
S2-22_MERGEDSQM: Syllabus for SQM Mid-Sem Test

1 message

K VENKATASUBRAMANIAN . <noreply@wilp.bits-pilani.ac.in>
Reply-To: "K VENKATASUBRAMANIAN ." <kvenkat@pilani.bits-pilani.ac.in>
To: "HEMANT TIWARI ." <2022mt93184@wilp.bits-pilani.ac.in>

Wed, Mar 1, 2023 at 12:50 PM

[S2-22_MERGEDSQM](#) » [Forums](#) » [Announcements](#) » [Syllabus for SQM Mid-Sem Test](#)



Syllabus for SQM Mid-Sem Test
by [K VENKATASUBRAMANIAN](#) . - Wednesday, 1 March 2023, 12:49 PM

Dear SQM Students.

The syllabus for the SQM Mid-Semester Tests (both Regular as well as Makeup) will include:

- 1. All topics covered in the live lectures till 26th February 2022.**
- 2. Software Engineering Courseware modules M1, M2, M3, M4.1, M7, and M9.1.**
- 3. Software Quality Management Courseware Modules M1, M2, and M3,**
- 4. Chapters 1, 2, 3, 4, 5, 7, 8, 19, 20, 21, 22, and 30 of the prescribed textbook [Pressman, Software Engineering, 8th (Indian) edition, McGraw Hill, 2019]**

The Mid-Semester Test will be completely Open Book type, wherein you can refer to any reference material including the textbooks, reference books, lecture slides, class notes, and any downloaded reference material, **all in electronic or print form**.

For practice, you may please ponder over the questions at the end of the relevant chapters of the prescribed textbook, **Pressman, Software Engineering, 8th (Indian) edition, McGraw Hill, 2019**.

There is no need to refer to any other textbook. Hope this helps.

Best wishes,

Instructor, SQM

[See this post in context](#)

[Change your forum digest preferences](#)



BITS Pilani
Pilani Campus

Course Name : Software Engineering

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



• 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

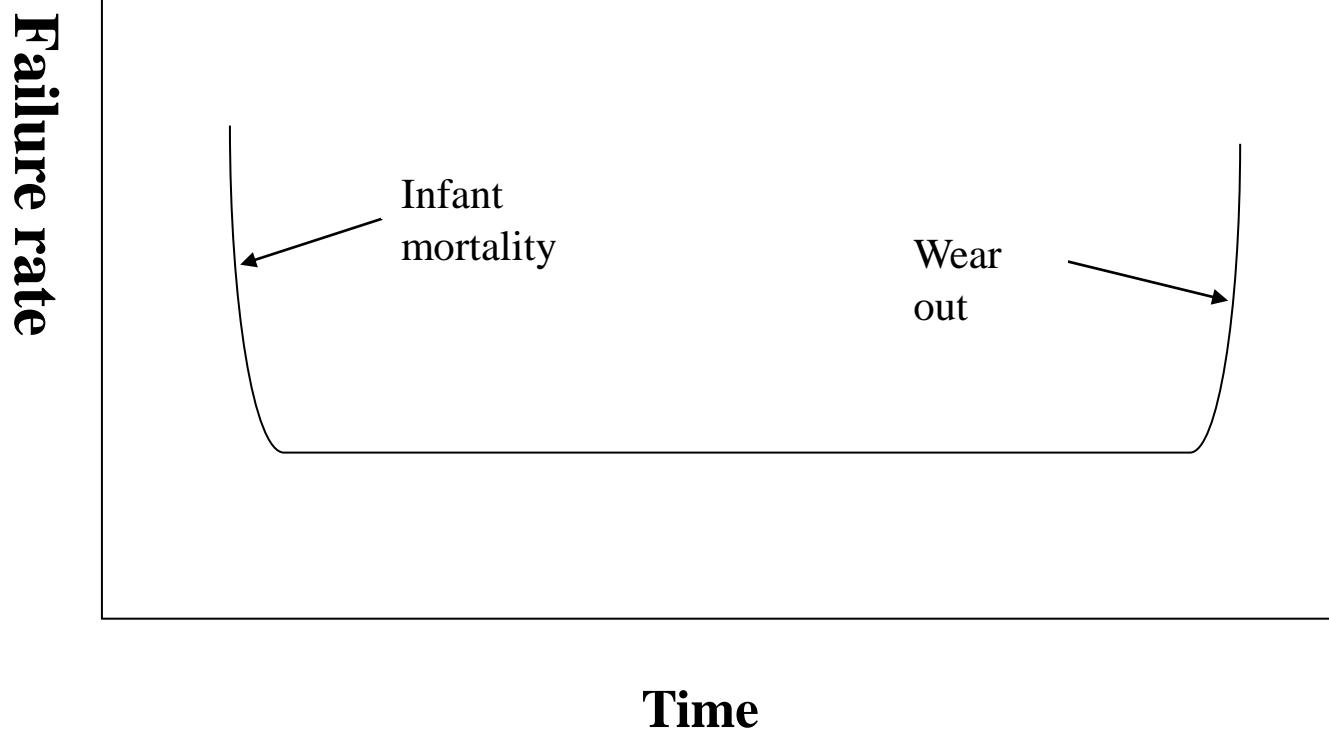
- Pattern recognition, neural networks...

Software Characteristics

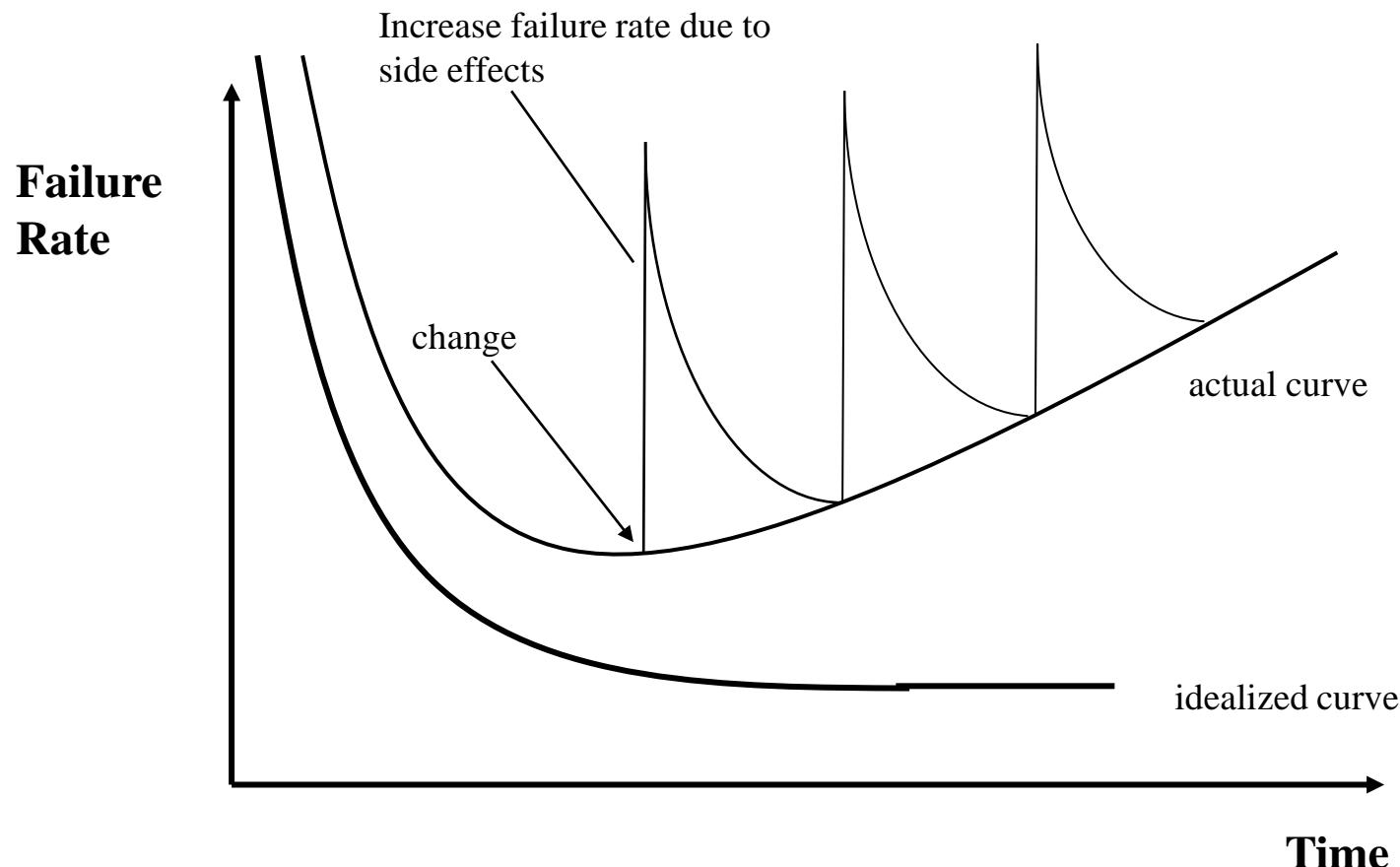


- Software is developed or engineered....Not manufactured in the classical sense
- Software does NOT “wear out”
 - Aging of software is very unlike material objects (hardware)
- Industry is moving towards component-based construction, but most of the software effort is for the custom-building

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 maintenance)

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



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

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



BITS Pilani
Pilani Campus

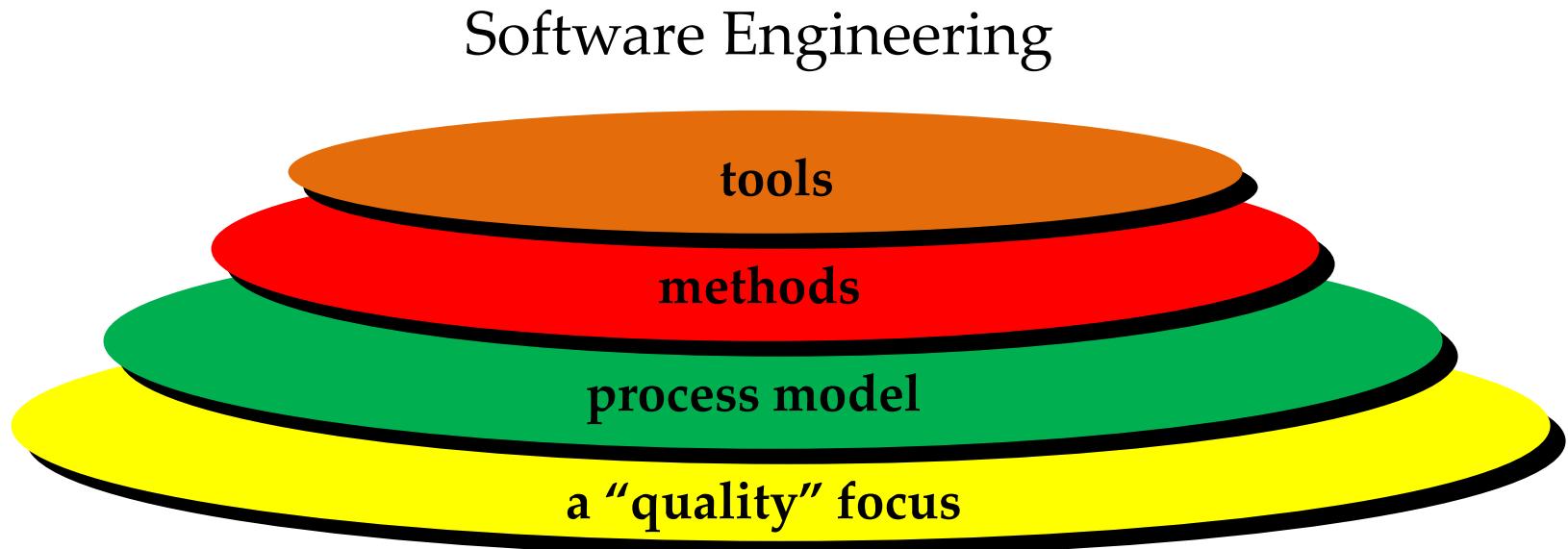
Course Name : Software Engineering

T V Rao
Software Engineering Definitions

Software Engineering - Definitions

- (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

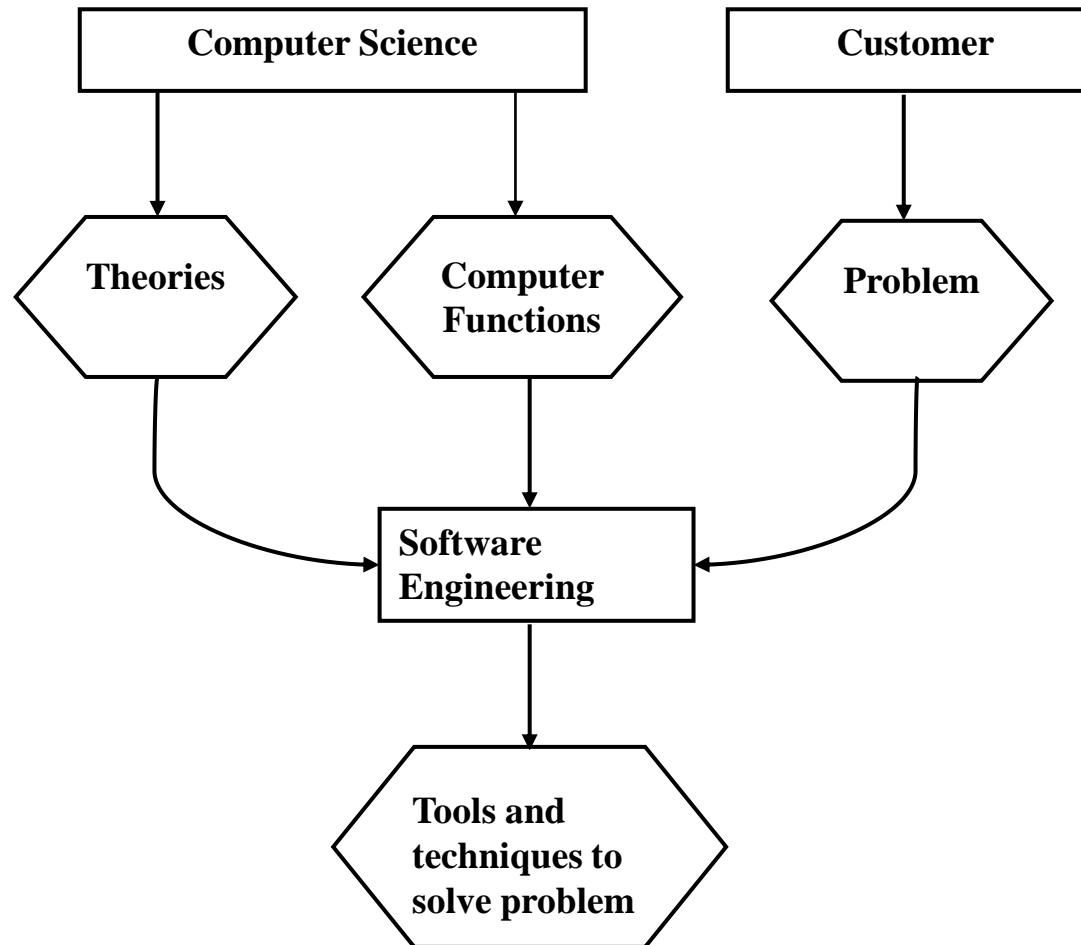
- A proven solution to a problem
 - Describes process-related problem
 - Environment in which it was encountered
- A template to describe solution
 - Ambler proposed a template for describing a process pattern
- Can be defined at any level of abstraction
 - Complete process model (e.g. Prototyping)
 - Framework 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

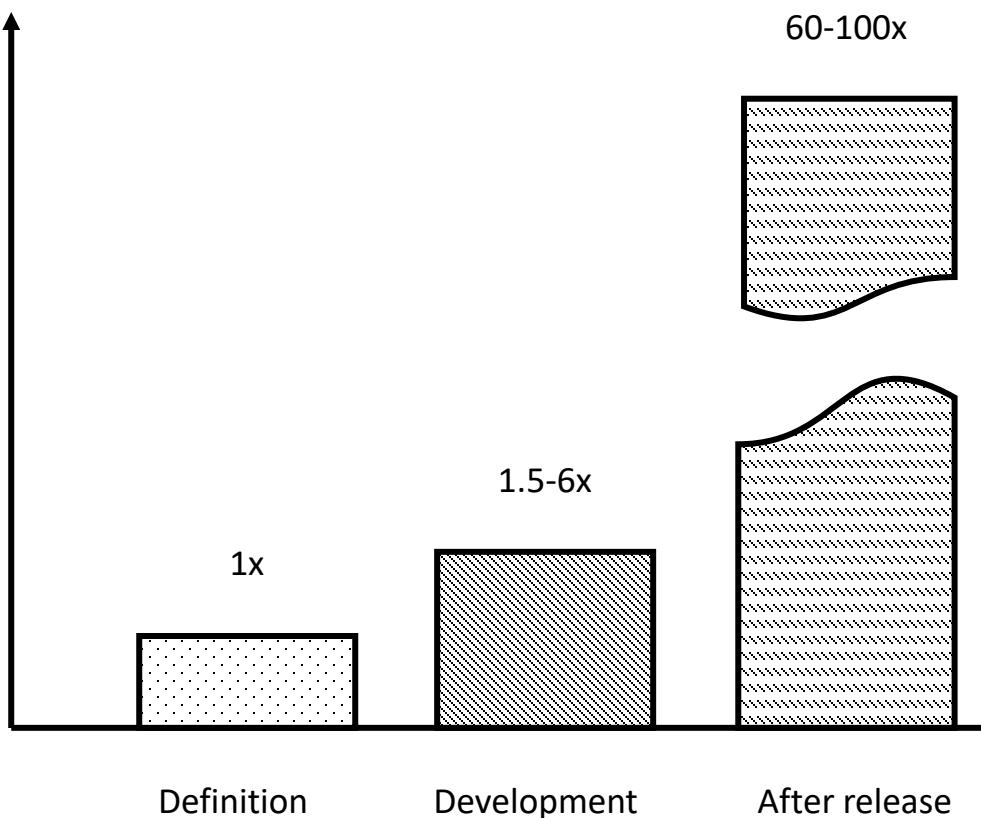
Software Myths - Management

- "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



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.*

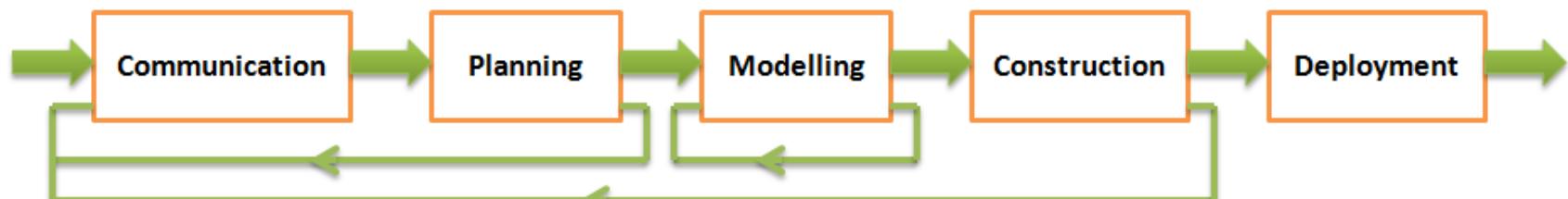
Prescriptive Process Model

- 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

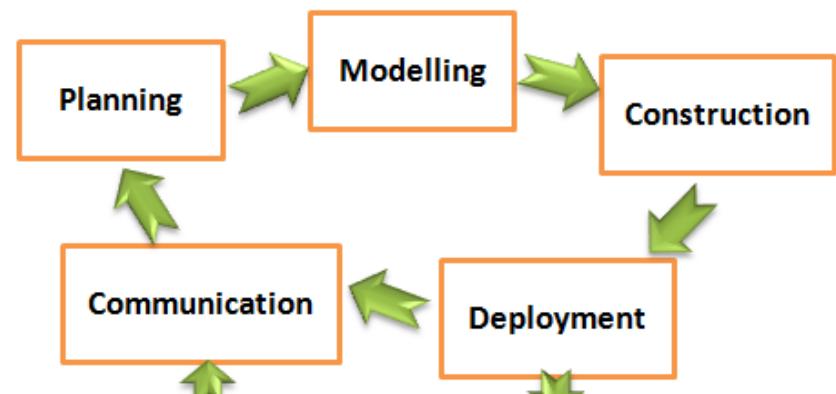


(a) Linear Process Flow



(b) Iterative Process Flow

An important variation among process models comes from flow of activities



(c) Evolutionary Process Flow

Waterfall Model



(Diagram)

Communication:

- Project Initiation
- Requirement Gatherings

Planning:

- Estimation
- Scheduling
- Tracking

Modelling:

- Analysis
- Design

Construction:

- Code
- Test

Deployment:

- Delivery
- Support
- Feedback

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.

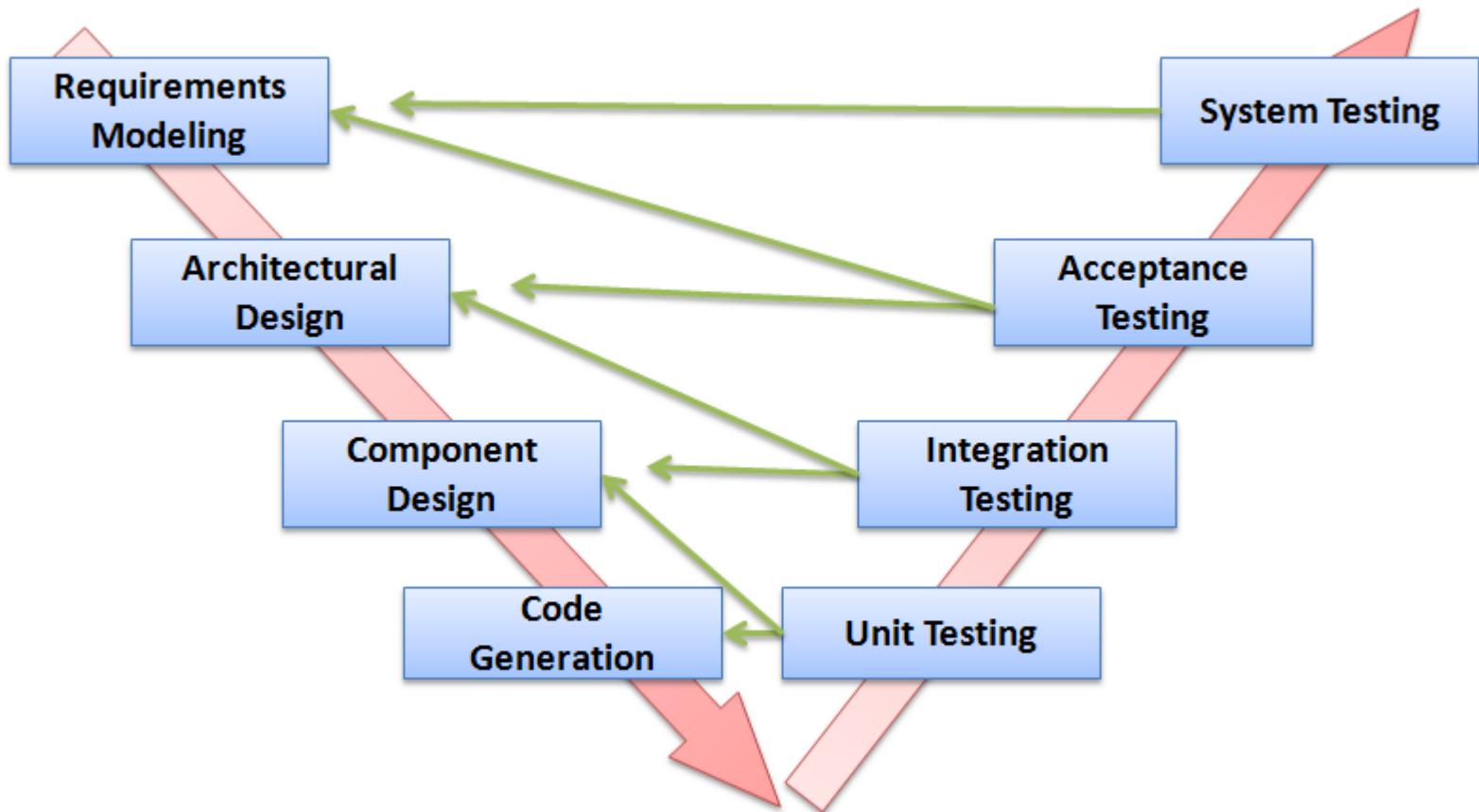
Critique of Waterfall Model



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

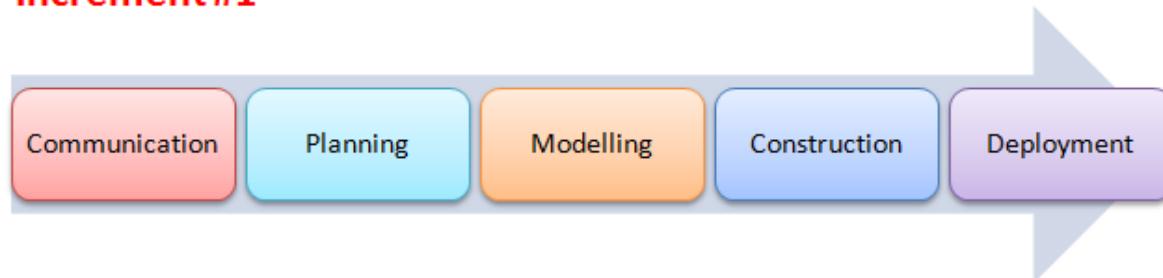


Incremental Model

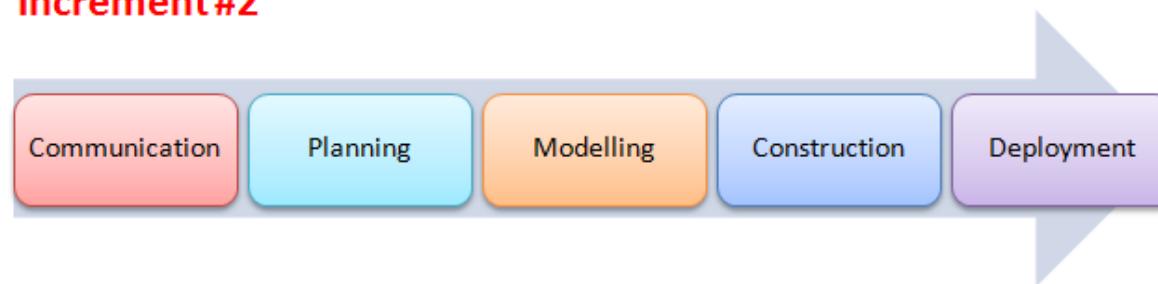
(Diagram)



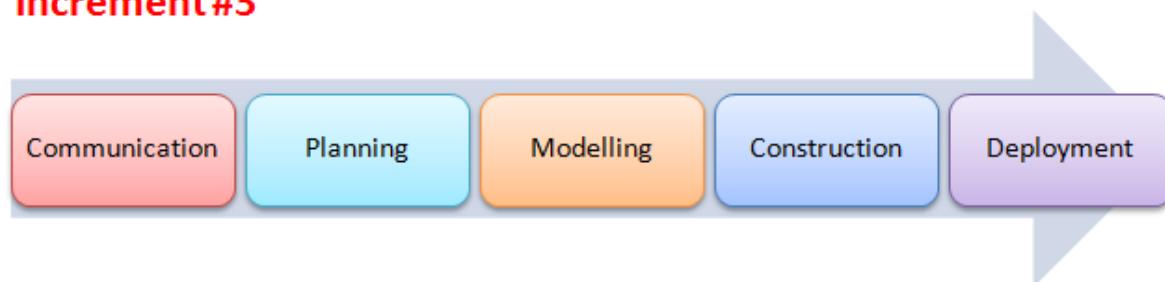
Increment#1



Increment#2



Increment#3



The Incremental Model

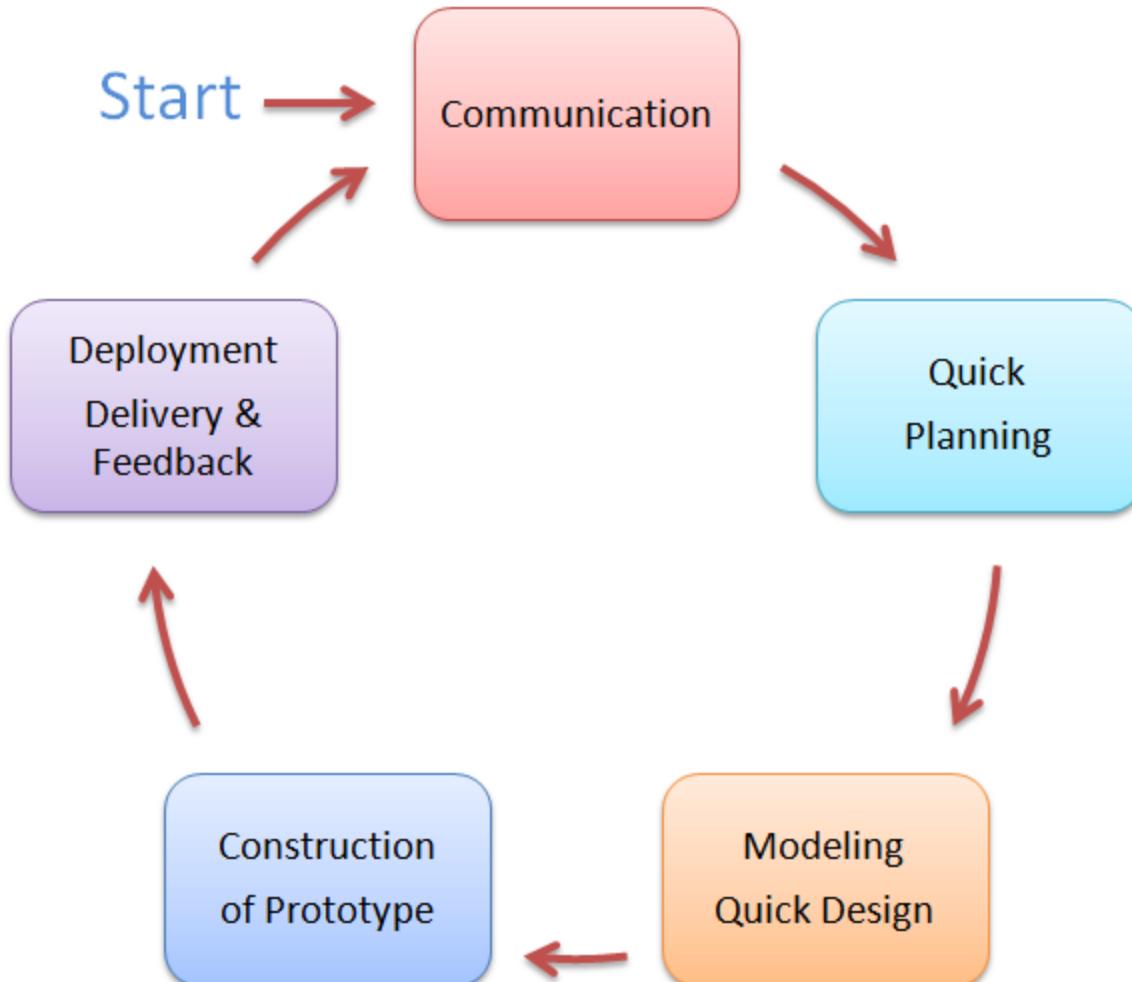
- Rather than deliver the system as a single delivery, the development and delivery is broken down into increments with each increment delivering part of the required functionality.
- First Increment is often core product
 - Includes basic requirement
 - Many supplementary features (known & unknown) 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

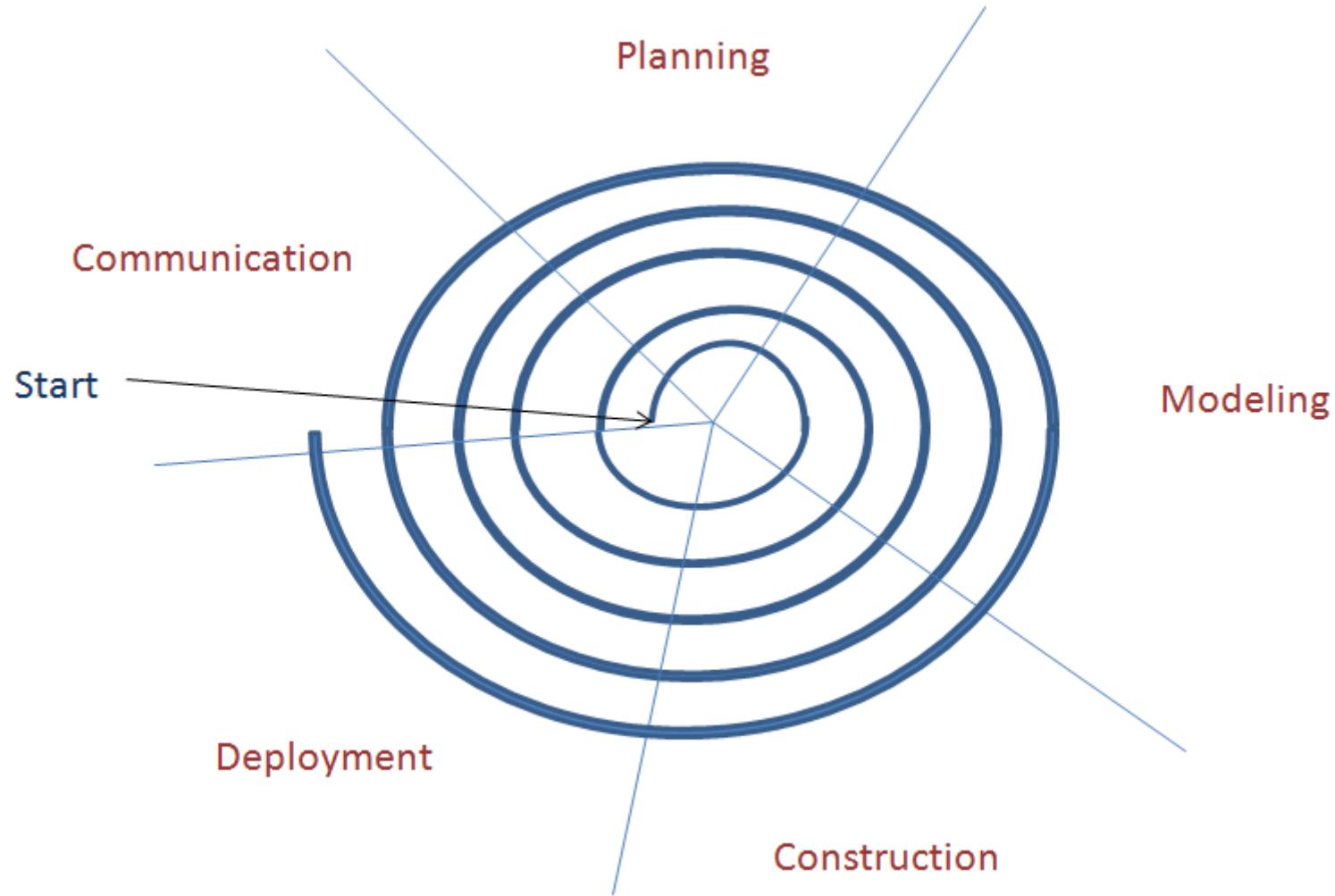
Prototyping Model

(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)



- 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)



- 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)



- 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

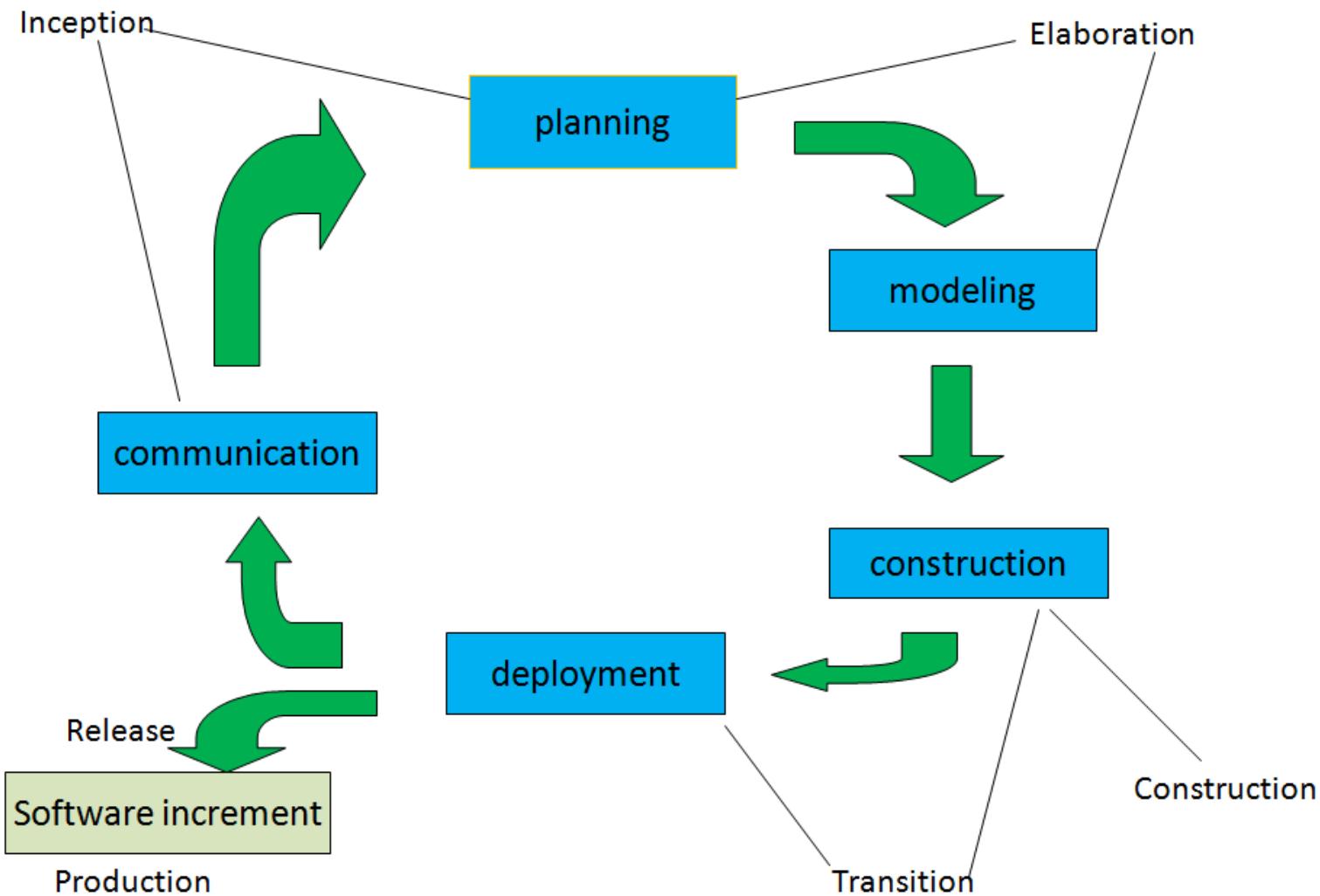
Background

(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

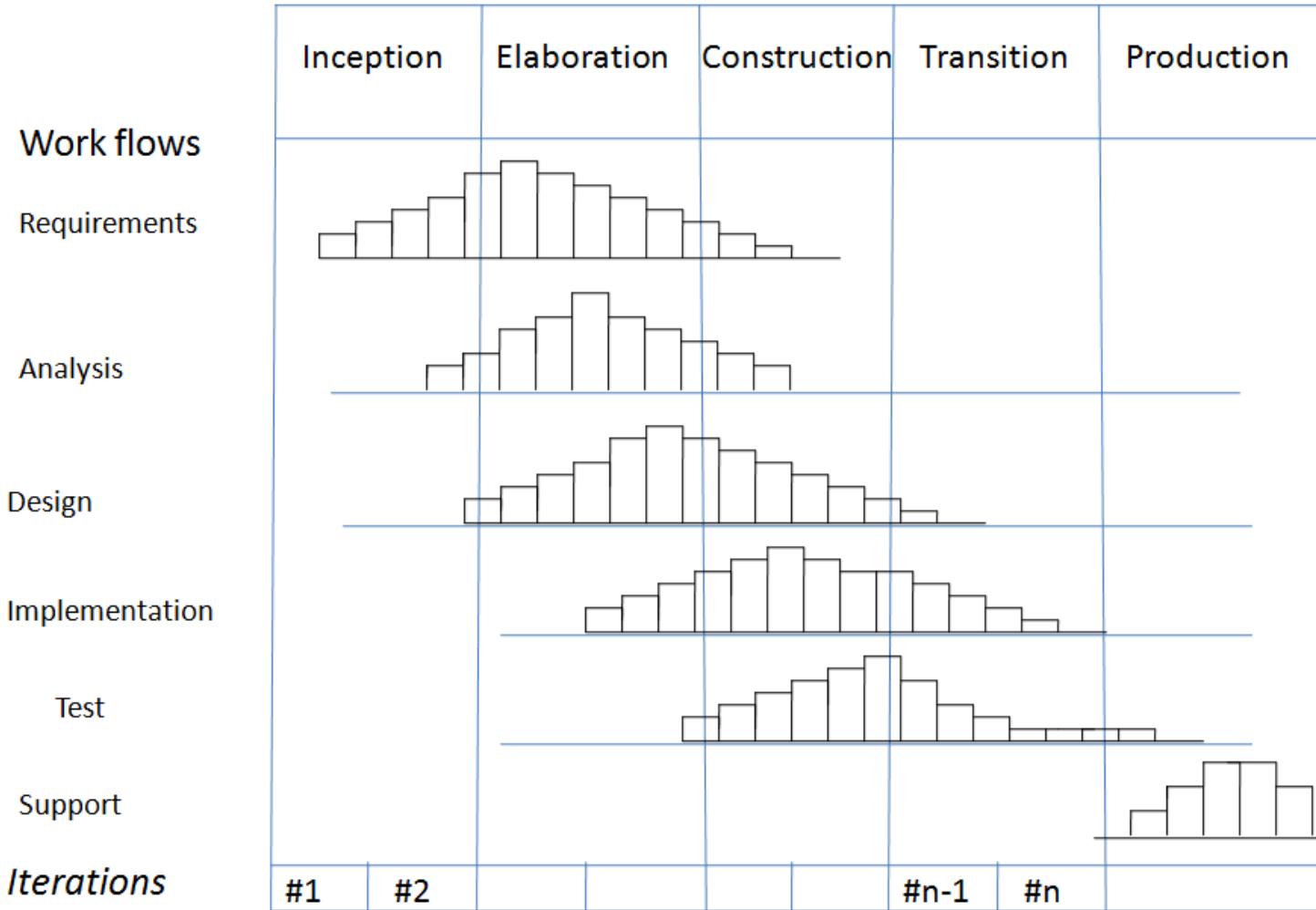


- Encompasses the construction activity of the generic process
- Uses the architectural model from the elaboration phase as input
- Develops or acquires the software components that make each use-case operational
- Analysis and design models from the previous phase are completed to reflect the final 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



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



BITS Pilani
Pilani Campus

Course Name : Software Engineering

T V Rao
Concept of Agility

A View on Software Process

- 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

- 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.

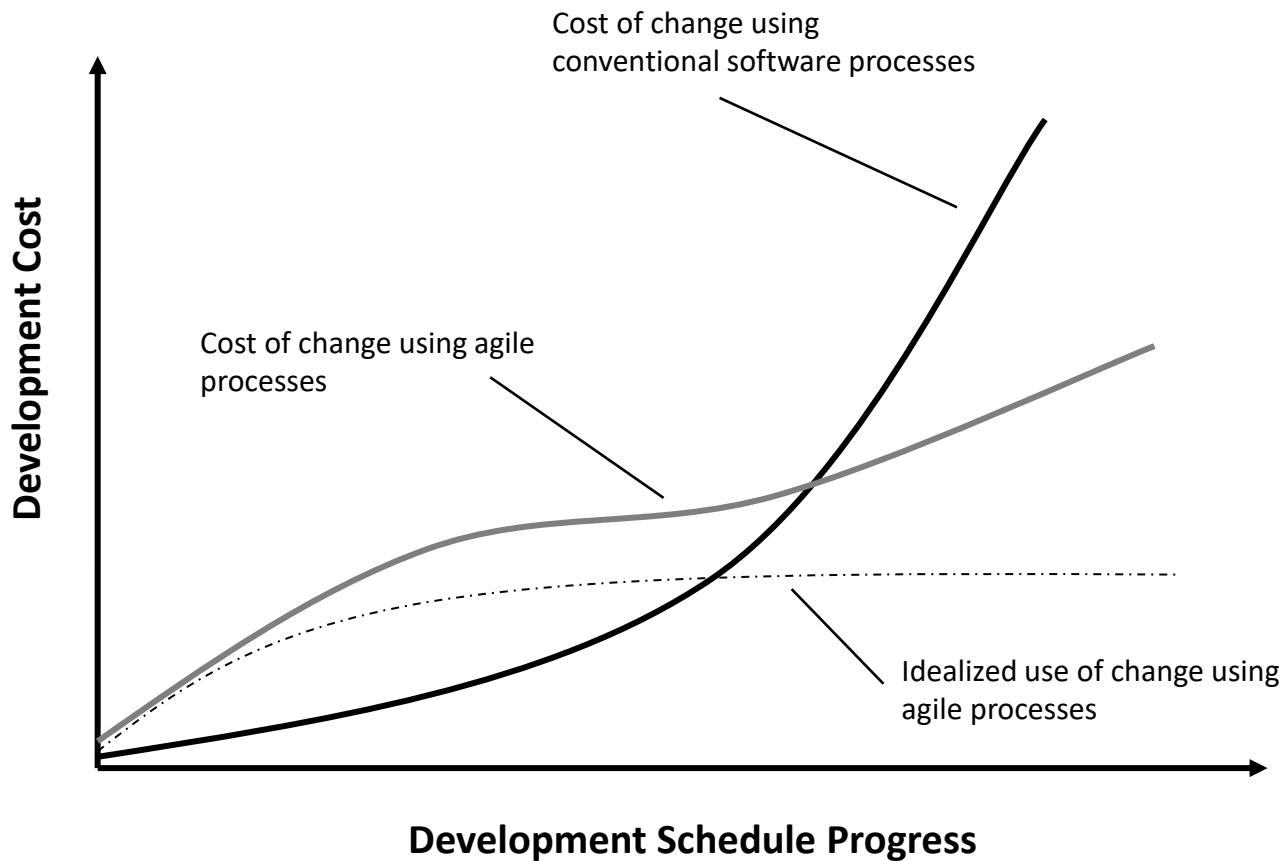
Agility Principles (contd)

- 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

- 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

... 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



Thank You...

Credits

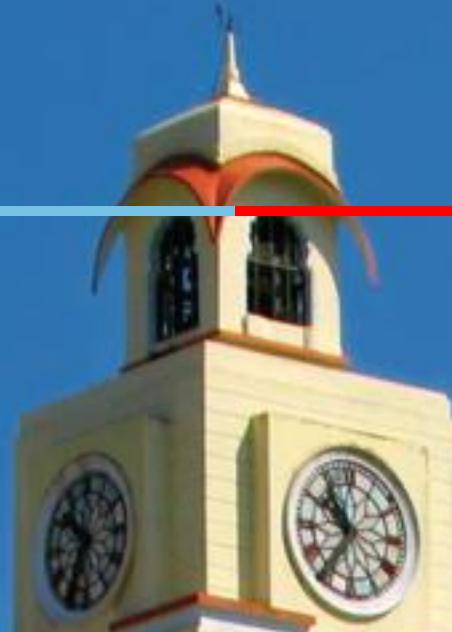
- Software Engineering 7/ed by Roger Pressman – Reference



innovate

achieve

lead



BITS Pilani
Pilani Campus

Course Name : Software Engineering

T V Rao
Agile Process Models

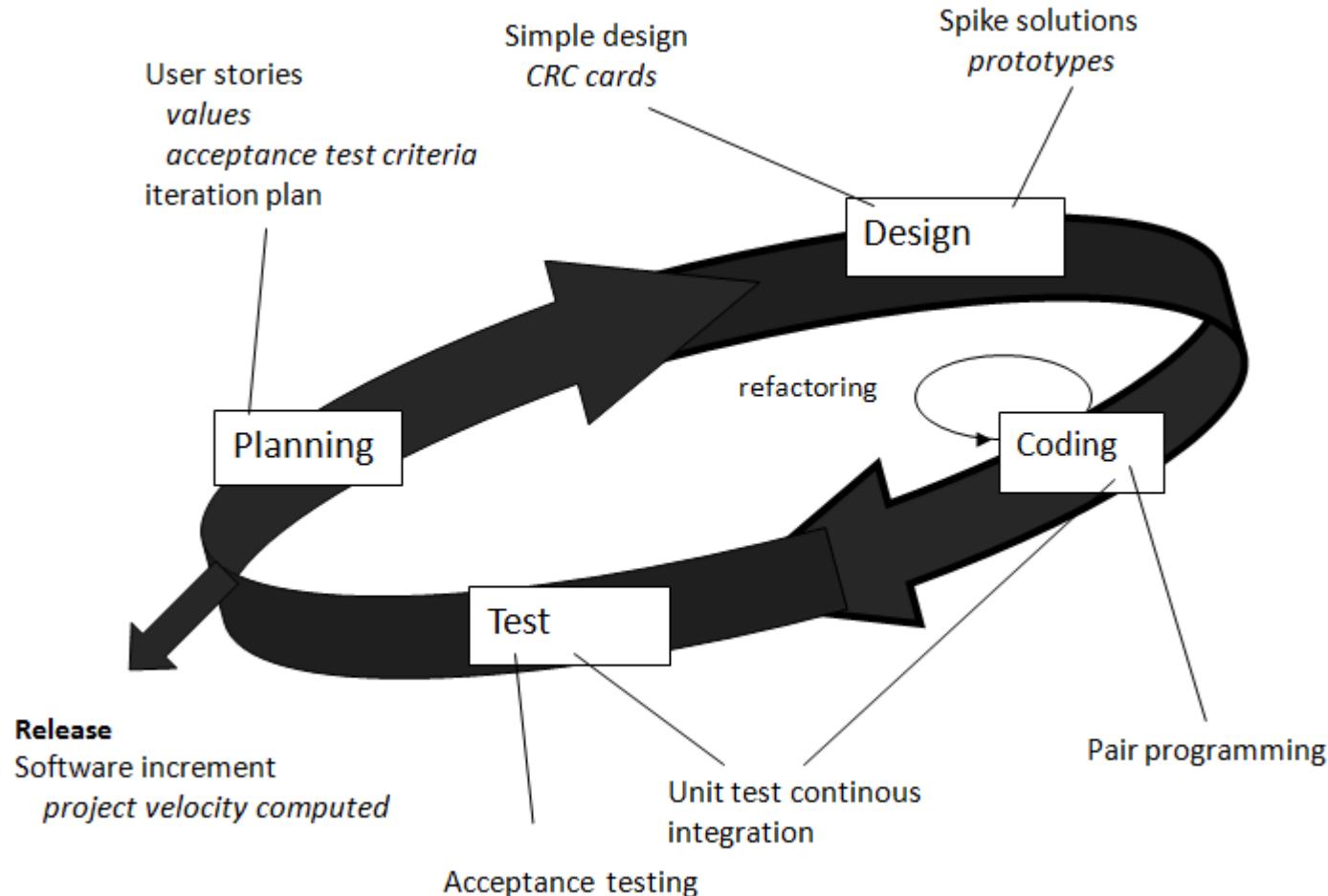
Extreme Programming (XP)

- 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 require 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

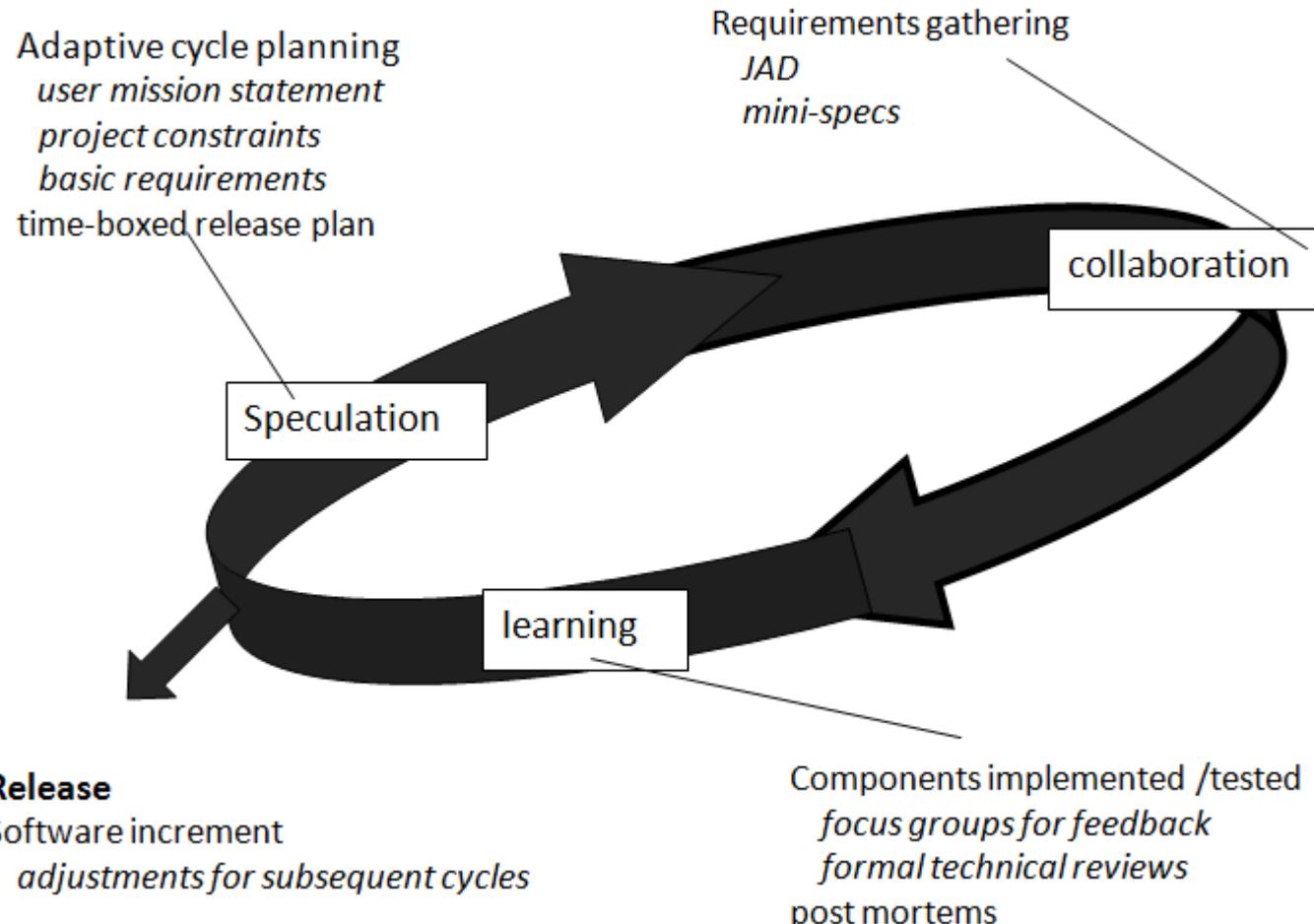
- Without architectural design, structure of the software lacks quality and maintainability

Adaptive Software Development



- 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



Release

Software increment
adjustments for subsequent cycles

Components implemented /tested
focus groups for feedback
formal technical reviews
post mortems

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 operational environment



SCRUM - Introduction

- 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

- 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

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

- 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



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





BITS Pilani
Pilani Campus

Module Name : Software Engineering Practice

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. 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



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





BITS Pilani
Pilani Campus

Module Name : Requirements Engineering

S Subramanian
Work-Integrated Learning Programme



Chapter 5

- 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.**

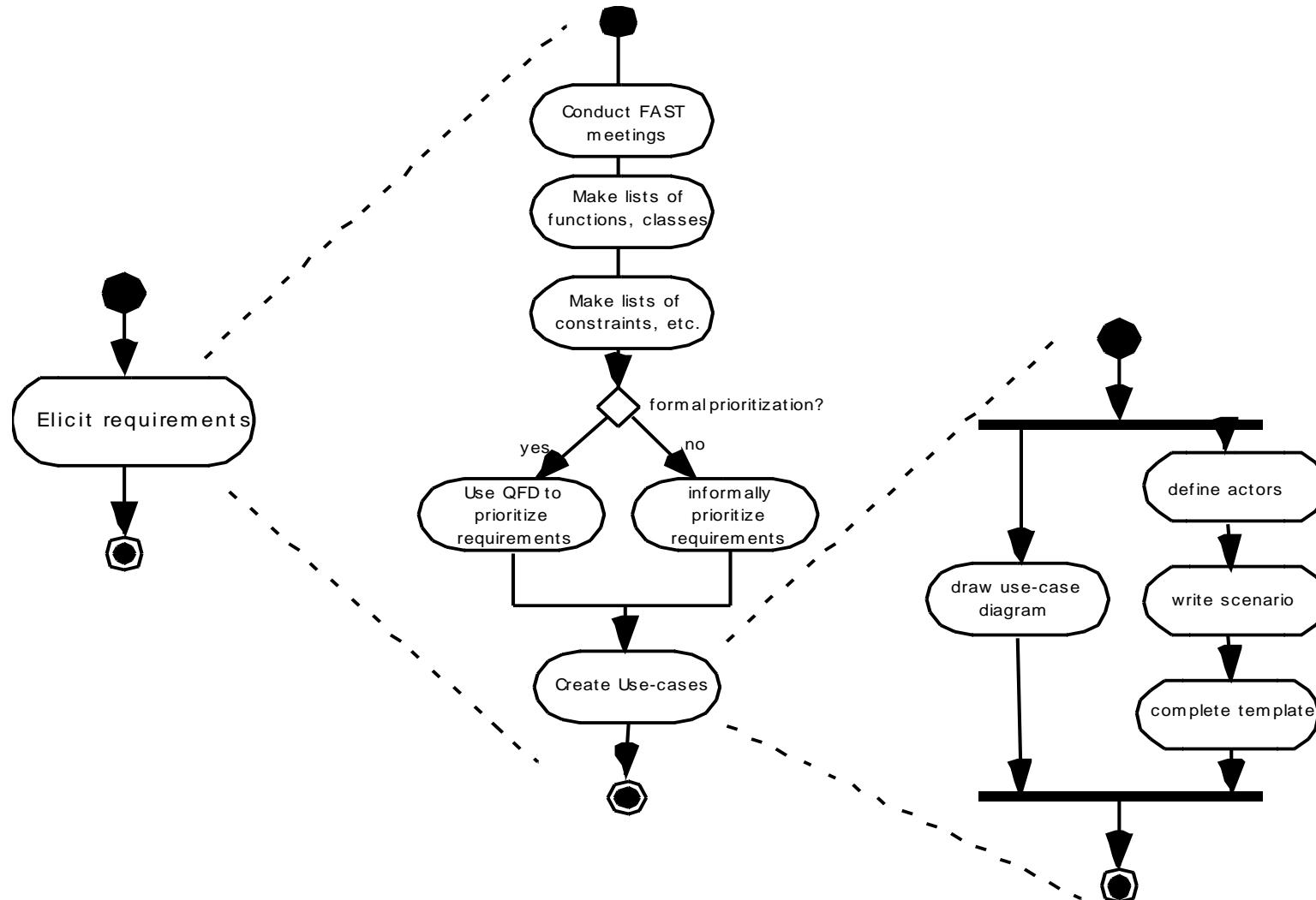
Quality Function Deployment

-
- 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



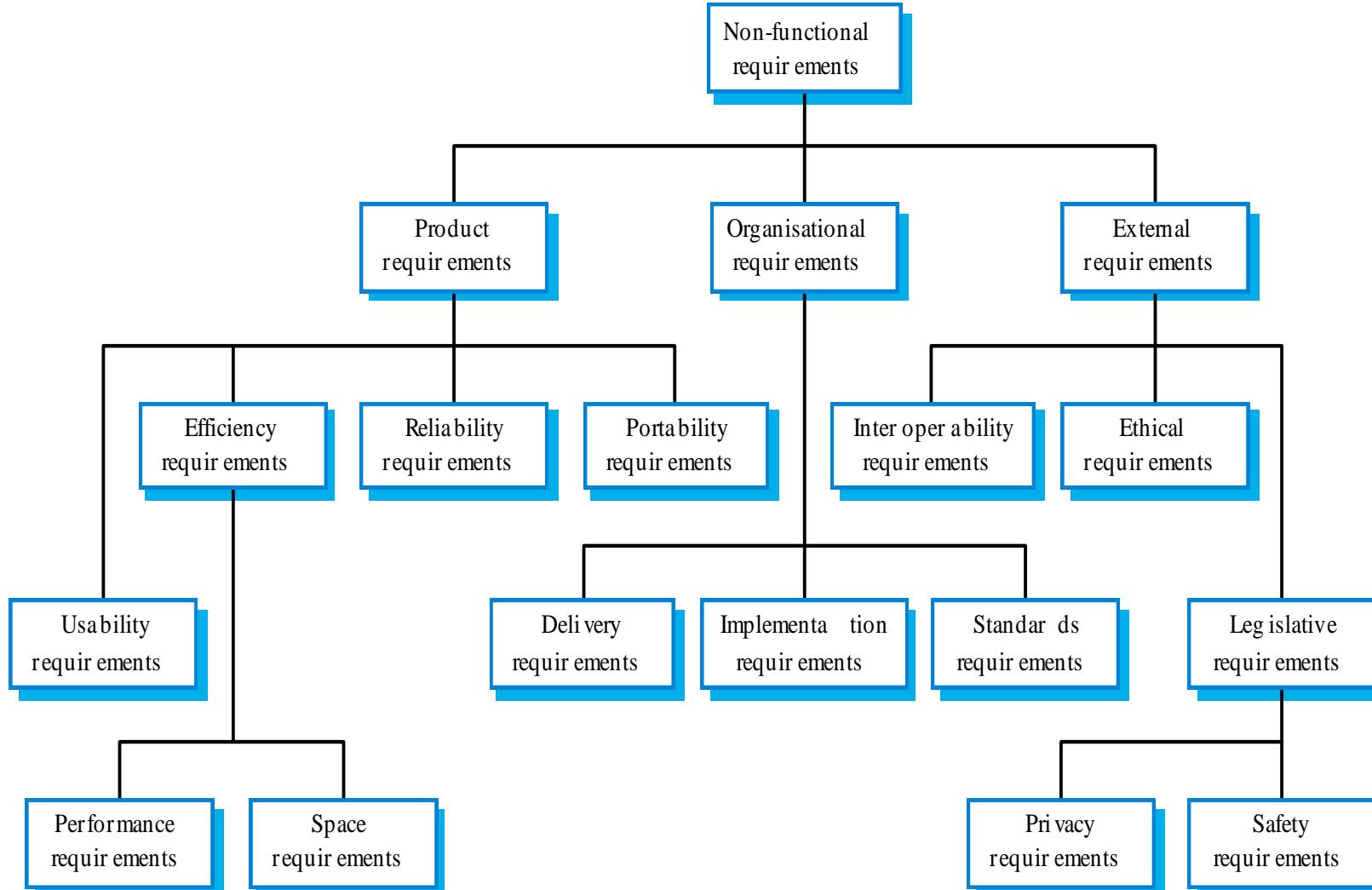
Requirements Phase output

- 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

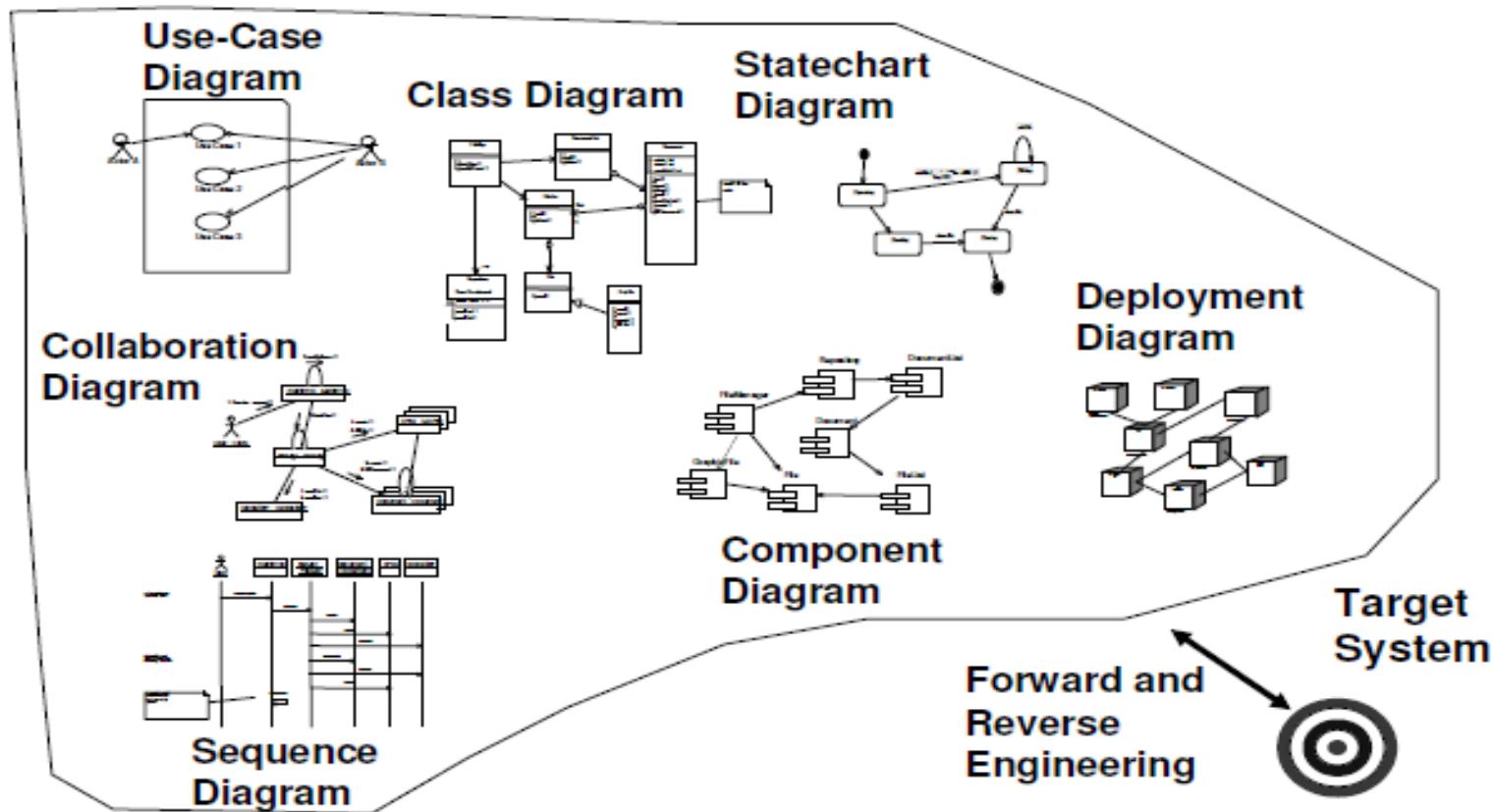
Non-functional Requirement Types



Building the Analysis Model



Visual Modeling Using UML Diagrams



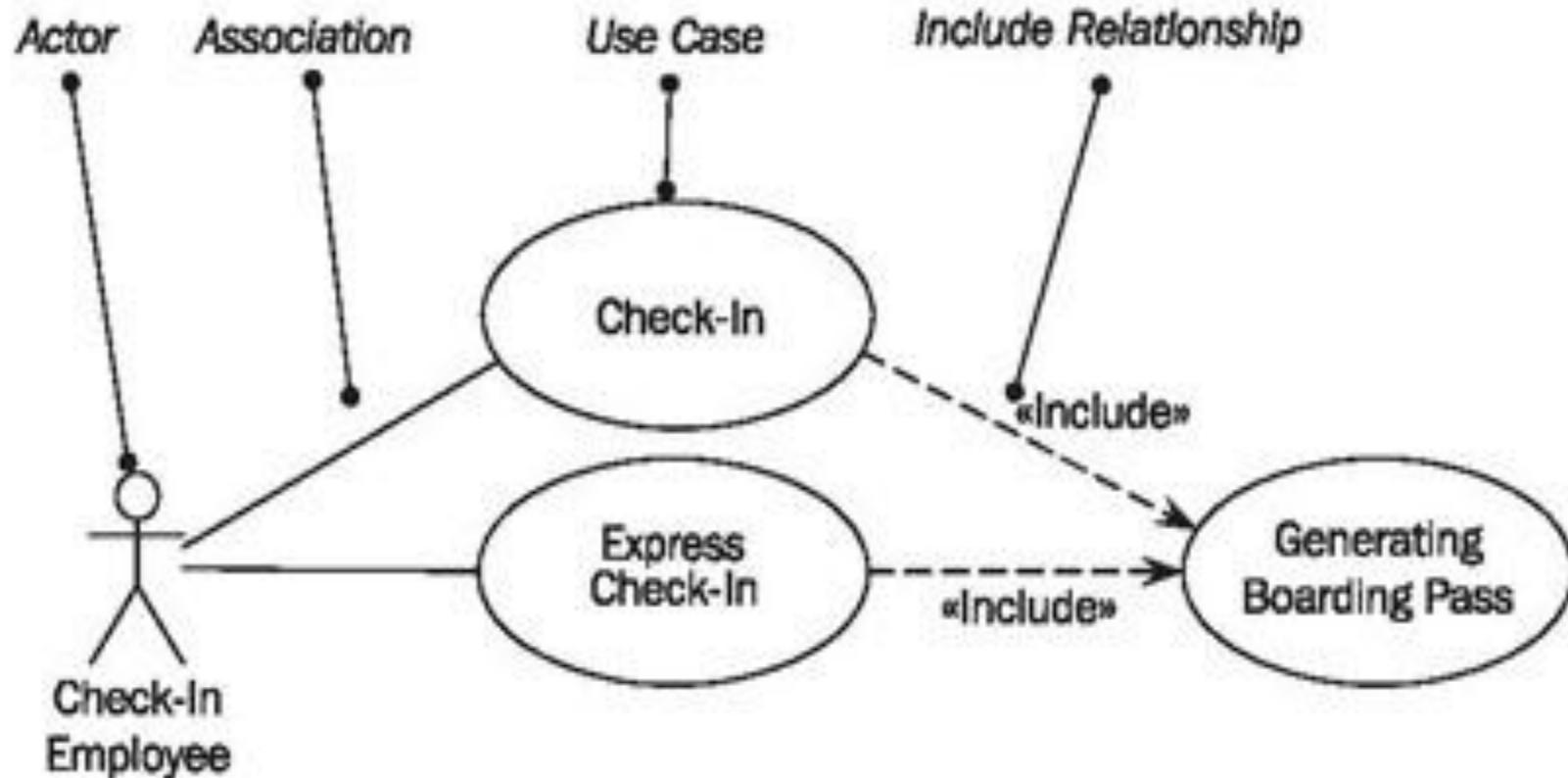
IBM

15

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?

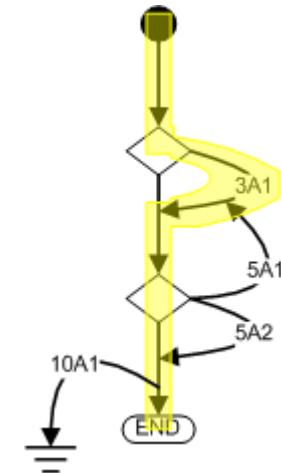
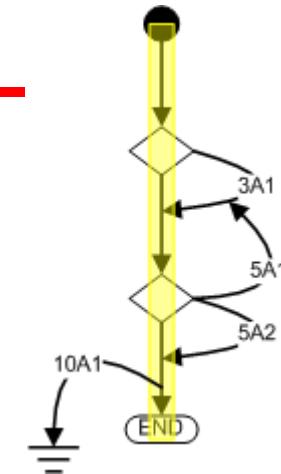
Use case Diagram



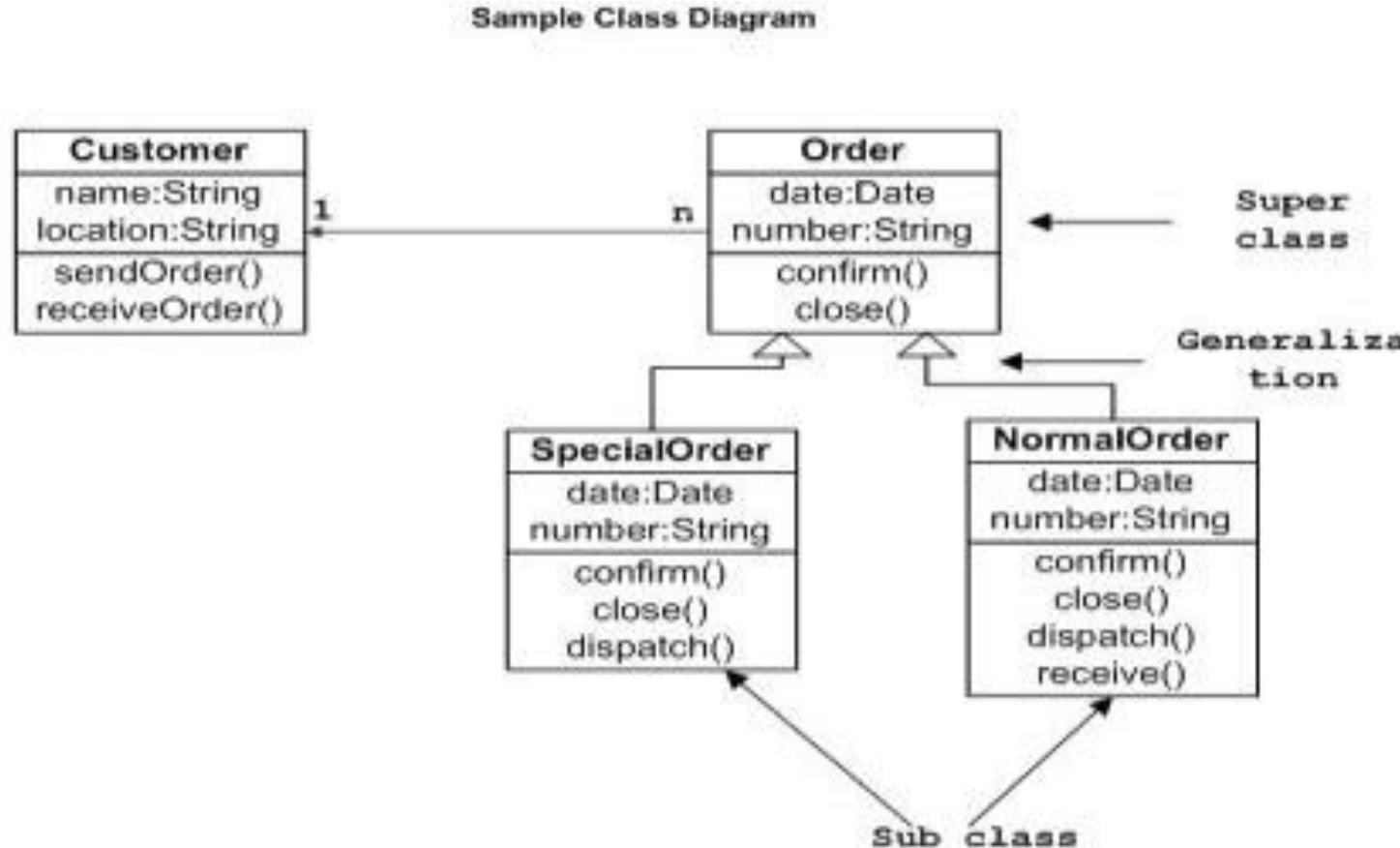
What Are Use Case Scenarios



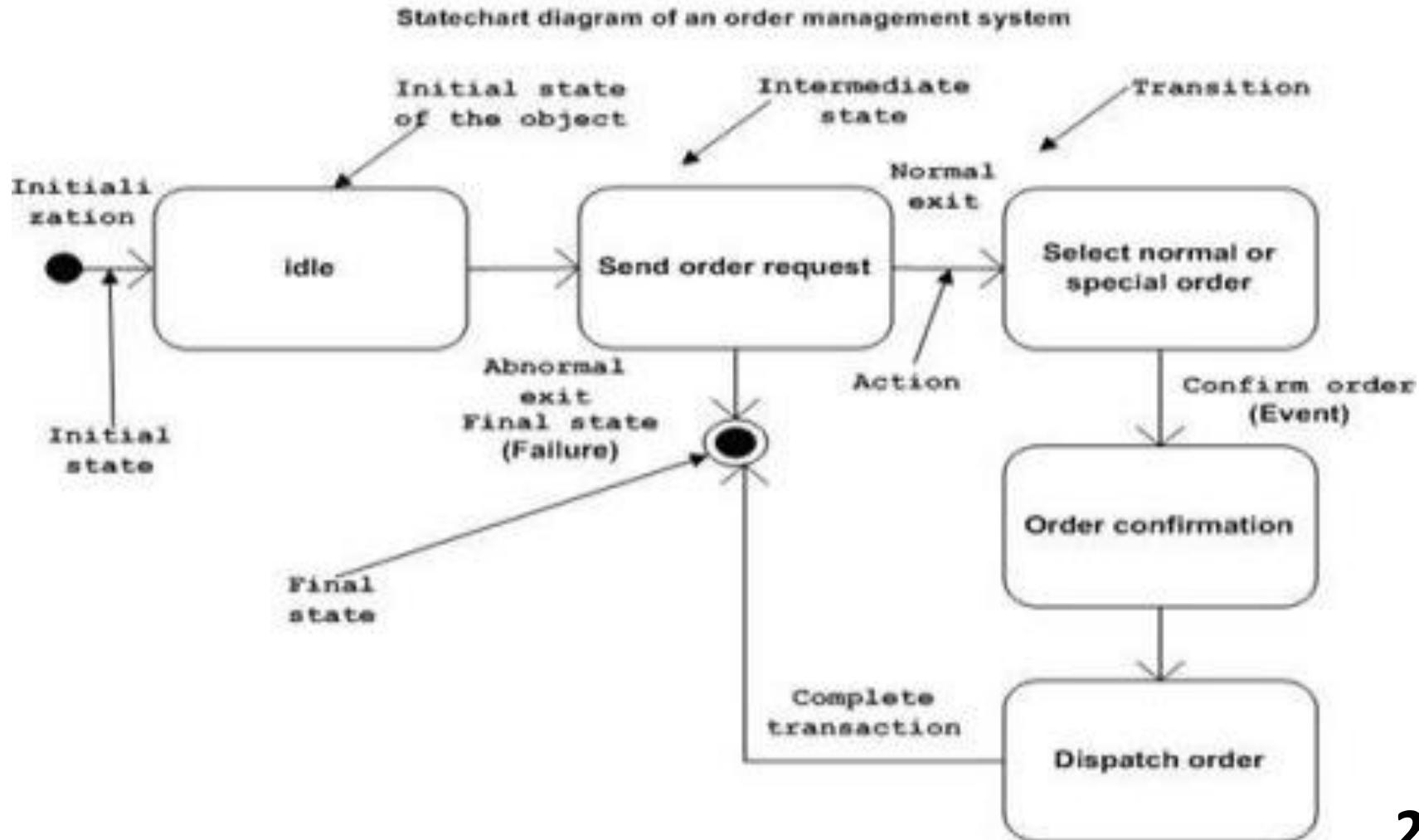
- A use case represents the actions that are required to enable or abandon a goal.
- A use case has multiple “paths” that can be taken by any user at any one time.
- A use **case scenario** is a single path through the use case



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



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





BITS Pilani
Pilani Campus

Module Name : Building Requirements Model - I

S Subramanian
Work-Integrated Learning Programme

Chapter 6

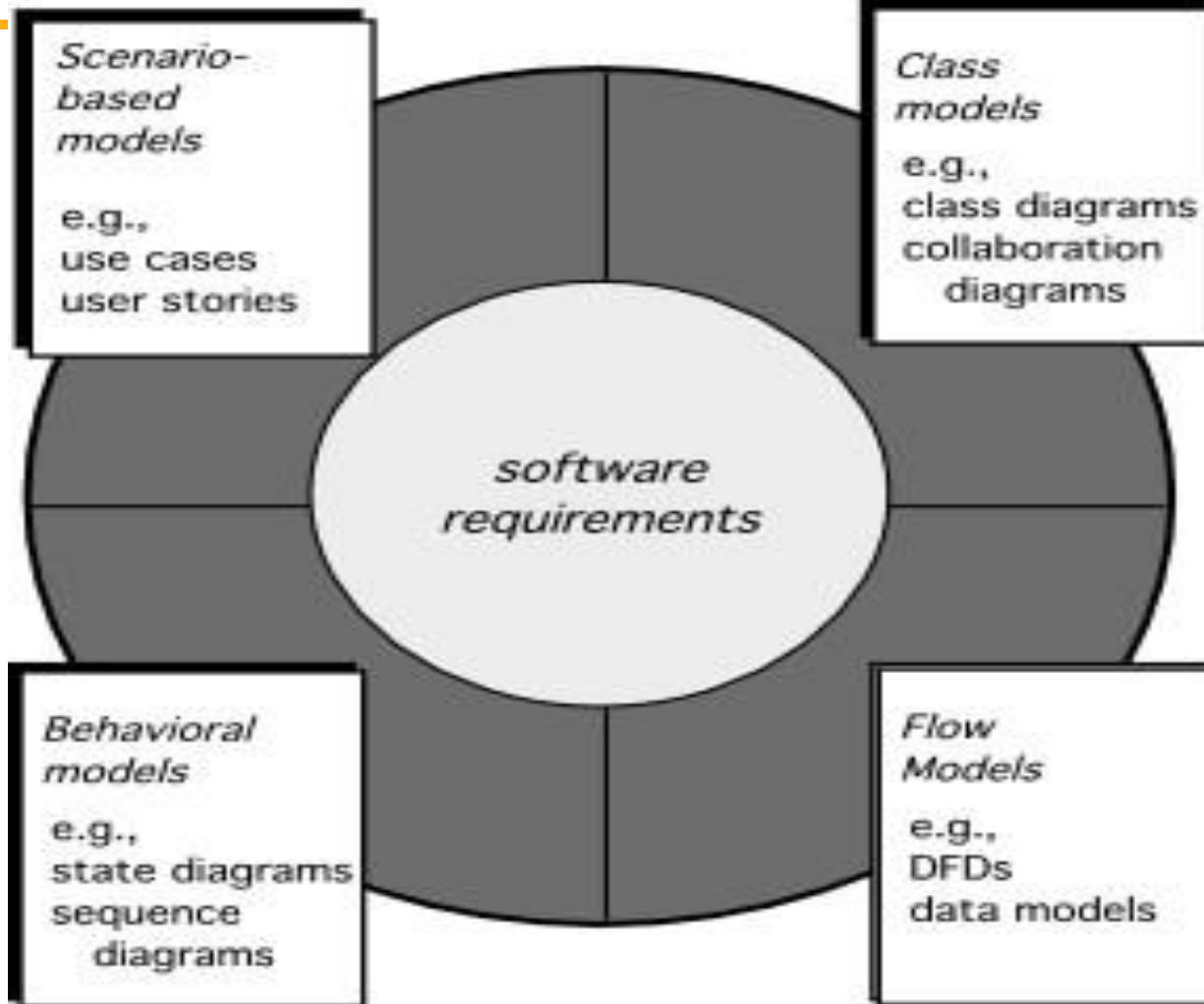
Building Requirements Model - I

Requirement Model

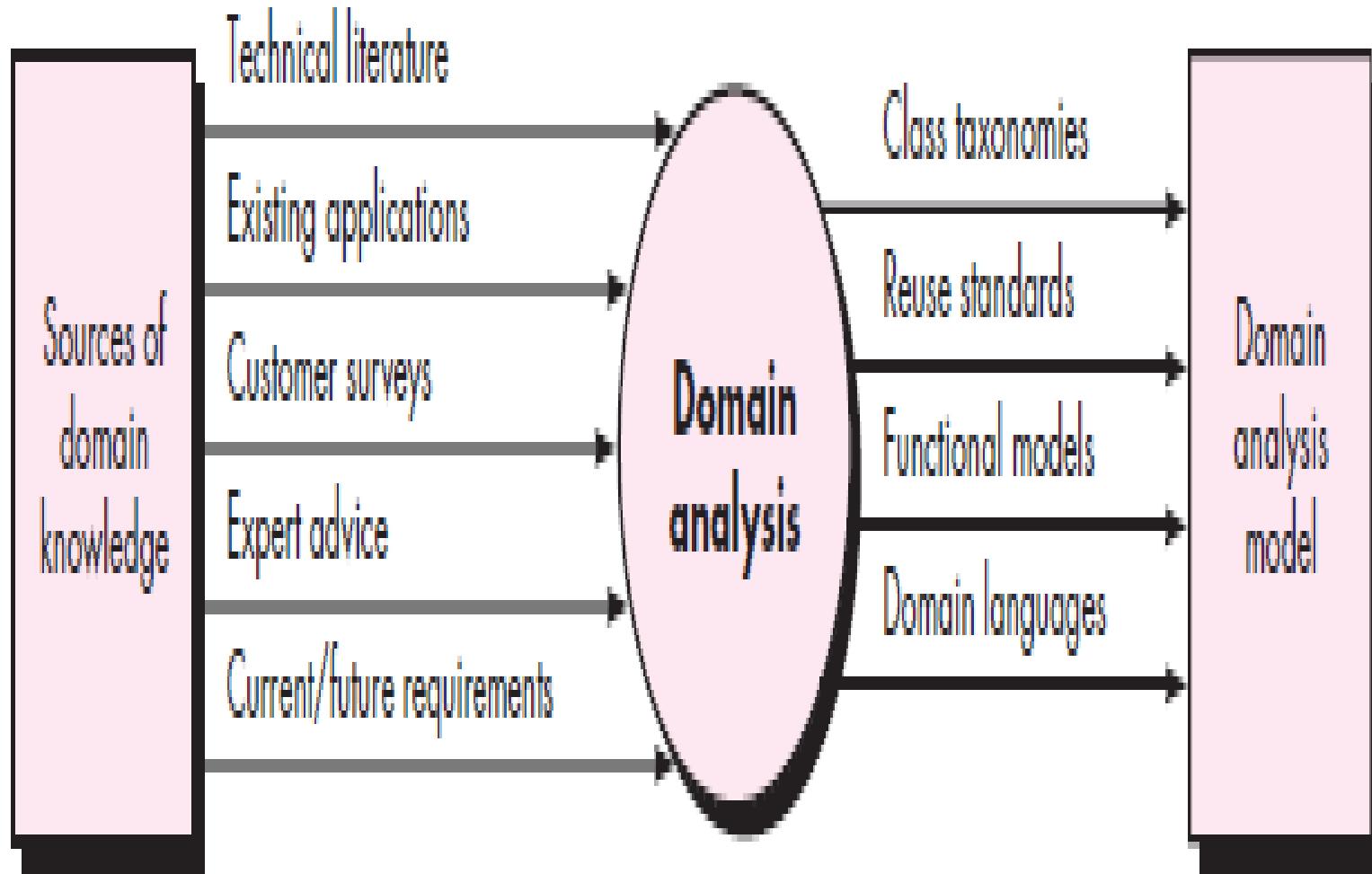
The requirements model must achieve three primary objectives:

1. to describe what the customer requires,
2. to establish a basis for the creation of a software design,
3. and to define a set of requirements that can be validated once the software is built.

Elements of Requirements Analysis



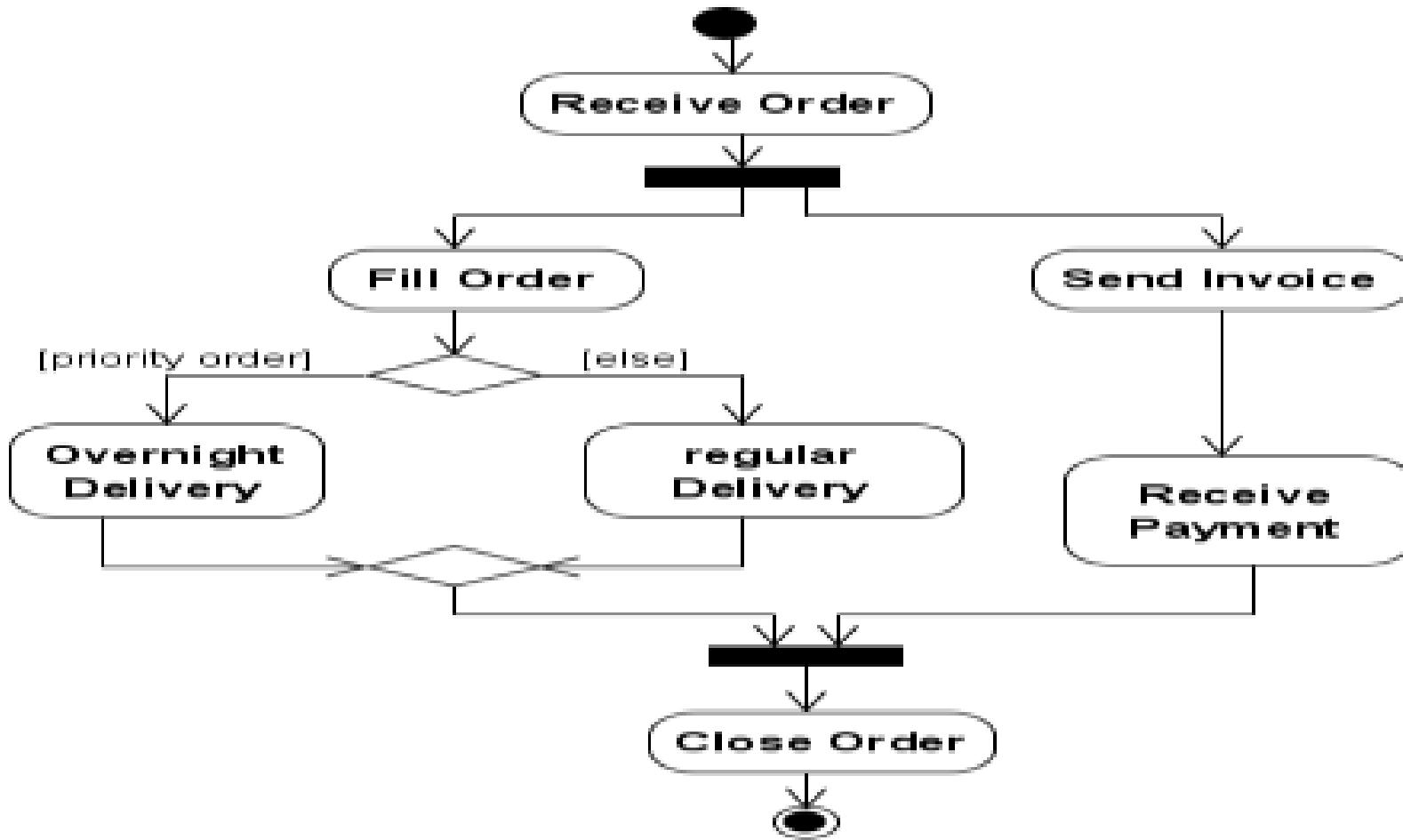
Domain Analysis



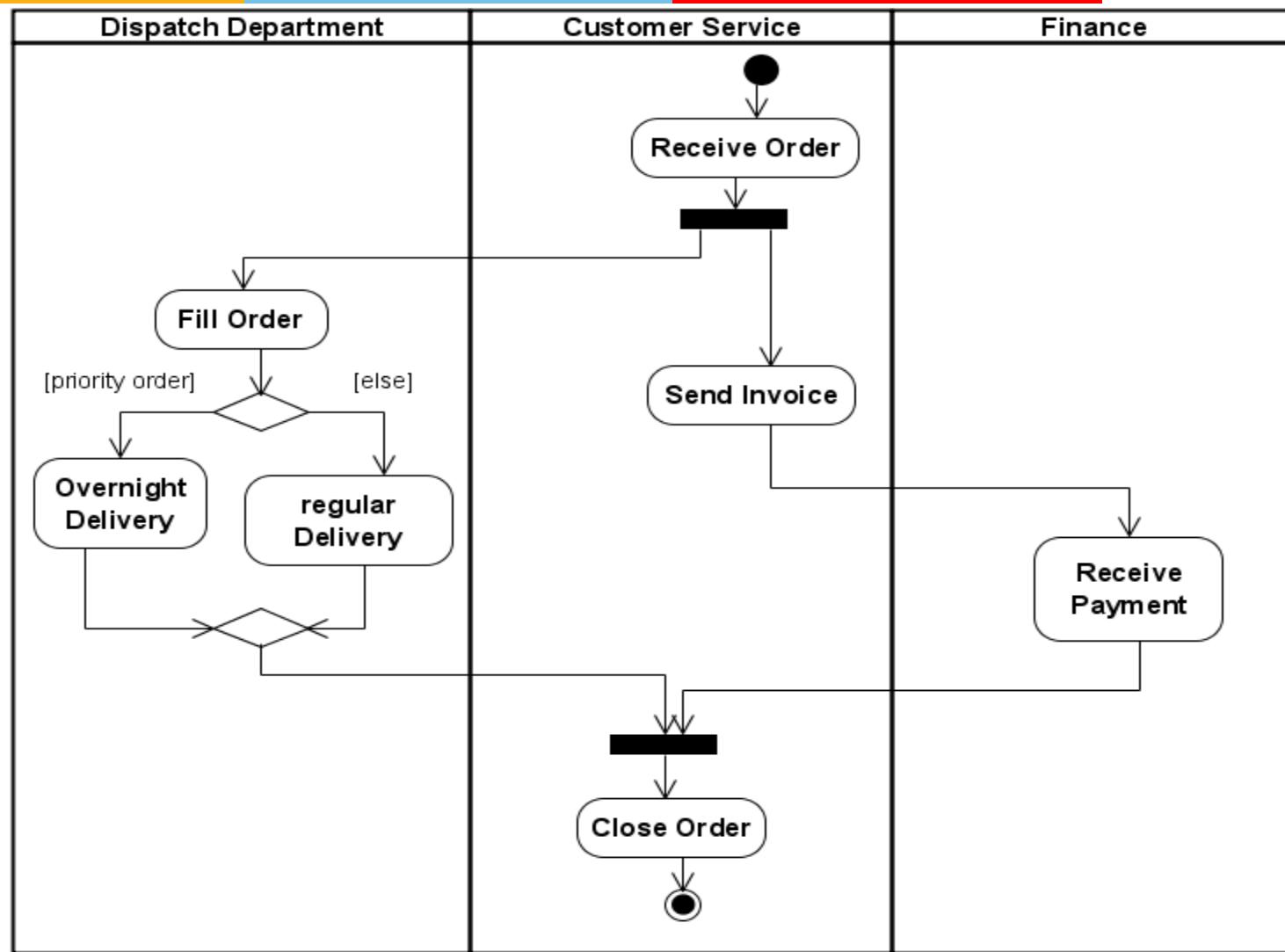
Recording the use cases

- For each actor, what interactions /"results" they require of the system –each is a use case
- What tasks does the actor want the system to perform
- What information must the actor provide to the system
- Are there events that the actor must tell system about
- Does actor need to be informed when something happens

Activity Diagram



Swimlanes



What is a Data Object?

Object —something that is described by a set of attributes (data items) and that will be manipulated within the software (system)

- each instance of an object (e.g., a book) can be identified uniquely (e.g., ISBN #)
- each plays a necessary role in the system i.e., the system could not function without access to instances of the object
- each is described by attributes that are themselves data items

What are some typical data objects?

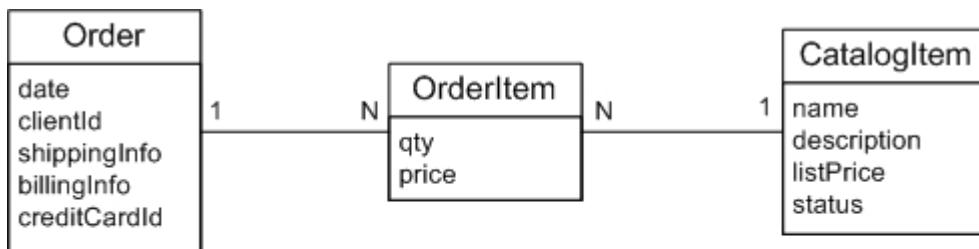
Data Modeling

- Data modeling is a common activity in the software development process of information systems, which typically use database management systems to store information.
- The output of this activity is the data model, which describes the static information structure in terms of data entities and their relationships.
- This structure is often represented graphically in entity-relationship diagrams (ERDs) or UML class diagrams.

Data model evolution



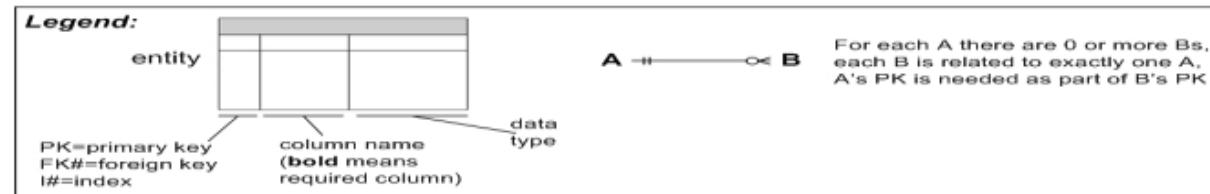
Conceptual Data Model



Logical Data Model

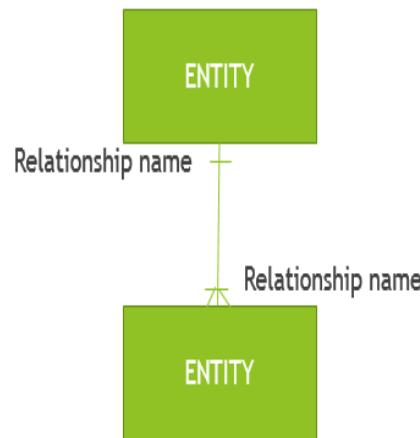


Physical Data Model



E-R Model

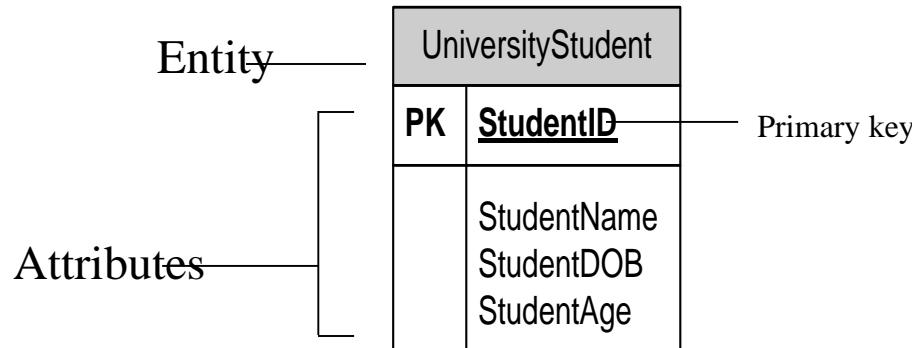
A data model consists of entities related to each other on a diagram:



Data model element	Definition
Entity	A real world thing or an interaction between 2 or more real world things.
Attribute	The atomic pieces of information that we need to know about entities.
Relationship	How entities depend on each other in terms of why the entities depend on each other (the relationship) and what that relationship is (the cardinality of the relationship).

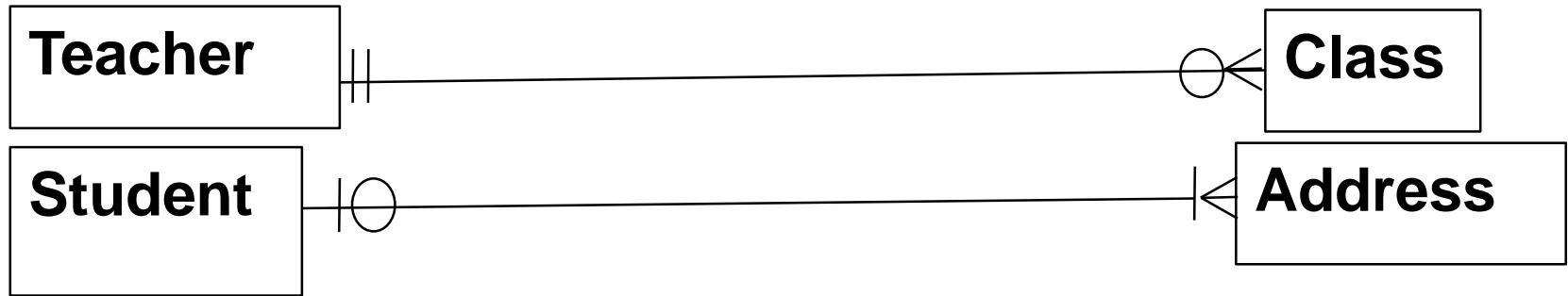
Entity-Relationship Model

- Entity Relationship Diagrams (ERD) as this is the most widely used
- ERDs have an advantage in that they are capable of being normalized



- Represent entities as rectangles
- List attributes within the rectangle

Crow's Foot Style ERD



Teacher teaches 0 to many classes

Classes have 1 and only 1 teacher

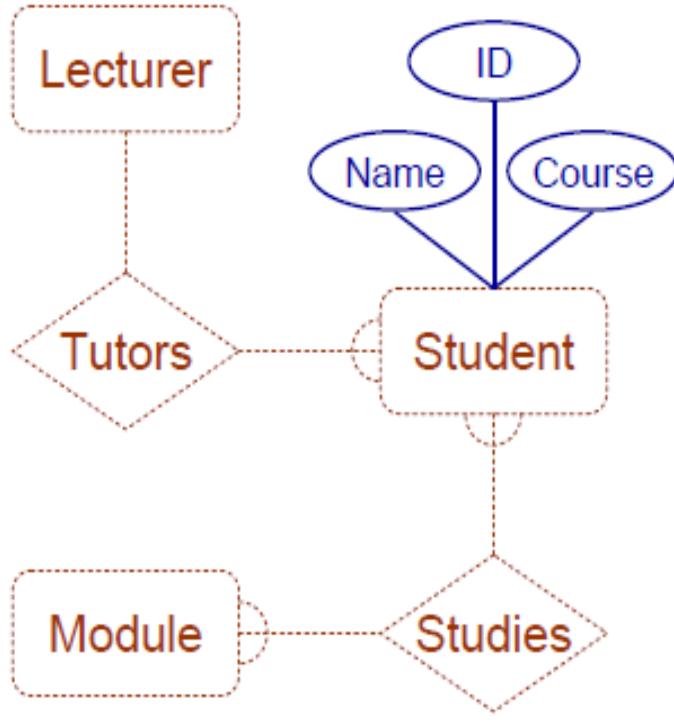
Students have 1 to many addresses

An address is for zero to one student (addresses may not be associated with multiple students)

First “thing” denotes optional or mandatory.

Second “thing” denotes cardinality (one or many)

Key Constraints



One to one (1:1)

- Each lecturer has a unique office

• One to many (1:M)

A lecturer may tutor many students, but each student has just one tutor

• Many to many (M:M)

Each student takes several modules, and each module is taken by several students

Class-Based Modeling

- Identify analysis classes by examining the problem statement
- Use a “grammatical parse” to isolate potential classes
- Identify the attributes of each class
- Identify operations that manipulate the attributes

Typical Classes (a reminder)

- *External entities* - printer, user, sensor
- *Things* - reports, displays, signals
- *Occurrences or events* (e.g., interrupt, alarm)
- *Roles* (e.g., manager, engineer, salesperson)
- *Organizational units* (e.g., division, team)
- *Places* (e.g., manufacturing floor or loading dock)
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers)

But, how do we select classes?

From NPs to classes or attributes



Consider the following problem description, analyzed for Subjects, Verbs, Objects:

The ATM verifies whether the customer's card number and PIN are correct.

S C V

R O

O A

O A

If it is, then the customer can check the account balance, deposit cash, and withdraw cash.

S R

V

O A

V

O A

V

O A

Checking the balance simply displays the account balance.

S M

O A

V

O A

Depositing asks the customer to enter the amount, then updates the account balance.

S M

V

O R

V

O A

V

O A

Withdraw cash asks the customer for the amount to withdraw; if the account has enough cash,

S M

O A

V

O R

O A

V

S C

V

O A

the account balance is updated. The ATM prints the customer's account balance on a receipt.

O A

V

S C

V

O A

O

Analyze each subject and object as follows:

- Does it represent a person performing an action? Then it's an actor, 'R'.
- Is it also a verb (such as 'deposit')? Then it may be a method, 'M'.
- Is it a simple value, such as 'color' (string) or 'money' (number)?
Then it is probably an attribute, 'A'.
- Which NPs are unmarked? Make it 'C' for class.

Verbs can also be classes, for example:

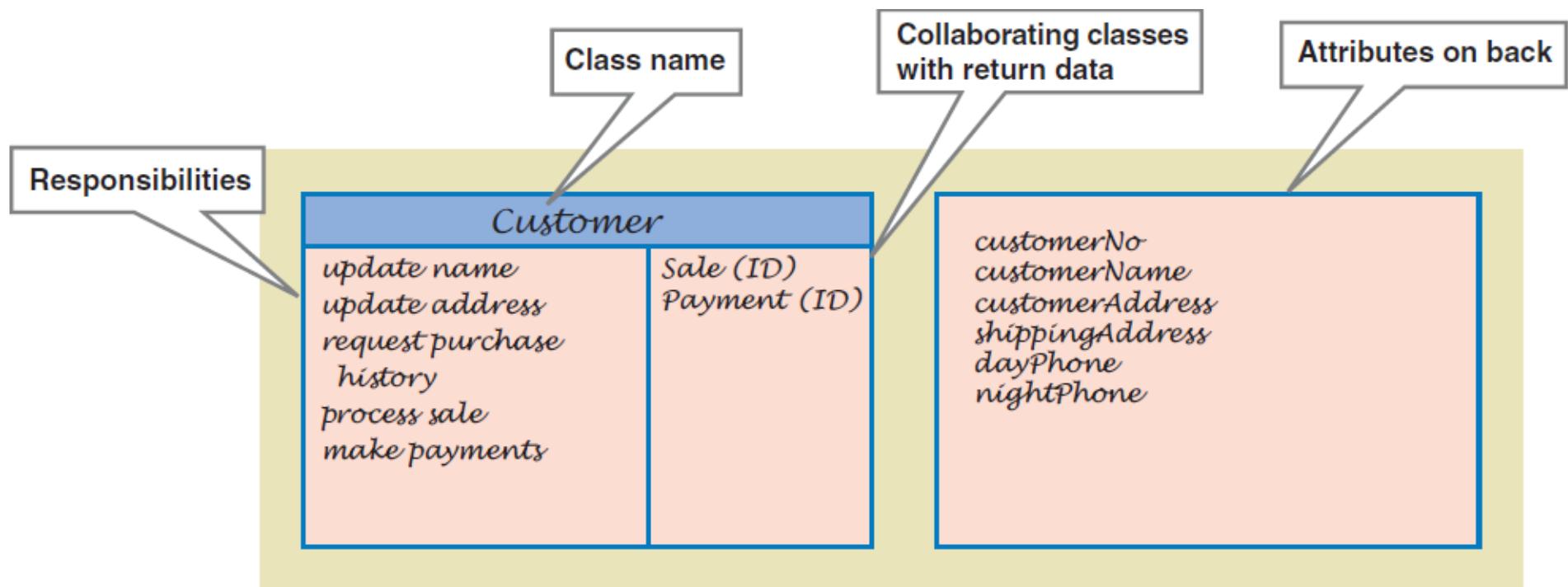
- Deposit is a class if it retains state information

• Selecting Classes—Criteria



- retained information
- needed services
- multiple attributes
- common attributes
- common operations
- essential requirements

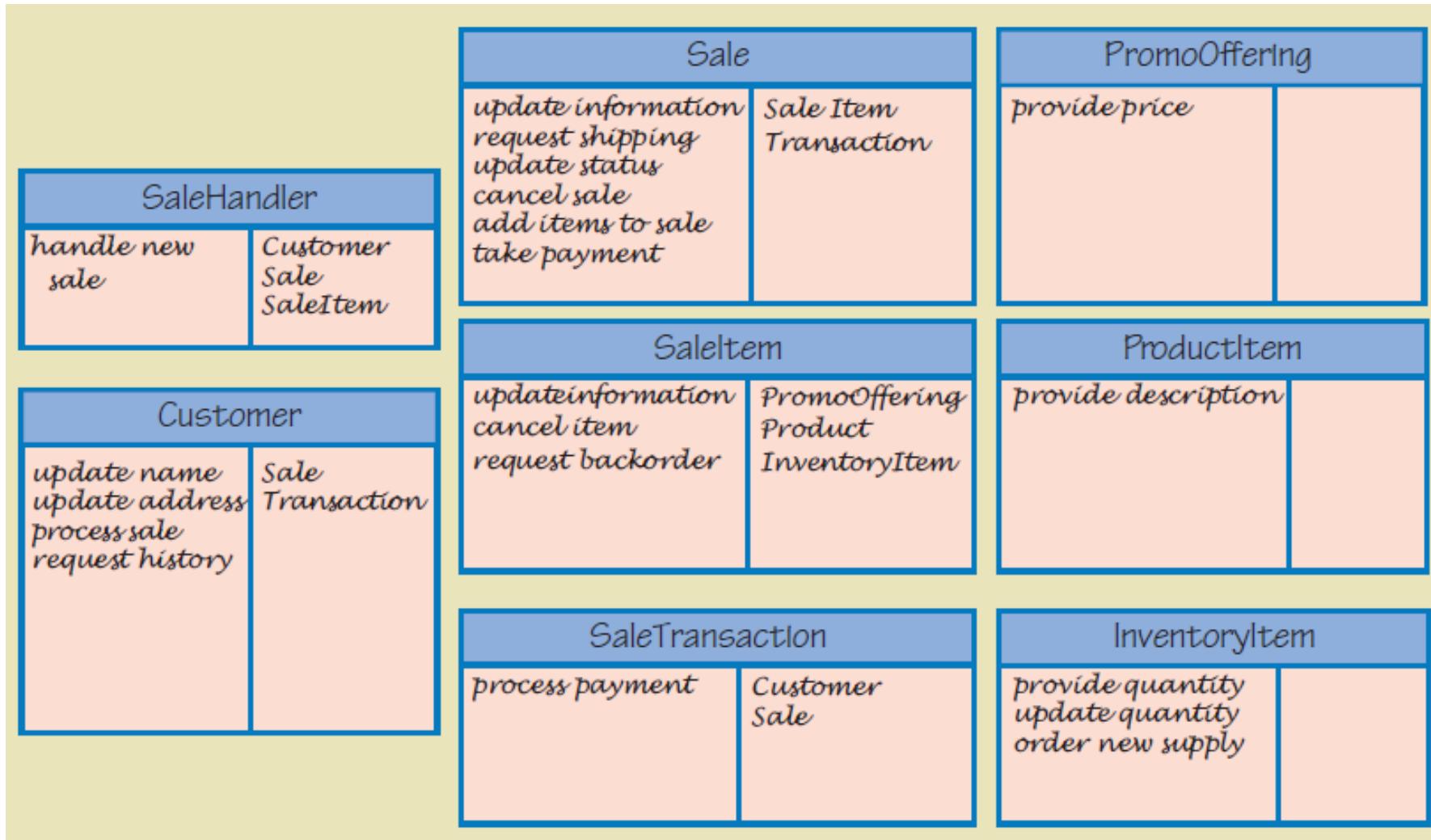
Example of CRC Card



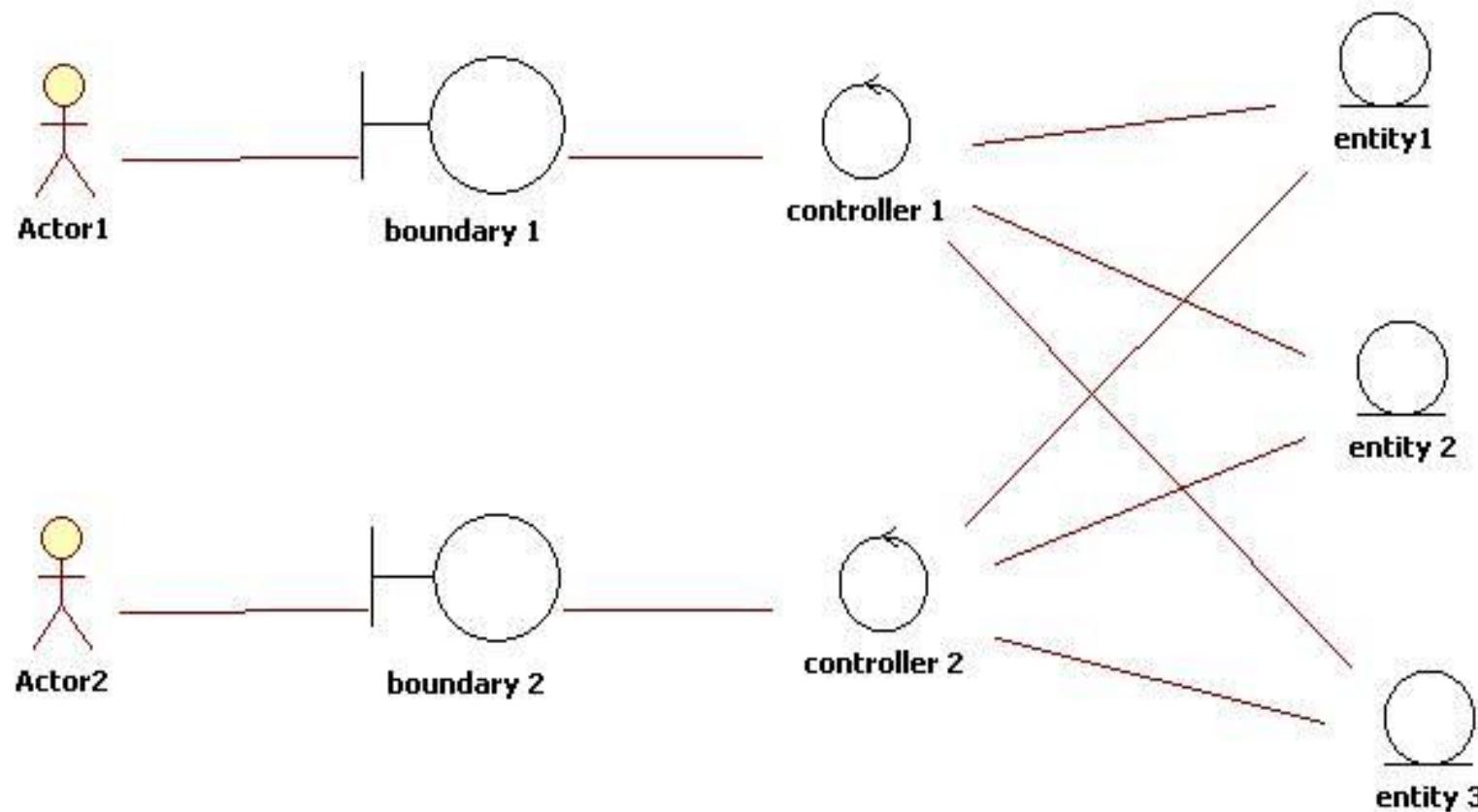
CRC Cards Results



Several Use Cases



Type of Classes



Fundamental Design Principles

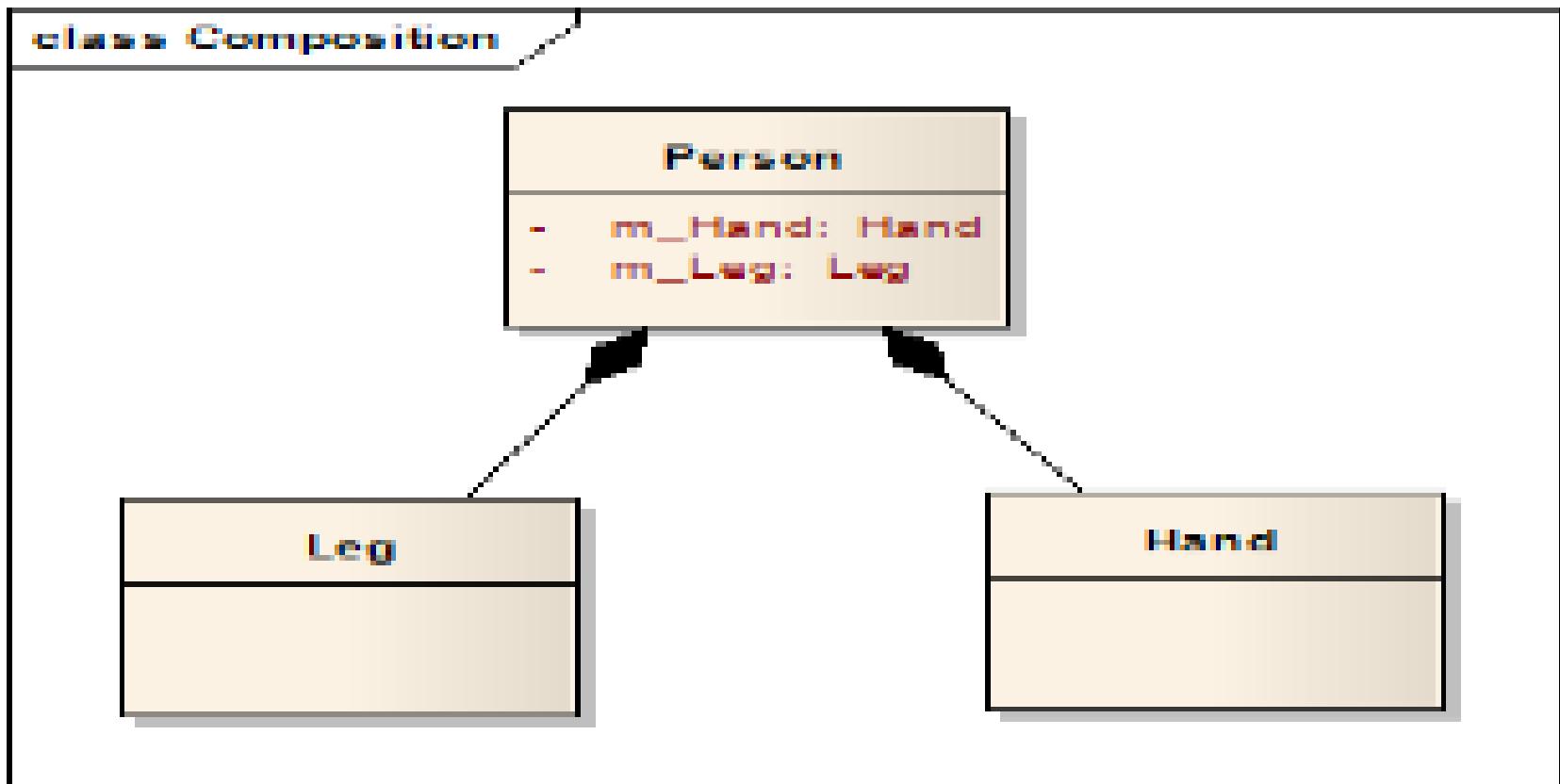


- Object Responsibility
 - A design principle that states objects are responsible for carrying out system processing
 - A fundamental assumption of OO design and programming
 - Responsibilities include “knowing” and “doing”
 - Objects know about other objects (associations) and they know about their attribute values. Objects know how to carry out methods, do what they are asked to do.
 - Information about one thing should be localized with a single class, not distributed across multiple classes.

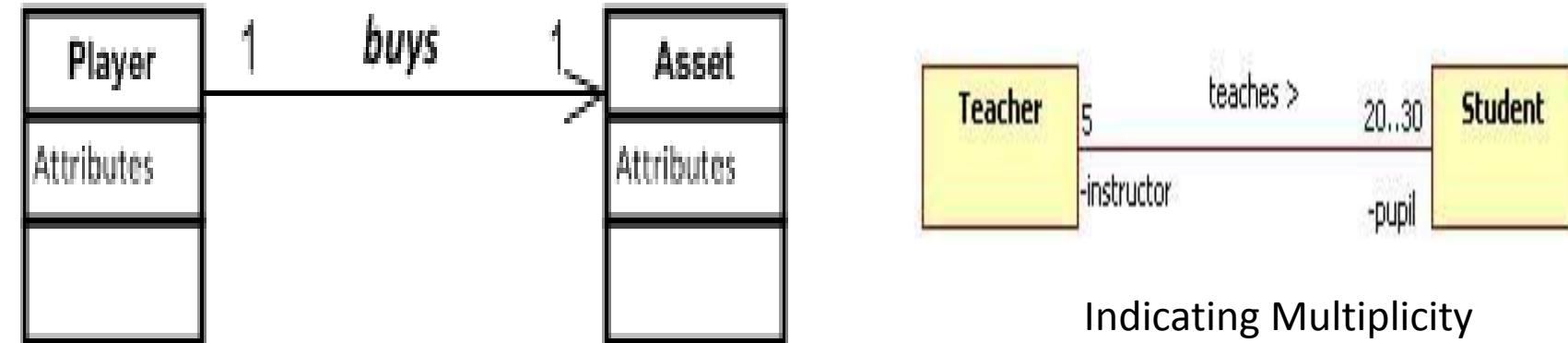
Collaborations

- Classes fulfill their responsibilities in one of two ways:
 - A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or
 - a class can collaborate with other classes.
- Collaborations identify relationships between classes
- Collaborations are identified by determining whether a class can fulfill each responsibility itself
- three different generic relationships between classes
 - the *is-part-of* relationship
 - the *has-knowledge-of* relationship
 - the *depends-upon* relationship

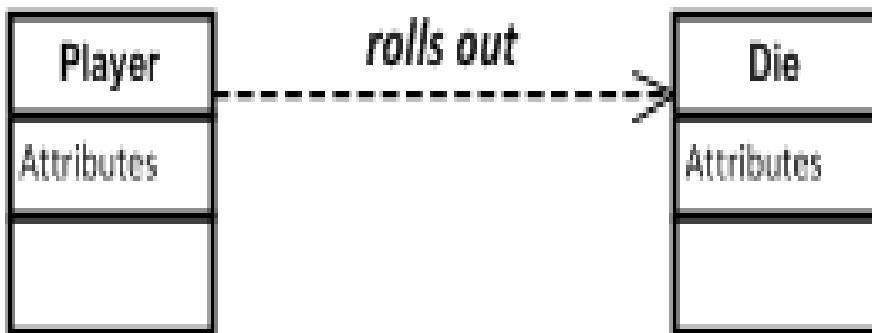
Composite Aggregate Class



Associations and Dependencies



Associations



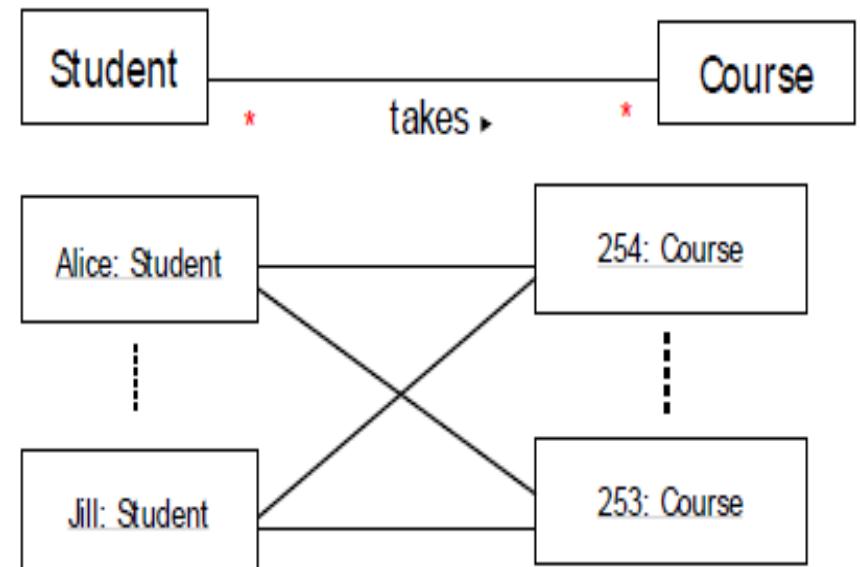
Dependencies

Multiplicity

Multiplicity can be expressed as,

- Exactly one - 1
- Zero or one - 0..1
- Many - 0..* or *
- One or more - 1..*
- Exact Number - e.g. 3..4 or 6
- Or a complex relationship – e.g. 0..1, 3..4, 6..* would mean any number of objects other than 2 or 5

- A **Student** can take many **Courses** and many **Students** can be enrolled in one **Course**.

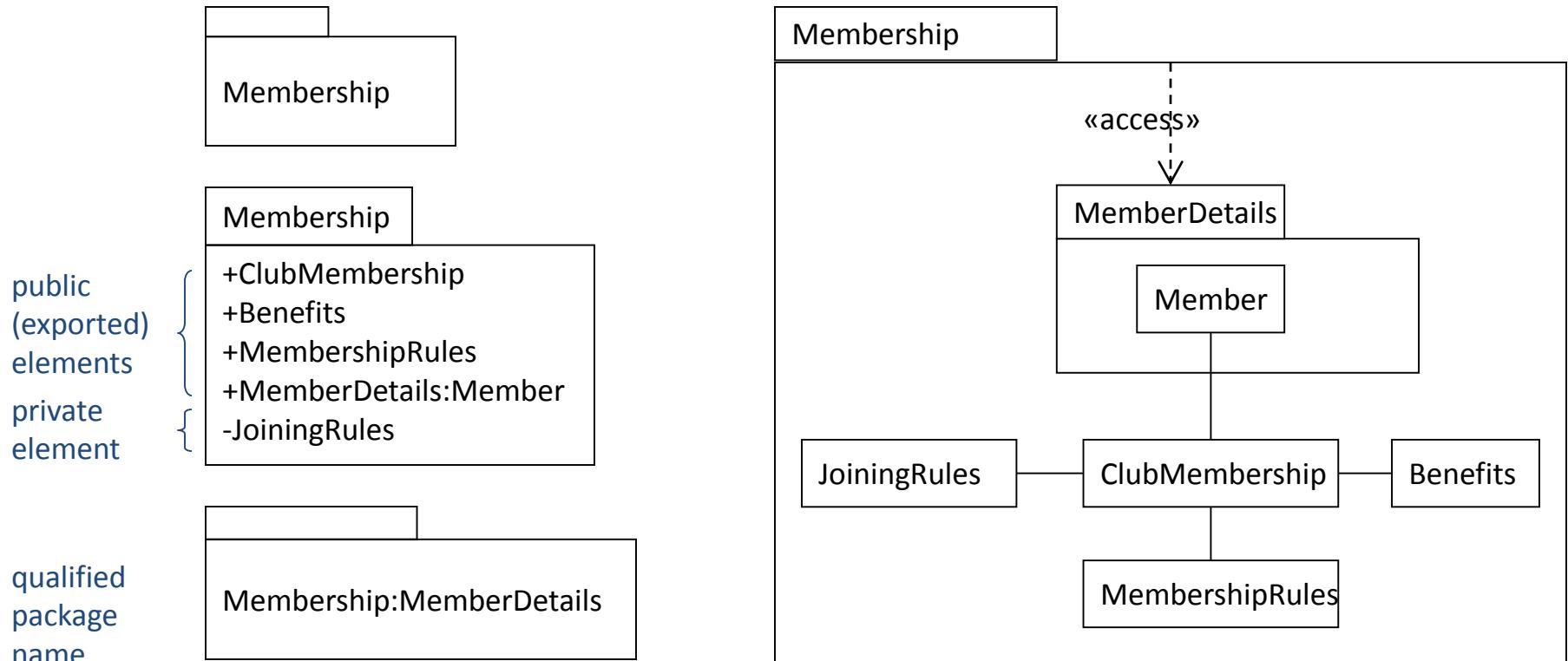


Analysis Packages

Package is the UML mechanism for grouping things. It can be used to:

- group semantically related elements;
 - define a “semantic boundary” in the model;
 - provide units for parallel working and configuration management;
 - package is used to provide an encapsulated namespace within which all names must be unique.
- Analysis packages contain:
 - use cases
 - analysis classes
 - use case realizations

Package syntax



standard UML 2 package stereotypes

«framework»	A package that contains model elements that specify a reusable architecture
«modelLibrary»	A package that contains elements that are intended to be reused by other packages

Credits

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

Thank You



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





BITS Pilani
Pilani Campus

Module Name : Building Requirements Model - II

S Subramanian
Work-Integrated Learning Programme

Chapter 7

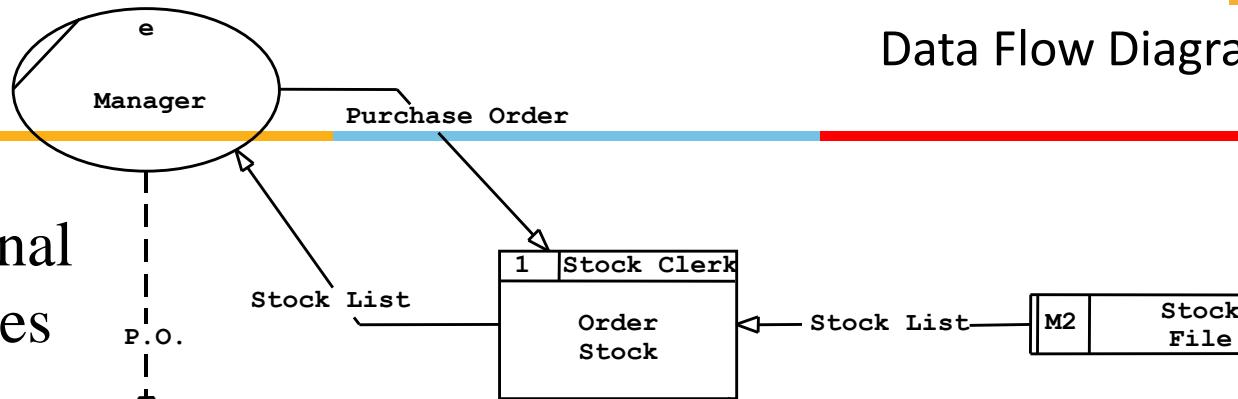
- Requirements Modeling: Flow, Behavior, Patterns.

Flow-Oriented Modeling

- Represents how data objects are transformed as they move through the system
- **Data Flow diagram (DFD)** is the diagrammatic form that is used
- Considered by many to be an “old school” approach, but continues to provide a view of the system that is unique—it should be used to supplement other analysis model elements

Data Flow Diagrams aid communication

External Entities



Processes

Purchase Order

Data Stores

Receive Stock

M1 Purchase Order Cabinet

Matched Orders

M2 Stock File

Delivered Goods

Sold Goods

Bought Goods

Data Flow Diagrams

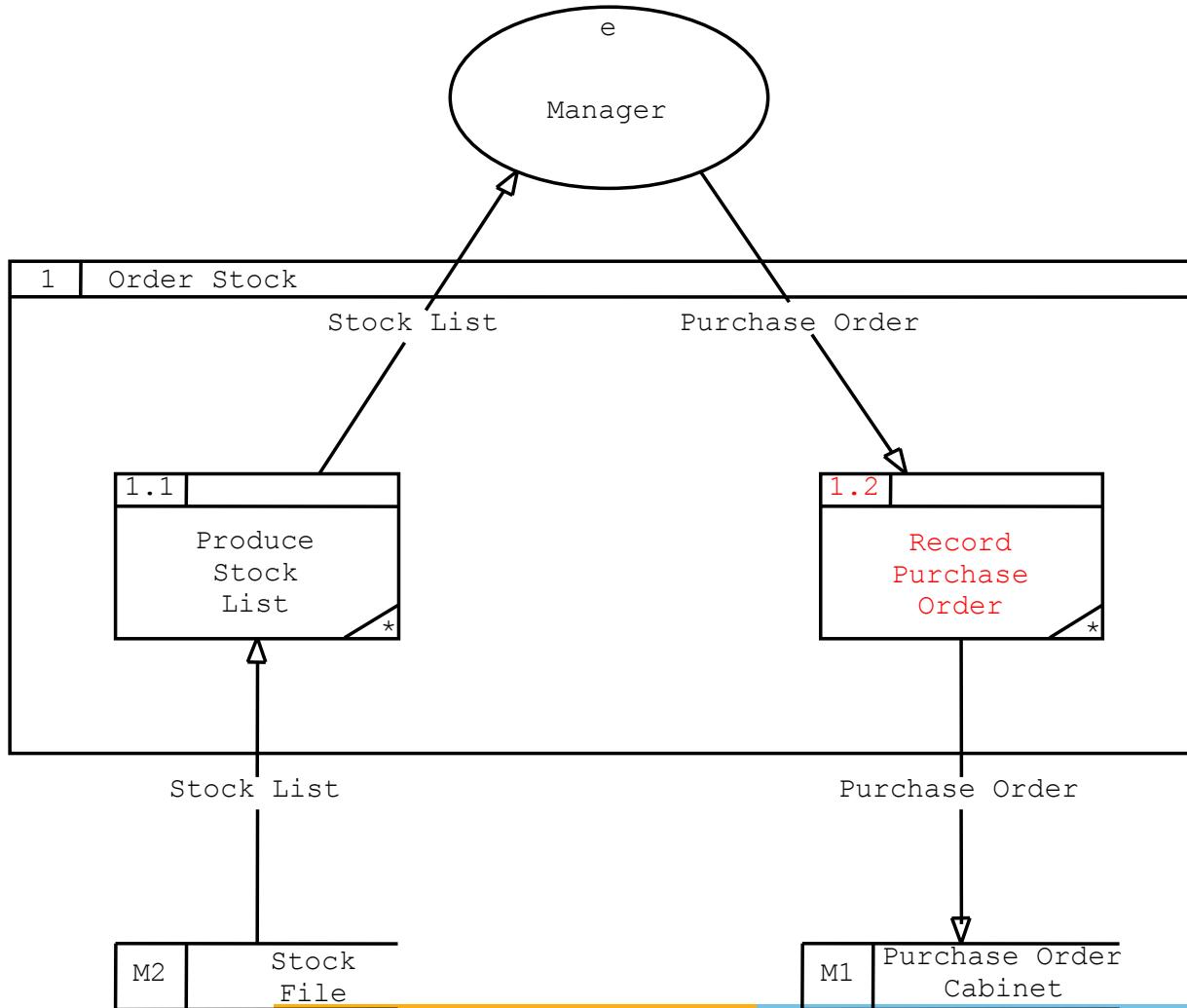
Decomposing Data Flow Diagrams



- A closer look at process 1 of the Small Stock System also shows that it is logically consistent and does indeed describe the activity of ordering stock
- On the other hand, it does not contain enough detail to understand exactly what happens when stock is ordered
- For example:

Data Flow Diagrams

Decomposing Data Flow Diagrams



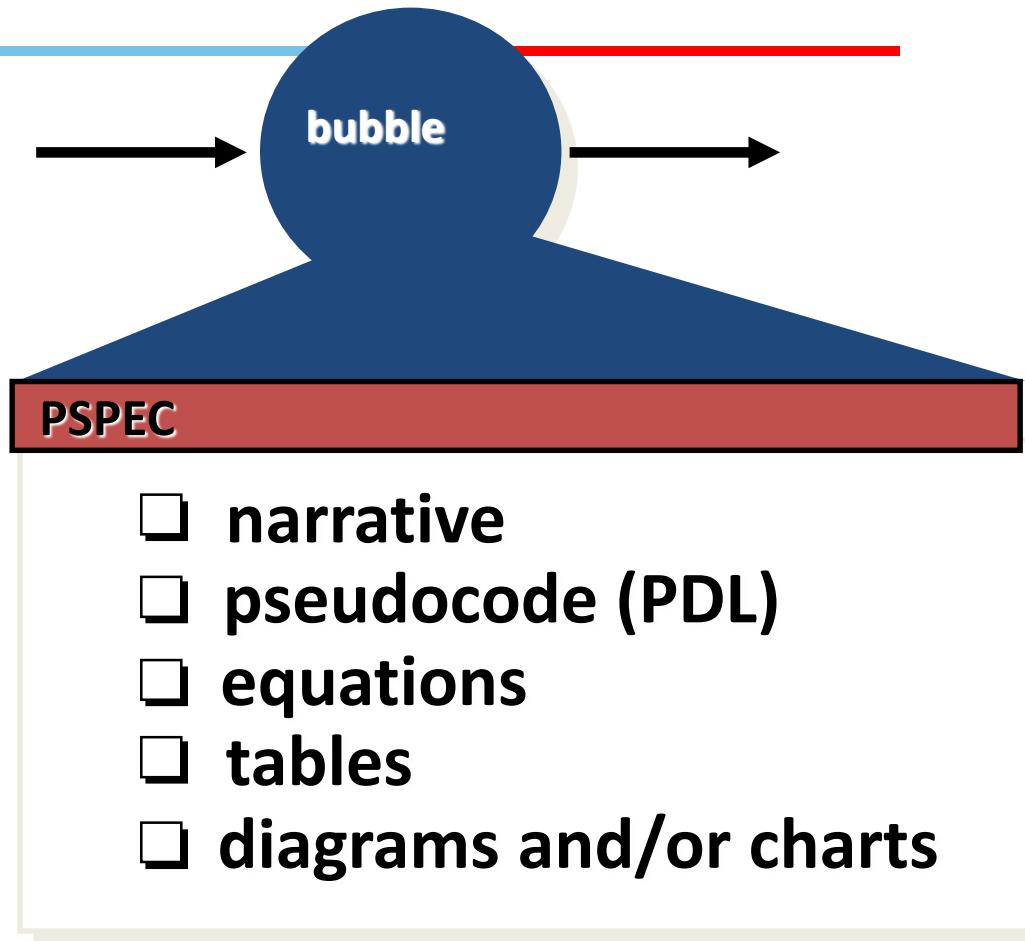
Data Flow Diagrams

Decomposing Data Flow Diagrams



- The decomposition of a DFD into lower level DFDs is known as **levelling**
- The DFD that shows the entire system is known as the ‘top level’ or ‘level 0’ DFD
- The DFDs that contain more detailed views of the level 0 processes make up ‘level 1’ DFDs
- Any level 1 process that is further decomposed gives rise to a level 2 DFD and so on

Process Specification (PSPEC)



Control Flow Modeling

- Represents “events” and the processes that manage events
- An “event” is a Boolean condition that can be ascertained by:
 - listing all sensors that are "read" by the software.
 - listing all interrupt conditions.
 - listing all "switches" that are actuated by an operator.
 - listing all data conditions.
 - recalling the noun/verb parse that was applied to the processing narrative, review all "control items" as possible CSPEC inputs/outputs.

Behavioural model

- Behavioural models are used to describe the overall behaviour of a system.
- Two types of behavioural model are:
 - Data processing models that show how data is processed as it moves through the system;
 - State machine models that show the systems response to events.
- These models show different perspectives so
- both of them are required to describe the system's behaviour.

Control Specification (CSPEC)

The CSPEC can be:

- state diagram
(sequential spec)
- state transition table
- decision tables
- activation tables



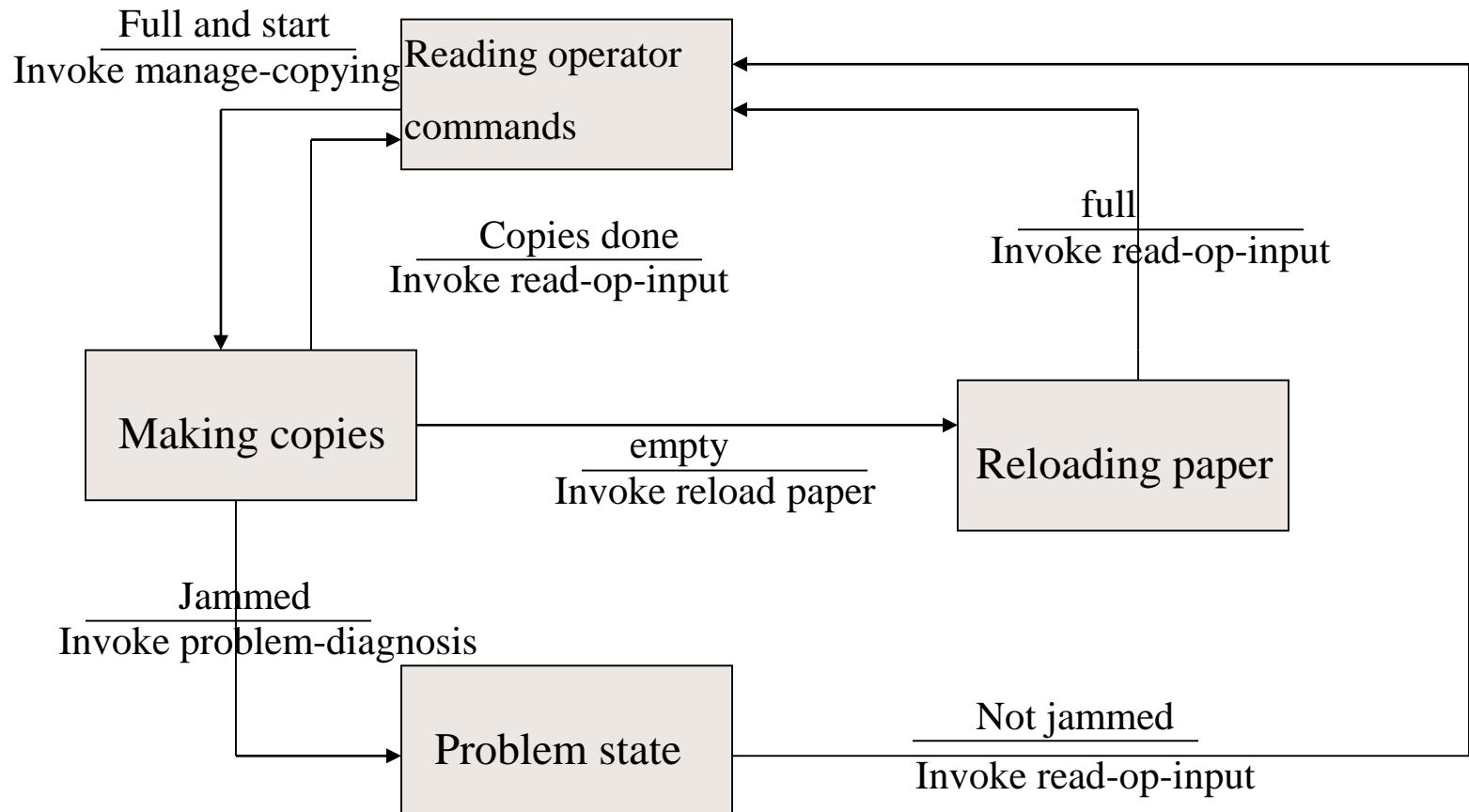
combinatorial spec

The States of a System

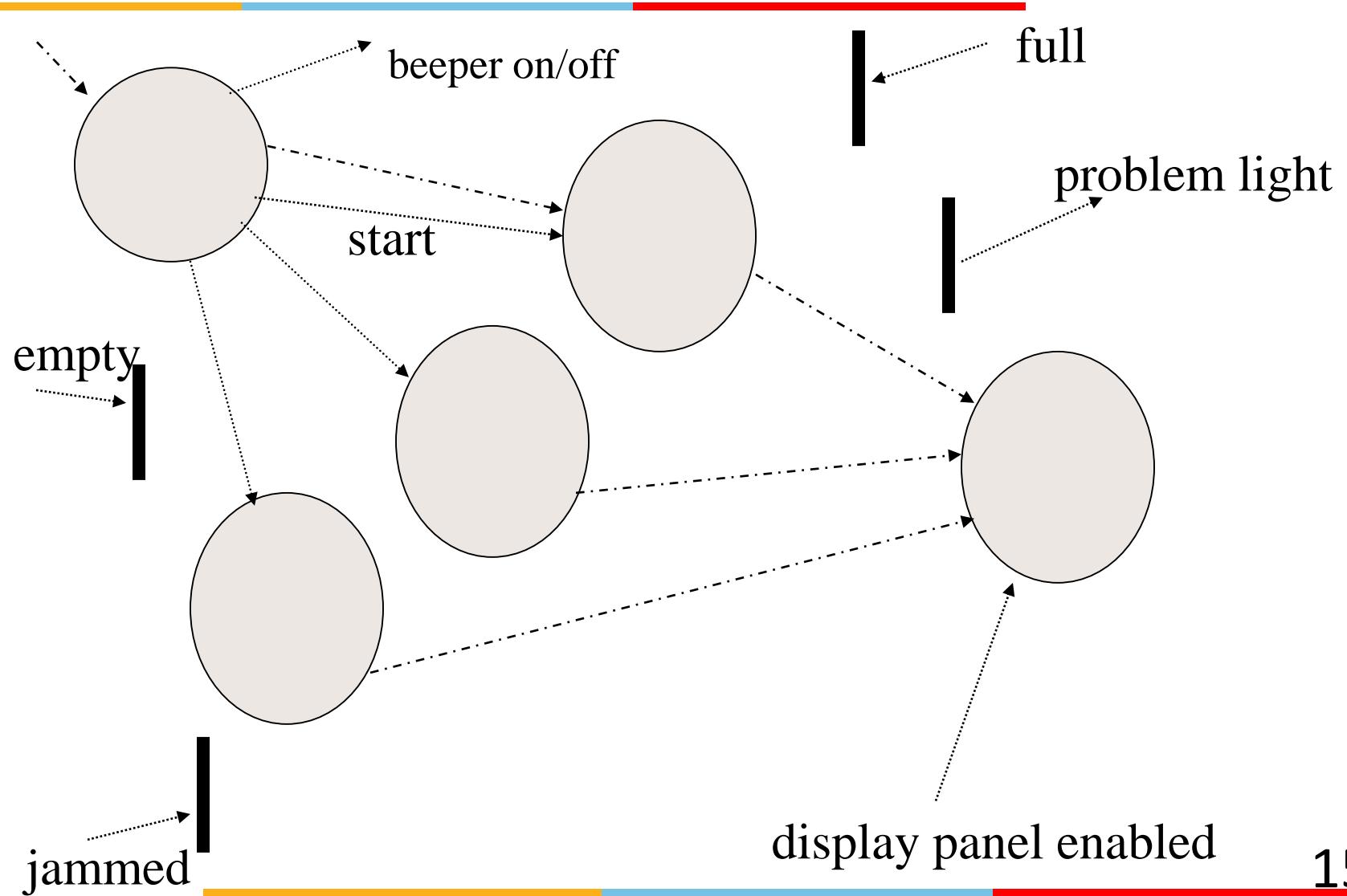


- State: a set of observable circumstances that characterizes the behavior of a system at a given time
- State transition: the movement from one state to another
- Event: an occurrence that causes the system to exhibit some predictable form of behavior
- Action- process that occurs as a consequence of making a transition

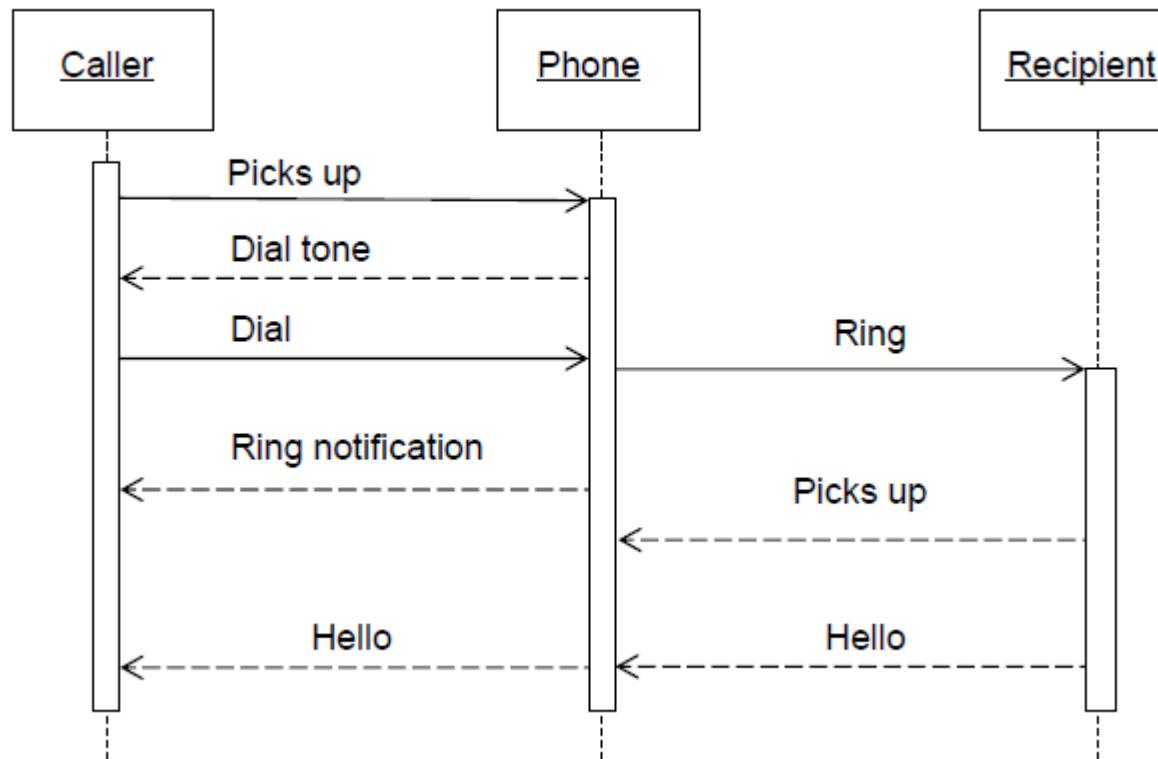
State Transition Diagram



Control flow diagram



Sequence Diagram

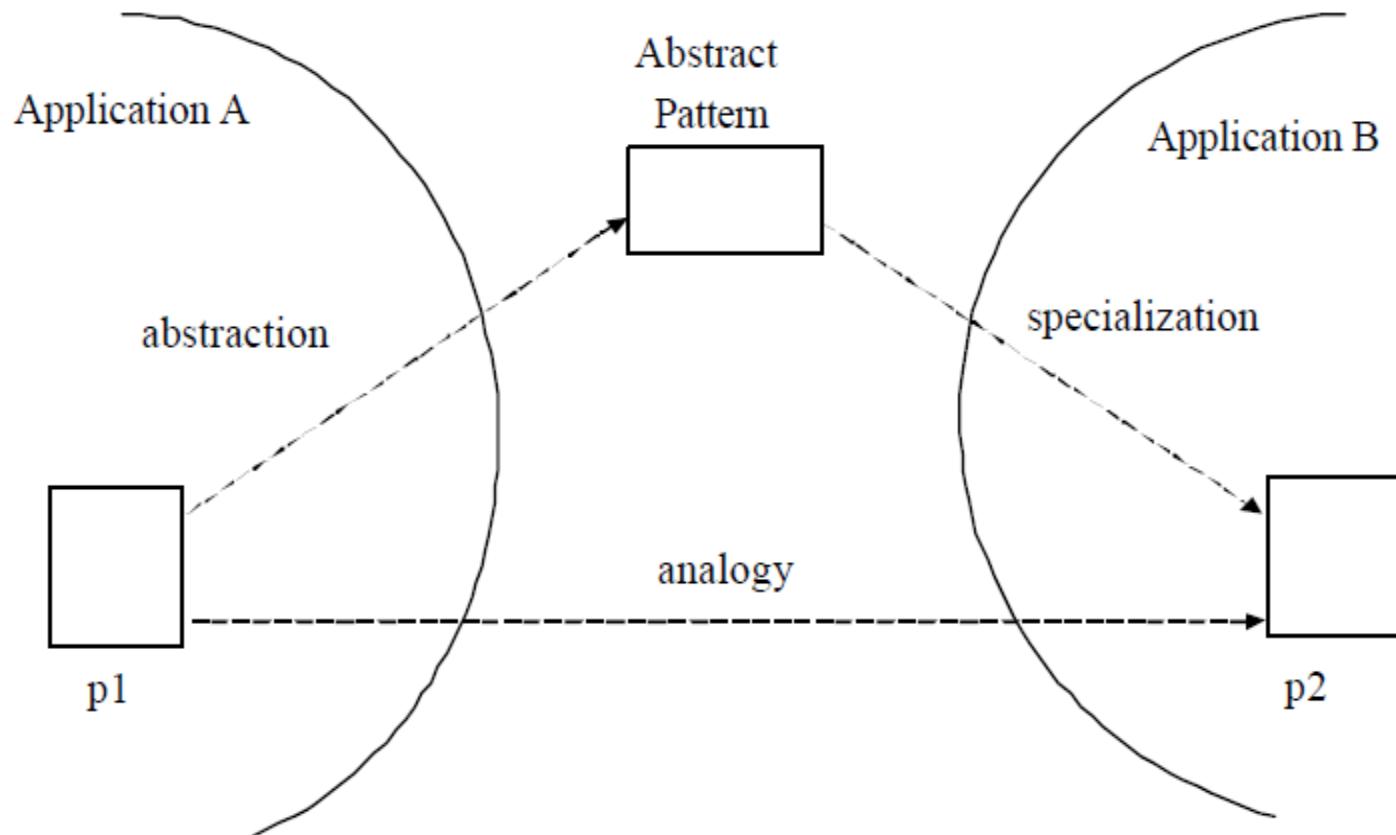


Patterns for Requirements Modeling



- Software patterns are a mechanism for capturing domain knowledge in a way that allows it to be reapplied when a new problem is encountered
 - domain knowledge can be applied to a new problem within the same application domain
 - the domain knowledge captured by a pattern can be applied by analogy to a completely different application domain.
- The original author of an analysis pattern does not “create” the pattern, but rather, *discovers* it as requirements engineering work is being conducted.
- Once the pattern has been discovered, it is documented

Discovering Analysis Patterns



Decision Table

Rules		
Condition	1	2
Actions	1	2

Condition				
N not numeric	T	F	F	F
N <= 1	-	T	F	F
N legal	-	-	T	F
N prime	-	-	T	F
Action				
Print “N prime”			X	
Print “N not prime”				X
Print error message	X	X		
Print “Good bye”			X	X
Input new value for N	X	X		
Stop			X	X

Credits

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

Thank You



BITS Pilani
Pilani Campus

Course Name : Software Engineering

T V Rao
Design Concepts

Purpose of Design

Design is where customer requirements, business needs, and technical considerations all come together in the formulation of a product or system

The design model provides detail about the software data structures, architecture, interfaces, and components

The design model can be assessed for quality and be improved before code is generated and tests are conducted

- Does the design contain errors, inconsistencies, or omissions?
- Are there better design alternatives?
- Can the design be implemented within the constraints, schedule, and cost that have been established?

How to Design

A designer must practice diversification and convergence -[Belady]

- The designer selects from design components, component solutions, and knowledge available through catalogs, textbooks, and experience
- The designer then chooses the elements from this collection that meet the requirements defined by requirements engineering and analysis modeling
- Convergence occurs as alternatives are considered and rejected until one particular configuration of components is chosen

Software design is an iterative process through which requirements are translated into a blueprint for constructing the software

- Design begins at a high level of abstraction that can be directly traced back to the data, functional, and behavioral requirements
- As design iteration occurs, subsequent refinement leads to design representations at much lower levels of abstraction

How to Design

Design is an intellectually challenging task

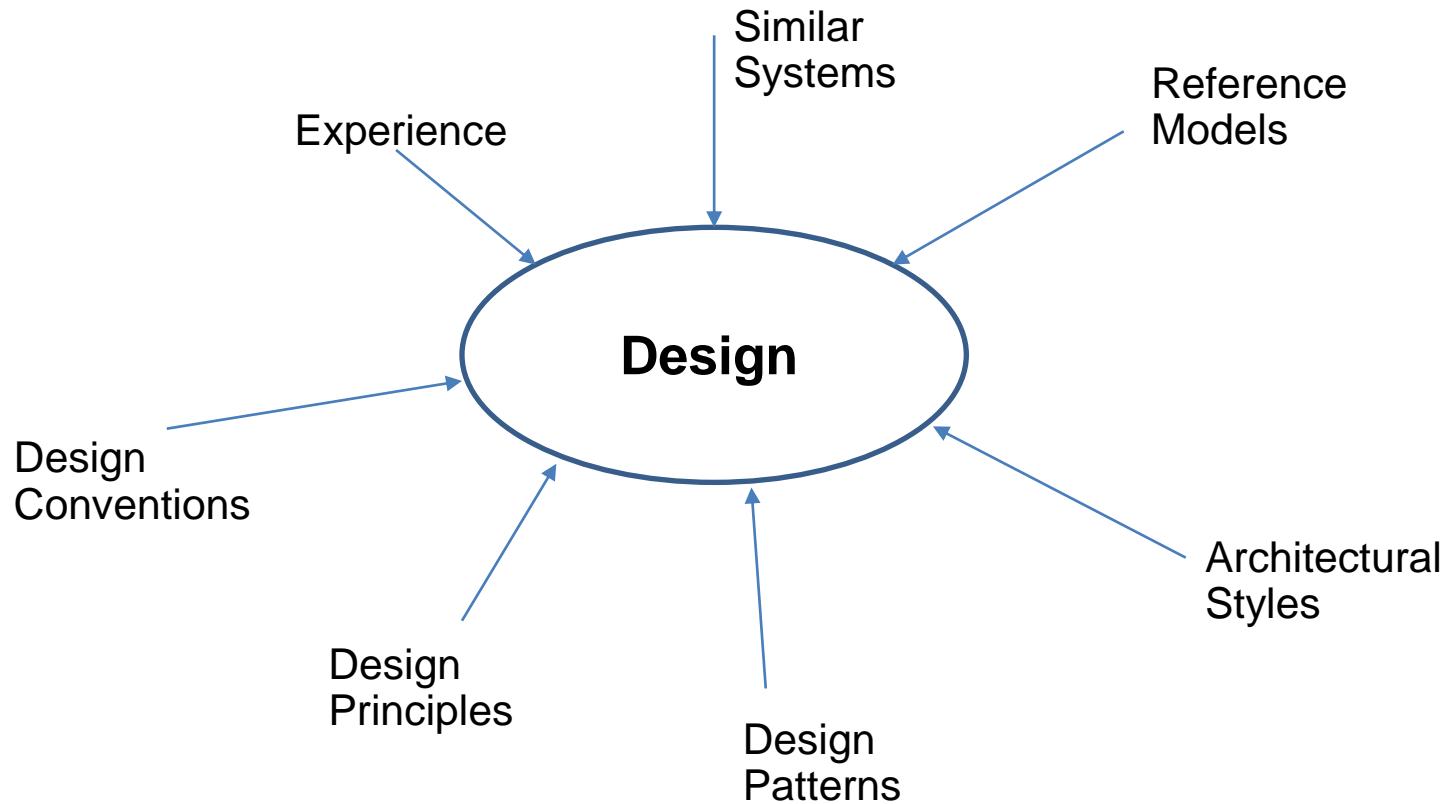
- Numerous possibilities the system must accommodate
- Nonfunctional design goals (e.g., ease of use, ease to maintain)
- External factors (e.g., standard data formats, government regulations)

We can improve our design by studying examples of good design. Many ways to leverage existing solutions

- Cloning: Borrow design/code in its entirety, with minor adjustments
- Reference models: Generic architecture that suggests how to decompose the system

The Design Process

Design is creative, but so many things influence it



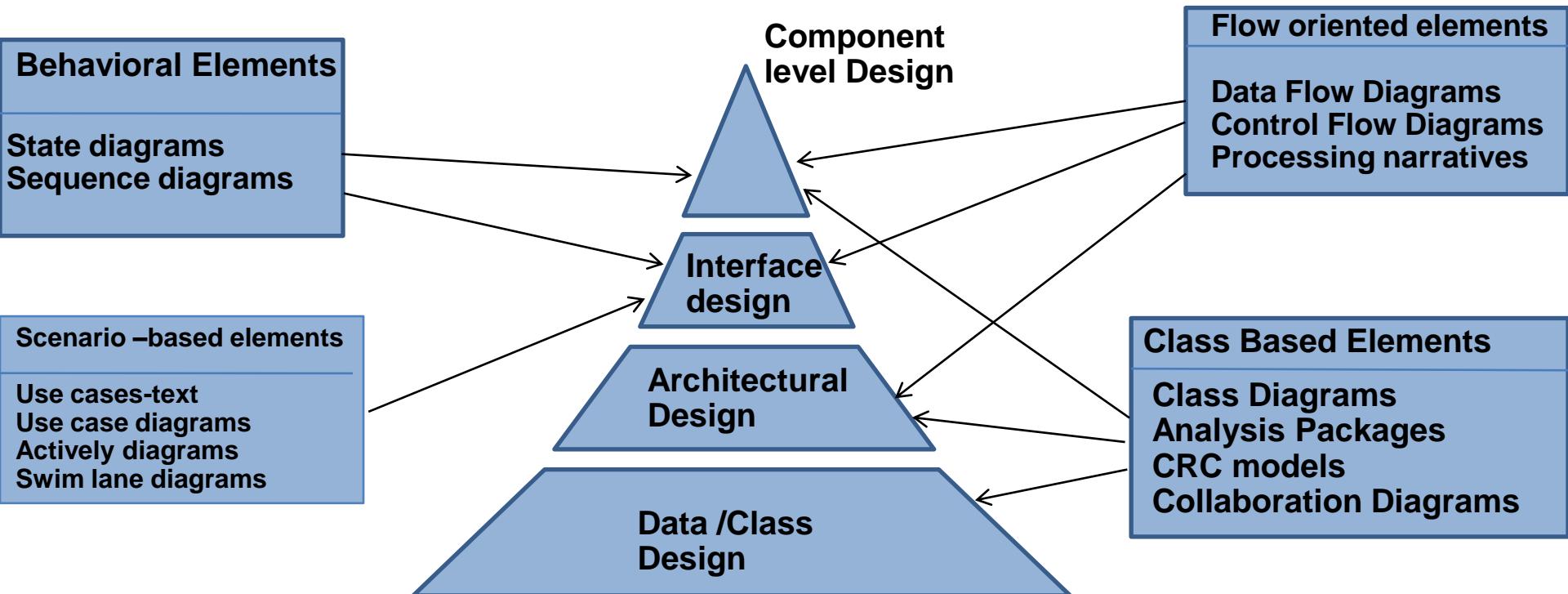
Analysis Model to Design Model



Each element of the analysis model provides information that is necessary to create the four design models

- The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
- The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
- The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
- The component-level design transforms structural elements of the software architecture into a procedural description of software components

Analysis → Design



Wisdom of Design

"Questions about whether design is necessary or affordable are quite beside the point; design is inevitable. The alternative to good design is bad design, [rather than] no design at all." **Douglas Martin (Author)**

"You can use an eraser on the drafting table or a sledge hammer on the construction site."
Frank Lloyd Wright (Architect, Structural Designer)

"The public is more familiar with bad design than good design. If is, in effect, conditioned to prefer bad design, because that is what it lives with; the new [design] becomes threatening, the old reassuring." **Paul Rand (Graphic Designer)**

"Every now and then go away, have a little relaxation, for when you come back to your work your judgment will be surer. Go some distance away because then the work appears smaller and more of it can be taken in at a glance and a lack of harmony and proportion is more readily seen." **Leonardo DaVinci (Genius)**

“Software Design Manifesto”

Mitch Kapor, the creator of Lotus 1-2-3, presented a “software design manifesto” in *Dr. Dobbs Journal*. He applied ideas of Roman architecture critic Vitruvius over software design. According to him, good software design should exhibit:

- *Firmness*: A program should not have any bugs that inhibit its function.
- *Commodity*: A program should be suitable for the purposes for which it was intended.
- *Delight*: The experience of using the program should be pleasurable one.

Task Set for Software Design

- 1) Examine the information domain model and design appropriate data structures for data objects and their attributes
- 2) Using the analysis model, select an architectural style (and design patterns) that are appropriate for the software
- 3) Partition the analysis model into design subsystems and allocate these subsystems within the architecture
 - a) Design the subsystem interfaces
 - b) Allocate analysis classes or functions to each subsystem
- 4) Create a set of design classes or components
 - a) Translate each analysis class description into a design class
 - b) Check each design class against design criteria; consider inheritance issues
 - c) Define methods associated with each design class
 - d) Evaluate and select design patterns for a design class or subsystem

Task Set for Software Design (continued)

- 5) Design any interface required with external systems or devices
- 6) Design the user interface
- 7) Conduct component-level design
 - a) Specify all algorithms at a relatively low level of abstraction
 - b) Refine the interface of each component
 - c) Define component-level data structures
 - d) Review each component and correct all errors uncovered
- 8) Develop a deployment model
 - Show a physical layout of the system, revealing which components will be located where in the physical computing environment

Fundamental Design Concepts

Abstraction—data, procedure, control

Architecture—the overall structure of the software

Patterns—"conveys the essence" of a proven design solution

Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces

Modularity—compartmentalization of data and function

Information Hiding—controlled interfaces

Functional independence—single-minded function and low coupling

Refinement—elaboration of detail for all abstractions

Aspects—a mechanism for understanding how global requirements affect design

Refactoring—a reorganization technique that simplifies the design

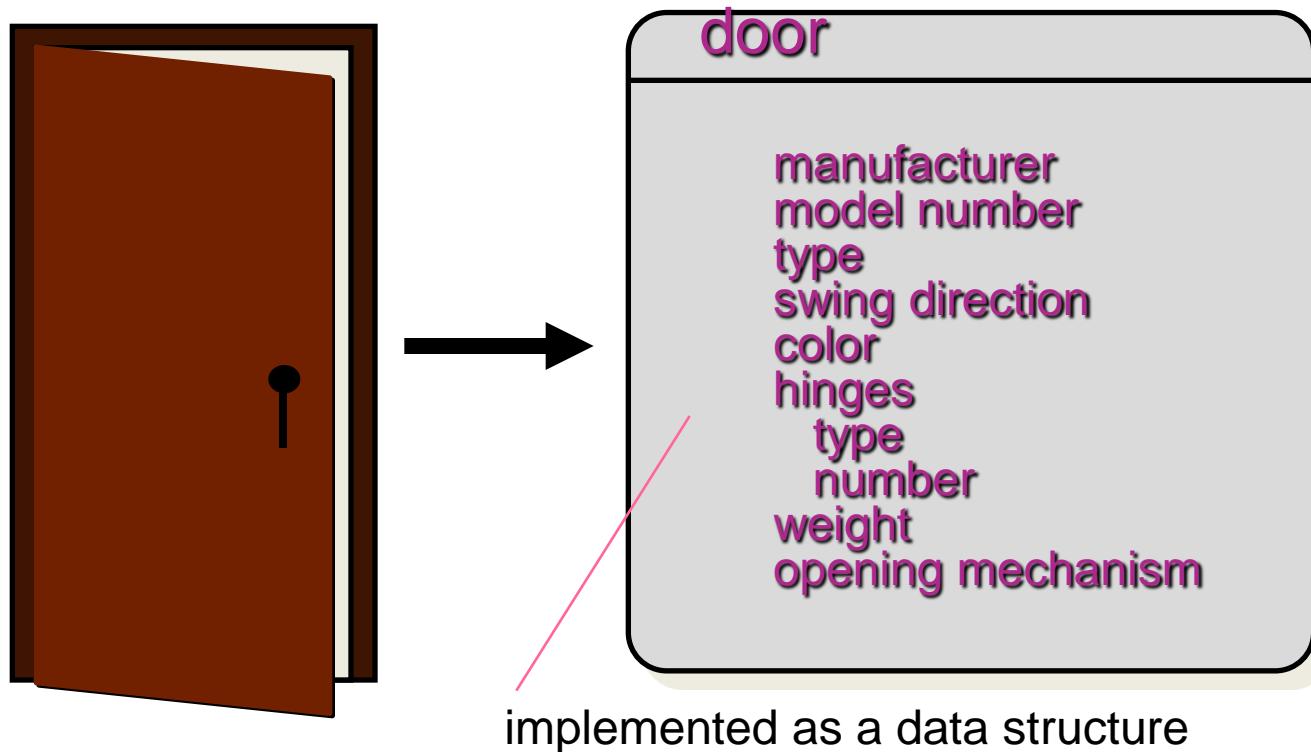
The beginning of wisdom (for a software engineer) is to recognize the difference between getting program to work, and getting it right

Abstraction et al.

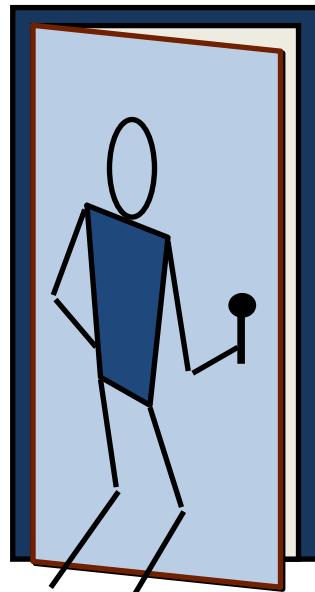
Abstraction

- process – extracting essential details
- entity – a model or focused representation
- Enablers
 - Information hiding
 - the suppression of inessential information
 - Encapsulation
 - process – enclosing items in a container
 - entity – enclosure that holds the items

Data Abstraction



Procedural Abstraction



implemented with a "knowledge" of the object that is associated with enter

Software Architecture

“The overall structure of the software and the ways in which that structure provides conceptual integrity for a system.”

The architectural model is derived from three sources:

- information about the application domain for the software to be built;
- specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand, and
- the availability of architectural patterns and styles.

Architectural Specifications

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Patterns

An **architectural pattern** expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

A **design pattern** provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

An **idiom** is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Design Patterns

The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution

A description of a design pattern may also consider a set of design forces.

- ***Design forces*** describe non-functional requirements (e.g., ease of maintainability, portability) associated the software for which the pattern is to be applied.

The **pattern characteristics** (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems.

Pattern Template

Design Pattern Template

Pattern name — describes the essence of the pattern in a short but expressive name

Intent — describes the pattern and what it does

Also-known-as — lists any synonyms for the pattern

Motivation — provides an example of the problem

Applicability — notes specific design situations in which the pattern is applicable

Structure — describes the classes that are required to implement the pattern

Participants — describes the responsibilities of the classes that are required to implement the pattern

Collaborations — describes how the participants collaborate to carry out their responsibilities

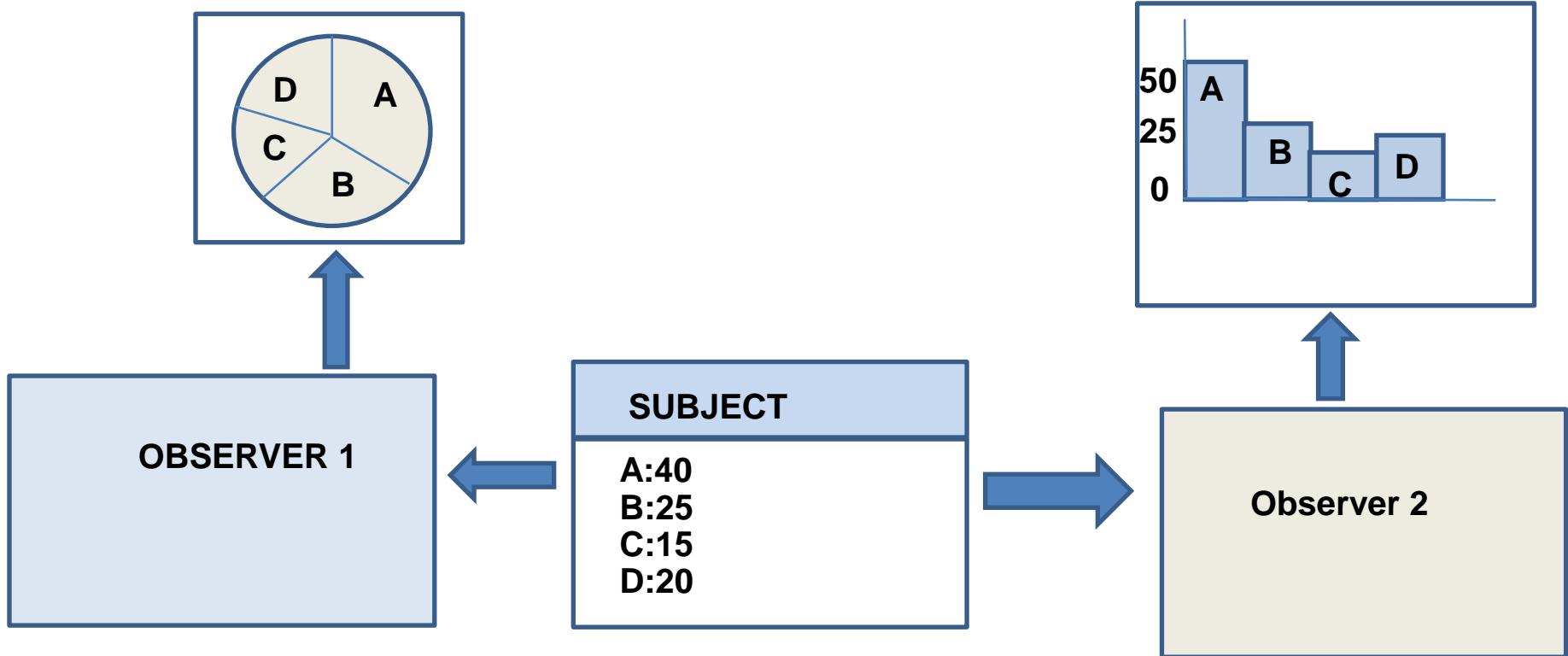
Consequences — describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented

Related patterns — cross-references related design patterns

The Observer pattern

Pattern name	Observer
Intent	Separates the display of the state of an object from the object itself and allows alternative displays to be provided.
Motivation	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified
Applicability	This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Structure	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations.
Participants & Collaborations	<p>The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.</p> <p>The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.</p>
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Multiple displays using the Observer pattern



Separation of Concerns

Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently

A *concern* is a feature or behavior that is specified as part of the requirements model for the software

By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

"modularity is the single attribute of software that allows a program to be intellectually manageable" – Glen Myers

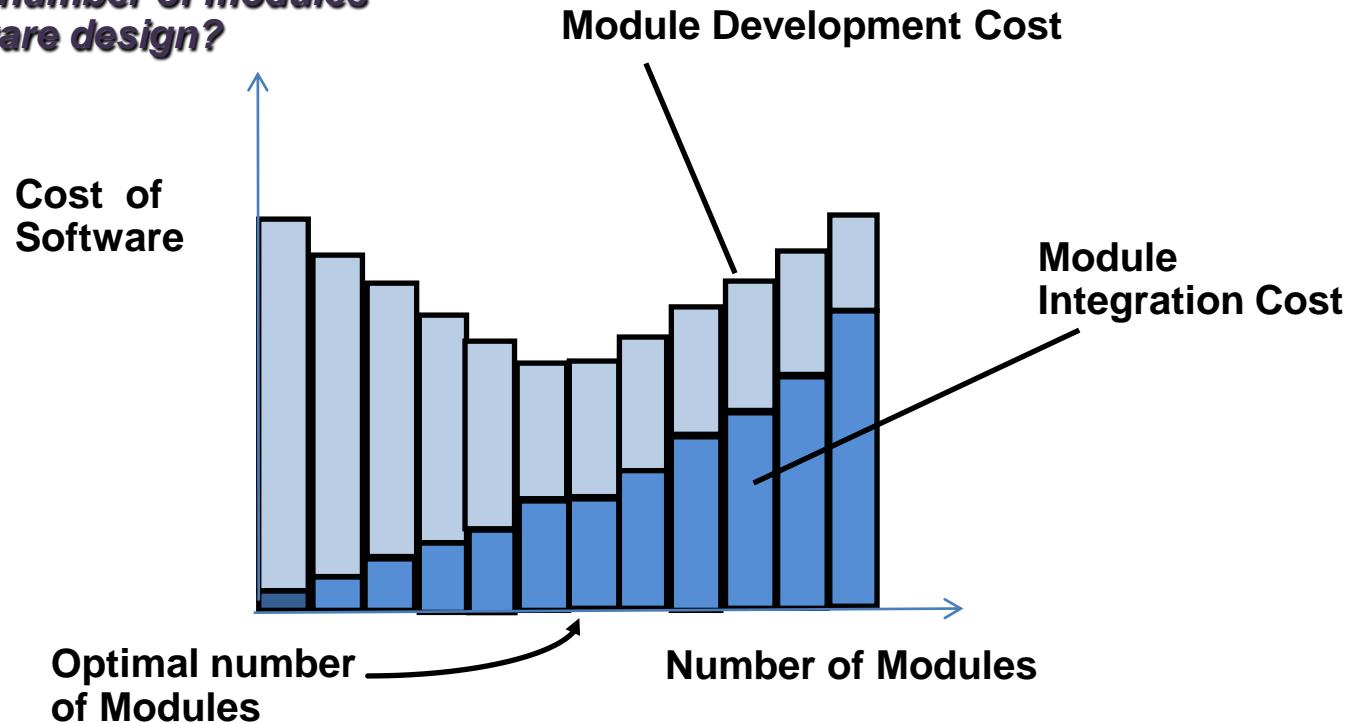
Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.

- The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.

In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

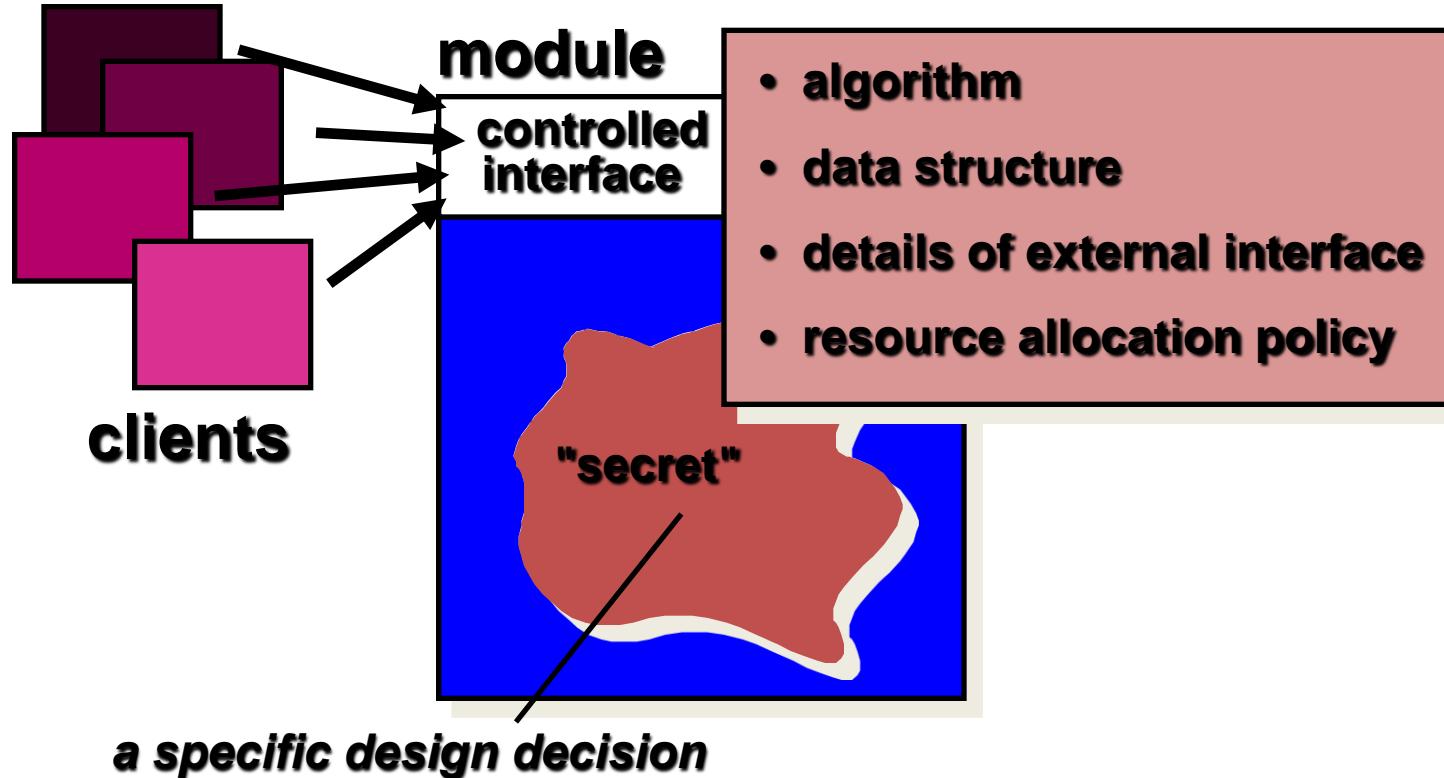
Modularity: Trade-offs

What is the "right" number of modules for a specific software design?



Information Hiding

Responds to “How do I decompose software solution to obtain best set of modules



Modules be “characterized by design decisions that hides from all others” - Parnas

Why Information Hiding?

- reduces the likelihood of “side effects”
 - limits the global impact of local design decisions
 - emphasizes communication through controlled interfaces
 - discourages the use of global data
 - leads to encapsulation—an attribute of high quality design
 - results in higher quality software
-

Functional Independence

Functional independence is achieved by developing modules with "**single-minded**" function and an "**aversion**" to excessive interaction with other modules.

Cohesion is an indication of the relative functional strength of a module.

- A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

Coupling is an indication of the relative interdependence among modules.

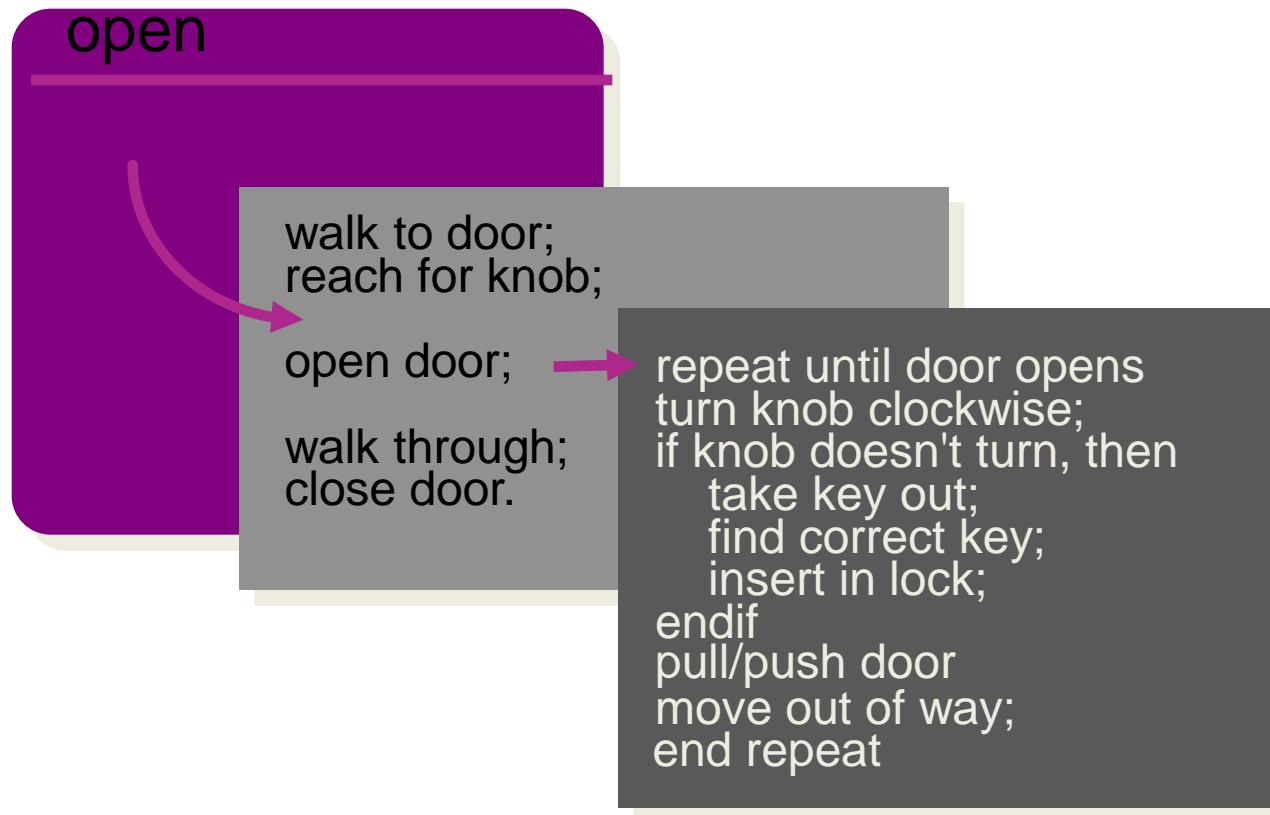
- Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Functional Independence

COHESION - the degree to which a module performs one and only one function.

COUPLING - the degree to which a module is "connected" to other modules in the system.

Stepwise Refinement



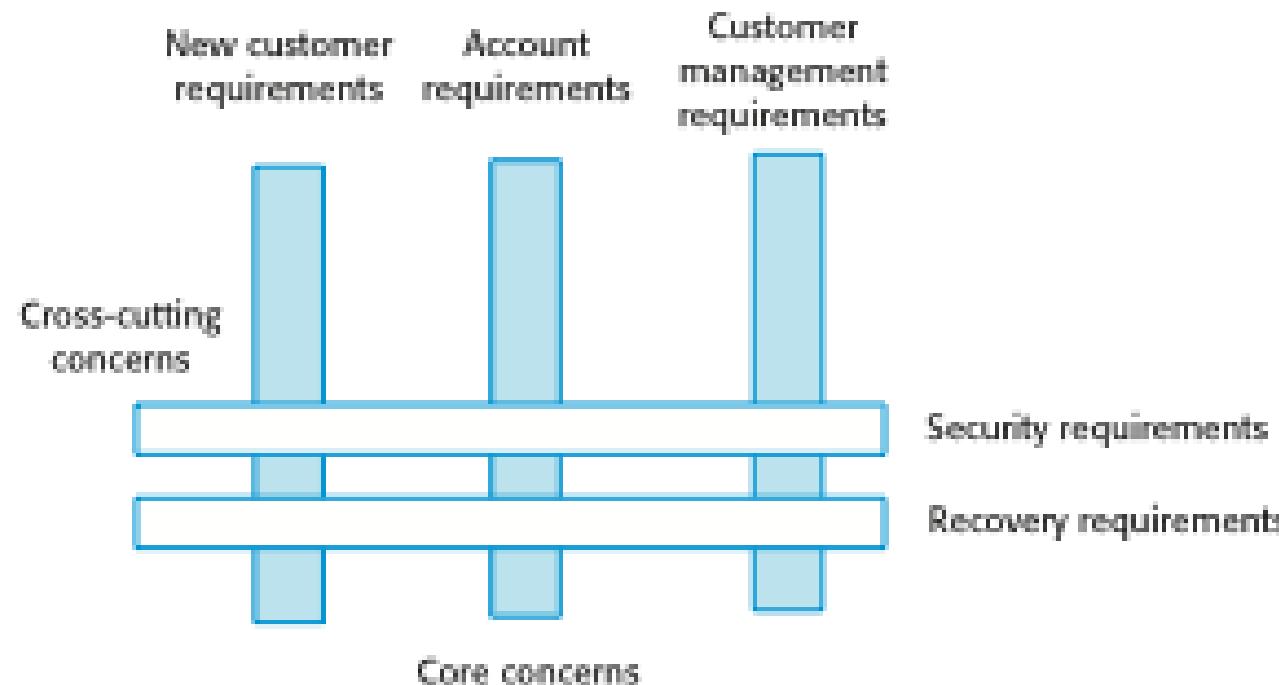
Aspects

Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* “if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account”

- Rosenhainer

An *aspect* is a representation of a cross-cutting concern.

Aspects are Cross-cutting concerns



Refactoring

Fowler [FOW99] defines refactoring in the following manner:

- "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure."

When software is refactored, the existing design is examined for

- redundancy
- unused design elements
- inefficient or unnecessary algorithms
- poorly constructed or inappropriate data structures
- or any other design failure that can be corrected to yield a better design.

Another Dimension of Design Challenge



According to industry observers, as of 2009, a large organization with over 250 applications would have used more than 50 different design languages and methodologies such as

- Various flavors of UML
- Flowcharts
- Decision tables,
- Data-flow diagrams
- HIPO (hierarchical Input-Process-Output) diagrams
- LePus3(Language for Pattern Uniform Specification)
- Express (data modeling language)
- Nassi-Schneiderman charts
- Jackson design
- Etc.

Due to incompatible design representations there is no easy way

- to pick out common features or patterns
- to identify library of reusable materials.

To Summarize

- A software design creates meaningful engineering representation (or model) of some software product that is to be built.
- Designers must strive to acquire a repertoire of alternative design information and learn to choose the elements that best match the analysis model.
- A design model can be traced to the customer's requirements and can be assessed for quality against predefined criteria. During the design process the software requirements model (data, function, behavior) is transformed into design models that describe the details of the data structures, system architecture, interfaces, and components necessary to implement the system.
- Each design product is reviewed for quality
 - identify and correct errors, inconsistencies, or omissions,
 - whether better alternatives exist, and
 - whether the design model can be implemented within the project constraints



Thank You...

Credits

- Software Engineering 7/ed by Roger Pressman – Reference
- Software Engineering 9/ed by Ian Sommerville - Reference



BITS Pilani

Pilani Campus

Course Name : Software Engineering

T V Rao
Architectural Design

Analysis Model to Design Model

Each element of the analysis model provides information that is necessary to create the four design models

- The data/class design transforms analysis classes into design classes along with the data structures required to implement the software
- The architectural design defines the relationship between major structural elements of the software; architectural styles and design patterns help achieve the requirements defined for the system
- The interface design describes how the software communicates with systems that interoperate with it and with humans that use it
- The component-level design transforms structural elements of the software architecture into a procedural description of software components

Definitions

- The software architecture of a program or computing system is the structure or structures of the system which comprise
 - The software components
 - The externally visible properties of those components
 - The relationships among the components
- Software architectural design represents the structure of the data and program components that are required to build a computer-based system
- An architectural design model is transferable
 - It can be applied to the design of other systems
 - It represents a set of abstractions that enable software engineers to describe architecture in predictable ways

Importance of Software Architecture

- Representations of software architecture are an enabler for communication between all stakeholders interested in the development of a computer-based system
- The software architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity
- The software architecture constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together

Uses of Architecture Descriptions

- **Understanding and communication:** An architecture description communicates the architecture to various stakeholders, including
 - users who will use the system,
 - clients who commissioned the system,
 - builders who will build the system, and,
 - other architects.
- **Reuse:** Architecture descriptions can help software reuse. The software engineering world has, for a long time, been working towards a discipline where software can be assembled from parts that are developed by different people and are available for others to use.
- **Construction and Evolution:** As architecture partitions the system into parts, some architecture provided partitioning can naturally be used for constructing the system, which also requires that the system be broken into parts such that different teams (or individuals) can separately work on different parts.
- **Analysis:** It is highly desirable if some important properties about the behaviour of the system can be determined before the system is actually built. This will allow the designers to consider alternatives and select the one that will best suit the needs.

Emphasis on Software Components

- A software architecture enables a software engineer to
 - Analyze the effectiveness of the design in meeting its stated requirements
 - Consider architectural alternatives at a stage when making design changes is still relatively easy
 - Reduce the risks associated with the construction of the software
- Focus is placed on the software component
 - A program module
 - An object-oriented class
 - A database
 - Middleware

Architectural Design Process



- Basic Steps
 - Creation of the data design
 - Derivation of one or more representations of the architectural structure of the system
 - Analysis of alternative architectural styles to choose the one best suited to customer requirements and quality attributes
 - Elaboration of the architecture based on the selected architectural style
- A database designer creates the data architecture for a system to represent the data components
- A system architect selects an appropriate architectural style derived during system engineering and software requirements analysis

Architecture addresses Non-functional Requirements



Non-functional requirements are the ones that don't appear in use cases. Rather than define *what* the application does, they are concerned with *how* the application provides the required functionality. There are three distinct areas of nonfunctional requirements that can impact architecture

- *Technical constraints*: Constrain design options by specifying certain technologies the application must use. "We only have Java developers, so we must develop in Java". "The existing database runs on Windows XP only".
- *Business constraints*: These constraint design options for business reasons. For example, "In order to widen our potential customer base, we must interface with XYZ tools". Another example is "The supplier of our middleware has raised prices prohibitively, so we're moving to an open source version".
- *Quality attributes*: These define an application's requirements in terms of scalability, availability, ease of change, portability, usability, performance, and so on. Quality attributes address issues of concern to application users, as well as other stakeholders like the project team itself or the project sponsor.

Architectural Style

- The software that is built for computer-based systems exhibit one of many architectural styles
- Each style describes a system category that encompasses
 - A set of component types that perform a function required by the system
 - A set of connectors (subroutine call, remote procedure call, data stream, socket) that enable communication, coordination, and cooperation among components
 - Semantic constraints that define how components can be integrated to form the system
 - A topological layout of the components indicating their runtime interrelationships

(Architectural) Style versus Pattern



- Architectural Style is a transformation that is imposed on the design of an entire system.
 - Architectural Pattern also imposes a transformation on the design, but differs from style in following ways:
 - The scope is less broad; focus on one aspect
 - Imposes a rule on architecture w.r.t. some functionality e.g. concurrency
 - Address specific behavioral issues e.g. synchronization or interrupt handling
-

Architectural Styles

Can be considered **semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

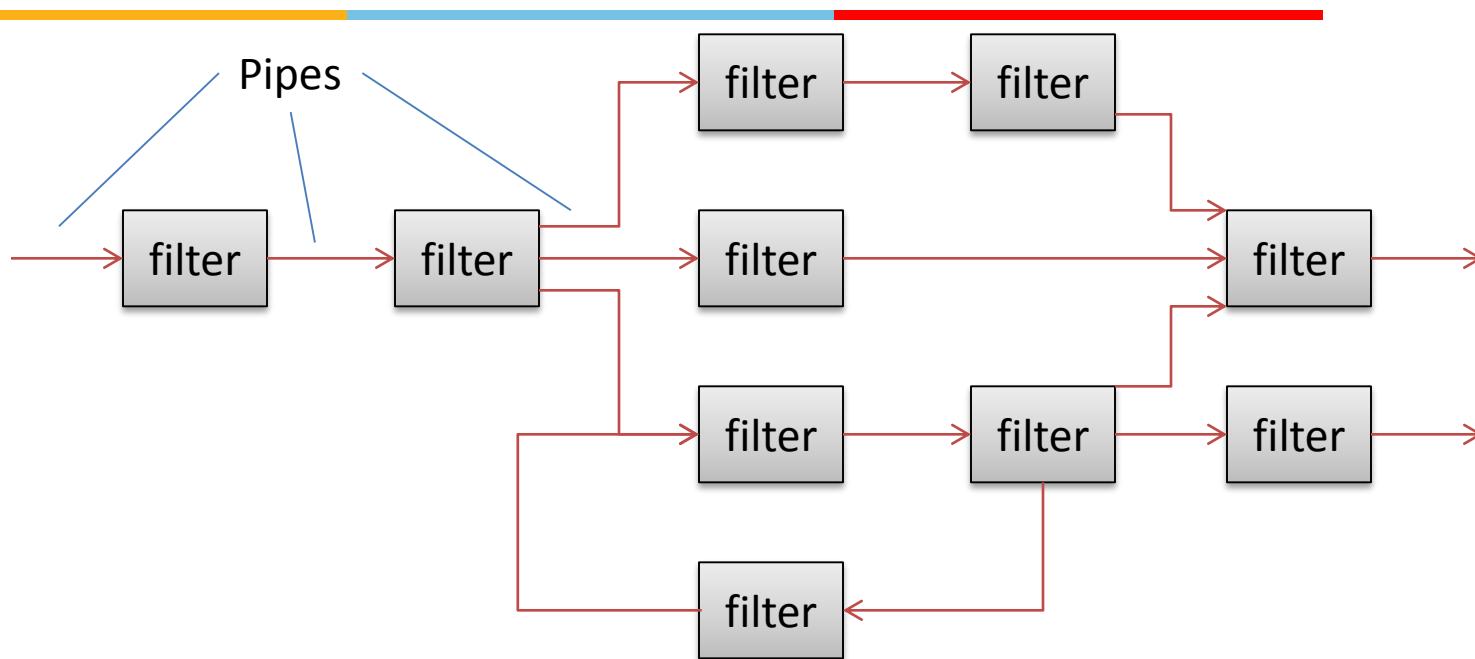
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures
- Data-centered architectures

Architectural Patterns

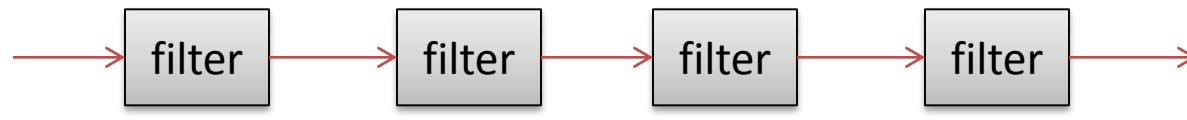
- **Concurrency**—applications must handle multiple tasks in a manner that simulates parallelism
 - *operating system process management* pattern
 - *task scheduler* pattern
- **Persistence**—Data persists if it survives past the execution of the process that created it. Two patterns are common:
 - a *database management system* pattern that applies the storage and retrieval capability of a DBMS to the application architecture
 - an *application level persistence* pattern that builds persistence features into the application architecture
- **Distribution**— the manner in which systems or components within systems communicate with one another in a distributed environment
 - A *broker* acts as a ‘middle-man’ between the client component and a server component.

Data Flow Architecture

(Architectural Style)



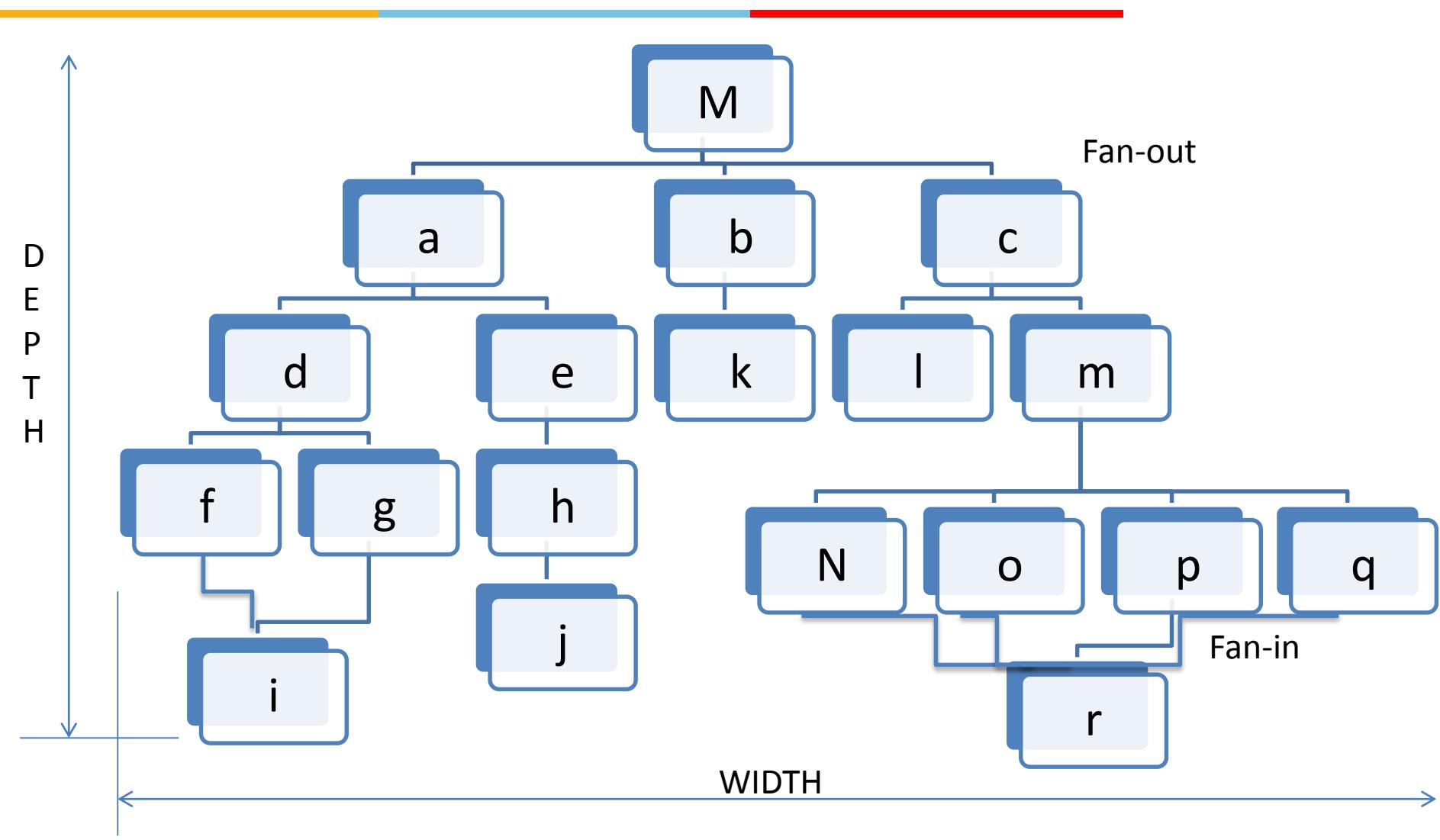
(a) Pipes and filters



(b) Batch sequential

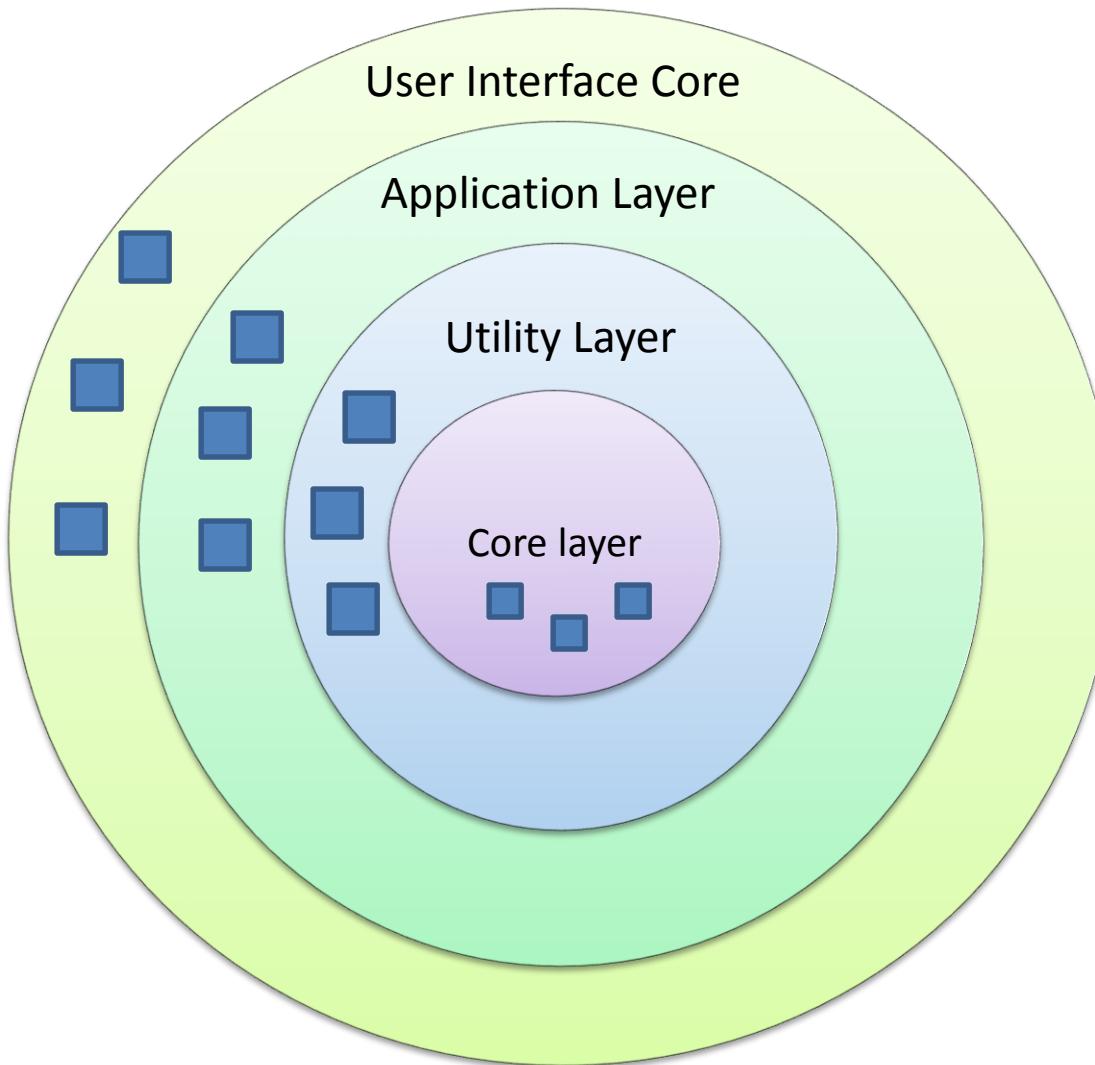
Call and Return Architecture

(Architectural Style)



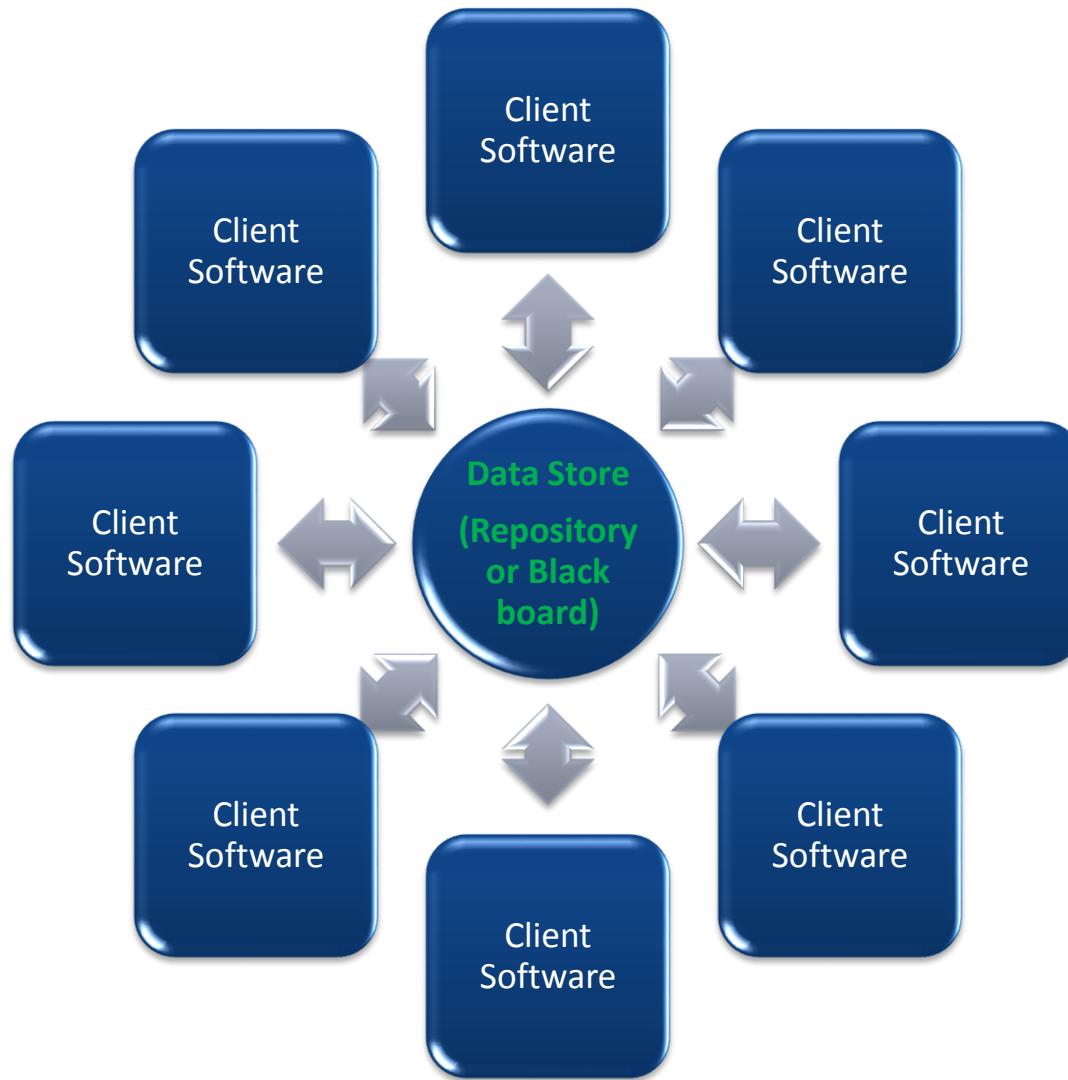
Layered Architecture

(Architectural Style)



Data-Centered Architecture

(Architectural Style)





Architectural Design Process

Architectural Design Steps

- 1) Represent the system in context
- 2) Define archetypes (well-understood term or symbol...so that you don't need to elaborate it hard)
- 3) Refine the architecture into components
- 4) Describe instantiations of the system

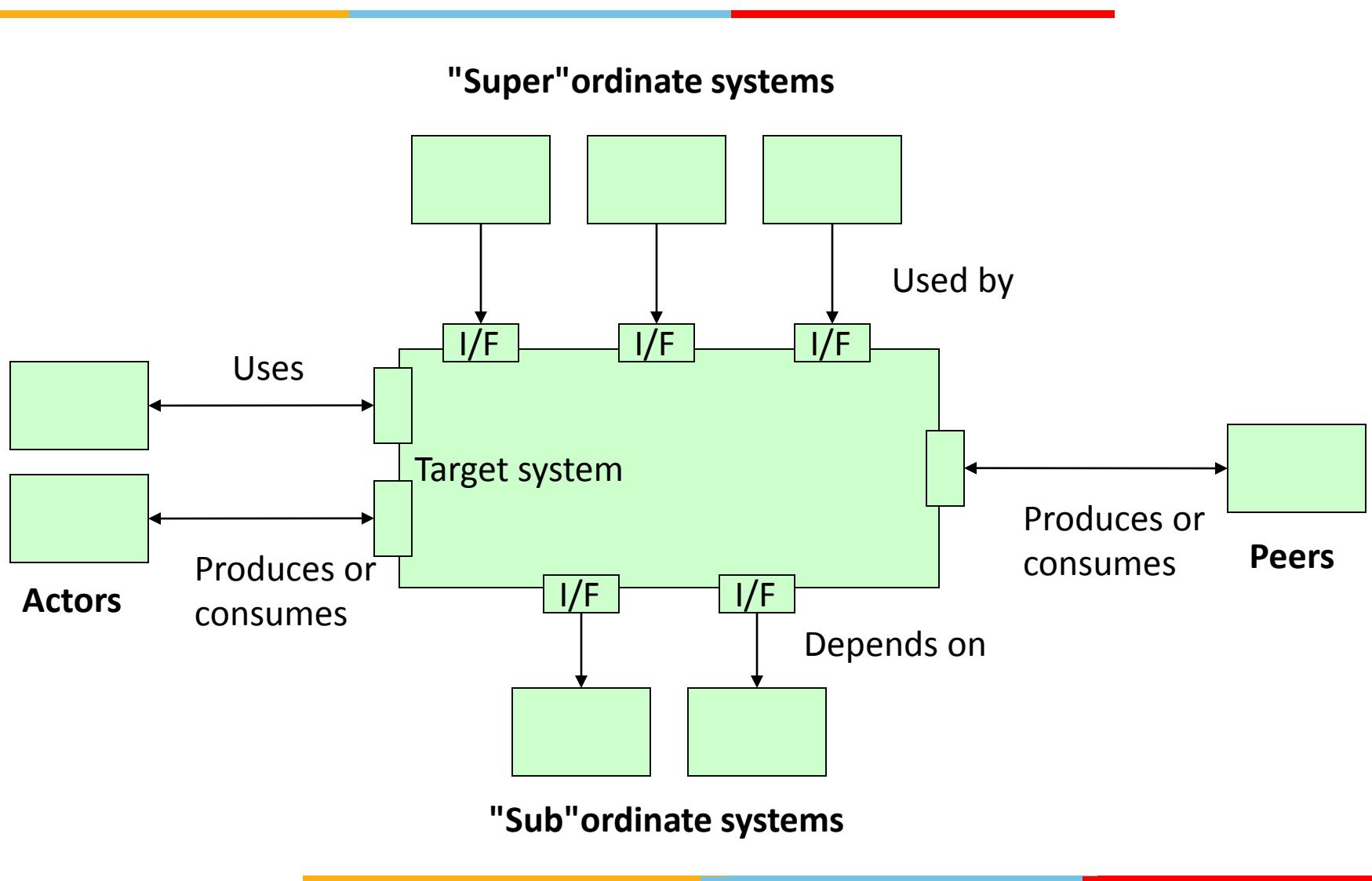
"A doctor can bury his mistakes, but an architect can only advise his client to plant vines." Frank Lloyd Wright

1. Represent the System in Context



- Use an architectural context diagram (ACD) that shows
 - The identification and flow of all information into and out of a system
 - The specification of all interfaces
 - Any relevant support processing from/by other systems
- An ACD models the manner in which software interacts with entities external to its boundaries
- An ACD identifies systems that interoperate with the target system
 - Super-ordinate systems
 - Use target system as part of some higher level processing scheme
 - Sub-ordinate systems
 - Used by target system and provide necessary data or processing
 - Peer-level systems
 - Interact on a peer-to-peer basis with target system to produce or consume data
 - Actors
 - People or devices that interact with target system to produce or consume data

1. Represent the System in Context



2. Define Archetypes

- Archetypes indicate the important abstractions within the problem domain (i.e., they model information)
 - An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system
 - It is also an abstraction from a class of programs with a common structure and includes class-specific design strategies and a collection of example program designs and implementations
 - Only a relatively small set of archetypes is required in order to design even relatively complex systems
 - The target system architecture is composed of these archetypes
 - They represent stable elements of the architecture
 - They may be instantiated in different ways based on the behavior of the system
 - They can be derived from the analysis class model
 - The archetypes and their relationships can be illustrated in a UML class diagram
-

Example Archetypes in Software Architecture

- Node
- Detector/Sensor
- Indicator
- Controller
- Manager

3. Refine the Architecture into Components

- Based on the archetypes, the architectural designer refines the software architecture into components to illustrate the overall structure and architectural style of the system
- These components are derived from various sources
 - The application domain provides application components, which are the domain classes in the analysis model that represent entities in the real world
 - The infrastructure domain provides design components (i.e., design classes) that enable application components but have no business connection
 - Examples: memory management, communication, database, and task management
 - The interfaces in the ACD imply one or more specialized components that process the data that flow across the interface
- A UML class diagram can represent the classes of the refined architecture and their relationships

4. Describe Instantiations of the System

- An actual instantiation of the architecture is developed by applying it to a specific problem
- This demonstrates that the architectural structure, style and components are appropriate
- A UML component diagram can be used to represent this instantiation



Representing Architecture

Multiple Views for Software Architecture

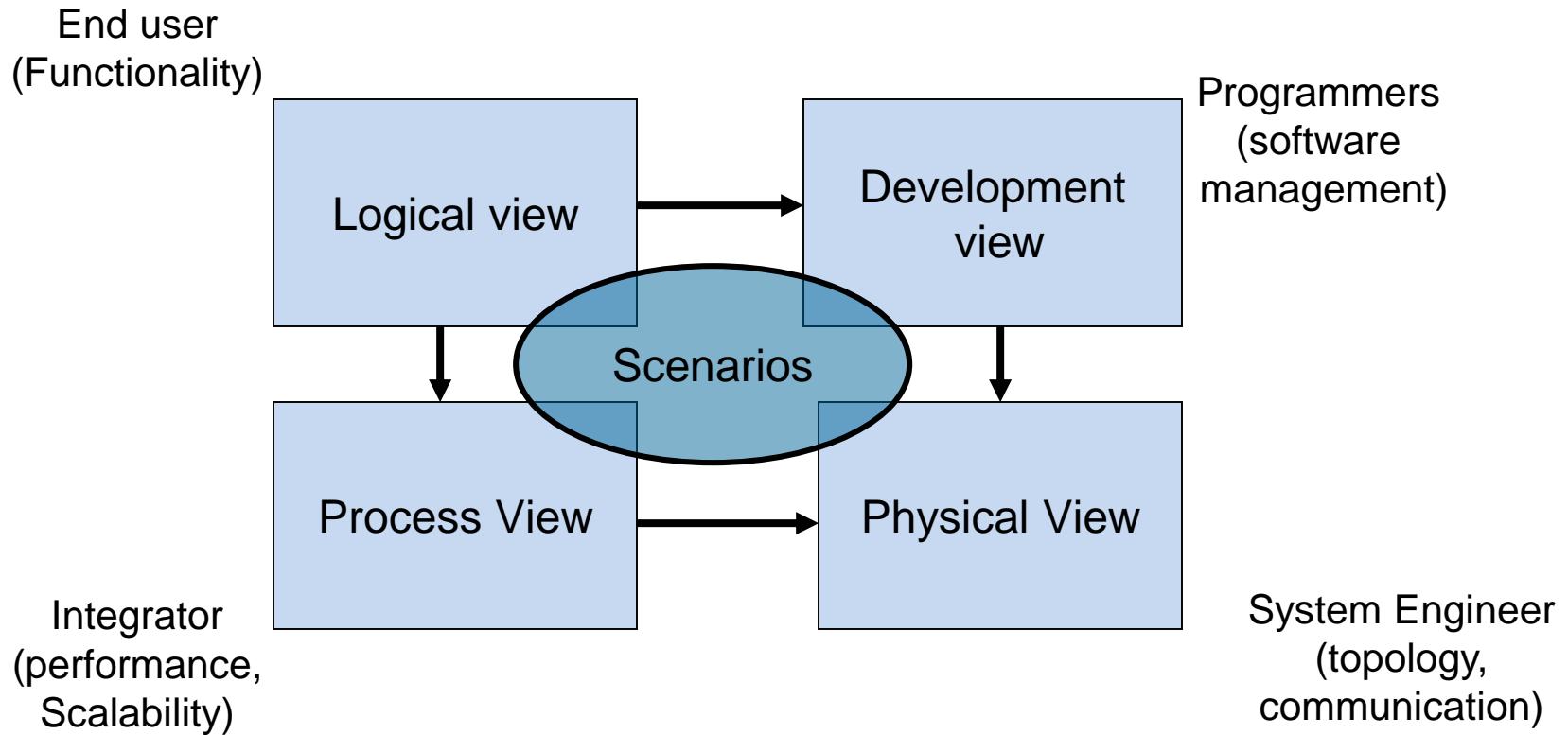


Boxes and Arrows of architecture diagram struggle to represent more on one blueprint than it can actually express

Do the boxes represent
running programs? Or
chunks of source code? Or
physical computers? Or
Merely logical groupings of functionality?

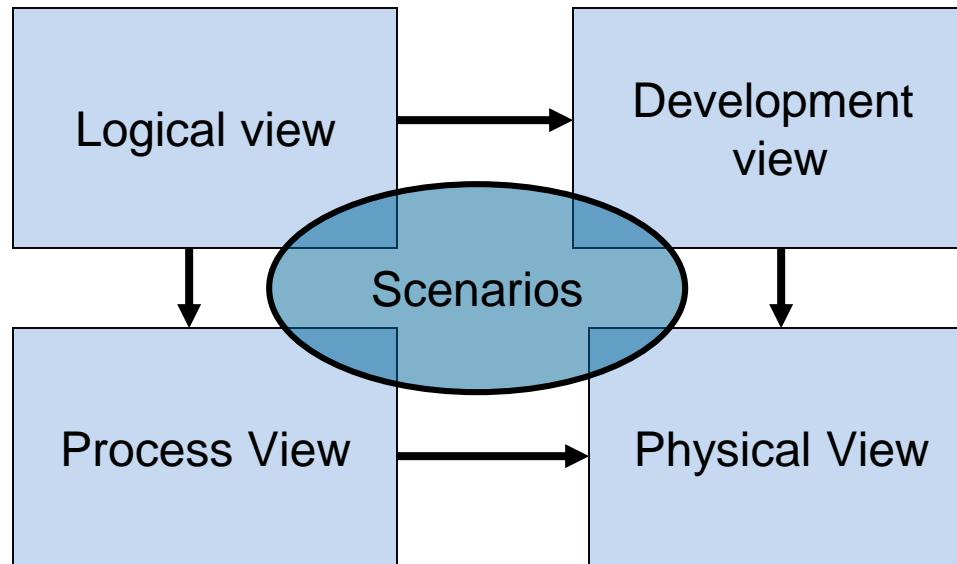
Do the arrows represent
compilation dependencies? Or
control flows? Or
data flows?

“4+1” Views of Architecture



“4+1” Views in UML

Class, object,
State diagram



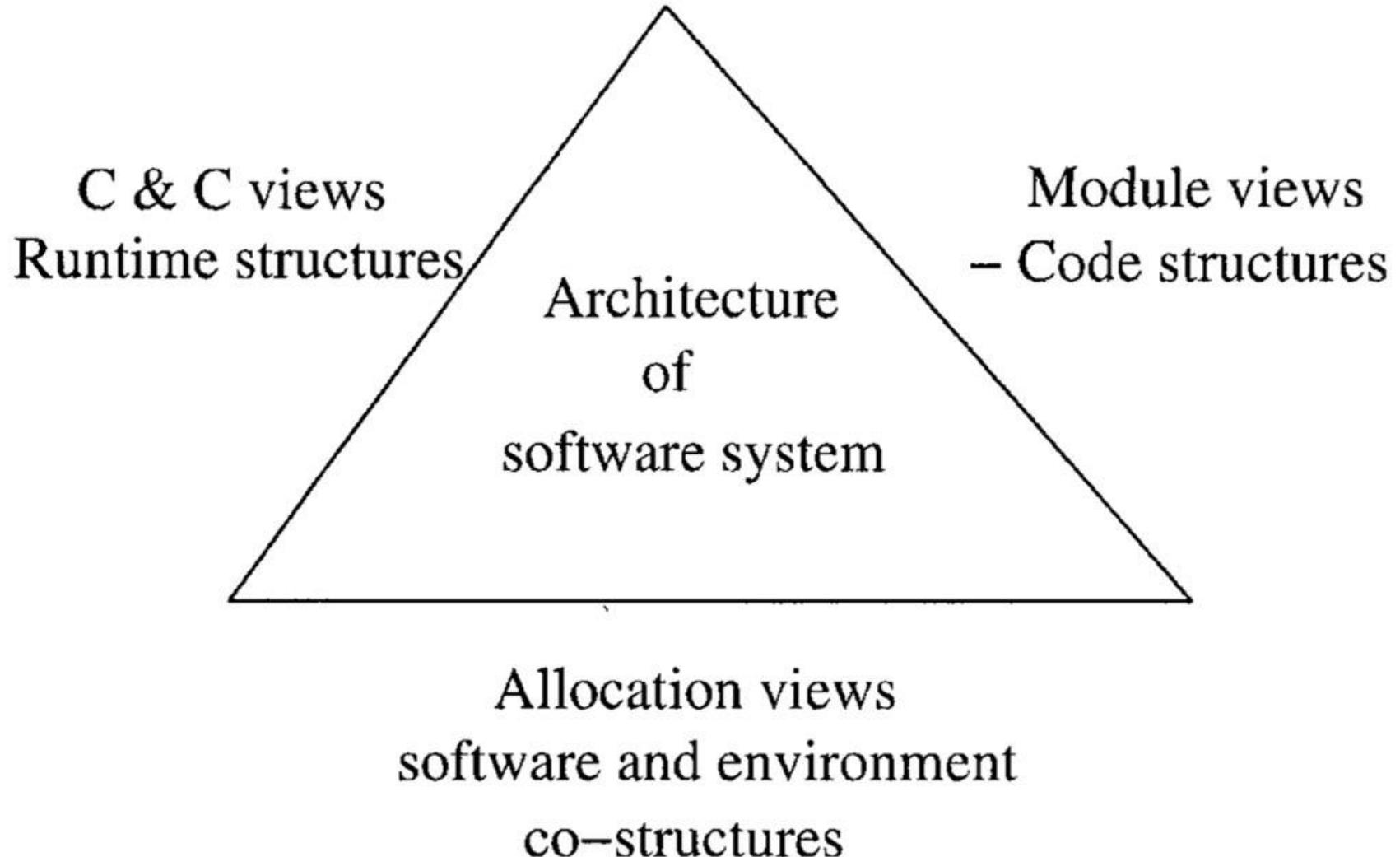
Component
diagram

Activity,
Sequence
diagrams

Scenarios = Use cases

Deployment
diagram

Another Set of Views





Assessing Alternative Architectural Designs

Architecture Assessment Methods

- A. Ask a set of questions that provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture
 - Assess the control in an architectural design (see next slide)
 - Assess the data in an architectural design (see upcoming slide)
- B. Apply the architecture trade-off analysis method
- C. Assess the architectural complexity

Approach A: Questions – Assessing Control in an Architectural Design

- How is control managed within the architecture?
 - Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
 - How do components transfer control within the system?
 - How is control shared among components?
 - What is the control topology (i.e., the geometric form that the control takes)?
 - Is control synchronized or do components operate asynchronously?
-

Approach A: Questions – Assessing Data in an Architectural Design

- How are data communicated between components?
- Is the flow of data continuous, or are data objects passed to the system sporadically?
- What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)
- Do data components exist (e.g., a repository or blackboard), and if so, what is their role?
- How do functional components interact with data components?
- Are data components passive or active (i.e., does the data component actively interact with other components in the system)?
- How do data and control interact within the system?

Approach B: Architecture Trade-off Analysis Method

- 1) Collect scenarios representing the system from the user's point of view
- 2) Elicit requirements, constraints, and environment description to be certain all stakeholder concerns have been addressed
- 3) Describe the candidate architectural styles that have been chosen to address the scenarios and requirements
- 4) Evaluate quality attributes by considering each attribute in isolation (reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability)
- 5) Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style by making small changes in the architecture
- 6) Critique the application of the candidate architectural styles (from step #3) using the sensitivity analysis conducted in step #5

Based on the results of steps 5 and 6, some architecture alternatives may be eliminated. Others will be modified and represented in more detail until a target architecture is selected

Approach C: Assessing Architectural Complexity

- The overall complexity of a software architecture can be assessed by considering the dependencies between components within the architecture
- These dependencies are driven by the information and control flow within a system
- Three types of dependencies
 - Sharing dependency $U \leftarrow \rightarrow \square \leftarrow \rightarrow V$
 - Represents a dependency relationship among consumers who use the same source or producer
 - Flow dependency $\rightarrow U \rightarrow V \rightarrow$
 - Represents a dependency relationship between producers and consumers of resources
 - Constrained dependency $U \text{ “XOR” } V$
 - Represents constraints on the relative flow of control among a set of activities such as mutual exclusion between two components

To Summarize

- A software architecture provides a uniform, high-level view of the system to be built
 - It depicts
 - The structure and organization of the software components
 - The properties of the components
 - The relationships (i.e., connections) among the components
 - Software components include program modules and the various data representations that are manipulated by the program
 - A number of different architectural styles are available that encompass a set of component types, a set of connectors, semantic constraints, and a topological layout
 - The choice of a software architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative architectures
-



Thank You...

References

- Software Engineering 7/ed by Roger Pressman
- Software Engineering 9/ed by Ian Sommerville
- Bass, Clements, and Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003
- Kruchten, Philippe. Architectural Blueprints—The “4+1” View Model of Software Architecture, IEEE Software Nov 1995
- An Integrated Approach to Software Engineering, Third Edition by Pankaj Jalote Springer © 2005



BITS Pilani
Pilani Campus

Course Name : Software Engineering

T V Rao
Component Design



What is a Component?

OMG Unified Modeling Language Specification [OMG01] defines a component as

“... a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”

OO view: a component contains a set of collaborating classes

Conventional view: a component contains processing logic, the internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.

Object-oriented View

- A component is viewed as a set of one or more collaborating classes
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
 - This also involves defining the interfaces that enable classes to communicate and collaborate
- This elaboration activity is applied to every component defined as part of the architectural design
- Once this is completed, the following steps are performed
 - 1) Provide further elaboration of each attribute, operation, and interface
 - 2) Specify the data structure appropriate for each attribute
 - 3) Design the algorithmic detail required to implement the processing logic associated with each operation
 - 4) Design the mechanisms required to implement the interface to include the messaging that occurs between objects

Conventional View

- A component is viewed as a functional element (i.e., a module) of a program that incorporates
 - The processing logic
 - The internal data structures that are required to implement the processing logic
 - An interface that enables the component to be invoked and data to be passed to it
- A component serves one of the following roles
 - A control component that coordinates the invocation of all other problem domain components
 - A problem domain component that implements a complete or partial function that is required by the customer
 - An infrastructure component that is responsible for functions that support the processing required in the problem domain

Conventional View (continued)

- Conventional software components are derived from the data flow diagrams (DFDs) in the analysis model
 - Each transform bubble (i.e., module) represented at the lowest levels of the DFD is mapped into a module hierarchy
 - Control components reside near the top
 - Problem domain components and infrastructure components migrate toward the bottom
 - Functional independence is strived for between the transforms
- Once this is completed, the following steps are performed for each transform
 - 1) Define the interface for the transform (the order, number and types of the parameters)
 - 2) Define the data structures used internally by the transform
 - 3) Design the algorithm used by the transform (using a stepwise refinement approach)

Process-related View

- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- As the software architecture is formulated, components are selected from the library and used to populate the architecture
- Because the components in the library have been created with reuse in mind, each contains the following:
 - A complete description of their interface
 - The functions they perform
 - The communication and collaboration they require

Component-level Design Principles

- **Open-closed principle**
 - A module or component should be open for extension but closed for modification
 - The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component
- **Liskov substitution principle**
 - Subclasses should be substitutable for their base classes
 - A component that uses a base class should continue to function properly if a subclass of the base class is passed to the component instead
- **Dependency inversion principle**
 - Depend on abstractions (i.e., interfaces); do not depend on concretions
 - The more a component depends on other concrete components (rather than on the interfaces) the more difficult it will be to extend
- **Interface segregation principle**
 - Many client-specific interfaces are better than one general purpose interface
 - For a server class, specialized interfaces should be created to serve major categories of clients
 - Only those operations that are relevant to a particular category of clients should be specified in the interface

Component Packaging Principles

- **Release reuse equivalency principle**
 - The granularity of reuse is the granularity of release
 - Group the reusable classes into packages that can be managed, upgraded, and controlled as newer versions are created
- **Common closure principle**
 - Classes that change together belong together
 - Classes should be packaged cohesively; they should address the same functional or behavioral area on the assumption that if one class experiences a change then they all will experience a change
- **Common reuse principle**
 - Classes that aren't reused together should not be grouped together
 - Classes that are grouped together may go through unnecessary integration and testing when they have experienced no changes but when other classes in the package have been upgraded

Designing O-O Components



-
- Step 1. Identify all design classes that correspond to the problem domain.
 - Step 2. Identify all design classes that correspond to the infrastructure domain.
 - Step 3. Elaborate all design classes that are not acquired as reusable components.
 - Step 3a. Specify message details when classes or component collaborate.
 - Step 3b. Identify appropriate interfaces for each component.
 - Step 3c. Elaborate attributes and define data types and data structures required to implement them.
 - Step 3d. Describe processing flow within each operation in detail.
 - Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
 - Step 5. Develop and elaborate behavioural representations for a class or component.
 - Step 6. Elaborate deployment diagrams to provide additional implementation detail.
 - Step 7. Refactor every component-level design representation and always consider alternatives.

Designing Conventional Components



The design of processing logic is governed by the basic principles of algorithm design and structured programming

Represent the algorithm at a level of detail that can be reviewed for quality options:

- graphical (e.g. flowchart, box diagram)

- pseudocode (PDL or program design language or structured english)

- decision table

- use structured programming to implement procedural logic

- use ‘formal methods’ to prove logic

The design of data structures is defined by the data model developed for the system

The design of interfaces is governed by the collaborations that a component must effect

Component-Based Development

For exploring possibilities of reuse, the software team must check:

- Are commercial off-the-shelf (COTS) components available to implement the requirement?
- Are internally-developed reusable components available to implement the requirement?
- Are the interfaces for available components compatible within the architecture of the system to be built?

At the same time, they are faced with the several impediments to reuse ...

Impediments to Reuse

- Few companies and organizations have anything that even slightly resembles a comprehensive software reusability plan.
 - Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them.
 - Relatively little training is available to help software engineers and managers understand and apply reuse.
 - Many software practitioners continue to believe that reuse is “more trouble than it’s worth.”
 - Many companies continue to encourage of software development methodologies which do not facilitate reuse
 - Few companies provide an incentives to produce reusable program components.
-

Component-Based Software Engineering



- A library of components must be available
- Components should have a consistent structure
- A standard should exist, e.g.,
 - OMG/CORBA
 - Microsoft COM
 - Sun JavaBeans

CBSE Activities

Component-Based Software Engineering requires additional activities

- Component qualification
- Component adaptation
- Component composition
- Component update

Component Qualification

Before a component can be used, you must consider:

- application programming interface (API)
 - development and integration tools required by the component
 - run-time requirements including resource usage (e.g., memory or storage), timing or speed, and network protocol
 - service requirements including operating system interfaces and support from other components
 - security features including access controls and authentication protocol
 - embedded design assumptions including the use of specific numerical or non-numerical algorithms
 - exception handling
-

Component Adaptation

The implication of “easy integration” is:

- (1) that consistent methods of resource management have been implemented for all components in the library;
- (2) that common activities such as data management exist for all components, and
- (3) that interfaces within the architecture and with the external environment have been implemented in a consistent manner.

Adaptation techniques

White-box wrapping – integration conflicts removed by making code-level modifications to the code

Grey-box wrapping – used when component library provides a component extension language or API that allows conflicts to be removed or masked

Black-box wrapping – requires the introduction of pre- and post-processing at the component interface to remove or mask conflicts

Component Composition

An infrastructure must be established to bind components together

Architectural ingredients for composition include:

- Data exchange model
- Automation
- Structured storage
- Underlying object model

Component development for reuse

- Components for reuse may be specially constructed by generalising existing components.
 - Component reusability
 - Should reflect stable domain abstractions;
 - Should hide state representation;
 - Should be as independent as possible;
 - Should publish exceptions through the component interface.
 - There is a trade-off between reusability and usability
 - The more general the interface, the greater the reusability but it is then more complex and hence less usable.
-

Changes for reusability

- Remove application-specific methods.
- Change names to make them general.
- Add methods to broaden coverage.
- Make exception handling consistent.
- Add a configuration interface for component adaptation.
- Integrate required components to reduce dependencies.

Component validation

- Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests.
 - The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests.
- As well as testing that a component for reuse does what you require, you may also have to check that the component does not include any malicious code or functionality that you don't need.

Ariane launcher failure – validation failure?

- In 1996, the 1st test flight of the Ariane 5 rocket ended in disaster when the launcher went out of control 37 seconds after take off.
 - The problem was due to a reused component from a previous version of the launcher (the Inertial Navigation System) that failed because assumptions made when that component was developed did not hold for Ariane 5.
 - The functionality that failed in this component was **not required** in Ariane 5.
-

Web services

- A web service is an instance of a more general notion of a service:
“an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production”.
- The essence of a service, therefore, is that the provision of the service is independent of the application using the service.
- Service providers can develop specialized services and offer these to a range of service users from different organizations.

Service-oriented architectures

- A means of developing distributed systems where the components are stand-alone services
- Services may execute on different computers from different service providers
- Standard protocols have been developed to support service communication and information exchange

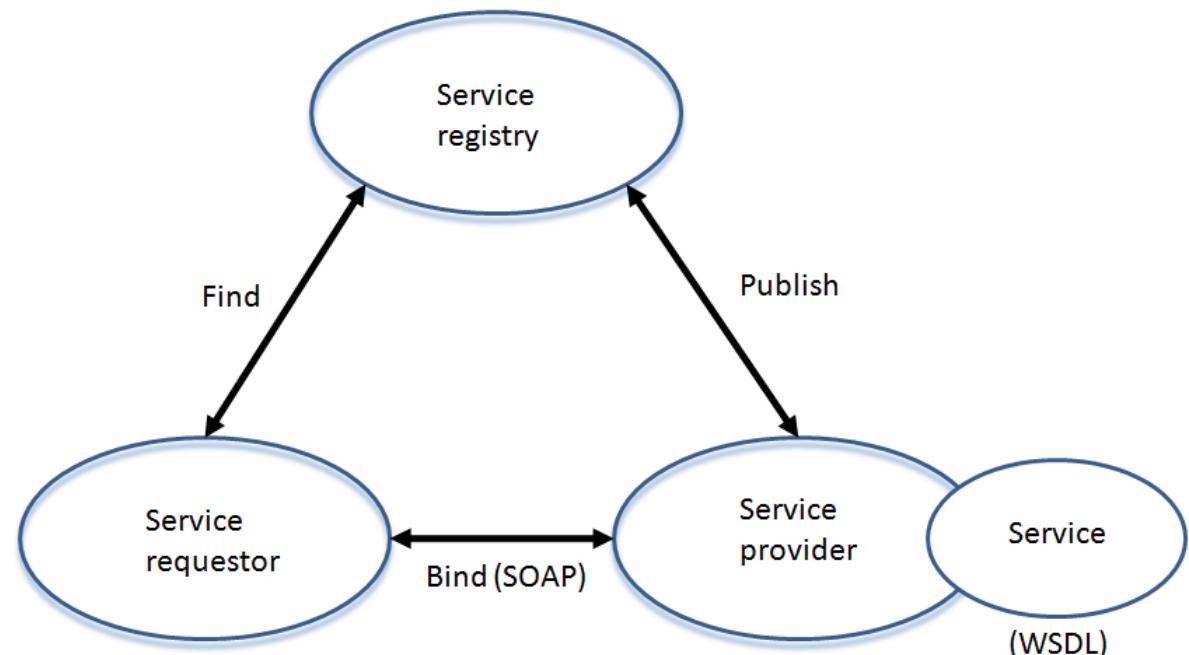
Service-oriented architecture

(Enabling protocols)



SOAP : Message interchange protocol that supports the communication between services. Defines essential and optional components of messages

WSDL: Web Services Definition Language is a standard for service interface definition. Sets out how service operations (operation names, parameters, and types) and service bindings should be defined



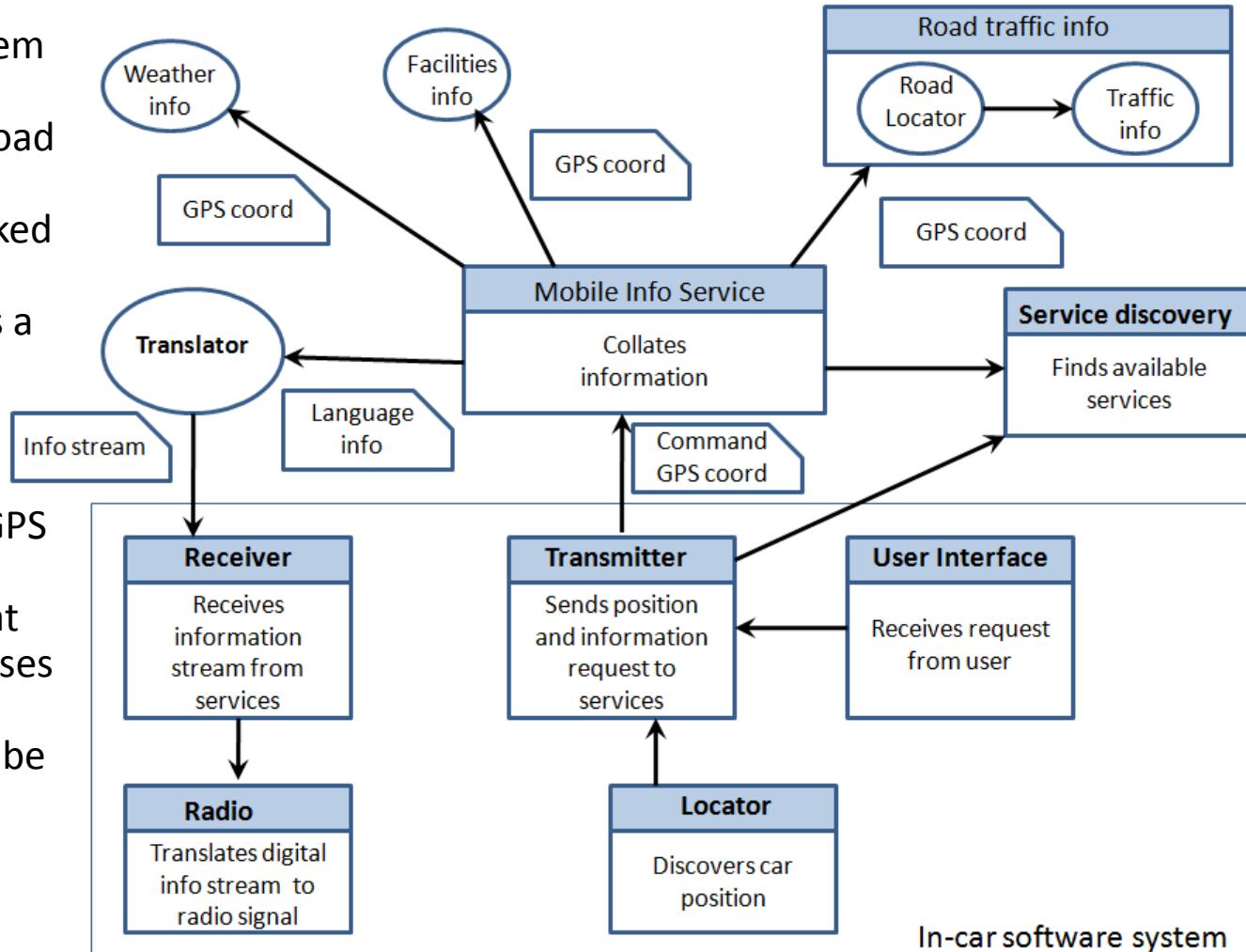
Benefits of SOA

- Services can be provided locally or outsourced to external providers
 - Services are language-independent
 - Investment in legacy systems can be preserved
 - Inter-organizational computing is facilitated through simplified information exchange
-

Example SOA

A service-based, in-car information system

- An in-car information system provides drivers with information on weather, road traffic conditions, local information etc. This is linked to car radio so that information is delivered as a signal on a specific radio channel.



- The car is equipped with GPS receiver to discover its position and, based on that position, the system accesses a range of information services. Information may be delivered in the driver's specified language.

To summarize

Component-level design defines data structures, algorithms, interface characteristics, and communication mechanisms for each component identified in the architectural design. Component-level design occurs after the data and architectural designs are established.

Component-level design may be represented using graphical, tabular, or text-based notation in a way that allows the designer to review it for correctness and consistency, before it is built.

Some of the components may be acquired from outside. In such a case lifecycle would be different.

SOA approach builds software solution that benefits from services which are published as per standard protocols. SOA protocols make components(services) platform-independent.



Thank You...

Credits

- Software Engineering 7/ed by Roger Pressman – Reference
- Software Engineering 9/ed by Ian Sommerville - Reference



BITS Pilani
Pilani Campus

Course Name : Software Engineering

T V Rao
User Interface Design



Where Do Design Guidelines Come From?

- The UI (User Interface) design rules are based on human psychology.
 - UI researchers study how people
 - perceive,
 - learn,
 - reason,
 - remember, and
 - convert intentions into action.
 - Many authors of design guidelines had at least some background in psychology that they applied to computer system design.
-

Interface Design

What do users expect?

- Easy to learn**
- Easy to use**
- Easy to understand**

What do they complain about?

- lack of consistency**
- too much memorization**
- no guidance / help**
- no context sensitivity**
- poor response**
- Arcane/unfriendly**



Golden Rules of UI Design

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent
 - Theo Mandel[97] , PhD in Cognitive Psychology
(Consultant, Author)

Place the User in Control



Define interaction modes in a way that does not force a user into unnecessary or undesired actions.

Provide for flexible interaction.

Allow user interaction to be interruptible and undoable.

Streamline interaction as skill levels advance and allow the interaction to be customized.

Hide technical internals from the casual user.

Design for direct interaction with objects that appear on the screen.

Reduce the User's Memory Load



Reduce demand on short-term memory.

Establish meaningful defaults.

Define shortcuts that are intuitive.

The visual layout of the interface should be based on a real world metaphor.

Disclose information in a progressive fashion.

Make the Interface Consistent

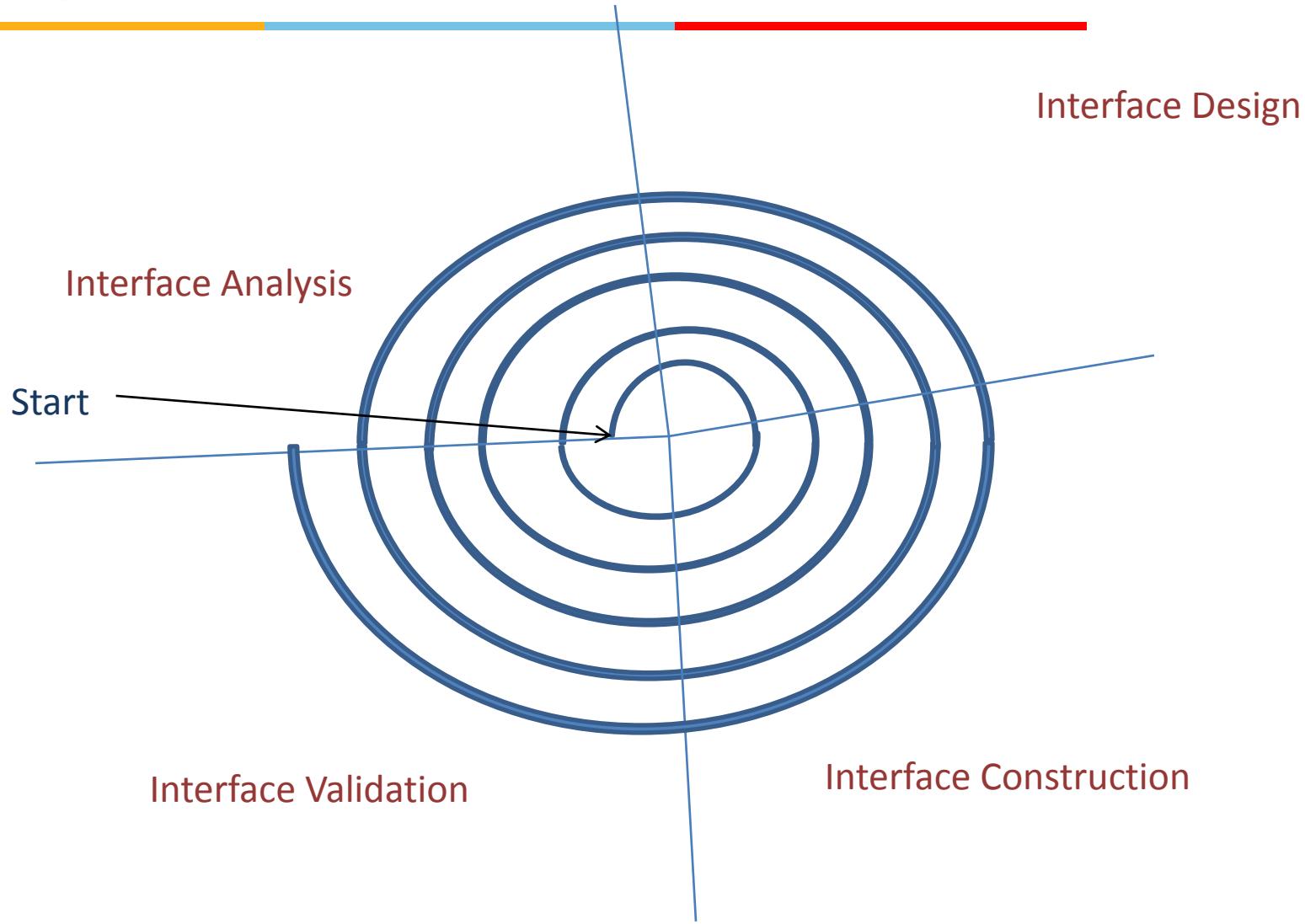


Allow the user to put the current task into a meaningful context.

Maintain consistency across a family of applications.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

User Interface Design Process



Interface Analysis

- Interface analysis means understanding
 - (1) the people (end-users) who will interact with the system through the interface;
 - (2) the tasks that end-users must perform to do their work,
 - (3) the content that is presented as part of the interface
 - (4) the environment in which these tasks will be conducted.
-

User Analysis

- Are users trained professionals, technician, clerical, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the age range of the user community?
- Will the users be represented predominately by one gender?
- How are users compensated for the work they perform?
- Do users work normal office hours or do they work until the job is done?
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology the sits behind the interface?

— Hackos & Redish [User & Task Analysis for Interface Design, 1998]

Task Analysis and Modeling

- Answers the following questions ...
 - What work will the user perform in specific circumstances?
 - What tasks and subtasks will be performed as the user does the work?
 - What specific problem domain objects will the user manipulate as work is performed?
 - What is the sequence of work tasks—the workflow?
 - What is the hierarchy of tasks?
 - Use-cases define basic interaction
 - Task elaboration refines interactive tasks
 - Object elaboration identifies interface objects (classes)
 - Workflow analysis defines how a work process is completed when several people (and roles) are involved
-

Analysis of Display Content

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
 - Can the user customize the screen location for content?
 - Is proper on-screen identification assigned to all content?
 - If a large report is to be presented, how should it be partitioned for ease of understanding?
 - Will mechanisms be available for moving directly to summary information for large collections of data.
 - Will graphical output be scaled to fit within the bounds of the display device that is used?
 - How will color be used to enhance understanding?
 - How will error messages and warning be presented to the user?
-

Interface Design Steps

- Using information developed during interface analysis, define interface objects and actions (operations).
 - Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
 - Depict each interface state as it will actually look to the end-user.
 - Indicate how the user interprets the state of the system from information provided through the interface.
-

Design Issues

- Response time
- Help facilities
- Error *handling*
- Menu and command labeling
- Application accessibility
- Internationalization

Web Application Interface Design



As per Dix[99], WebApp interface answer three primary questions for the user

- *Where am I? The interface should*
 - provide an indication of the WebApp that has been accessed
 - inform the user of her location in the content hierarchy.
- *What can I do now?* The interface should always help the user understand his current options
 - what functions are available?
 - what links are live?
 - what content is relevant?
- *Where have I been, where am I going?* The interface must facilitate navigation.
 - Provide a “map” (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

Tognozzi's WebApp Interface Design Principles

- **Anticipation**—A WebApp should be designed so that it anticipates the user's next move.
- **Communication**—The interface should communicate the status of any activity initiated by the user
- **Consistency**—The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout)
- **Controlled autonomy**—The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- **Efficiency**—The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.
- **Focus**—The WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- **Fitt's Law**—“The time to acquire a target is a function of the distance to and size of the target.”
- **Human interface objects**—A vast library of reusable human interface objects has been developed for WebApps.

Tognozzi's WebApp Interface Design Principles

- **Latency reduction**—The WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
 - **Learnability**— A WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.
 - **Maintain work product integrity**—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
 - **Readability**—All information presented through the interface should be readable by young and old.
 - **Track state**—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
 - **Visible navigation**—A well-designed WebApp interface provides “the illusion that users are in the same place, with the work brought to them.”
-

Aesthetic Design

- Don't be afraid of white space.
 - Emphasize content.
 - Organize layout elements from top-left to bottom right.
 - Group navigation, content, and function geographically within the page.
 - Don't extend your real estate with the scrolling bar.
 - Consider resolution and browser window size when designing layout.
-

User Interface (UI) Patterns



GUIs have become very important elements of software development over last 3 decades. Several popular patterns emerged w.r.t. the following categories :

- Whole UI. Provide design guidance for top-level structure and navigation throughout the entire interface.
- Page layout. Address the general organization of pages (for Websites) or distinct screen displays (for interactive applications)
- Forms and input. Consider a variety of design techniques for completing form-level input.
- Tables. Provide design guidance for creating and manipulating tabular data of all kinds.
- Direct data manipulation. Address data editing, modification, and transformation.
- Navigation. Assist the user in navigating through hierarchical menus, Web pages, and interactive display screens.
- Searching. Enable content-specific searches through information maintained within a Web site or contained by persistent data stores that are accessible via an interactive application.
- Page elements. Implement specific elements of a Web page or display screen.
- E-commerce. Specific to Web sites, these patterns implement recurring elements of ecommerce applications.

To Summarize

User interface design creates an effective communication between a human and a computer.

Proper interface design begins with careful analysis of the user, the task and the environment.

Based on user's tasks, user scenarios are created and validated.

Good user interfaces are designed, they don't happen by chance.

Prototyping is a common approach to user interface design.

Early involvement of the user in the design process makes him or her more likely to accept the final product.

User interfaces must be field tested and validated prior to general release



Thank You...

Credits

- Software Engineering 7/ed by Roger Pressman
 - Reference
- Software Engineering 9/ed by Ian Sommerville
 - Reference



BITS Pilani

Pilani Campus

Course Name : Software Engineering

T V Rao
Software Quality & Testing Overview

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

- 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

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

“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

Critical success factors:

Software Engineering Methods

Project Management Techniques

Quality Control

Quality Assurance

Achieving Software Quality

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

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**



Software Testing Overview

Software Testing

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

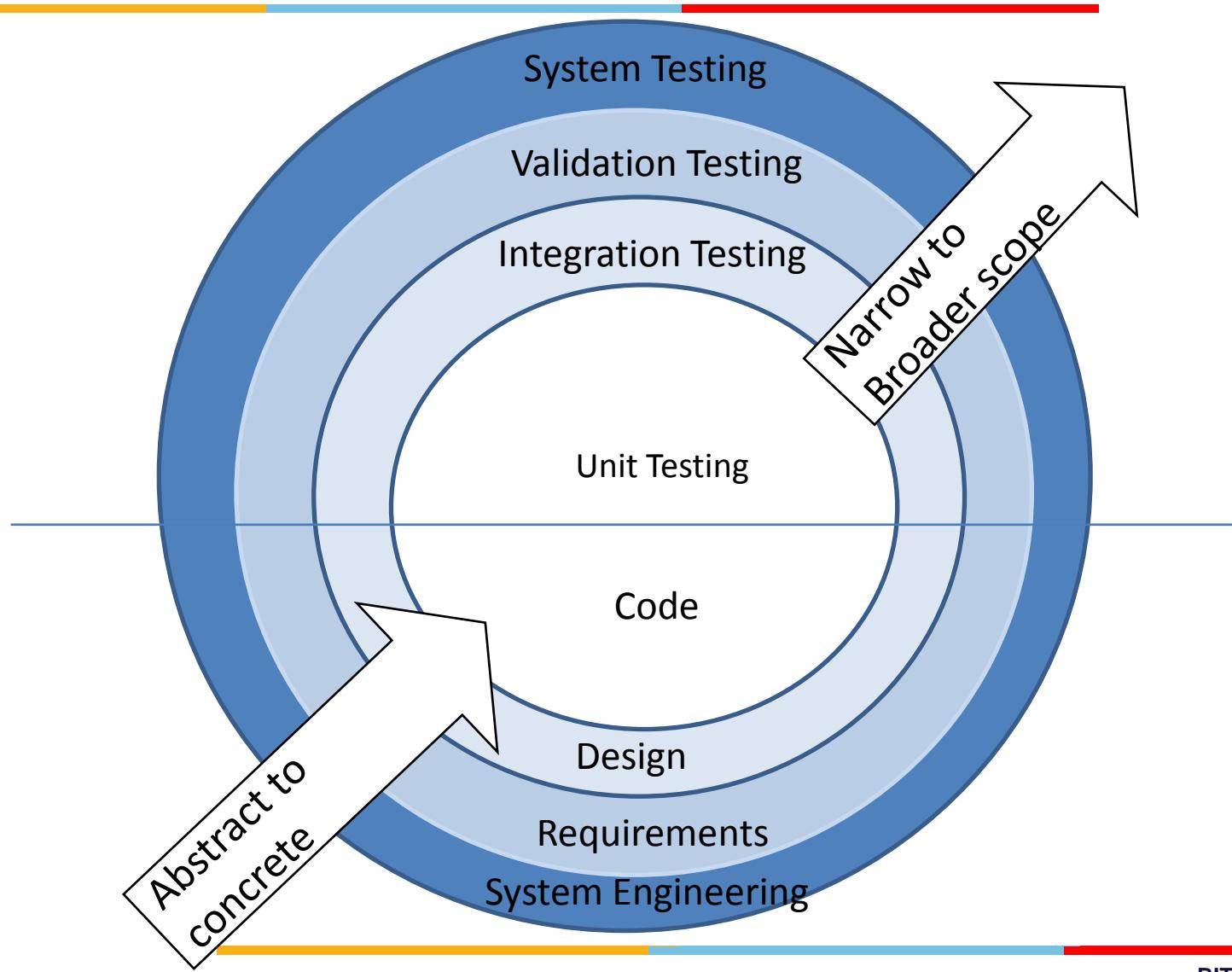
- Testing should aim at "breaking" the software
- Some misconceptions
 - The developer of software should do no testing at all
 - The software should be given to a secret team of testers who will test it unmercifully
 - The testers get involved with the project only when the testing steps are about to 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



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





Module Name : 8.1 OO Testing

BITS Pilani
Pilani Campus

S Subramanian
Work-Integrated Learning Programme

Chapter 19

- OO Testing

A Broader View of OO Testing

- Nature of OO systems influence both testing strategy and methods
- In Object Oriented systems the view of testing is broadened to encompass Analysis and Design
- *“It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level.”*
- Allows early circumvention of later problems

Consistency of OOA and OOD Models

- Consider the relationships among entities in the model.
- Examine each class and its connections to other classes.
- Revisit the CRC model and the object-relationship model.
- Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition

Some Challenges in OO Testing

- Encapsulation
- Polymorphism
- Inheritance

OO Testing Strategies

- Unit testing
 - the concept of the unit changes
 - the smallest testable unit is the encapsulated class
 - a single operation can no longer be tested in isolation (the conventional view of unit testing) but rather, as part of a class

OO Testing - Integration

- *Thread-based testing* integrates the set of classes required to respond to one input or event for the system
- *Use-based testing* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) of server classes. After the independent classes are tested, the next layer of classes, called *dependent classes*
- *Cluster testing* defines a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

OO – testing - Validation

- Validation Testing
 - Conventional Blackbox testing
 - Use object-behavior model
 - Use Analysis model

TEST CASE DESIGN FOR OO SOFTWARE

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.
2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain:
 - A list of specified states for the object that is to be tested.
 - A list of messages and operations that will be exercised as a consequence of the test
 - A list of exceptions that may occur as the object is tested.
 - A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test).
 - Supplementary information that will aid in understanding or implementing the test.

OBJECT ORIENTED TESTING TECHNIQUES

- Class Testing
- The white-box testing method can be applied to the operations defined for a class.
- Fault-Based Testing
- Scenario – Based Testing:

Partition Class Testing

- Reduces the number of test cases required (similar to equivalence partitioning)
- State-based partitioning
 - Categorize and test methods separately based on their ability to change the state of a class
- Attribute-based partitioning
 - Categorize and test operations based on the attributes that they use
- Category-based partitioning
 - Categorize and test operations based on the generic function each performs

OO testing Technique – Inter class Testing

- Class collaboration testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioural testing

Behavior Model Testing

- Test cases must cover all states in the state transition diagram
- Breadth first traversal of the state model can be used (test one transition at a time and only make use of previously tested transitions when testing a new transition)
- Test cases can also be derived to ensure that all behaviors for the class have been adequately exercised

Credits

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

Thank You



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





BITS Pilani
Pilani Campus

Module Name :M8.2 Web Testing

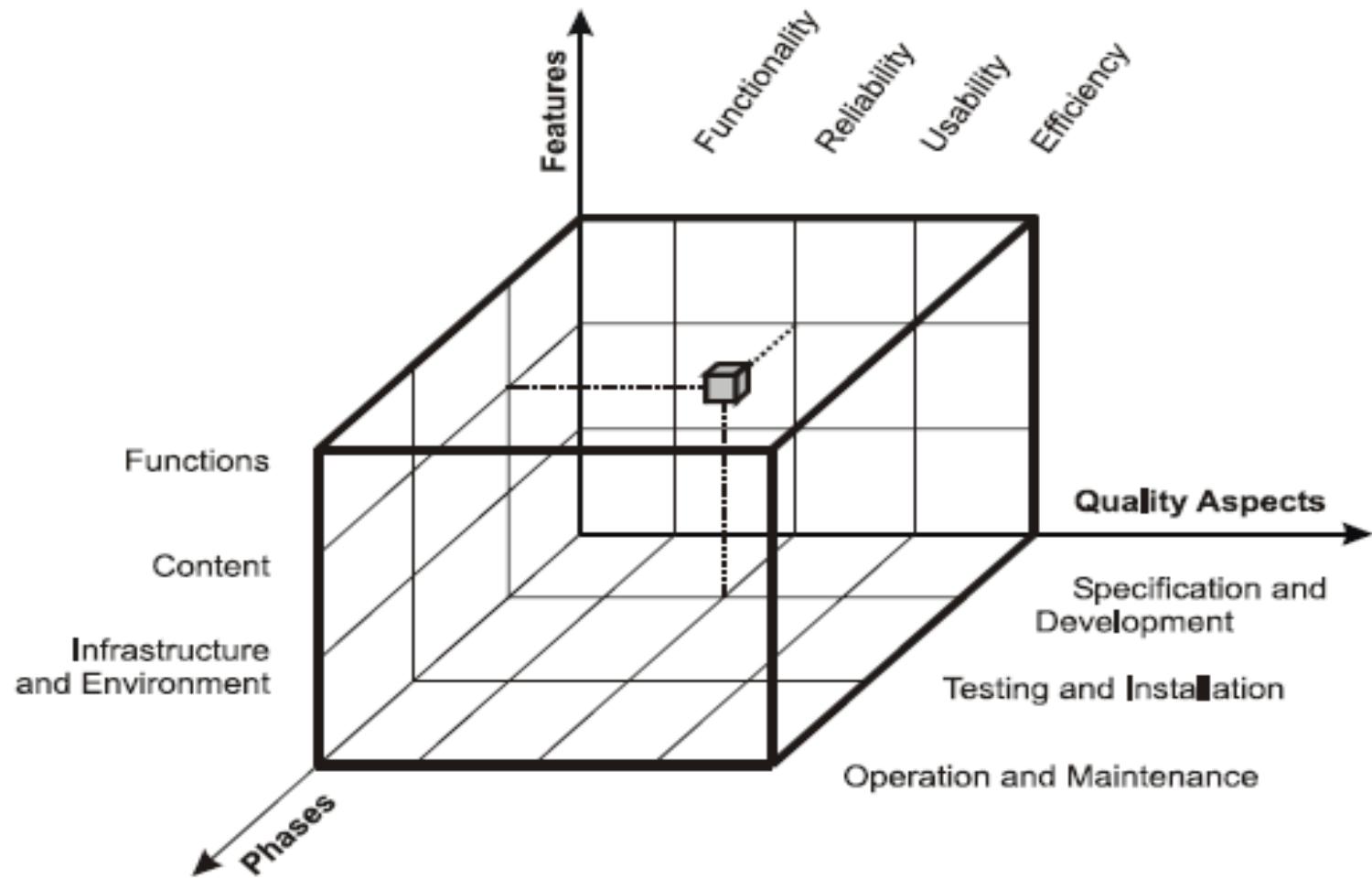
S Subramanian
Work-Integrated Learning Programme



Chapter 20

- **Testing Web applications.**

Quality Cube



Dimensions of Quality [1/2]

- Content
 - Syntactic and semantic
- Function
 - Conformance to requirements
- Structure
 - Extensible
- Usability
 - User interface
- Navigability
 - Links to other pages

Dimensions of Quality [2/2]

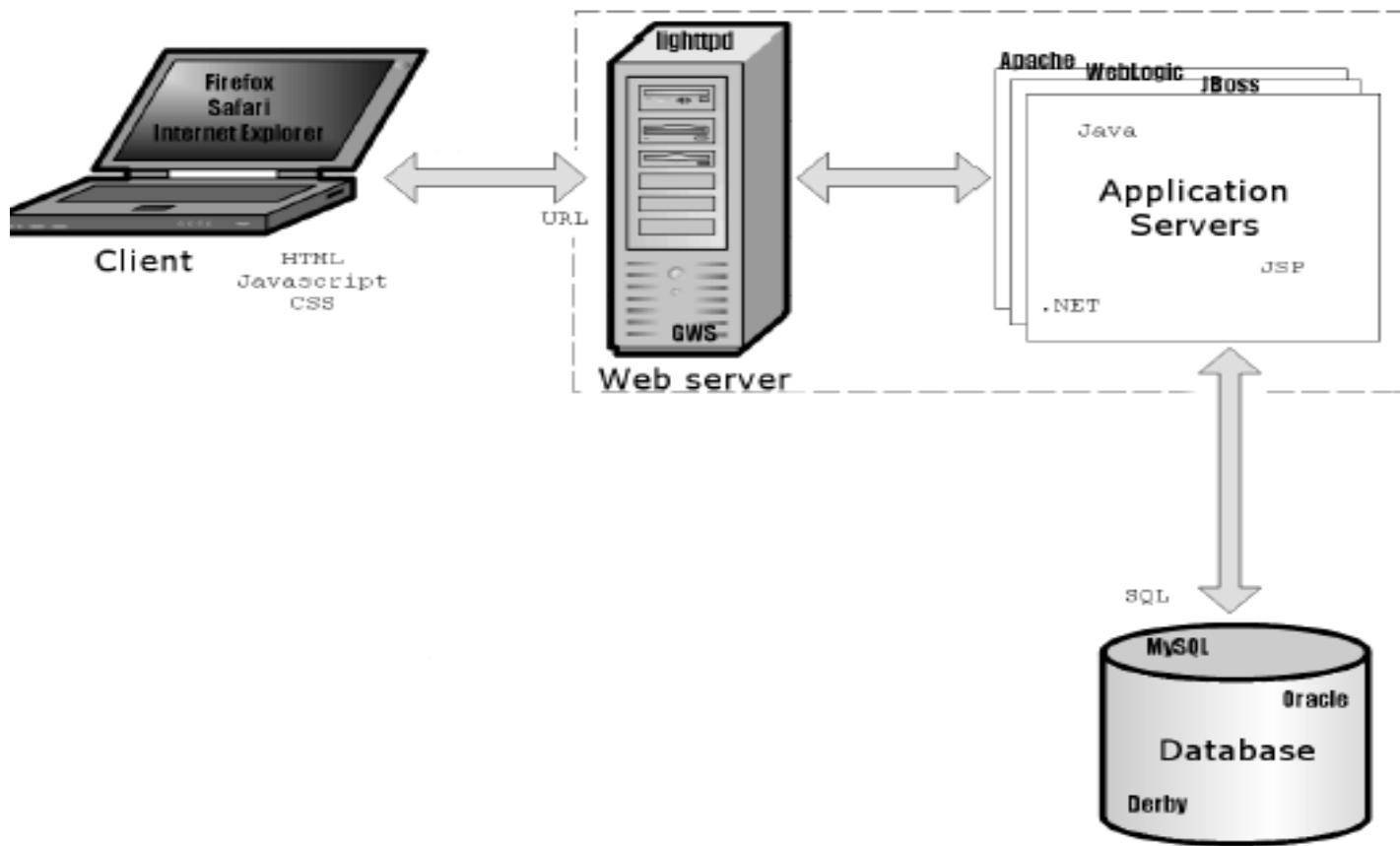
- Performance
 - Number of users
- Compatibility
 - Different configurations
- Interoperability
 - Other applications
- Security
 - Potential vulnerabilities

Challenges for Testing Web-based Applications



- High Rate of Change
- Dynamic Content Generation
- Persistent State and Concurrency
- Because WebApps reside within a client/server architecture, errors can be difficult to trace across three architectural layers: the client, the server, or the network itself.

Typical Webapp architecture



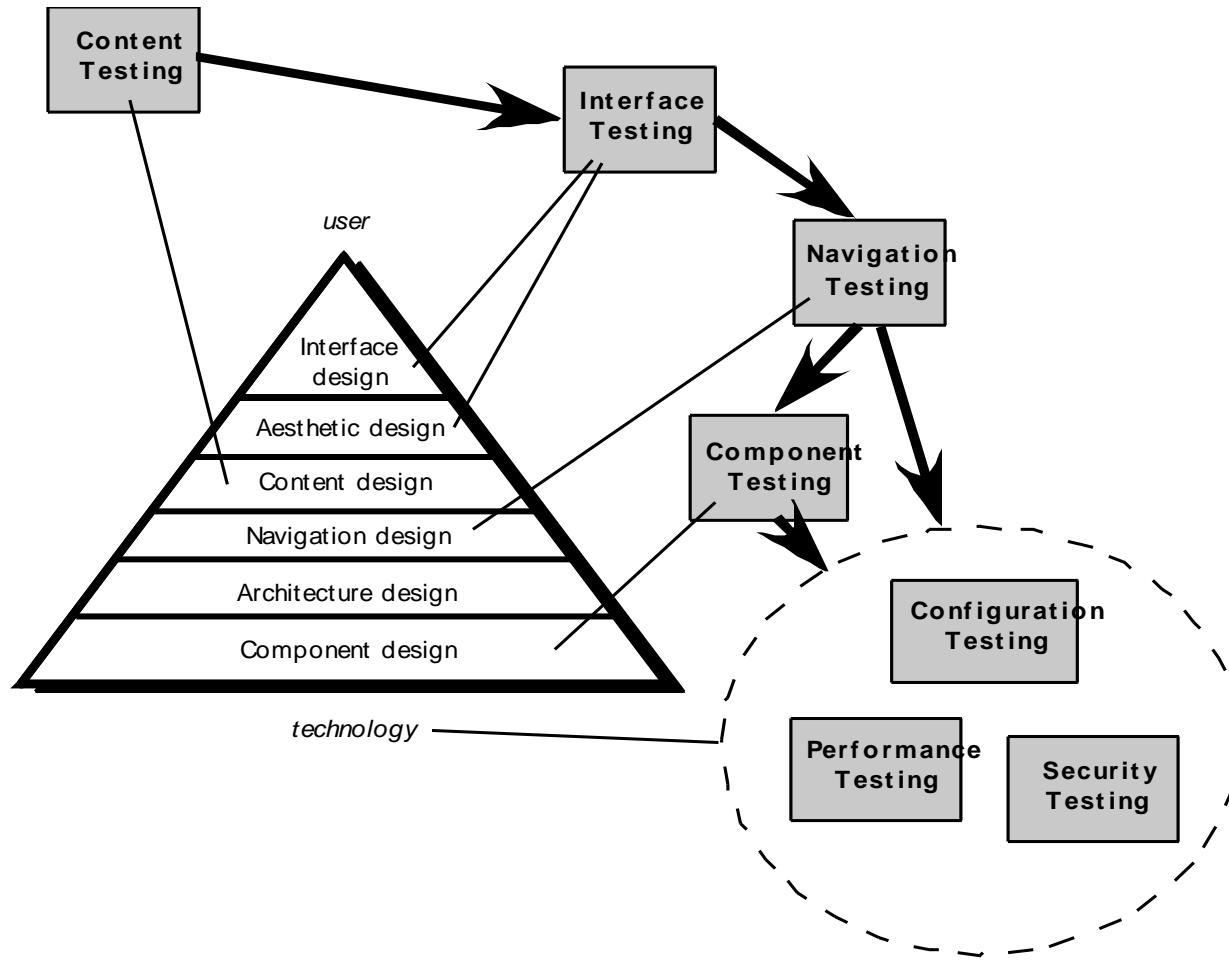
WebApp Testing Strategy - I

- Web application testing - Functional & Non functional.
 - The interface model is reviewed to ensure that all use-cases can be accommodated.
 - The design model for the WebApp is reviewed to uncover navigation errors.
 - The user interface is tested to uncover errors in presentation and/or navigation mechanics.
 - Selected functional components are unit tested.

WebApp Testing Strategy-II

- 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.

The Testing Process



Functional and Usability Issues

- The first tests for a website should focus on the site's intended behavior by assessing the following issues:
 - Functionality
 - Usability
 - Navigation
 - Forms
 - Page content

Why is Functionality Testing Important?

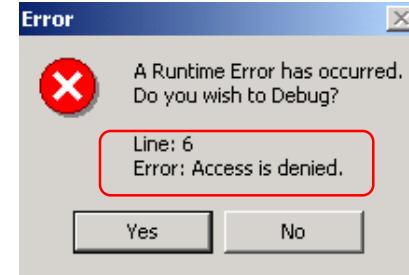


The page cannot be displayed

The page you are looking for is currently unavailable. The Web site might be experiencing technical difficulties, or you may need to adjust your browser settings.

Please try the following:

- Click the [Refresh](#) button, or try again later.
- If you typed the page address in the Address bar, make sure that it is spelled correctly.
- To check your connection settings, click the **Tools** menu, and then click **Internet Options**. On the **Connections** tab, click **Settings**. The settings should match those provided by your local area network (LAN) administrator or Internet service provider (ISP).
- If your Network Administrator has enabled it, Microsoft Windows can examine your network and automatically discover network connection settings.
If you would like Windows to try and discover them, click [Detect Network Settings](#)



Graphical User Interface Testing

- Interface features are tested to ensure that design rules, aesthetics, and related visual content is available for the user without error.
- Each interface mechanism is tested within the context of a use-case or NSU for a specific user category.
- The interface is tested within a variety of environments (e.g., browsers) to ensure that it will be compatible

Interface mechanisms - I

When a user interacts with a webapp, the interaction occurs through one or more mechanisms.

- **Links**
- **Forms**
- **Client-side scripting**
- **Dynamic HTML**
- **Pop-up windows:**

Interface mechanisms - II

- **Audio/Video Streaming content**
 - Push vs. pull
- **Cookies**
 - On the server side,
 - On the client side
- **Web 2.0 access**

Navigation mechanisms

- **Navigation**
 - Internal, External links and anchors within a specific web page should be tested to ensure that proper content or functionality is reached when the link is chosen.
- **Bookmarks**
 - Even it is a browser function, the webapp should be tested to ensure that a meaningful page title can be extracted as the bookmark is created.
- **Redirects**
 - Redirects should be tested by requesting incorrect internal links or external URLs and assessing how the webapp handles these requests.
- **Internal search engines**
- **Site Maps**

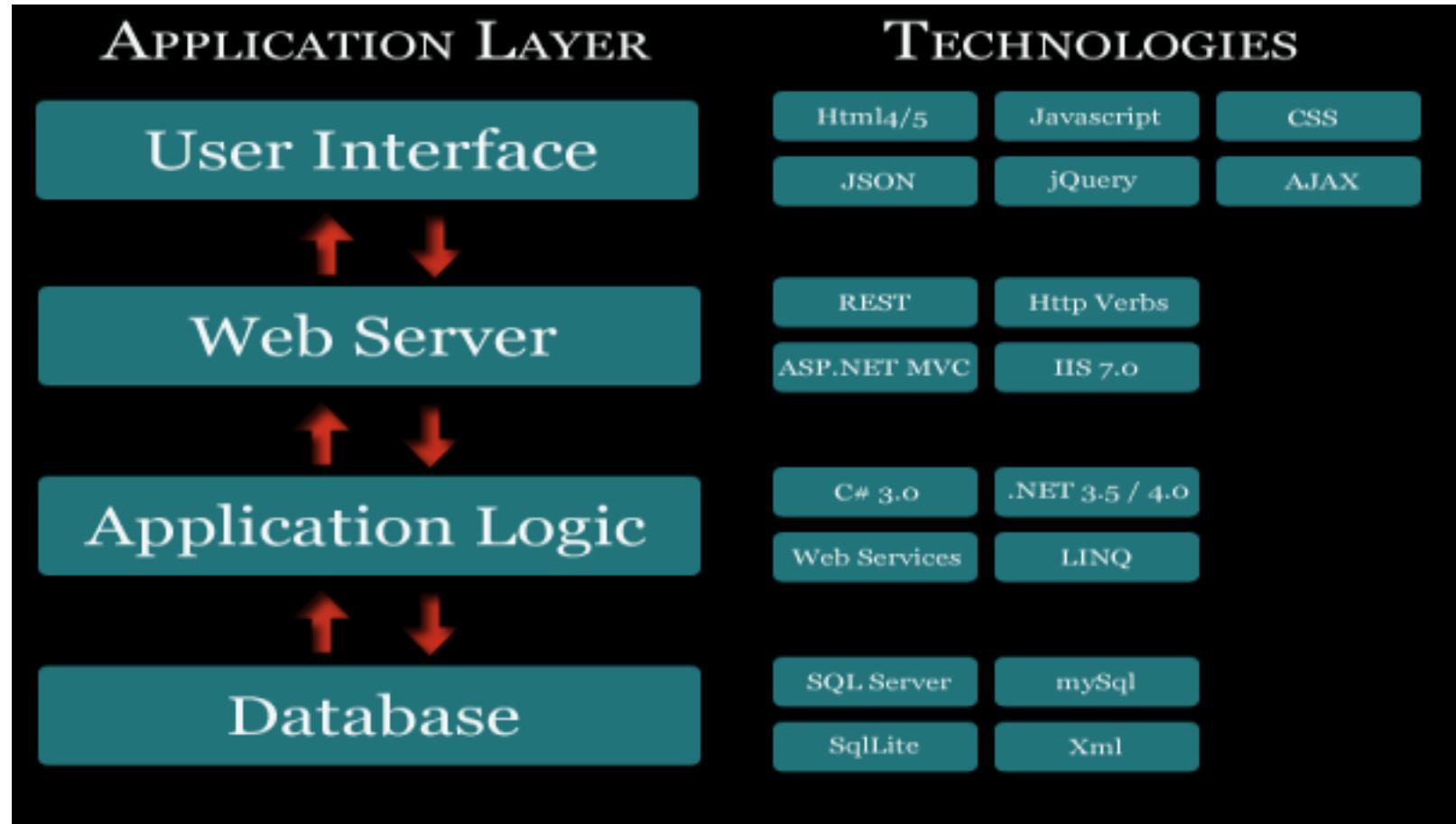
Configuration Testing - I

- Server-side
 - Is the WebApp fully compatible with the server OS?
 - Are system files, directories, and related system data created correctly when the WebApp is operational?
 - Do system security measures (e.g., firewalls or encryption) allow the WebApp to execute and service users without interference or performance degradation?
 - Has the WebApp been tested with the distributed server configuration (if one exists) that has been chosen?
 - Is the WebApp properly integrated with database software? Is the WebApp sensitive to different versions of database software?
 - If proxy servers are used, have differences in their configuration been addressed with on-site testing?

Configuration Testing - II

- Client-side
 - *Hardware*—CPU, memory, storage and printing devices
 - *Operating systems*—Linux, Macintosh OS, Microsoft Windows, a mobile-based OS
 - *Browser software*—Internet Explorer, Mozilla/Netscape, Opera, Safari, and others
 - *User interface components*—Active X, Java applets and others
 - *Plug-ins*—QuickTime, RealPlayer, and many others
 - *Connectivity*—cable, DSL, regular modem, T1
- The number of configuration variables must be reduced to a manageable number

Data & Transaction flow testing



Performance testing

- Performance testing ensures that the website server responds to browser requests within defined parameters. As part of performance testing:
 - Scalability testing assesses the website's ability to meet the load requirements.
 - Load testing evaluates how the system functions when processing many simultaneous requests from a multitude of users.
 - To collect metrics that will lead to design modifications to improve performance

Load Vs Stress Testing

load testing: Measuring system's behavior and performance when subjected to particular heavy tasks.

- Often tested at ~1.5x the expected SWL (safe working load)
- Testing a web app by having several users connect at once

stress testing: An extreme load test where the system is subjected to much higher than usual load.

- see how gracefully the system handles unreasonable conditions
- discover any data loss or corruption problems when under load
- test recoverability and recovery time



Security Testing - I

To protect against these vulnerabilities, one or more of the following security elements is implemented

- Firewall
- Authentication.
- Encryption
- Authorization
- Network vulnerabilities:

Security Testing - II

-
- **Web Application Vulnerability**
 - Other than virus attacks and network vulnerabilities
 - Weakness in custom Web Application, architecture, design, configuration, or code.

OWASP Top Ten Application Vulnerabilities

- A1. Cross-Site Scripting (XSS)
- A2. Injections Flaws
- A3. Malicious File Execution
- A4. Insecure Direct Object Reference
- A5. Cross Site Request Forgery (CSRF)
- A6. Information Leakage & Improper Error Handling
- A7. Broken Authentication & Session Management
- A8. Insecure Cryptographic Storage
- A9. Insecure Communications
- A10. Failure to Restrict URL Access

Test Automation

- Automated testing makes use of programs or tools to aid in test planning, execution and analysis.
- Automation can be done either fully or partially, unattended with occasional human intervention
- **Speed** –As automated tests are run by tools, these are run much faster than human users which adds to the first benefit of saving time.
- **Repeatability** – the same tests can be re-run in exactly the same manner eliminating the risk of human errors

Credits

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

Thank You



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





BITS Pilani
Pilani Campus

Module Name : M9.1 product Metrics

S Subramanian
Work-Integrated Learning Programme

Chapter 23

- **Software Metrics**



+

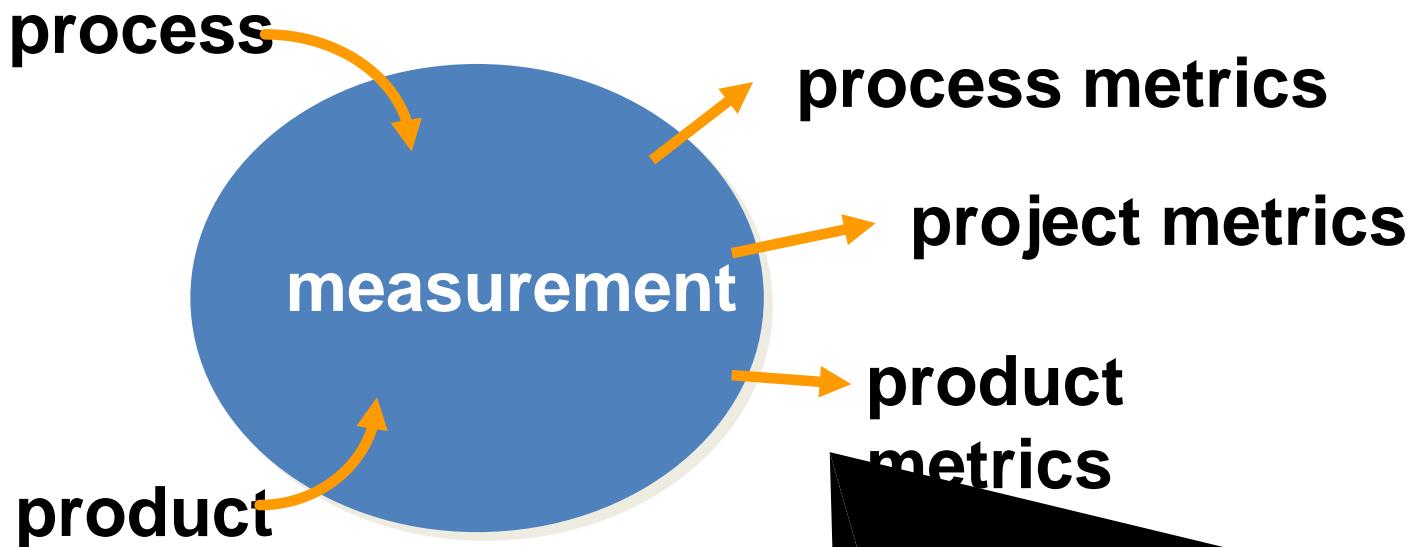


measures

McCall's Triangle of Quality



A Good Manager Measures

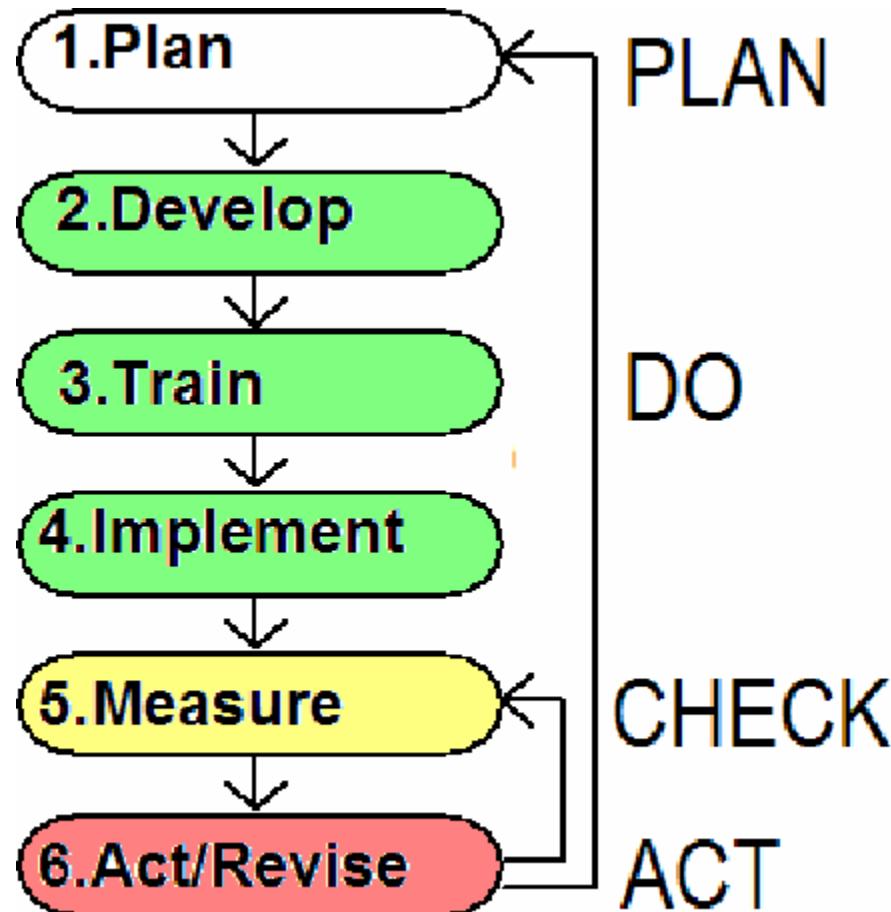


“Not everything that can be counted counts, and not everything that counts can be counted.” - Einstein

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			Total
	Low	Average	High	
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, \dots, 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 $V(g)$

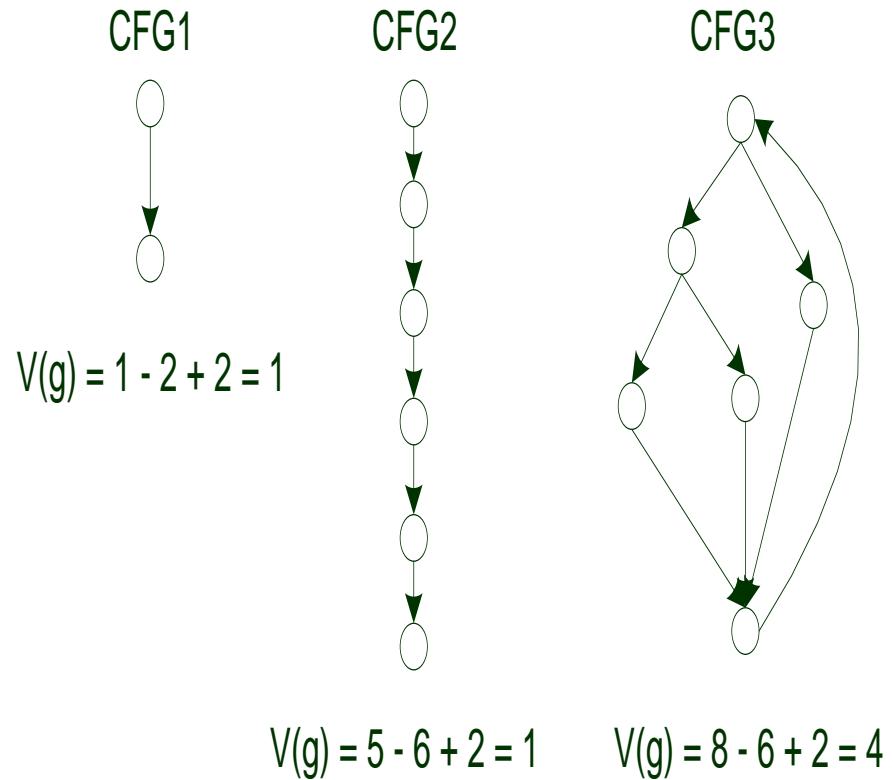
=

number of *independent paths*
through the control flow
graph

=

$$e - n + 2$$

$$(edges - \underline{nodes} + 2)$$



Credits

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

Thank You



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





Module Name : M9.2 Estimation

BITS Pilani
Pilani Campus

S Subramanian
Work-Integrated Learning Programme

Chapter 26

- **Software Planning & Estimation**

Project Planning

- SW project management process begins with *project planning*
- Objective of SW project planning - to provide a framework for manager to make reasonable *estimates of resources, costs and schedules*

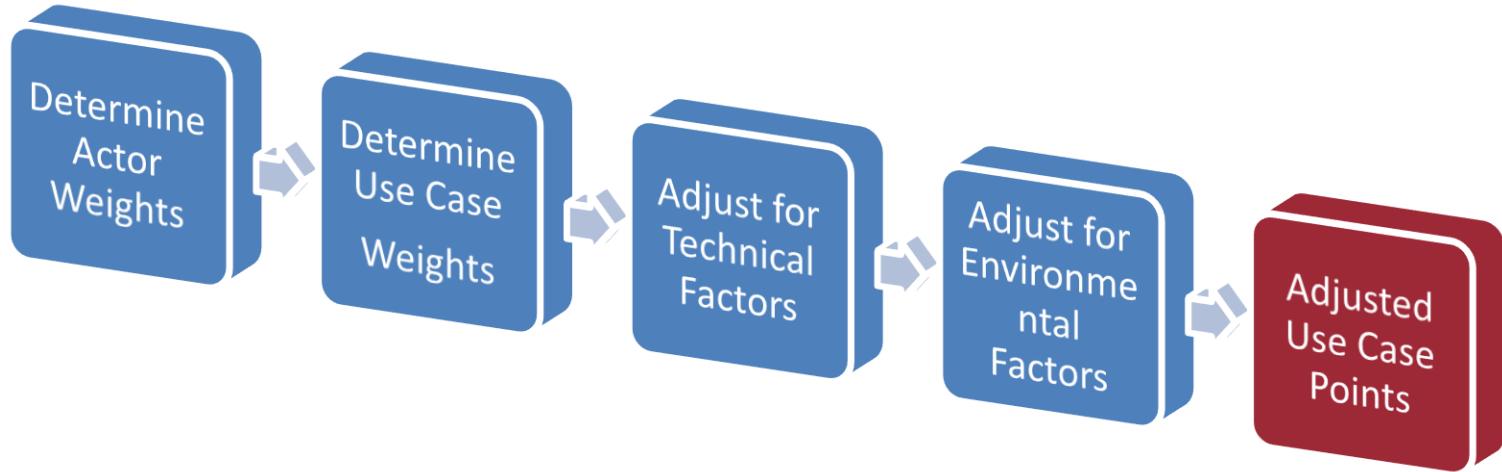
3 Common (subjective) estimation models

- Expert Judgment
- Analogy
- Price to win

Estimation

- History is your best ally
 - Especially when using LOC, function points, UCP, PERT (three point) etc.
- Use multiple methods if possible
 - This reduces your risk
 - If using “experts”, use two
- Get buy-in
- Remember: it's an iterative process!

Use Case Point Estimation – Process - I



Use Case Point Estimation – Process - II (Actor)

Actor Type	Description	Weight
Simple	The actor represents another system with a defined Application Programming Interface (API)	1
Average	The actor represents another system interacting through a protocol-driver Interface	2
Complex	The actor is a person interacting via a Graphical User Interface (GUI)	3

Unadjusted Actor Weight

- Actor initiates an interaction with the system to accomplish some goals.
- Does not represent the *physical* people or systems, but their *role*.

Use Case Point Estimation – Process - III (UC)

Use Case Type	Description	Weight
Simple	Less than 3 transactions	5
Average	4 to 7 transactions	10
Complex	More than 7 transactions	15

Unadjusted Use Case Weight

- Based on the total number of activities contained in all the use case scenarios

UUCP is the sum of Unadjusted Actor Weight (UAW) and Unadjusted Use Case Weight (UUCW).

$$\text{UUCP} = \text{UAW} + \text{UUCW}$$

Use Case Point Estimation – Process - IV (TCF)



Technical Factor	Description	Weight
T1	Distributed System	2
T2	Performance	1
T3	End User Efficiency	1
T4	Complex Internal Processing	1
T5	Reusability	1
T6	Installability	0.5
T7	Usability	0.5
T8	Portability	2
T9	Modifiability	1
T10	Concurrency	1
T11	Includes special security requirements	1
T12	Provides direct access by third parties	1
T13	Special User training facilities are required	1

Technical Complexity Factor - TCF

10

Use Case Point Estimation – Process - IV

- Each factor is given a perceived complexity value from 0 to 5 according to its impact
- Technical Total Factor is computed as the sum of all the weights multiplied by their corresponding perceived values
- Technical Complexity Factor (TCF):

$$\text{TCF} = 0.6 + (0.01 * \text{Technical Total Factor})$$

Use Case Point Estimation – Process - VI

Environment Factor	Description	Weight
F1	Familiarity with Life-Cycle model used	1.5
F2	Application domain experience	0.5
F3	Experience with development methodologies used	1
F4	Analyst capability	0.5
F5	Team motivation	1
F6	Stability of requirements	2
F7	Use of part-time team members	-1
F8	Use of difficult programming language	-1

Environment Complexity Factor - ECF

- Reflects the development team's experience

$$\text{ECF} = 1.4 + (-0.03 * \text{Environment Total Factor})$$

- Environment Total Factor is the sum of each factor weight multiplied by the perceived value

Productivity Factor (PF)

- Convert the UCP number into meaningful value in terms of man-hours required.

$$\mathbf{UCP = UUCP * TCF * ECF}$$

- The value may range from 16 to 30 man-hour/UCP. (Productivity Factor – PF)

$$\mathbf{Effort = UCP * PF}$$

Basic Model Schedule Equation



- MTDEV (Minimum time to develop) = $2.5 * (\text{Effort})^{\text{exponent}}$
- 2.5 is constant for all modes
- Exponent based on mode
 - organic = 0.38
 - semi = 0.35
 - embedded = 0.32
- Note that MTDEV does not depend on number of people assigned. Effort: is in Person months

Effort : in person
months

Organic Mode

Developed in familiar, stable environment Product similar to previously developed product <50,000 DSIs (ex: accounting system)

Semidetached Mode

somewhere between Organic and Embedded

Embedded Mode

new product requiring a great deal of innovation inflexible constraints and interface requirements (ex: real-time systems)

Agile Estimation - overview

- Rank Stories 1 to 5
- Rank for each story for Complexity 1 to 5
- Using these two vectors, effort of a particular User Story $ES = Complexity \times Size$
- Effort for the complete project. $E = \sum_i(ES) \ i = 1 \text{to } n$
- Determining Agile Velocity

Credits

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

Thank You



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





Module Name : M9.3 Project Scheduling

BITS Pilani
Pilani Campus

S Subramanian
Work-Integrated Learning Programme

Chapter 27

- Project Scheduling

Why Software Is Delivered Late?

- An unrealistic deadline
- Changing but unpredicted customer requirements
- Underestimation of efforts needed
- Risks not considered at the project start
- Unforeseen technical difficulties
- Unforeseen human difficulties
- Miscommunication among project staff
- Failure to recognize that project is falling behind schedule

Basic Principles for SE Scheduling

- **Work Breakdown Structure** – define distinct tasks
- **Linking of WBS tasks** - parallel and sequential tasks
- **Time allocation** – based of the effort estimation - assigned person days, start time, ending time
- **Effort validation** - be sure right resources are available on time
- **Defined responsibilities** — people must be assigned
- **Defined Outcomes**- identify all major and minor deliverables.

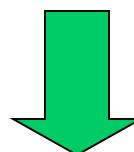
People and Effort



“If we fall behind schedule we can always add more programmers and catch up late in the project”

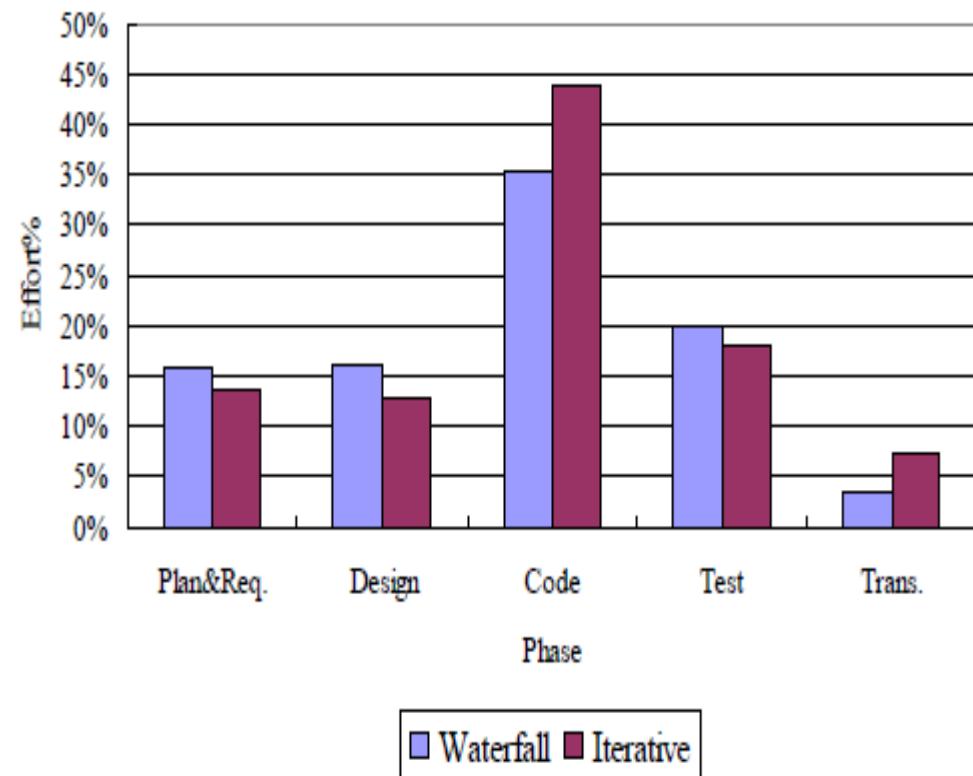


Has a disruptive effect on the project



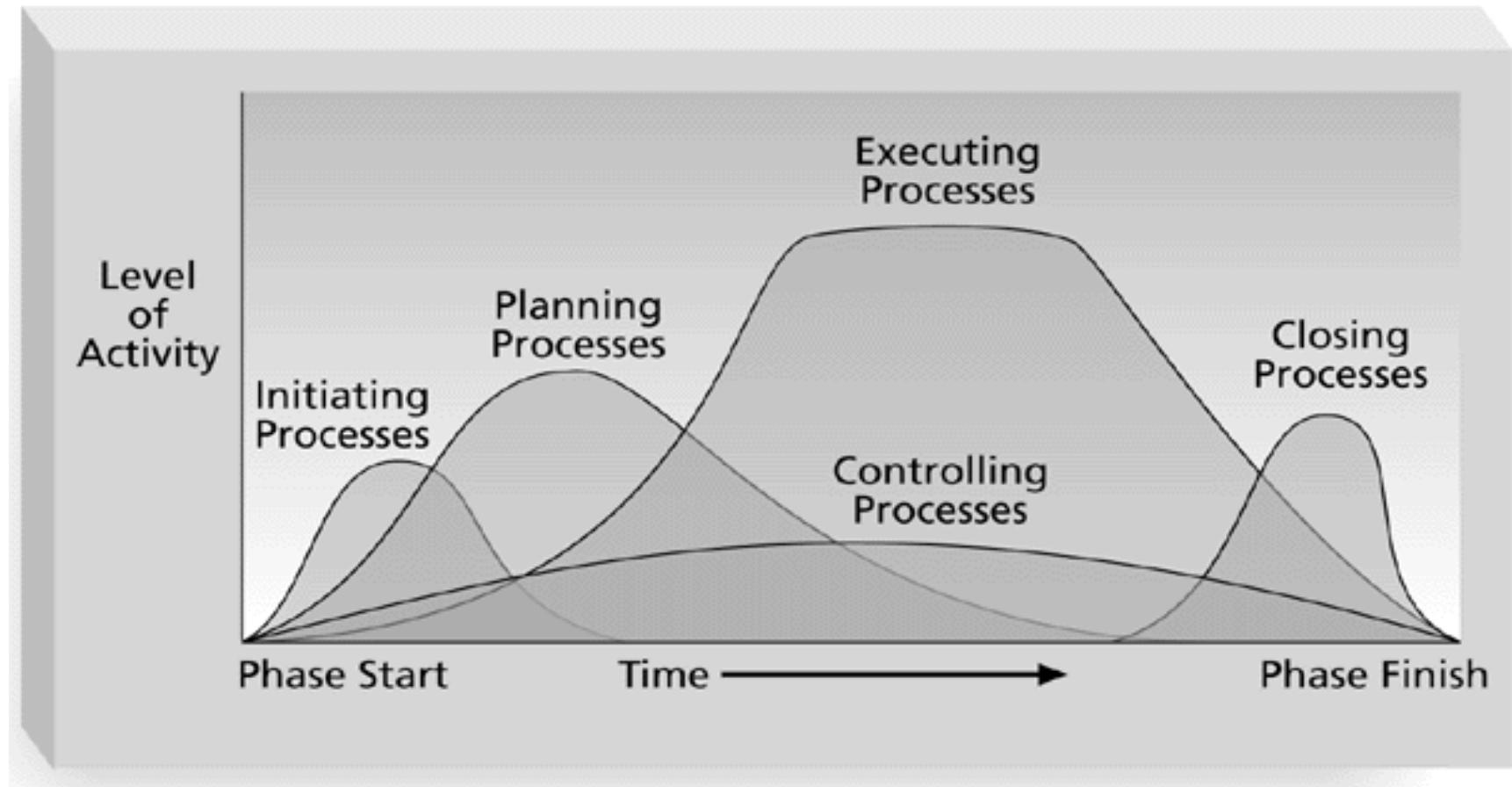
Schedules slip even further

Effort distribution in SDLC



Phase	Activities Included
Plan	Plan, Preliminary Requirement Analysis
Requirement	Requirement Analysis
Design	Product Design, Detailed Design
Code	Code, Unit Test, Integration
Test	System Test
Transition	Installation, Transition, Acceptance Test, User Training, Support

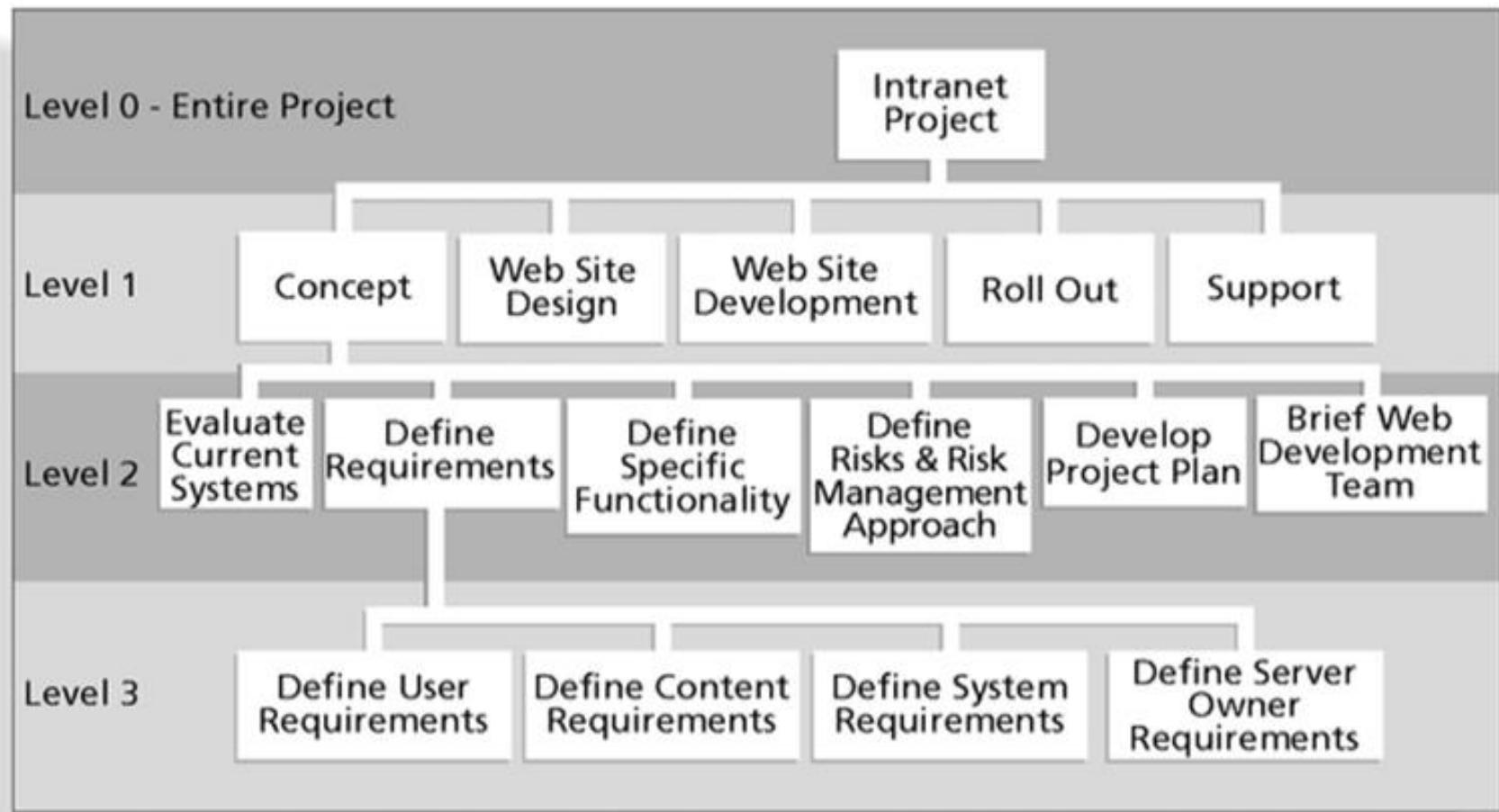
Application Lifecycle Management



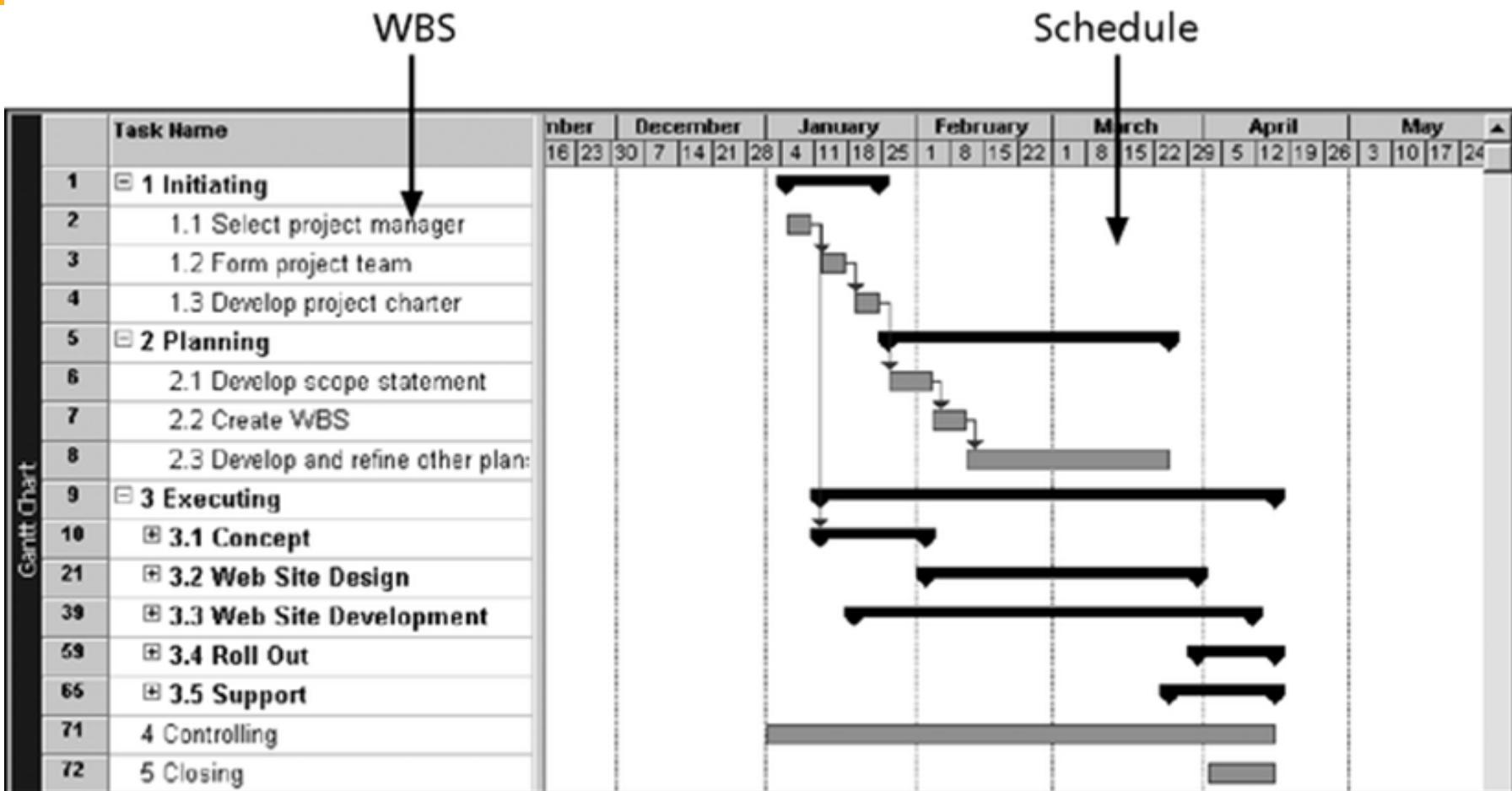
Project Planning

- The main purpose of project planning is to *guide execution*
- Every knowledge area includes planning information
- Key outputs include:
 - A team contract
 - A scope statement (project charter)
 - A work breakdown structure (WBS)
 - A project schedule, in the form of a Gantt chart with all dependencies and resources entered.
 - Development & testing environment & standards
 - A list of prioritized risks and mitigation strategy.
 - A deployment plan
 - A Software Quality Assurance Plan

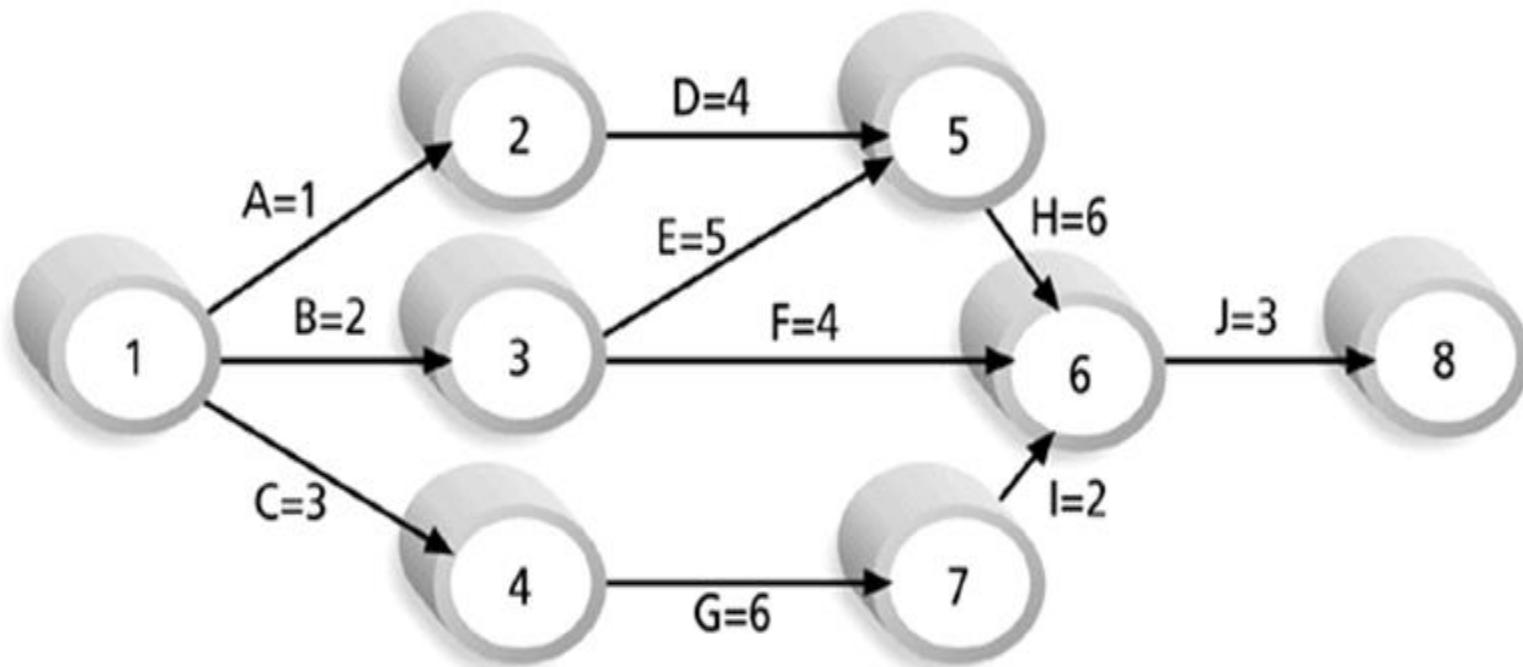
Sample Intranet Organized by Phase



Sample Gantt Chart



Critical path



Note: Assume all durations are in days.

Apply Earned Value Analysis

- Earned Value Analysis (EVA)
 - Earned Value Analysis is an objective method to measure project performance in terms of scope, time and cost
 - EVA metrics are used to measure project health and project performance

Earned Value Characteristics

- Point in Time Evaluation
- How much work did you PLAN to complete?
(Planned Value)
- How much work did you ACTUALLY complete?
(Earned Value)
- How much did you spend to complete the
work? (Actual Cost)

EVA Example

A \$10,000 software project is scheduled for 4 weeks. At the end of the third week, the project is 50% complete and the actual costs to date is \$9,000

Planned Value (PV) = \$7,500

Earned Value (EV) = \$5,000

Actual Cost (AC) = \$9,000

What is the project health?

Schedule Variance

$$= EV - PV = \$5,000 - \$7,500 = -\$2,500$$

Schedule Performance Index (SPI)

$$= EV/PV = \$5,000 / \$7,500 = .66$$

Cost Variance

$$= EV - AC = \$5,000 - \$9,000 = -\$4,000$$

Cost Performance Index (CPI)

$$= EV/AC = \$5,000 / \$9,000 = .55$$

- Objective metrics indicate the project is behind **schedule and over budget.**
- On-target projects have an **SPI and CPI of 1 or greater**

Forecasting Costs

- If the project continues at the current performance, what is the true cost of the project?
 - Estimate At Complete
 - = Budget At Complete (BAC) / CPI
 - = \$10,000 / .55 = \$18,181
- At the end of the project, the total project costs will be \$18,181

Credits

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

Thank You



BITS Pilani
Pilani Campus

Course Name : Software Engineering

S Subramanian
Work-Integrated Learning Programme





BITS Pilani
Pilani Campus

Module Name : M10 Emerging Trends

S Subramanian
Work-Integrated Learning Programme

Chapter 31

- Emerging Trends

For Software, Change is Constant

Reuse/patterns

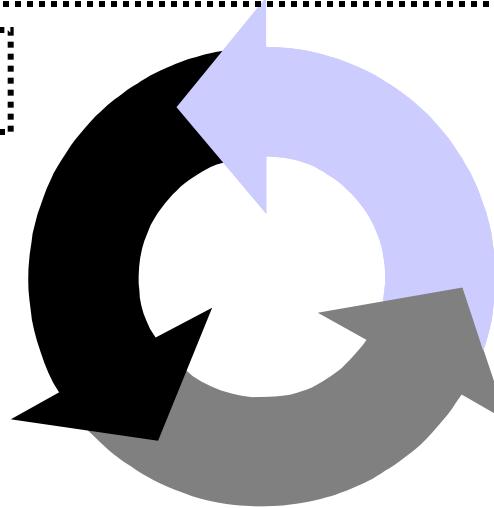
Streamlined processes

Architecture-first

Process paradigms

Metrics-based
management

Best engineering
practices



Extreme programming

Other silver bullets

Agile methods

COTS-based systems

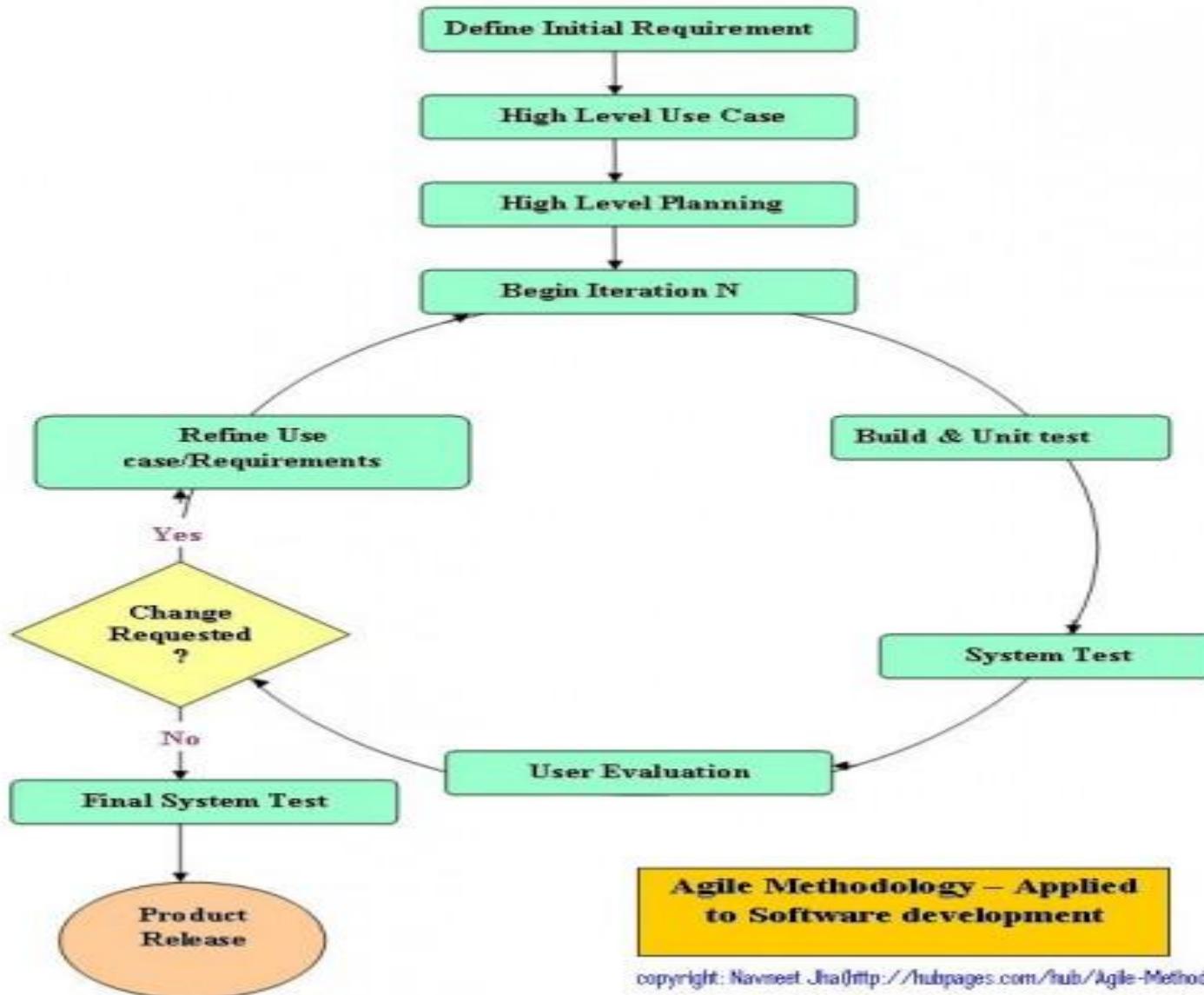
Experience factory

Component-based
composition

Product-lines

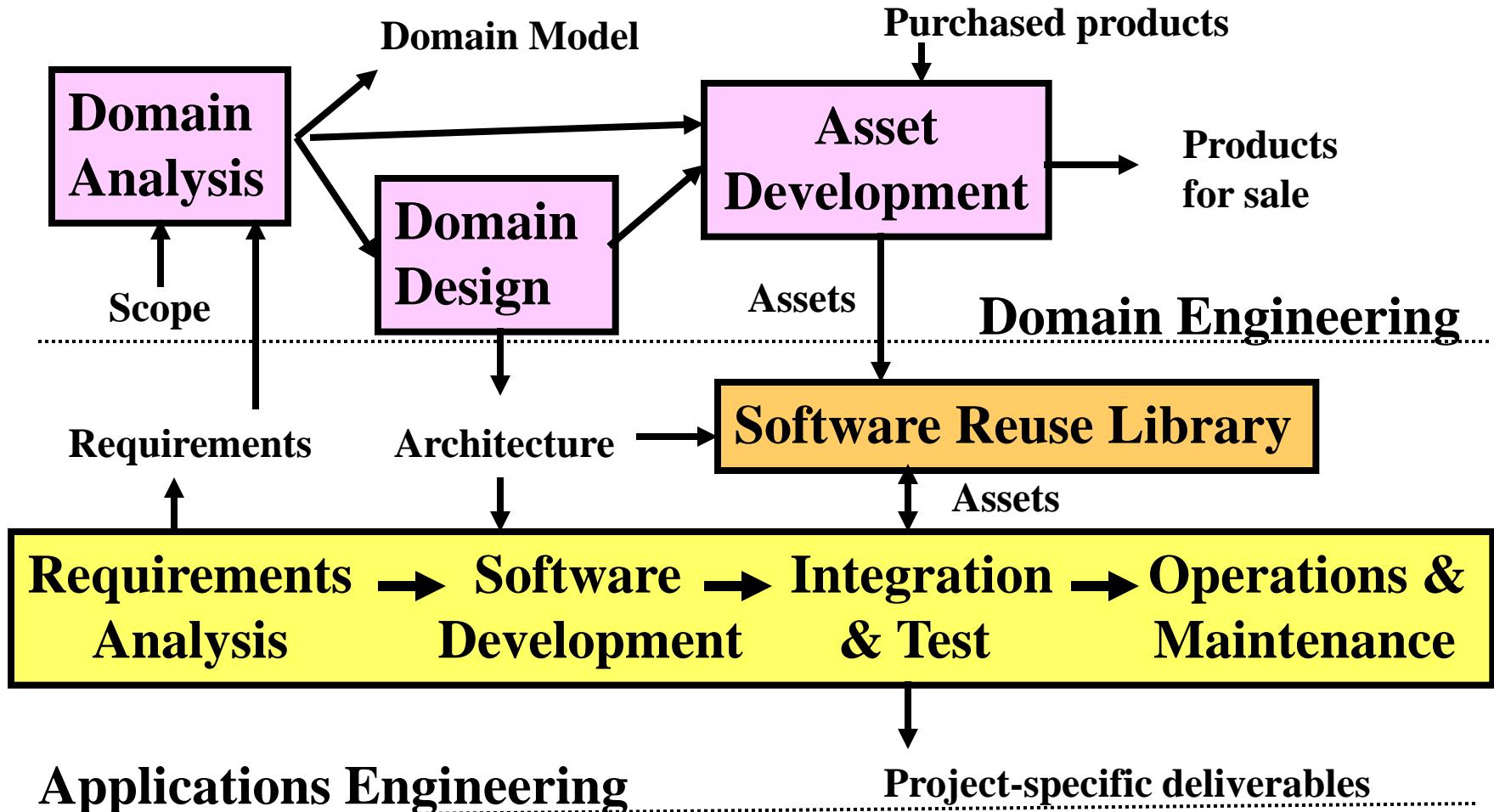
Searching for ways to do the job faster, better and cheaper

AGILE



copyright: Navneet Jha (<http://hubpages.com/hub/Agile-Methodology-A-Brief-overview>)

Reuse-Based Development Paradigm



Gartner Hype cycle



2013 Hype Cycle for Emerging Technologies



Plateau will be reached in:

less than 2 years

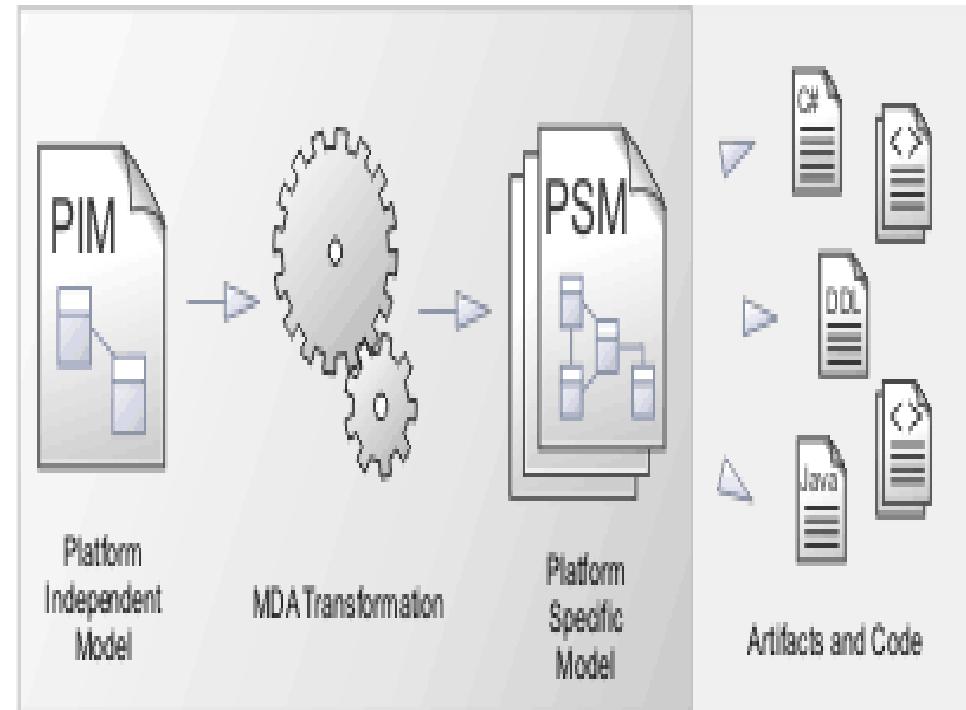
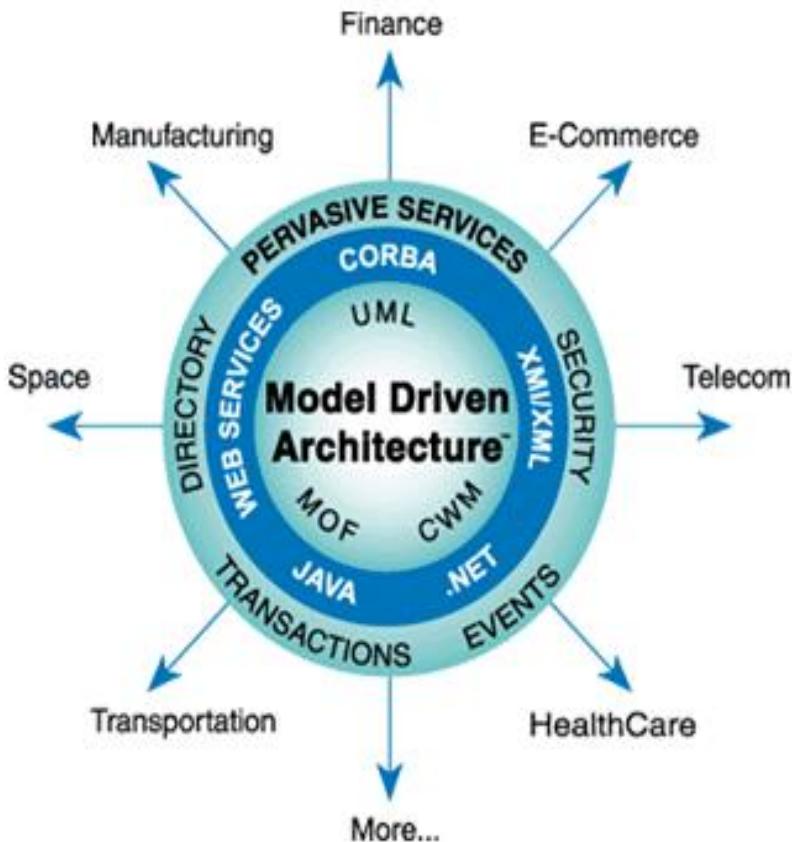
2 to 5 years

5 to 10 years

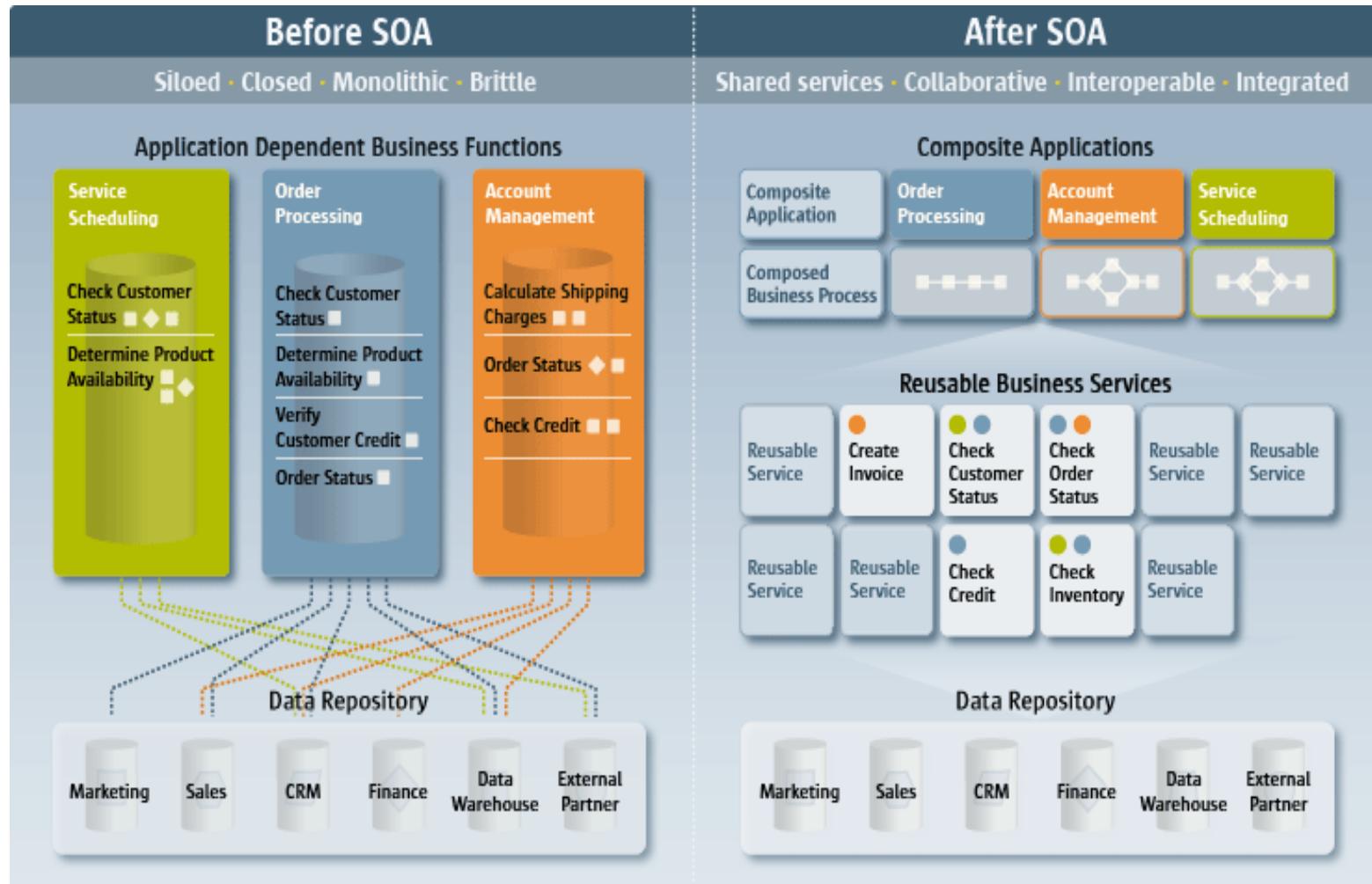
more than 10 years

obsolete
before plateau

Model Driven Architecture



Before SOA – After SOA



source:IBM

Leveraging OS - Snapshot of OSDP4J product stack

innovate

achieve

lead

Requirements
Collection

- Nimble OSRMT



Analysis and
Design



Build



PMD5.0

FindBugs™

sonar



JUnit

CPD

Checkstyle

JavaNCSS

JDepend

Metrics



ORACLE JDeveloper

ORACLE JSF

Cobertura



mybatis

Apache Server



Apache Tomcat



JBoss



JBoss ESB

Mule ESB



Testing

HUDSON
Extensible continuous
integration server



QAliber



Dyna Trace



SoapUI



Nikto2



Paros Proxy

InfraRED

Odysseus

JENSOR - Java Profiler

Spike

WebScarab



Web Services

Note: The above images are copied from the internet and it is respective organization proprietary

Trends in Testing



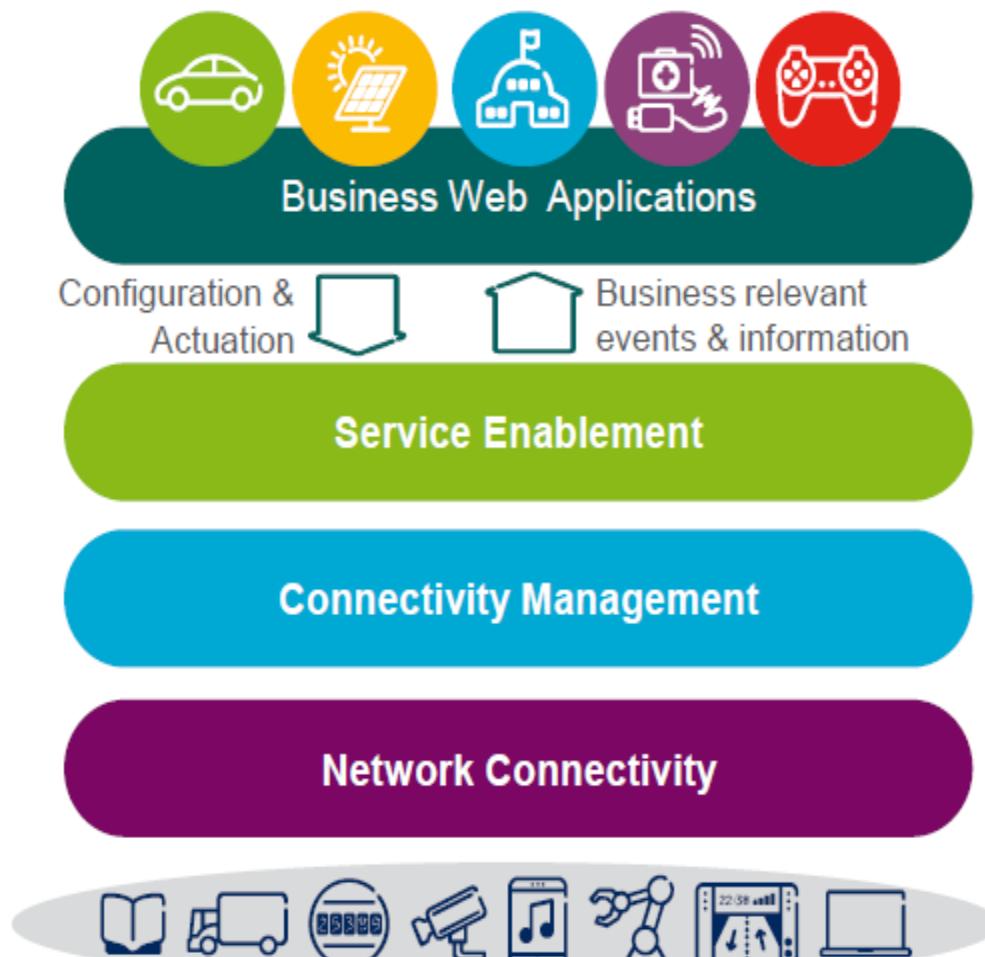
- **DEVICE TESTING**
- **SCENARIO BASED EXPLORATORY TESTING**
- **RISK BASED TESTING**
- **PERFORMANCE AND SECURITY TESTING**
- **USER EXPERIENCE TESTING**

Internet of Things



Business Web:

First-class Internet of Things Integration



Cloud application development



Credits

- Software Engineering 7/ed by Roger Pressman
- Based on Donald J. Reifer's lecture **and**
- Other Internet sources.

Thank You