
- Conflicting customer needs
  - -team cannot reconcile conflicting demands
- Requirements are expressed informally
  - -omissions, inconsistencies, errors
- Lack of formal design
  - -Without architectural design, structure of the software lacks quality and maintainability

Adaptive Software Development

innovate achieve lead

- Originally proposed by Jim Highsmith
- ASD — distinguishing features
  - *Mission-driven* planning
  - *Component-based focus*
  - Uses "*time-boxing*"
  - Explicit consideration of *risks*
  - Emphasizes *collaboration* for requirements gathering
  - Emphasizes "*learning*" throughout the process

Adaptive Software Development



Dynamic Systems Development Method

- Promoted by the DSDM Consortium (www.dsdm.org)
- DSDM—distinguishing features
  - Similar in most respects to XP and/or ASD
  - Nine guiding principles
    - Active user involvement is imperative.
    - DSDM teams must be empowered to make decisions.
    - The focus is on frequent delivery of products.
    - Fitness for business purpose is the essential criterion for acceptance of deliverables.
    - Iterative and incremental development is necessary to converge on an accurate business solution.
    - All changes during development are reversible.
    - Requirements are baselined at a high level
    - Testing is integrated throughout the life-cycle.

Dynamic Systems Development Method

- DSDM life cycle has three iterative cycles preceded by two activities
  - Feasibility study

    Is the application a viable candidate for the DSDM process?
  - Business study

    Establish functional & information requirements to provide business value
  - Functional model iteration

    Build incremental prototypes to demonstrate functionality
  - Design & build iteration

    Revisit prototype & engineer it to provide operational business value
  - Implementation iteration

    Place the latest software increment (an “operationalized” prototype) into the oprational environment

SCRUM - Introduction

innovate achieve lead

- Scrum is an Agile Software Development Process.
- Scrum is not an acronym
- name taken from the sport of Rugby, where everyone in the team pack acts together to move the ball down the field
- analogy to development is the team works together to successfully develop quality software

Scrum

- Proposed by Schwaber and Beedle
- Scrum—distinguishing features
  - Development work partitioned into packets makes up "Backlog"
  - Testing and documentation are on-going as the product is constructed
  - Work occurs in "sprints" and is derived from a "backlog" of existing requirements
  - Meetings are very short and sometimes conducted without chairs
    - What did you do since last team meeting
    - What obstacles are you encountering?
    - What do you plan to accomplish by the next team meeting?
  - "demos" are delivered to the customer with the time-box allocated

Agile Unified Process

innovate achieve lead

- Adopts “serial in the large” and “iterative in the small” philosophy suggested by Scott Ambler
- Adopts UP phased activities – inception, elaboration, construction, and transition
- Within each of the activity, team iterates to achieve agility.
- UML representations are used, though modeling is kept to bare minimum

Agile Process and Documentation

According to Matt Simons (“Internationally Agile”) , there are two keys to successful documentation on agile projects.

- Finding the point of "just enough" documentation. This is difficult to determine and will vary by project. Fortunately, the iterative nature of agile development allows you to experiment until you get it right.

- Not get attached to it or have unrealistic hopes of keeping it updated.

“Documentation must be created to serve a specific purpose, and after it has served that purpose you'll all probably have more important things to do than keep updating the documents. It may seem counterintuitive, but it's often better to produce fresh documentation the next time some is clearly required. A side benefit of starting over each time you need to document part of your project is that it's great incentive to keep your documentation efficient!”

Agile Process with Offshore Development

Martin Fowler suggests following successful practices:

- Use Continuous Integration to Avoid Integration Headaches
- Have Each Site Send Ambassadors to the Other Sites
- Use Contact Visits to build trust
- Don't Underestimate the Culture Change
- Use wikis to contain common information
- Use Test Scripts to Help Understand the Requirements
- Use Regular Builds to Get Feedback on Functionality

Agile Process with Offshore Development (contd)

Martin Fowler suggests following successful practices:

- Use Regular Short Status Meetings
- Use Short Iterations
- Use an Iteration Planning Meeting that's Tailored for Remote Sites
- When Moving a Code Base, Bug Fixing Makes a Good Start
- Separate teams by functionality not activity
- Expect to need more documents.
- Get multiple communication modes working early

Article by Martin Fowler, Author and Member, Agile Alliance

Some challenges with agile methods

- It can be difficult to keep the interest of customers who are involved in the process.
- Team members may be unsuited to the intense involvement that characterises agile methods.
- Prioritising changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.

Agile methods and software maintenance

- Most organizations spend more on maintaining existing software than they do on new software development. So, if agile methods are to be successful, they have to support maintenance as well as original development.
- Two key issues:
  - Are systems that are developed using an agile approach maintainable, given the emphasis in the development process of minimizing formal documentation?
  - Can agile methods be used effectively for evolving a system in response to customer change requests?
- Problems may arise if original development team cannot be maintained.

Large systems development

- Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones.
- Large systems are ‘brownfield systems’, that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don’t really lend themselves to flexibility and incremental development.
- Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development.

Large systems development

innovate achieve lead

- Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed.
- Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
- Large systems usually have a diverse set of stakeholders.It is practically impossible to involve all of these different stakeholders in the development process.

Thank You…

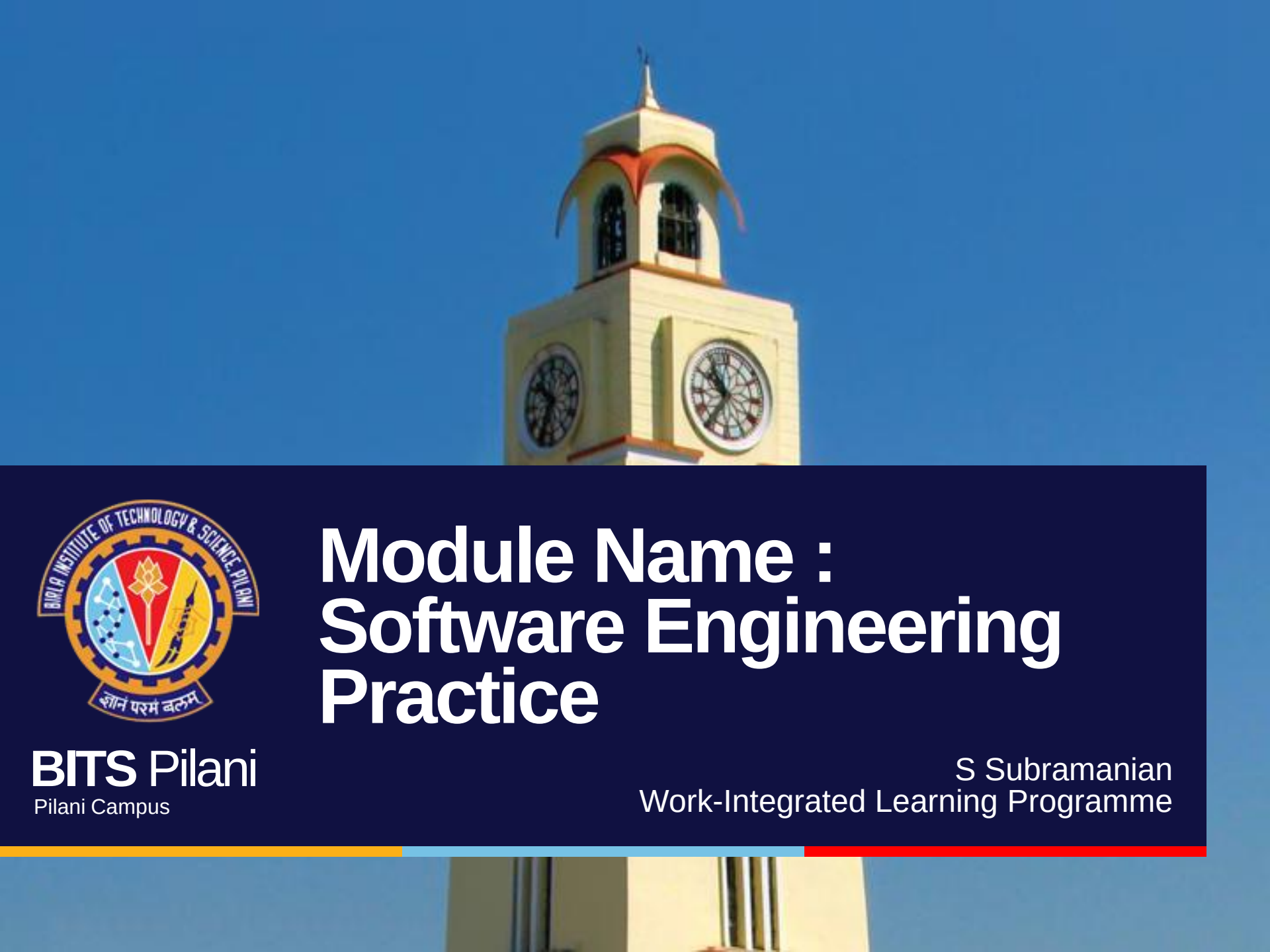
Credits

- Software Engineering 7/ed by Roger Pressman – Reference

Course Name : Software Engineering

BITS Pilani
Pilani Campus

S Subramanian
Work-Integrated Learning Programme



Module Name : Software Engineering Practice

BITS Pilani
Pilani Campus

S Subramanian
Work-Integrated Learning Programme

Software Engineering Knowledge

- *You often hear people say that software development knowledge has a 3-year half-life: half of what you need to know today will be obsolete within 3 years. In the domain of technology-related knowledge, that's probably about right. But there is another kind of software development knowledge—a kind that I think of as "software engineering principles"—that does not have a three-year half-life. These software engineering principles are likely to serve a professional programmer throughout his or her career.*

Steve McConnell

Principles that Guide Process - I

- **Principle #1. *Be agile.*** Basic tenets of agile development should govern your approach. Every aspect of the work you do should emphasize economy of action—keep your technical approach as simple as possible, keep the work products you produce as concise as possible

- **Principle #2. *Focus on quality at every step.*** Take pride in what you do and quality output will follow.

- **Principle #3. *Be ready to adapt.*** When necessary, adapt your approach to constraints imposed by the problem, the people, and the project itself.

- **Principle #4. *Build an effective team.*** People produce outputs; they are important. Build a self-organizing team that has mutual trust and respect.

Principles that Guide Process - II

- **Principle #5. Establish mechanisms for communication and coordination.** Keep all stakeholders informed – project resources, senior management, client etc. Both good and bad news. On one likes surprises.

- **Principle #6. Manage change.** Change is constant . Manage change proactively, informally and formally. If Change is not managed properly, project fail invariably.

- **Principle #7. Assess risk.** Lots Continuous risk assessment throughout the project life cycle is required and also establish contingency plans. Review risk plan as often as possible.

- **Principle #8. Create work products that provide value for others.** Create only those work products that provide value for other process activities, actions or tasks. Avoid duplication and redundant tasks.

Principles that Guide Practice

- **Principle #1.** ***Divide and conquer.*** Use O-O principles, component based development design . Build system assembling components.
- **Principle #2.** ***Understand the use of abstraction.*** The intent of an abstraction is to eliminate the need to communicate details. Following standards, OO principle, build frameworks which achieves specific business objectives..
- **Principle #3. Strive for consistency.** Consistency suggests that a familiar context makes software easier to use. As an example, consider the design of a user interface for a WebApp. Consistent placement of menu options, the use of a consistent color scheme, and the consistent use of recognizable icons all help to make the interface ergonomically sound.familiar context makes software easier to use.
- **Principle #4.** ***Focus on the transfer of information.*** Pay special attention to the analysis, design, construction, and testing of interfaces – Application level – end to end quality not just at code level.

Principles that Guide Practice

- **Principle #5.** ***Build software that exhibits effective modularity.*** Create common enterprise message bus. Modules / objects should hang on them, each module should exhibit low coupling to other modules, to data sources, and to other environmental aspects.of concerns
- **Principle #6.** ***Look for patterns.*** The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Ex: Java design patterns – Gang of four.

- **Principle #7.** ***When possible, represent the problem and its solution from a number of different perspectives. -*** When a problem and its solution are examined from a number of different perspectives, it is more likely that greater insight will be achieved and that errors and omissions will be uncovered.-

- **Principle #8.** ***Remember that someone will maintain the software.*** Write maintainable, documented code.

Communication Principles

- **Principle #1.** ***Listen.*** Listen carefully. Show respect. Try to focus on the speaker's words, rather than formulating your response to those words.

- **Principle # 2.** ***Prepare before you communicate.*** Do your homework well. Spend the time to understand the problem before you meet with others.

- **Principle # 3.** ***Someone should facilitate the activity.*** Manage and control meetings so that it does not loose focus. Disagreements will happen, mage it . Show leadership.

- **Principle #4.** ***Face-to-face communication is best.*** Adopt both formal & informal communication. Required on many occasions.

Communication Principles

- **Principle # 5.** ***Take notes and document decisions.*** Produce minutes the meeting and follow up for status and completion.

- **Principle # 6.** ***Strive for collaboration.*** Each small collaboration serves to build trust among team members and creates a common goal for the team.

- **Principle # 7.** ***Stay focused, modularize your discussion.*** *Have clear agenda, stick to agenda and box the time for open discussion.*

- **Principle # 8.** ***If something is unclear, draw a picture.***

- **Principle # 9.** ***(a) Once you agree to something, move on; (b) If you can't agree to something, move on; (c) If a feature or function is unclear and cannot be clarified at the moment, move on. –*** Don't wait for all information/data to be available for decision making, document assumptions and move forward.

- **Principle # 10.** ***Negotiation is not a contest or a game. It works best when both parties win.*** Negotiation will demand compromise from all parties

Planning Principles

- **Principle #1.** ***Understand the scope of the project.*** It's impossible to use a roadmap if you don't know where you're going. Scope provides the software team with a destination with controlled changes. Agile methodology can be adopted.
- **Principle #2.** ***Involve the customer in the planning activity.*** Share project plan and assumption with clients as early as possible to define priorities and establish project constraints.
- **Principle #3.** ***Recognize that planning is iterative.*** A project plan is a living document. As work begins, it very likely that things will change.
- **Principle #4.** ***Estimate based on what you know.*** The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done. Very important to document the assumptions.

Planning Principles

- **Principle #5.** ***Consider risk as you define the plan.*** Consider Risks and issues separately and evaluate the impact and probability of occurrences.
- **Principle #6.** ***Be realistic.*** Plan for optimum resource utilization. Productivity varies and not constant every day**.** People don't work 100 percent of every day.
- **Principle #7.** ***Adjust granularity as you define the plan.*** *Granularity* refers to the level of detail that is introduced as a project plan is developed. Not micro manage but stay reasonable.
- **Principle #8.** ***Define how you intend to ensure quality.*** The plan should identify how the software team intends to ensure quality.
- **Principle #9.** ***Describe how you intend to accommodate change.*** Even the best planning can be obviated by uncontrolled change. Establish CCBs and agree with customer.
- **Principle #10.** ***Track the plan frequently and make adjustments as required.*** Software projects fall behind schedule one day at a time. Keep the plan current, else plan is of no use.

Modeling Principles

- In software engineering work, two classes of models can be created:
  - *Requirements models* (also called *analysis models*) represent the customer requirements by depicting the software in three different domains: the information domain, the functional domain, and the behavioral/performance domain.
  - *Design models* represent characteristics of the software that help practitioners to construct it effectively: the architecture, the user interface, and component-level detail.

Requirements Modeling Principles

- **Principle #1.** ***The information domain of a problem must be represented and understood.-*** Functionality, data and business/transaction flow
- **Principle #2.** ***The functions that the software performs must be defined. –*** that delivers direct business functions and NFRs***.***
- **Principle #3.** ***The behavior of the software (as a consequence of external events) must be represented.-*** Define system behavior in relation to external system interfaces.
- **Principle #4.** ***The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion. –*** Break business requirements into components and integrate with work/data flow.
- **Principle #5.** ***The analysis task should move from essential information toward implementation detail. –*** Explain in an automated system how a business function will be executed by the end user – the workflow and screen navigation

Design Modeling Principles

- **Principle #1.** ***Design should be traceable to the requirements model -*** The design model translates requirements into an architecture, a set of subsystems that implement major functions, and a set of components that are the realization of requirements classes. The elements of the design model should be traceable to the requirements model.
- **Principle #2.** ***Always consider the architecture of the system to be built.*** - For all of these reasons, design should start with architectural considerations. Only after the architecture has been established should component-level issues be considered
- **Principle #3.** ***Design of data is as important as design of processing functions.*** – data engineering , schema model
- **Principle #5. User interface design should be tuned to the needs of the end-user. However, in every case, it should stress ease of use.** – GUI, inter process /module , database communication
- **Principle #6. Component-level design should be functionally independent. -** The functionality that is delivered by a component should be cohesive—that is, it should focus on one and only one function or sub function.

Design Modeling Principles

- **Principle #7. Components should be loosely coupled to one another and to the external environment.** Design enterprise message bus and components should hang on them

- **Principle #8. Design representations (models) should be easily understandable.** Use standard document techniques for design, data modelling and program specification.

- **Principle #9. The design should be developed iteratively. With each iteration, the designer should strive for greater simplicity.**

Construction Principles

- The construction activity encompasses a set of coding and testing tasks that lead to operational software that is ready for delivery to the customer or end-user.
- Coding principles and concepts are closely aligned programming style, programming languages, and programming methods.
- Testing principles and concepts lead to the design of tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

Preparation Principles

- ***Before you write one line of code, be sure you:***
  - Understand of the problem you're trying to solve.
  - Understand basic design principles and concepts. – Architecture, re-usable artifacts, data model
  - Pick a programming language that meets the needs of the software to be built and the environment in which it will operate.- Customer my also decide the environment.
  - Select a programming environment that provides tools that will make your work easier**. Leverage Integrated Development Environment (IDE)**
  - Create a set of unit tests that will be applied once the component you code is completed. – Try and automate unit test code generation

Coding Principles

- ***As you begin writing code, be sure you:***
  - Constrain your algorithms by following structured programming practice.
  - Consider the use of pair programming
  - Select frameworks that will meet the needs of the design.
  - Understand the software architecture and create interfaces that are open and standards based .
  - Keep conditional logic as simple as possible.
  - Create nested loops in a way that makes them easily testable.
  - Follow naming convention. Select meaningful variable names and follow other local coding standards.
  - Document code as you develop code.
  - Use tools as appropriate for document generation

Validation Principles

- ***After you've completed your first coding pass, be sure you:***
  - Conduct a code walkthrough when appropriate.
  - Perform unit tests and correct errors you've uncovered.
  - Refactor the code and optimize code

Testing Principles

- **Principle #1.** ***All tests should be traceable to customer requirements. – RTM***

- **Principle #2.** ***Tests should be planned long before testing begins. Parallel activity along with Design***

- **Principle #3.** ***The Pareto principle applies to software testing. –*** *80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components.*

- **Principle #4.** ***Testing should begin "in the small" and progress toward testing "in the large."***

- **Principle #5.** ***Exhaustive testing is not possible.*** *To adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.*

Deployment Principles

- **Principle #1.** ***Customer expectations for the software must be managed.*** Too often, the customer expects more than the team has promised to deliver, and disappointment occurs immediately. **Under promise over deliver**

- **Principle #2.** ***A complete delivery package should be assembled and tested.-*** Auto install on target environment (different from dev. Environment)

- **Principle #3.** ***A support regime must be established before the software is delivered.*** An end-user expects responsiveness and accurate information when a question or problem arises.

- **Principle #4.** ***Appropriate instructional materials must be provided to end-users. – online help***

- **Principle #5.** ***Buggy software should be fixed first, delivered later.*** Make sure all critical defects are fixed and tested

Credits

- Software Engineering 7/ed by Roger Pressman and
- Other Internet sources.


Thank You

Course Name : Software Engineering

BITS Pilani
Pilani Campus

S Subramanian
Work-Integrated Learning Programme

Module Name : Requirements Engineering

BITS Pilani
Pilani Campus

S Subramanian
Work-Integrated Learning Programme

innovate achieve lead

- **Understanding Requirements**

Requirements Engineering-I

- Inception—ask a set of questions that establish …
  - Business users identify a opportunity for automation
  - Customer IT dept. and users have a basic understanding of the problem
  - A high level understanding of the nature of the solution that is desired, and
  - the effectiveness of preliminary communication and collaboration between the customer and the developer
  - Requirements are subject to change
- Elicitation—elicit requirements from all stakeholders. POC, Prototype
- Elaboration—create an analysis model that identifies data, function and behavioral requirements – Functional requirements
- Negotiation—agree on a deliverable system that is realistic for developers and customers. Phased / Incremental implementation. Set functional priorities

Requirements Engineering-II

- Specification—can be any one (or more) of the following:
  - A written document – SRS standards based UML
  - A set of models – RUP
  - A formal mathematical - Algorithm
  - A collection of user scenarios (use-cases)
  - A prototype/POC – GUI
- Business rules & functionality Validation—a review mechanism that looks for
  - errors in content or interpretation
  - missing information
  - inconsistencies (a major problem when large products or systems are engineered)
  - conflicting or unrealistic (unachievable) requirements.
- Requirements management – Change request

Inception

- Identify stakeholders
  - “who else do you think I should talk to?”
  - Who pays for the project
- Recognize multiple points (multi geography) of view
- Work toward collaboration – conflict resolution
- The first questions
  - Who is behind the request for this work (business users)?
  - 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?

Eliciting Requirements

- **The goal is**
  - to identify the problem
  - propose elements of the solution
  - negotiate different approaches, and
  - specify a preliminary set of solution requirements
- **Process to be adopted:**
  - Meeting with key stakeholders
  - Workshop with user communities
  - Use of domain specialist by service provider
  - Use of templates & questionnaire
  - All meetings are conducted and attended by both software engineers and customers
  - Interim deliverables – minutes, draft module requirements, GUI screes, standards – approval by customer.

Some characteristics of well-stated requirements

- **Complete**: Each requirement (Functional and non functional) must fully describe the functionality to be delivered. It contains all of the information necessary for the Developer to design and implement that functionality.
- **Correct**: Each requirement must accurately describe the functionality to be built.
- **Feasible:** It must be possible to implement each requirement within the known capabilities and limitations of the system and its environment.
- **Unambiguous: All readers should arrive at a single, consistent interpretation of the requirement**.

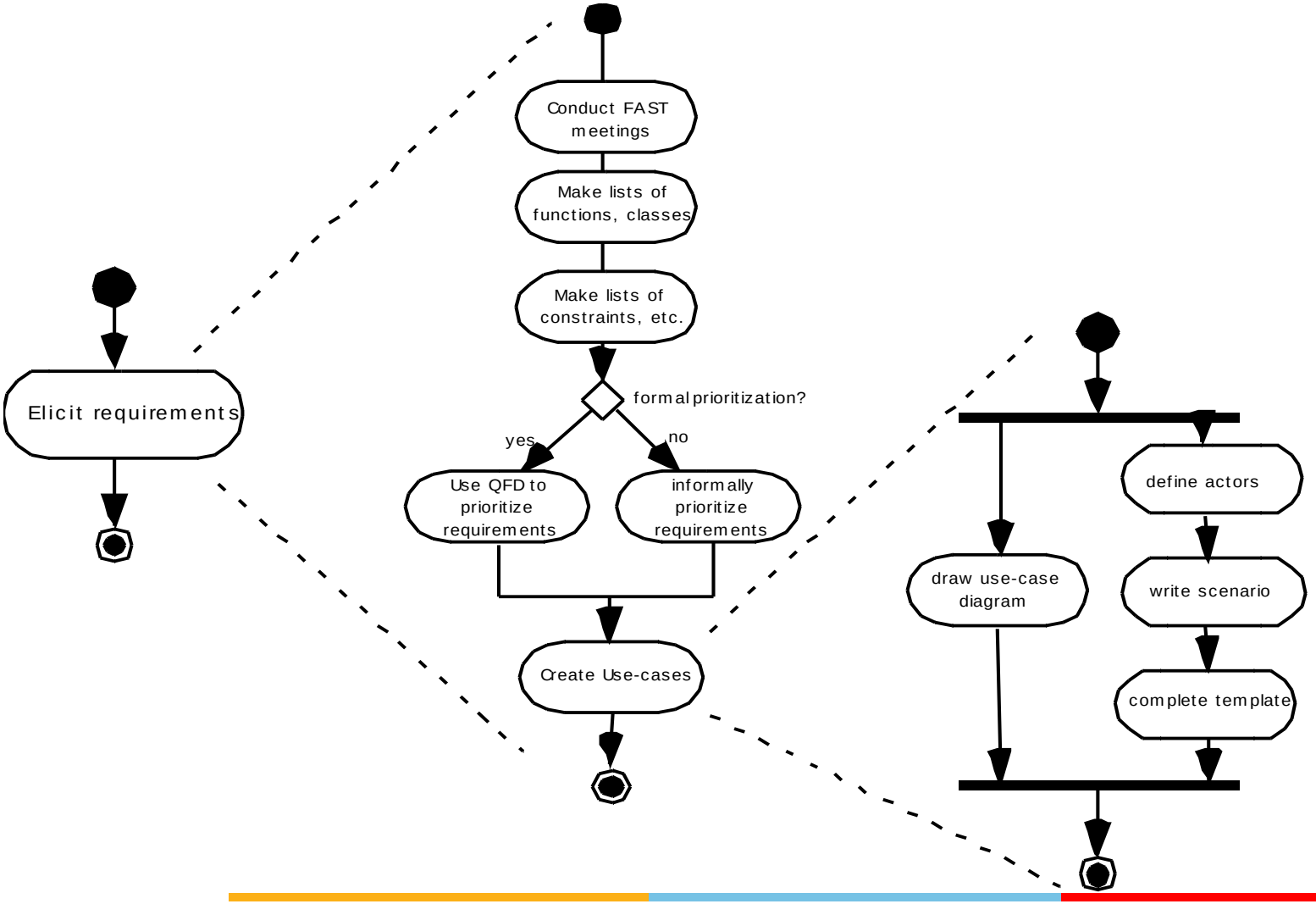
Quality Function Deployment

- Identify customer wants
- Identify how the good/service will satisfy customer wants
- Relate customer wants to product hows
- Identify relationships between the firm's hows
- Develop importance ratings
- Compare performance to desirable technical attributes

Quality Function Deployment (QFD)

- QFD emphasizes an understanding of what is valuable to the customer and then deploys these values throughout the engineering process
- **Normal requirements**
- **Expected requirements**
  - These 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**
  - delight every user of the product

Eliciting Requirements



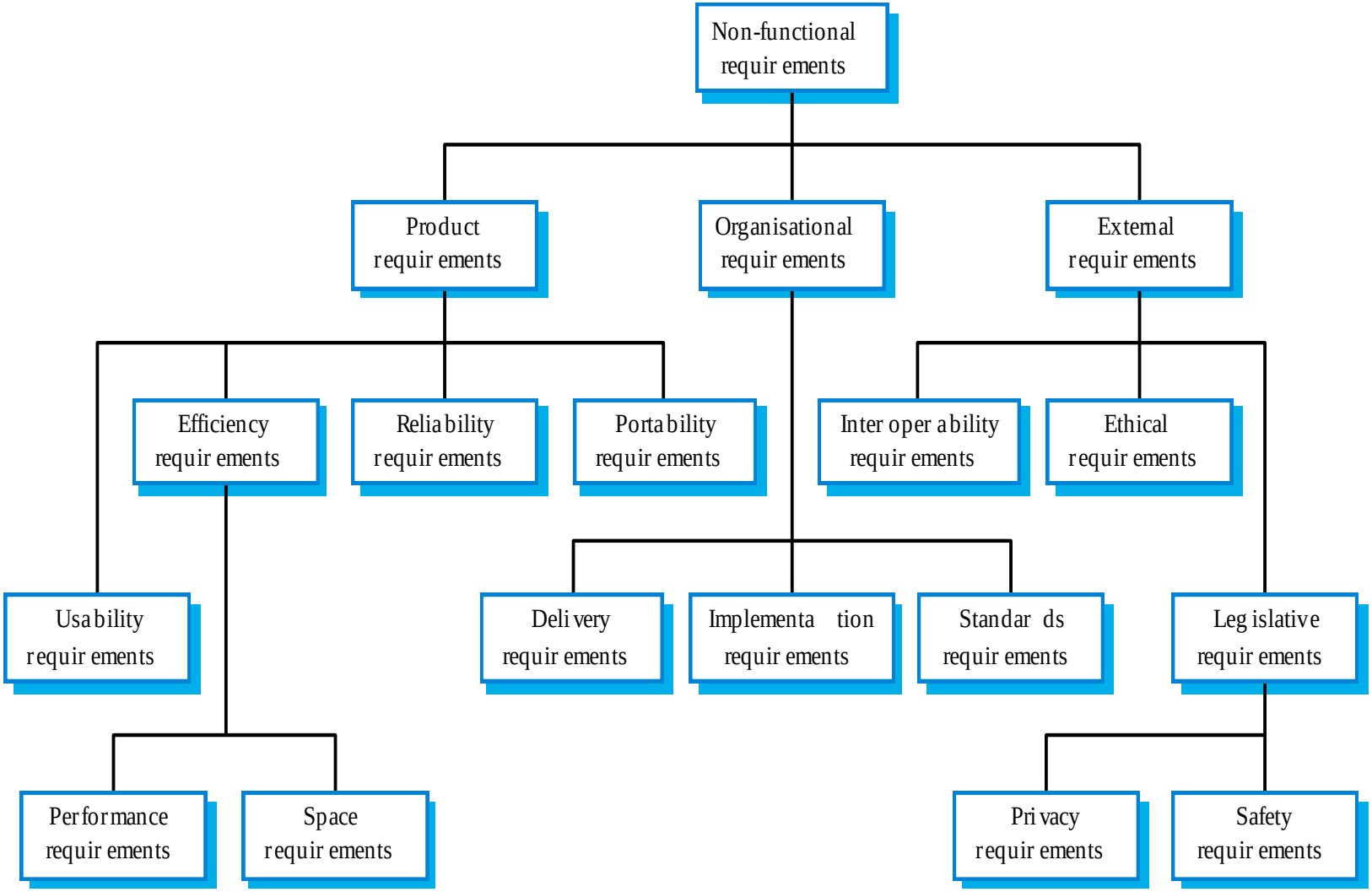
Requirements Phase output

- Business requirements document (with UC) with boundaries
- A list of customers, users, and other stakeholders who participated in requirements elicitation
- High level system's technical environment.
- List assumptions and dependencies.
- Description Non functional Requirements.
- GUI and Reports screen and layouts
- Prototype / PoC developed to better define requirements.

Validating Requirements

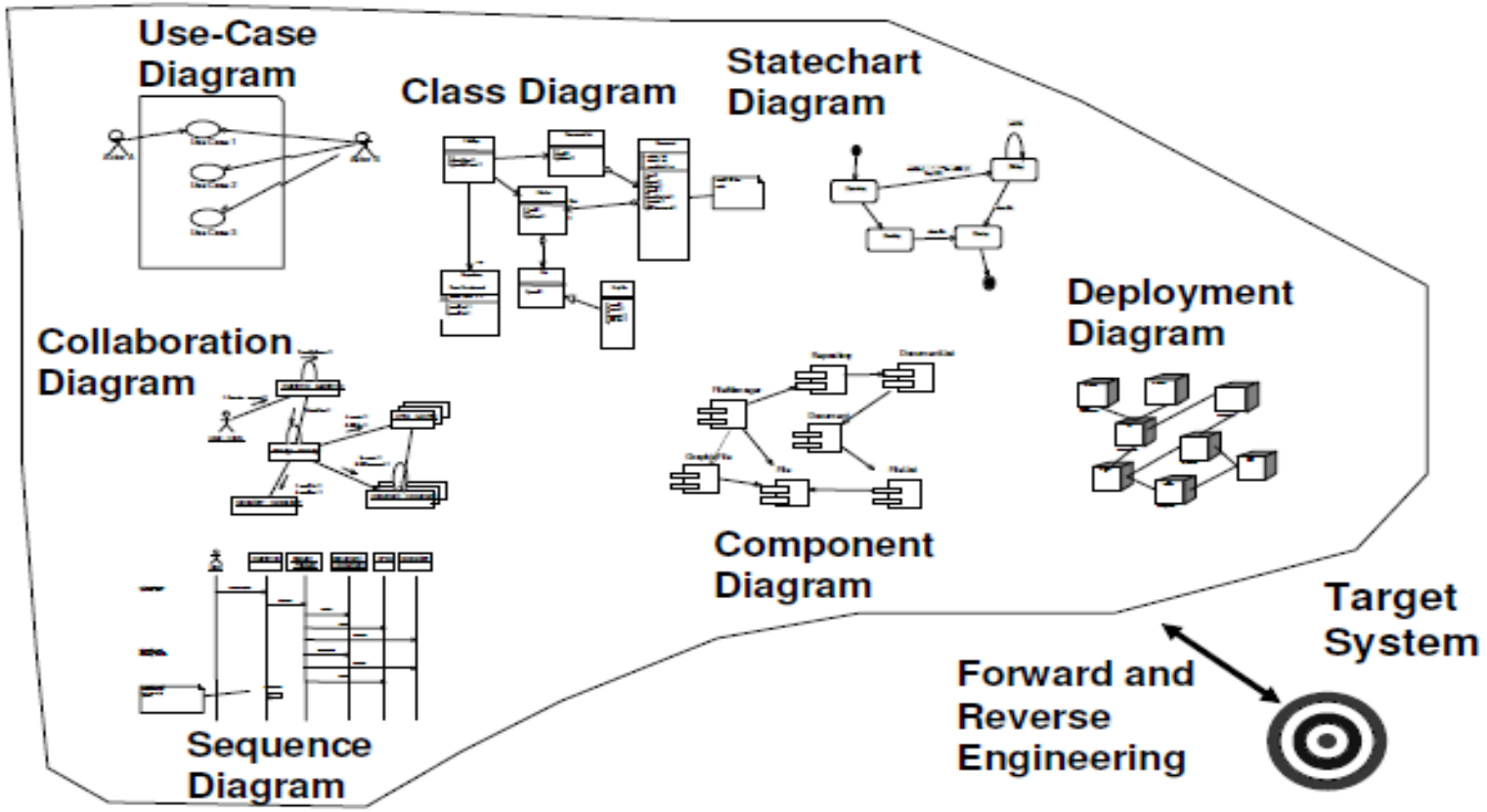
- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Do any requirements conflict with other requirements?
- Is each requirement achievable in the technical environment that will house the system or product?
- Have requirements patterns been used to simplify the requirements model

Non-functional Requirement Types



Building the Analysis Model

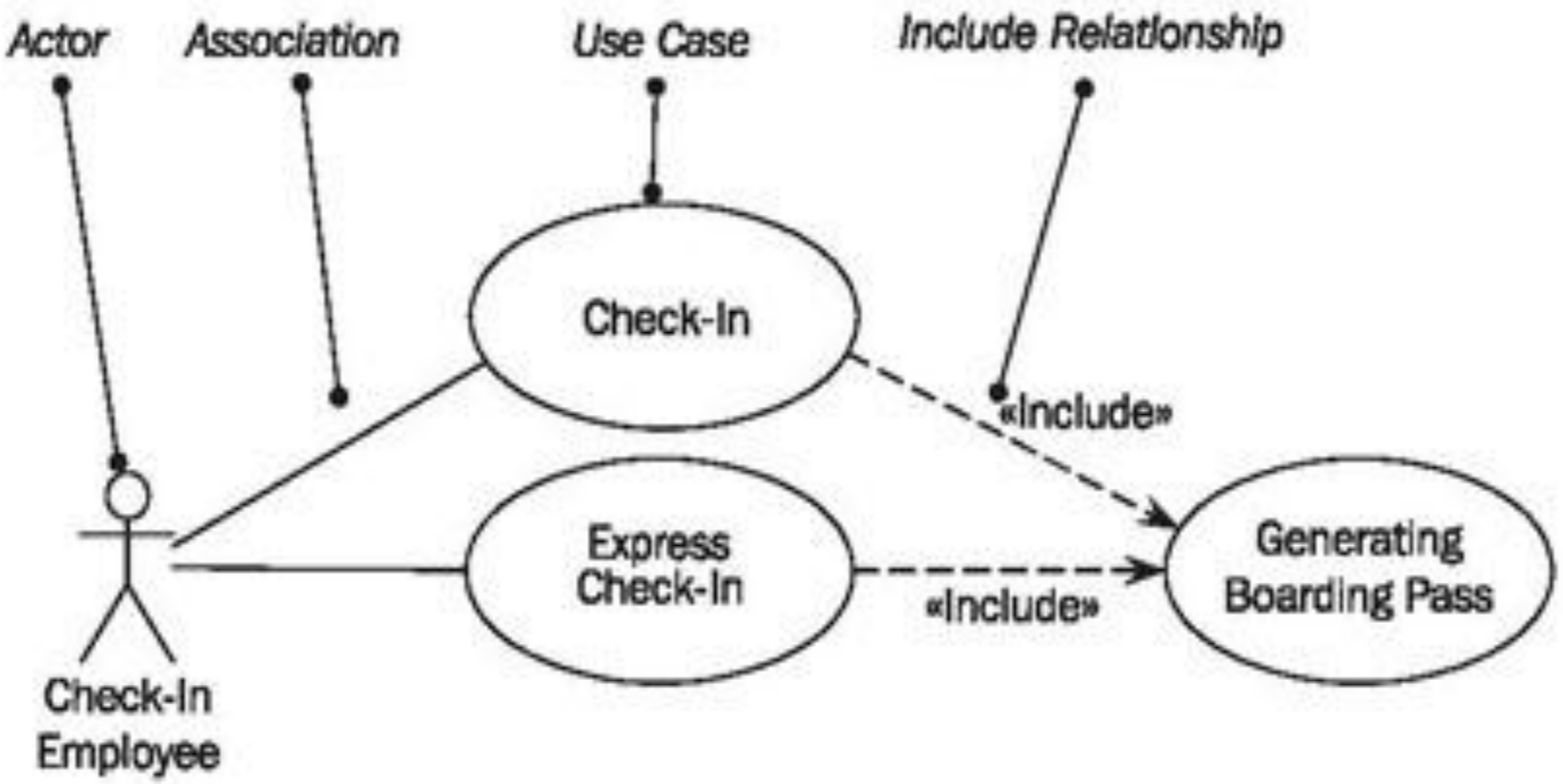
Visual Modeling Using UML Diagrams



Use-Cases

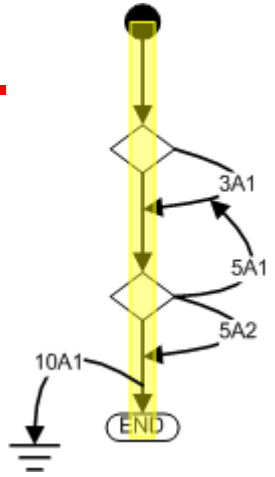
- A collection of user scenarios that describe the thread of usage of a system
- Each scenario is described from the point-of-view of an "actor"—a person or device that interacts with the software in some way
- Each scenario answers the following questions:
  - Who is the primary actor, the secondary actor (s)?
  - What are the actor's goals?
  - What preconditions should exist before the story begins?
  - What main tasks or functions are performed by the actor?
  - What extensions might be considered as the story is described?
  - What variations in the actor's interaction are possible?
  - What system information will the actor acquire, produce, or change?
  - Will the actor have to inform the system about changes in the external environment?
  - What information does the actor desire from the system?
  - Does the actor wish to be informed about unexpected changes?

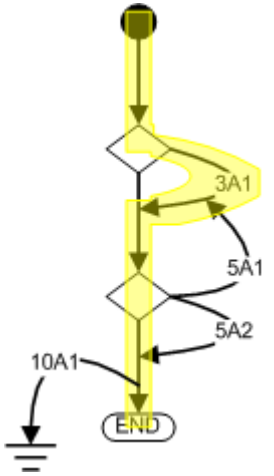
Use case Diagram



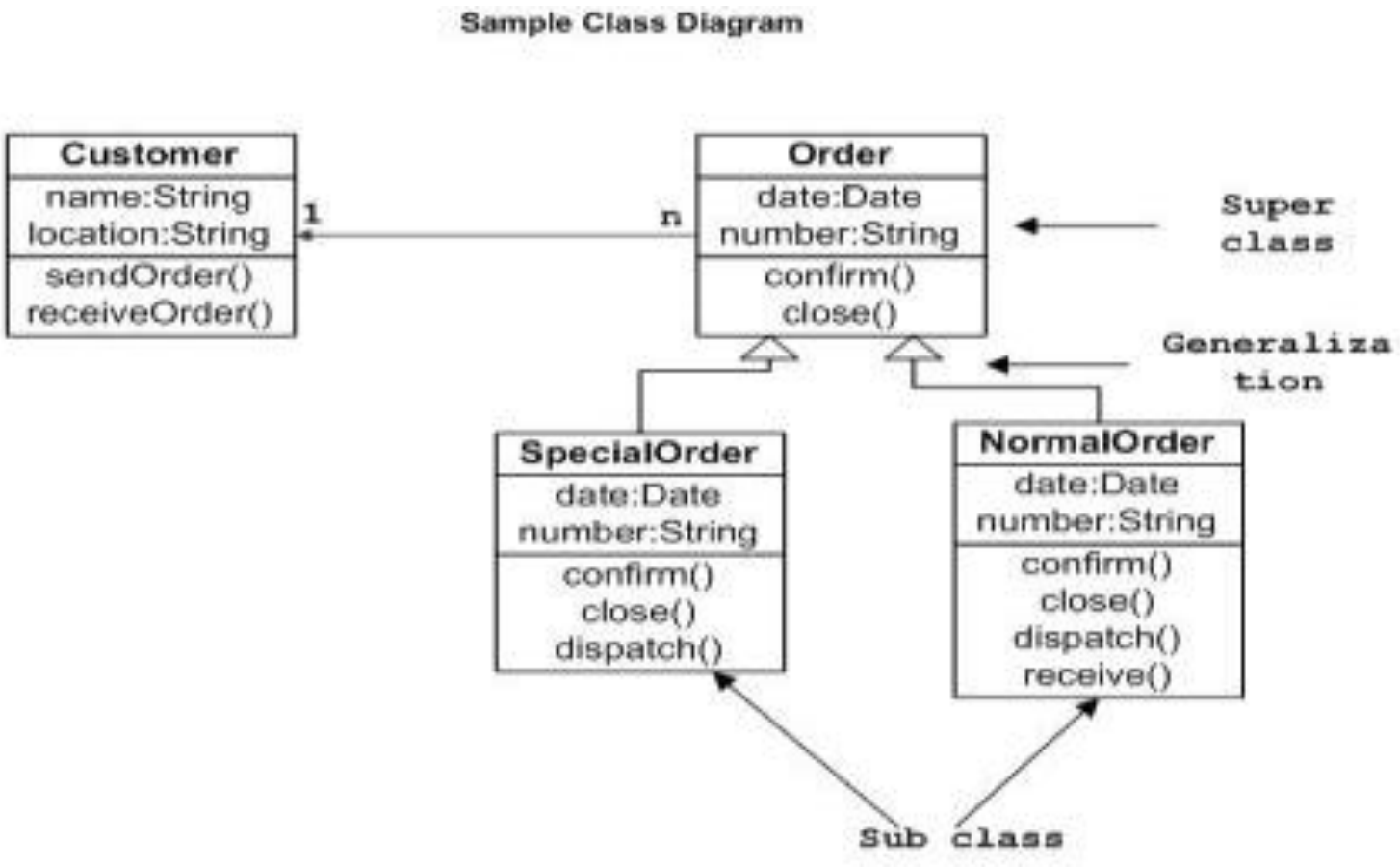
What Are Use Case Scenarios

- A use case represents the actions that are required to enable or abandon a goal.
- A use case has multiple “paths” that can be taken by any user at any one time.
- A use case scenario is a single path through the use case

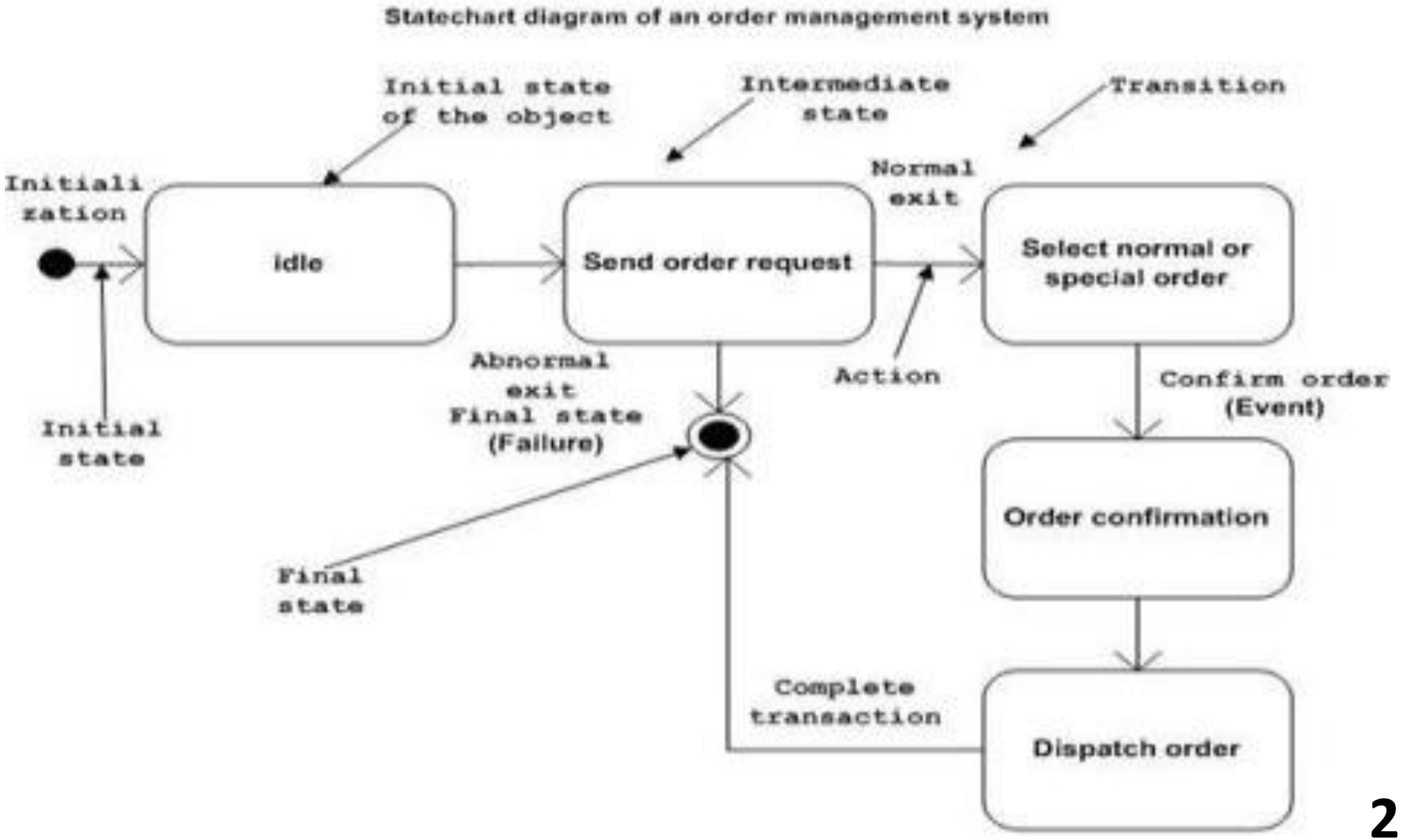




Class Diagram



State Diagram



Analysis Patterns

- These *analysis patterns* suggest solutions (e.g., a class, a function, a behavior) within the application domain that can be reused when modeling many applications.
- Analysis patterns are integrated into the analysis model by reference to the pattern name. They are also stored in a repository so that requirements engineers can use search facilities to find and apply them
- Re-usable artifacts. Business objects

Negotiating Requirements

- Identify the key stakeholders
  - These are the people who will be involved in the negotiation
- Determine each of the stakeholders "win conditions"
  - Win conditions are not always obvious
- Negotiate
  - Work toward a set of requirements that lead to "win-win"

Credits

- Software Engineering 7/ed by Roger Pressman and
- Other Internet sources.


Thank You

Course Name : Software Engineering

T V Rao

Software Quality & Testing Overview

BITS Pilani
Pilani Campus

Quality—A Philosophical View

Robert Persig [74] commented on the thing we call *quality:*

Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile? Obviously some things are better than others . . . but what's the betterness? . . . So round and round you go, spinning mental wheels and nowhere finding anyplace to get traction. What the hell is Quality? What is it?

Quality—A Pragmatic View

The *transcendental view argues (like Persig) that quality* is something that you immediately recognize, but cannot explicitly define.

The *user view sees quality in terms of an end-user's* specific goals. If a product meets those goals, it exhibits quality.

The *manufacturer's view defines quality in terms of the* original specification of the product. If the product conforms to the spec, it exhibits quality.

The *product view suggests that quality can be tied to* inherent characteristics (e.g., functions and features) of a product.

Finally, the *value-based view measures quality based on* how much a customer is willing to pay for a product. In reality, quality encompasses all of these views and more.

Software Quality

Software quality can be defined as:

*An **effective software process** applied in a manner that creates a **useful product** that provides measurable **value** for those who produce it and those who use it.*

- J Bessin, “The Business Value of Quality” (IBM DeveloperWorks)

Effective Software Process

innovate achieve lead

- An *effective software process* establishes the infrastructure that supports any effort at building a high quality software product.
- The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality.
- Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high quality software.
- Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

Useful Product

- A *useful product* delivers the content, functions, and features that the end-user desires
- But as important, it delivers these assets in a reliable, error free way.
- A useful product always satisfies those requirements that have been explicitly stated by stakeholders.
- In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high quality software.

Adding Value

- By *adding value for both the producer and user* of a software product, high quality software provides benefits for the software organization and the end-user community.
- The software organization gains added value because high quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.
- The user community gains added value because the application provides a useful capability in a way that expedites some business process.
- The end result is:
  - (1) greater software product revenue,
  - (2) better profitability when an application supports a business process, and/or
  - (3) improved availability of information that is crucial for the business.

Quality Dimensions

as per David Garvin[87]

Performance Quality. Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end-user?

Feature quality. Does the software provide features that surprise and delight first-time end-users?

Reliability. Does the software deliver all features and capability without failure? Is it available when it is needed? Does it deliver functionality that is error free?

Conformance. Does the software conform to local and external software standards that are relevant to the application? Does it conform to de facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

Quality Dimensions (contd...)

as per David Garvin[87]

Durability. Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? Will changes cause the error rate or reliability to degrade with time?

Serviceability. Can the software be maintained (changed) or corrected (debugged) in an acceptably short time period. Can support staff acquire all information they need to make changes or correct defects?

Aesthetics. Most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious “presence” that are hard to quantify but evident nonetheless.

Perception. In some situations, you have a set of prejudices that will influence your perception of quality.

Cost of Quality

innovate achieve lead

Prevention costs:

Quality planning
Formal technical reviews
Test equipment
Training

Internal failure costs:

Rework
Repair
failure mode analysis

Appraisal Costs:

Technical reviews
Data Collection
Testing & Debugging

External failure costs:

complaint resolution
product return and replacement
help line support
warranty work

Quality and Risk

innovate achieve lead

"People bet their jobs, their comforts, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right."

Example:

Throughout the month of November, 2000 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, five of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.

Negligence and Liability

The story is all too common. A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based "system" to support some major activity.

The system might support a major corporate function (e.g., pension management) or some governmental function (e.g., healthcare administration or homeland security).

Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad.

The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval.

Litigation ensues.

Quality and Security

"Software security relates entirely and completely to quality. You must think about security, reliability, availability, dependability—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process]. Even people aware of the software security problem have focused on late lifecycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. People pay too much attention to bugs and not enough on flaws."

Gary McGraw (ComputerWorld)

Achieving Software Quality

innovate achieve lead

Critical success factors:

Software Engineering Methods

Project Management Techniques

Quality Control

Quality Assurance

Achieving Software Quality

innovate achieve lead

Software Engineering Methods

- Understand the problem to be solved
- Create design that conforms to the problem
- Design and Software exhibit quality dimensions

Project Management Techniques

- Use estimation for achievable delivery dates
- Understand schedule dependencies and avoid short cuts
- Conduct risk planning

Achieving Software Quality

innovate achieve lead

Quality Control

- Reviews
- Inspection
- Testing
- Measurements and Feedback

Quality Assurance

- Establish infrastructure for software engineering
- Audit effectiveness and completeness of quality control

Software Quality Assurance

- **Standards**
- **Audits and Reports**
- **Error/defect collection and analysis**
- **Change management**
- **Education**
- **Vendor management**
- **Security management**
- **Safety**
- **Risk management**

innovate achieve lead

BITS Pilani

Pilani Campus

Software Testing Overview

Software Testing

innovate achieve lead

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Effective Testing

- To perform effective testing, a software team should conduct effective formal technical reviews
- Testing begins at the component level and work outward toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects) by an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

Testing Strategy

- A strategy for software testing integrates the design of software test cases into a well-planned series of steps that result in successful development of the software
- The strategy provides a road map that describes the steps to be taken, when, and how much effort, time, and resources will be required
- The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation
- The strategy provides guidance for the practitioner and a set of milestones for the manager
- Because of time pressures, progress must be measurable and problems must surface as early as possible

Successful Strategies for Testing

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective technical reviews as a filter prior to testing
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

-Tom Gilb

Verification and Validation

- Software testing is part of a broader group of activities called verification and validation (V & V)
- Verification (Are the algorithms coded correctly?)
  - The set of activities that ensure that software correctly implements a specific function or algorithm
  - Are we building the product right? [Boehm]
- Validation (Does it meet user requirements?)
  - The set of activities that ensure that the software that has been built is traceable to customer requirements
  - Are we building the right product? [Boehm]

When is Testing Complete?

- There is no definitive answer to this question
- Every time a user executes the software, the program is being tested
- Sadly, testing usually stops when a project is running out of time, money, or both
- One approach is to divide the test results into various severity levels
  - Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated

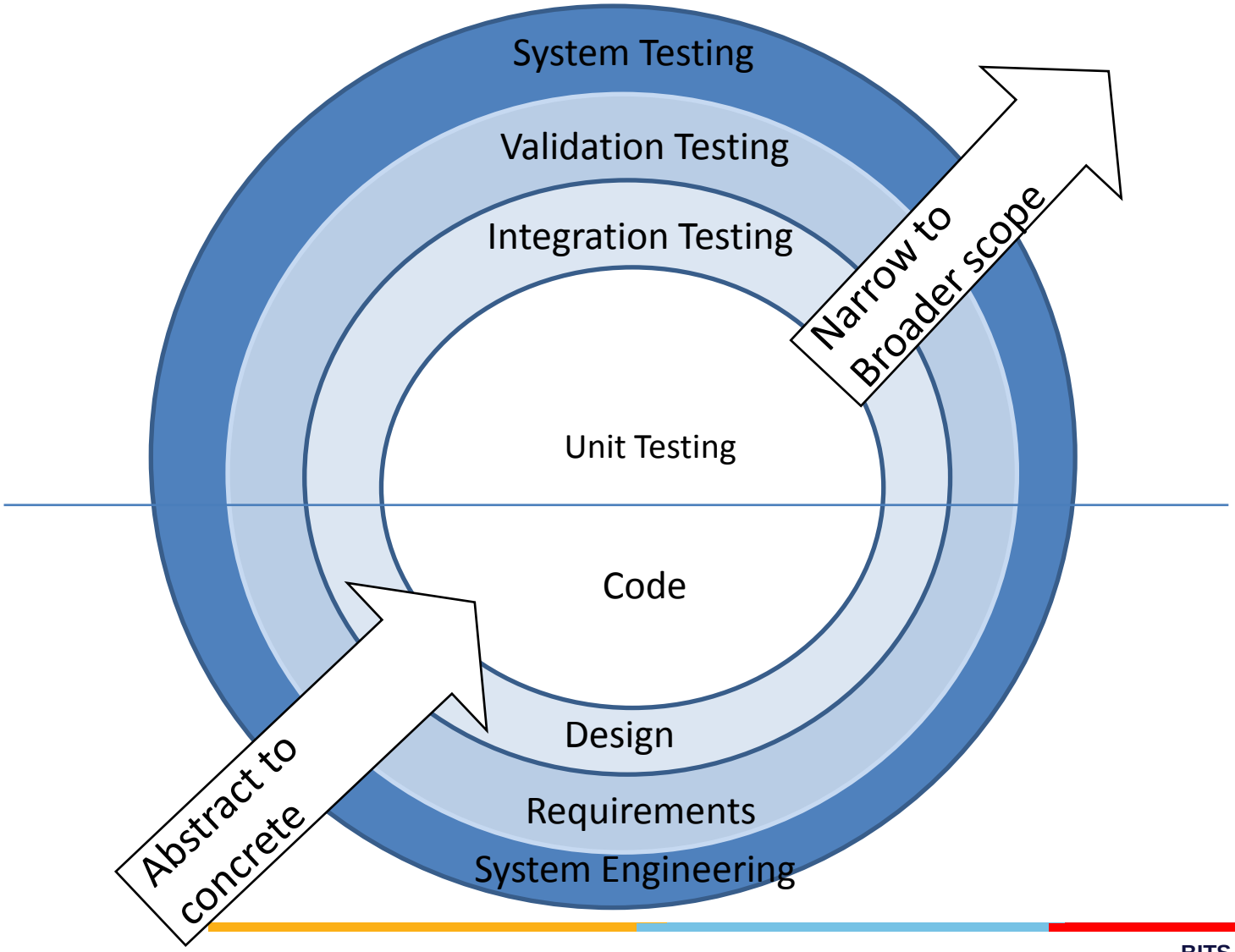
Organizing for Software Testing

- Testing should aim at "breaking" the software
- Some misconceptions
  - The developer of software should do no testing at all
  - The software should be given to a secret team of testers who will test it unmercifully
  - The testers get involved with the project only when the testing steps are about to begin
- Reality: Independent test group
  - Removes the inherent problems associated with letting the builder test the software that has been built
  - Removes the conflict of interest that may otherwise be present
  - Works closely with the software developer during analysis and design to ensure that thorough testing occurs

Levels of Testing for Software

- Unit testing
  - Concentrates on each component/function of the software as implemented in the source code
- Integration testing
  - Focuses on the design and construction of the software architecture
- Validation testing
  - Requirements are validated against the constructed software
- System testing
  - The software and other system elements are tested as a whole

Levels of Testing for Software (contd…)



Levels of Testing for Software (contd…)

- Unit testing
  - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
  - Components are then assembled and integrated
  - Can be white-box or black-box
- Integration testing
  - Focuses on inputs and outputs, and how well the components fit together and work together
  - Can take forms of regression testing and smoke testing
- Validation testing
  - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- System testing
  - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Black-box Testing and White-box Testing

- Black-box testing
  - Knowing the specified function that a product has been designed to perform, test to see if that function is fully operational and error free
  - Includes tests that are conducted at the software interface
  - Not concerned with internal logical structure of the software

- White-box testing
  - Knowing the internal workings of a product, test that all internal operations are performed according to specifications and all internal components have been exercised
  - Involves tests that concentrate on close examination of procedural detail
  - Logical paths through the software are tested
  - Test cases exercise specific sets of conditions and loops

Test Strategies for Object-Oriented Software

- Must broaden testing to include detections of errors in analysis and design models
- With object-oriented software, you can no longer test a single operation in isolation (conventional thinking)
- Traditional top-down or bottom-up integration testing has little meaning
- Use the same philosophy but different approach as in conventional software testing
- Class testing for object-oriented software is the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class
- Test "in the small" and then work out to testing "in the large"
  - Testing in the small involves class attributes and operations; the main focus is on communication and collaboration within the class
  - Testing in the large involves a series of regression tests to uncover errors due to communication and collaboration among classes

Test Strategies for Object-Oriented Software (contd..)

- Different object-oriented testing strategies (for integration)
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—start with independent classes & proceed to integrate dependent classes
  - cluster testing—integrates the set of classes required to demonstrate one collaboration
- Finally, the system as a whole is tested to detect errors in fulfilling requirements

WebApp Testing

- The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- Each functional component is unit tested.
- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end-users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

High Order Testing

- Validation testing
  - Focus is on software requirements
- Alpha/Beta testing
  - Focus is on customer usage
- System testing
  - Focus is on system integration
- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - test the run-time performance of software within the context of an integrated system

Debugging Process

- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction

Why is Debugging Difficult?

- The symptom and the cause may be geographically remote
- The symptom may disappear (temporarily) when another error is corrected
- The symptom may actually be caused by nonerrors (e.g., round-off accuracies)
- The symptom may be caused by human error that is not easily traced
- The symptom may be a result of timing problems, rather than processing problems
- It may be difficult to accurately reproduce input conditions, such as asynchronous real-time information
- The symptom may be intermittent such as in embedded systems involving both hardware and software
- The symptom may be due to causes that are distributed across a number of tasks running on different processes

Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies
  - Brute force
  - Backtracking
  - Cause elimination

Three Questions to ask Before Correcting the Error

- Is the cause of the bug reproduced in another part of the program?
  - Similar errors may be occurring in other parts of the program
- What next bug might be introduced by the fix that I'm about to make?
  - The source code (and even the design) should be studied to assess the coupling of logic and data structures related to the fix
- What could we have done to prevent this bug in the first place?
  - This is the first step toward software quality assurance
  - By correcting the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs

To Summarize

- Software testing must be planned carefully to avoid wasting development time and resources.
- Testing begins “in the small” and progresses “to the large”.
- Initially individual components are tested and debugged.
- After the individual components have been tested and added to the system, integration testing takes place.
- Once the full software product is completed, system testing is performed.
- The Test Specification document should be reviewed like all other software engineering work products.

Thank You…

References

- Software Engineering 7/ed by Roger Pressman

Course Name : Software Engineering

BITS Pilani
Pilani Campus

S Subramanian
Work-Integrated Learning Programme

Module Name : M9.1 product Metrics

BITS Pilani
Pilani Campus

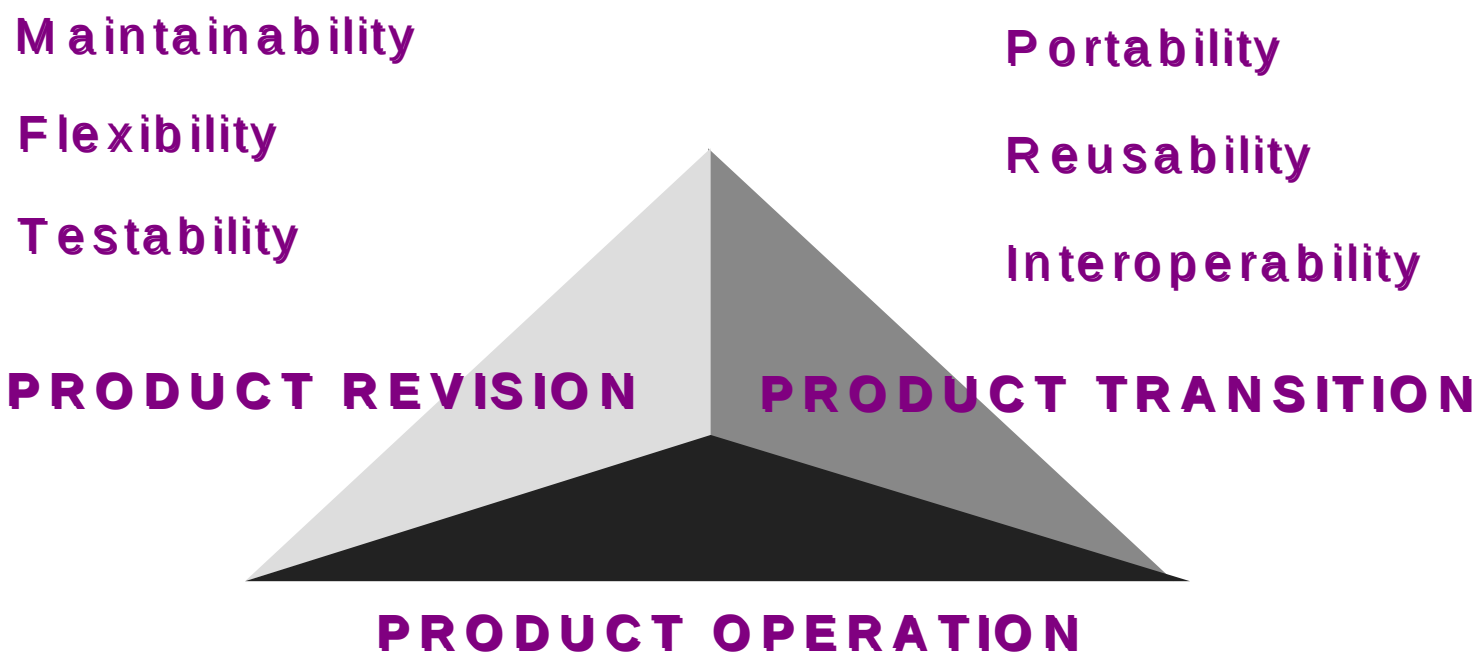
S Subramanian
Work-Integrated Learning Programme

Chapter 23

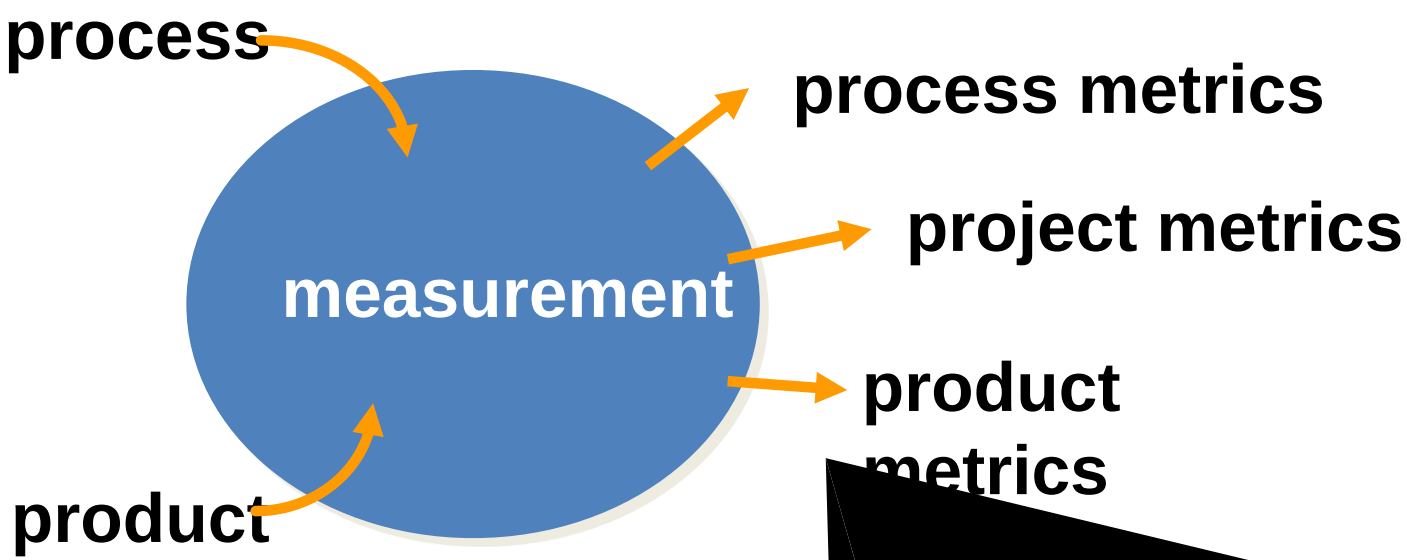
- **Software Metrics**

software + measures

McCall's Triangle of Quality



A Good Manager Measures

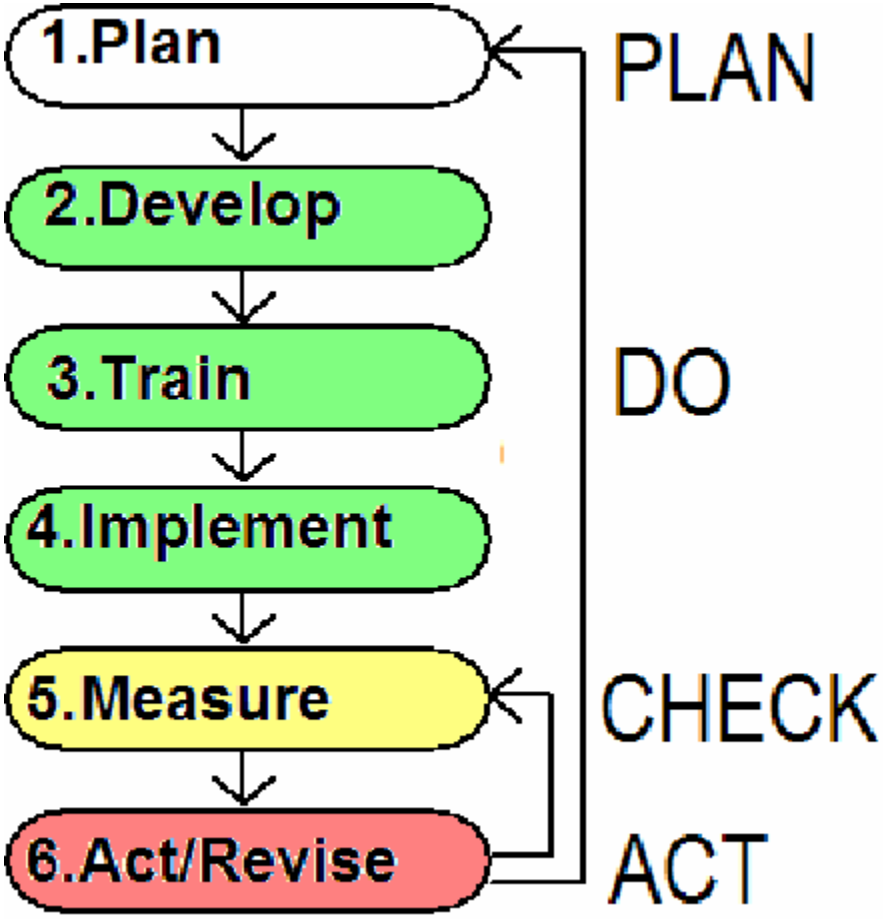


"Not everything that can be counted counts, and not everything that counts can be counted." - Einstein

What do we use as a basis?

- size?
- function?

Steps to implement software metrics



What Can Be Measured?

- Direct measures
  - Lines of codes (LOC), Churn, DRE, speed, cost, memory size, errors, ...
- Indirect measures
  - Quality, functionality, complexity, reliability, efficiency, maintainability, …

.

Example of Size-Oriented Metrics

- Productivity = Size / Effort
  = kLOC / person-month
- Quality = Errors / Size
  = Errors / kLOC
- Cost = $ / kLOC
- Documentation = pages / kLOC
- Other metrics can also be developed like: errors/KLOC, page/KLOC...etc.
- Or errors/person-month, LOC/person-month, cost/page.

Function-Oriented Metrics

- Based on “functionality” delivered by the software as the normalization value.
- Functionality is measured indirectly
- Function points (FP) measure- derived using an empirical relationship based on countable (direct) measures of software’s information domain and assessments of software complexity
- Number of requirements errors found (to assess quality)
- Change request frequency
  - To assess stability of requirements.
  - Frequency should decrease over time. If not, requirements analysis may not have been done properly.

Function-Based Metrics

- The *function point metric* (FP), first proposed by Albrecht [ALB79], can be used effectively as a means for measuring the functionality delivered by a system.
- Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity
- Information domain values are defined in the following manner:
  - **number of external inputs (EIs)**
  - **number of external outputs (EOs)**
  - **number of external inquiries (EQs)**
  - **number of internal logical files (ILFs)**
  - **Number of external interface files (EIFs)**

Total Unadjusted Function Points

Type of Component	Complexity of Components			
	Low	Average	High	Total
External Inputs	___ x 3 = ___	___ x 4 = ___	___ x 6 = ___	
External Outputs	___ x 4 = ___	___ x 5 = ___	___ x 7 = ___	
External Inquiries	___ x 3 = ___	___ x 4 = ___	___ x 6 = ___	
Internal Logical Files	___ x 7 = ___	___ x 10 = ___	___ x 15 = ___	
External Interface Files	___ x 5 = ___	___ x 7 = ___	___ x 10 = ___	
		Total Number of Unadjusted Function Points		_____

TDI = Sum of individual degree of influence of all 14 GSCs

VAF Formula : VAF = (65 + TDI)/100

FP = Unadjusted FP Count * VAF

Product Metrics

- Focus on the quality of deliverables
- Product metrics are combined across several projects to produce process metrics
- Metrics for the product:
  - Measures of the Analysis Model
  - Complexity of the Design Model
  - Internal algorithmic complexity
  - Architectural complexity
  - Data flow complexity
  - Code metrics

Software Design Metrics

- Number of parameters in modules
- Number of modules.
- Number of modules called (estimating complexity of maintenance)
- Data Bindings
- Triplet (p,x,q) where p and q are modules and X is variable within scope of both p and q
  - Actual data binding:
  - Used data binding:
  - Potential data binding:

OO Metrics: Distinguishing Characteristics

- The following characteristics require that special OO metrics be developed:
  - Encapsulation
  - Information hiding
  - Inheritance
  - Abstraction
- **Conclusion: the class is the fundamental unit of measurement**

OO Project Metrics

- Number of Scenario Scripts (Use Cases)
- Number of Key Classes (Class Diagram)
- Number of Subsystems (Package Diagram):

OO Analysis and Design Metrics

- Complexity:
  - Weighted Methods per Class (WMC): Assume that *n* methods with cyclomatic complexity are defined for a class C: $c_1, c_2, \ldots, c_n$

$$WMC = \sum c_i$$

  - Depth of the Inheritance Tree (DIT): The maximum length from a leaf to the root of the tree. Large DIT leads to greater design complexity but promotes reuse
  - Number of Children (NOC): Total number of children for each class. Large NOC may dilute abstraction and increase testing

Further OOA&D Metrics

- **Coupling:**
  - Coupling between Object Classes (COB): Total number of collaborations listed for each class in CRC cards. Keep COB low because high values complicate modification and testing
  - Response For a Class (RFC): Set of methods potentially executed in response to a message received by a class. High RFC implies test and design complexity
- **Cohesion:**
  - Lack of Cohesion in Methods (LCOM): Number of methods in a class that access one or more of the same attributes. High LCOM means tightly coupled methods

Cyclomatic complexity

- Among attempts to measure complexity, only cyclomatic complexity is still commonly collected

cyclomatic complexity V(g)

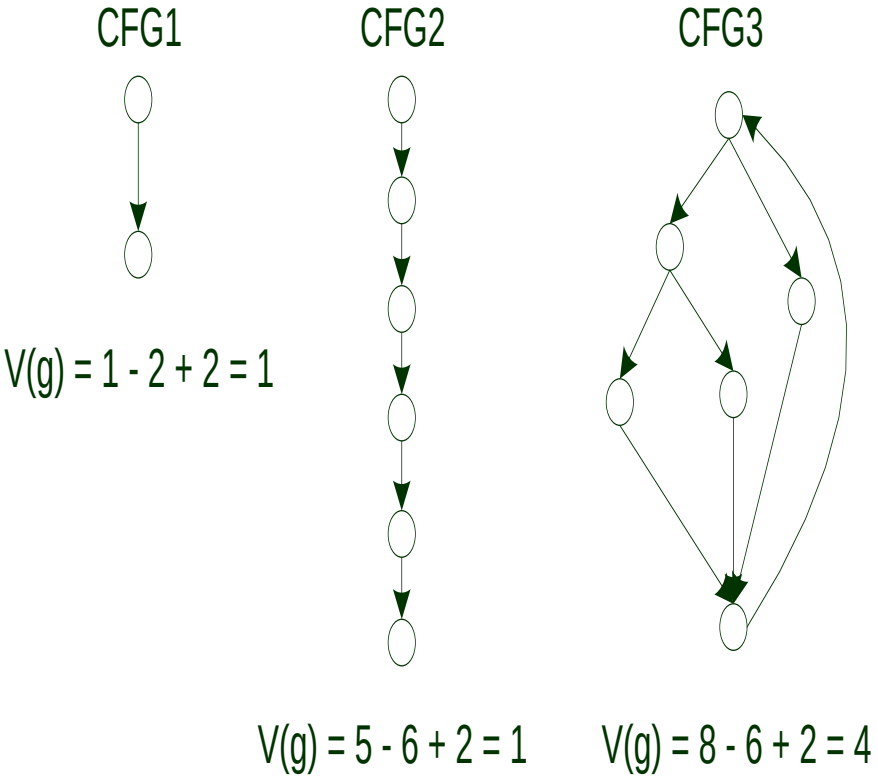
=

number of *independent paths* through the control flow graph

=

e - n + 2

(edges - nodes + 2)



Credits

- Software Engineering 7/ed by Roger Pressman and
- Other Internet sources.


Thank You