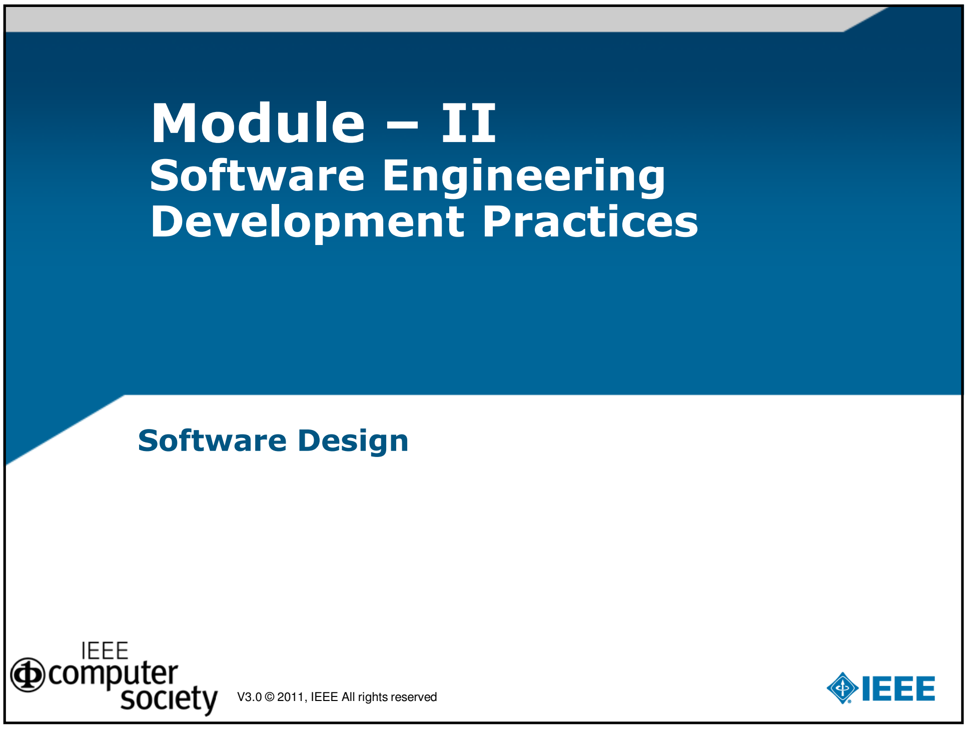
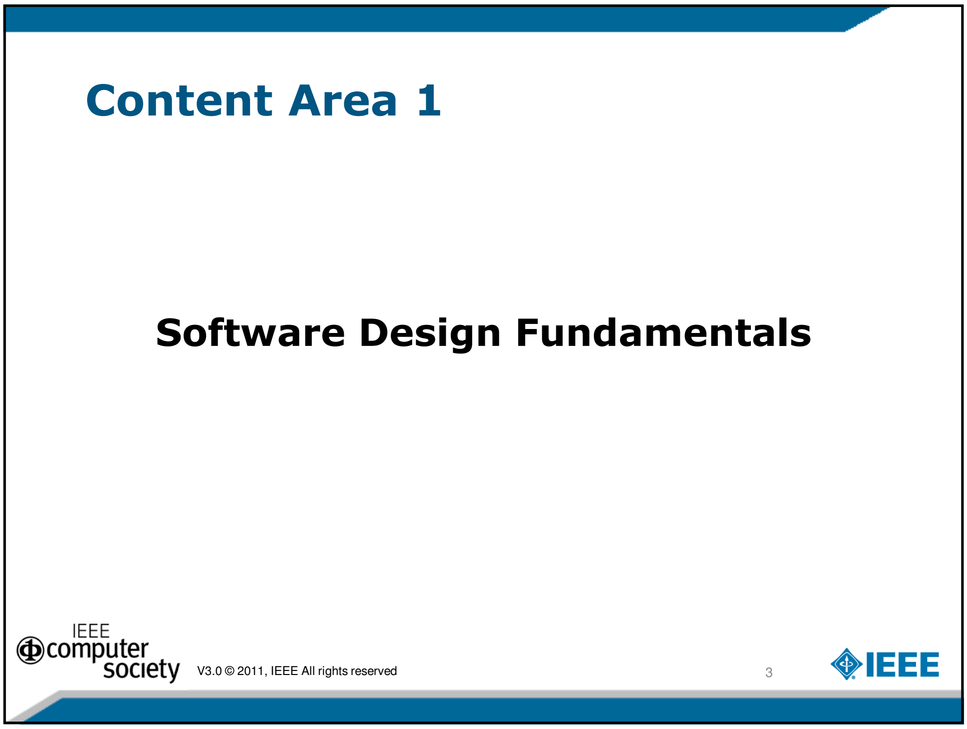


Module – II
Software Engineering Development Practices

Software Design

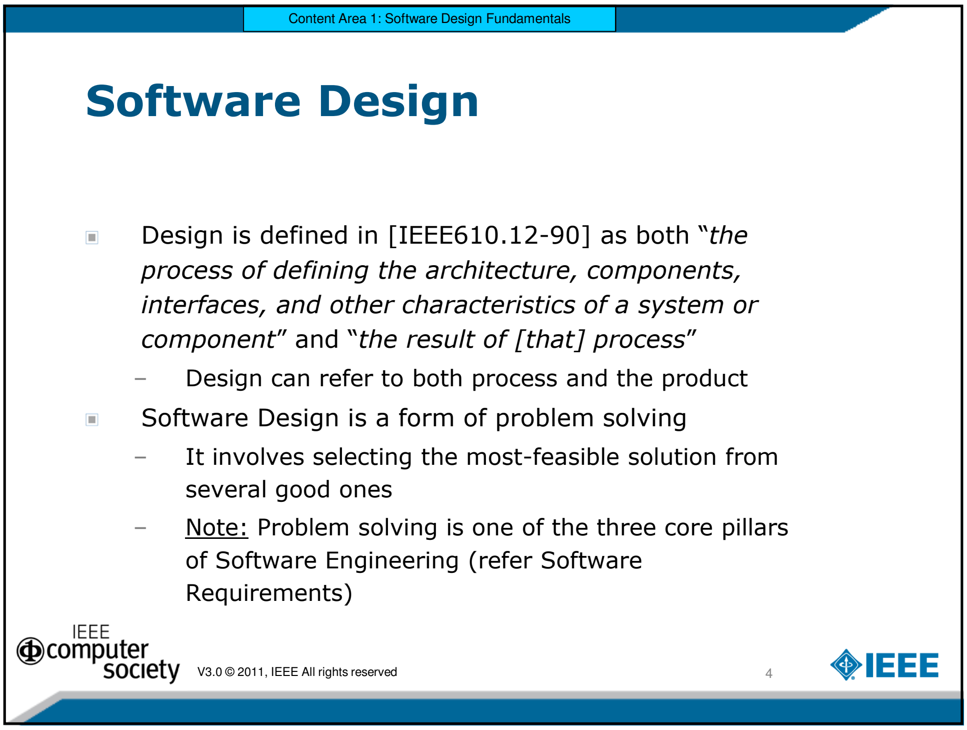
Software Design - Roadmap

- Software Design Fundamentals
- Key Issues in Software Design
- Software Structure and Architecture
- Human-Computer Interface Design
- Software Design Quality Analysis and Evaluation
- Software Design Notations
- Software Design Strategies and Methods



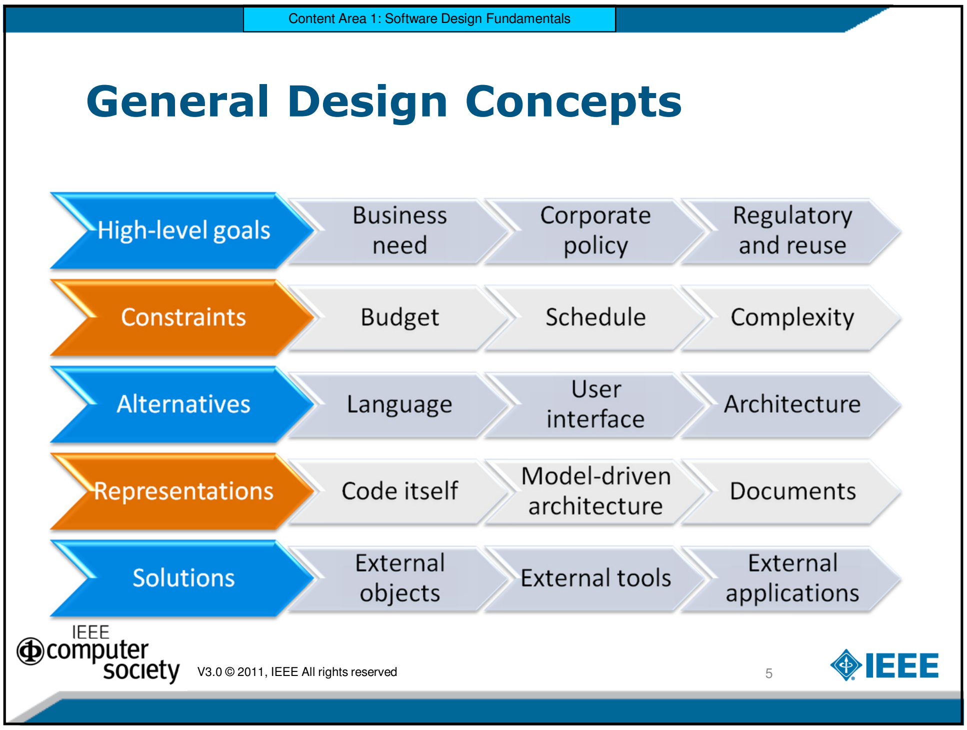
Content Area 1

Software Design Fundamentals



Software Design

- Design is defined in [IEEE610.12-90] as both “*the process of defining the architecture, components, interfaces, and other characteristics of a system or component*” and “*the result of [that] process*”
  - Design can refer to both process and the product
- Software Design is a form of problem solving
  - It involves selecting the most-feasible solution from several good ones
  - <u>Note:</u> Problem solving is one of the three core pillars of Software Engineering (refer Software Requirements)

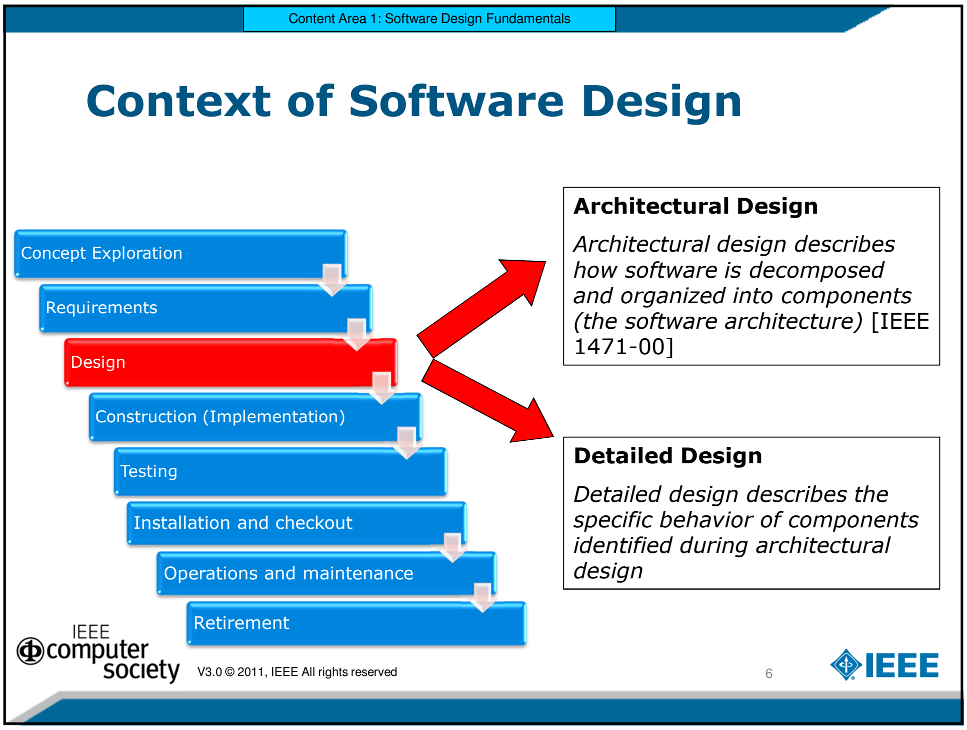


Content Area 1: Software Design Fundamentals

General Design Concepts

High-level goals	Business need	Corporate policy	Regulatory and reuse
Constraints	Budget	Schedule	Complexity
Alternatives	Language	User interface	Architecture
Representations	Code itself	Model-driven architecture	Documents
Solutions	External objects	External tools	External applications

IEEE computer society



Content Area 1: Software Design Fundamentals

Context of Software Design

Concept Exploration
Requirements
Design
Construction (Implementation)
Testing
Installation and checkout
Operations and maintenance
Retirement

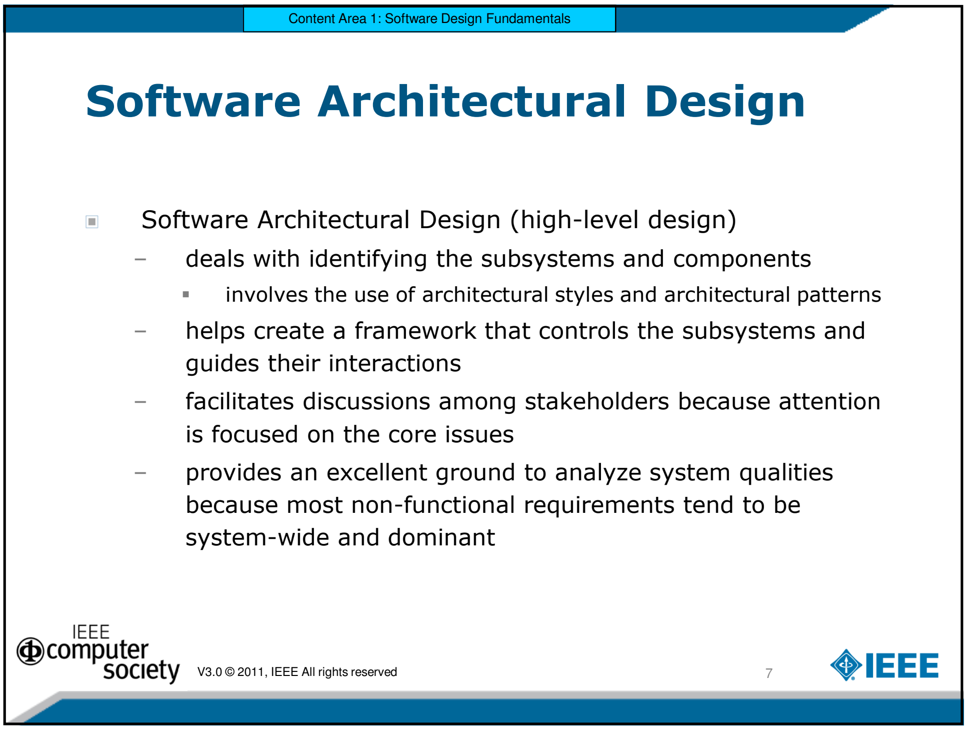
Architectural Design

Architectural design describes how software is decomposed and organized into components (the software architecture) [IEEE 1471-00]

Detailed Design

Detailed design describes the specific behavior of components identified during architectural design

IEEE computer society



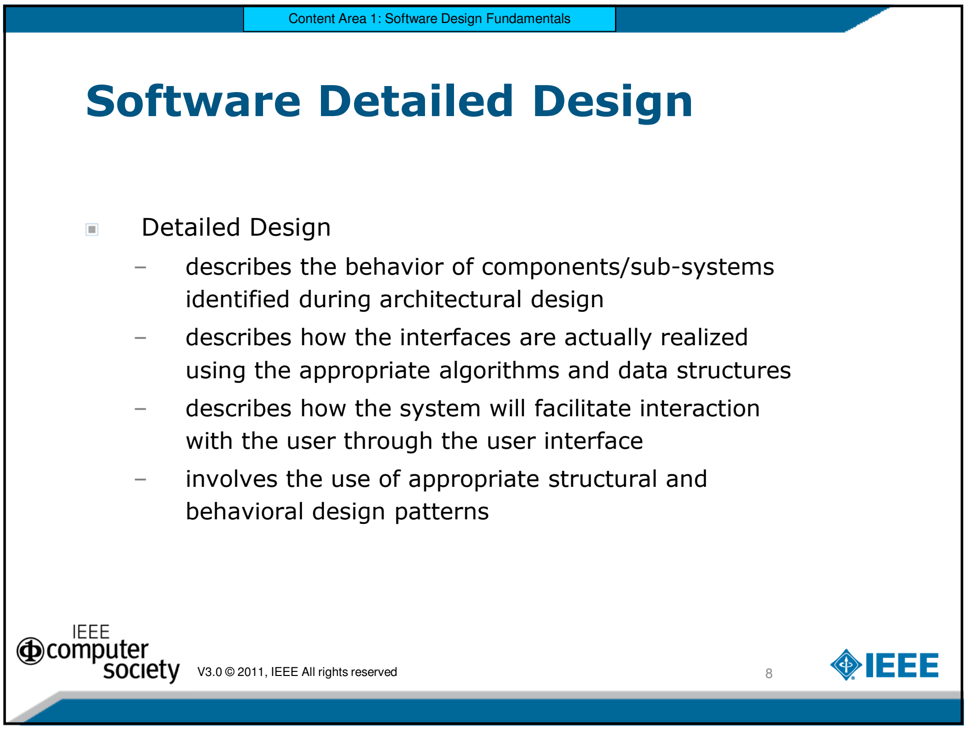
Content Area 1: Software Design Fundamentals

Software Architectural Design

- Software Architectural Design (high-level design)
  - deals with identifying the subsystems and components
    - involves the use of architectural styles and architectural patterns
  - helps create a framework that controls the subsystems and guides their interactions
  - facilitates discussions among stakeholders because attention is focused on the core issues
  - provides an excellent ground to analyze system qualities because most non-functional requirements tend to be system-wide and dominant

V3.0 © 2011, IEEE All rights reserved

7



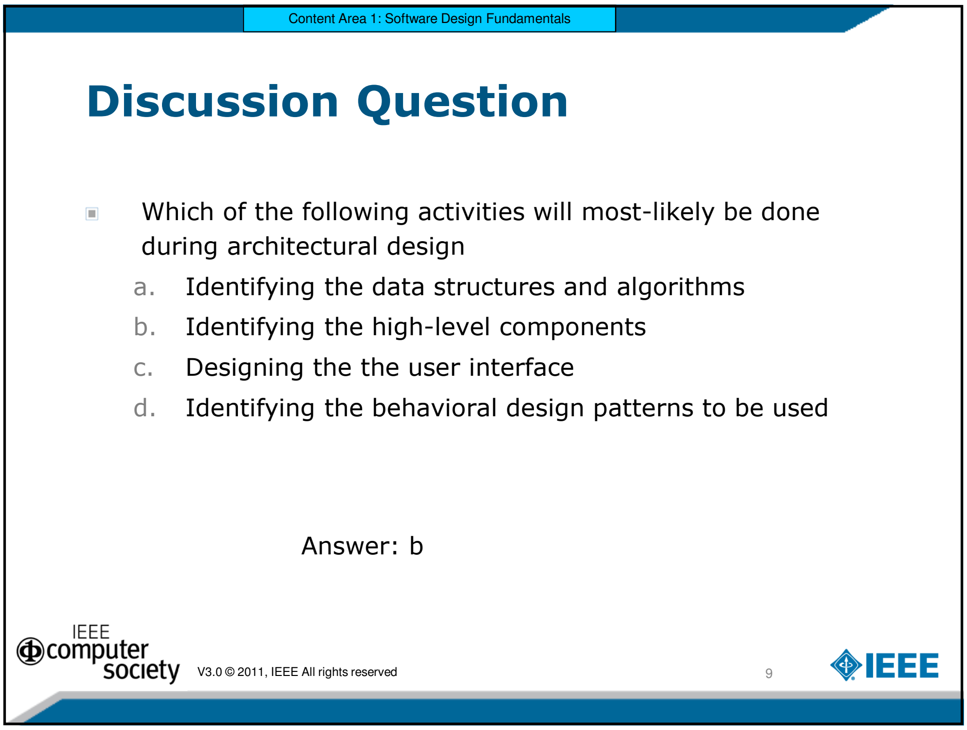
Content Area 1: Software Design Fundamentals

Software Detailed Design

- Detailed Design
  - describes the behavior of components/sub-systems identified during architectural design
  - describes how the interfaces are actually realized using the appropriate algorithms and data structures
  - describes how the system will facilitate interaction with the user through the user interface
  - involves the use of appropriate structural and behavioral design patterns

V3.0 © 2011, IEEE All rights reserved

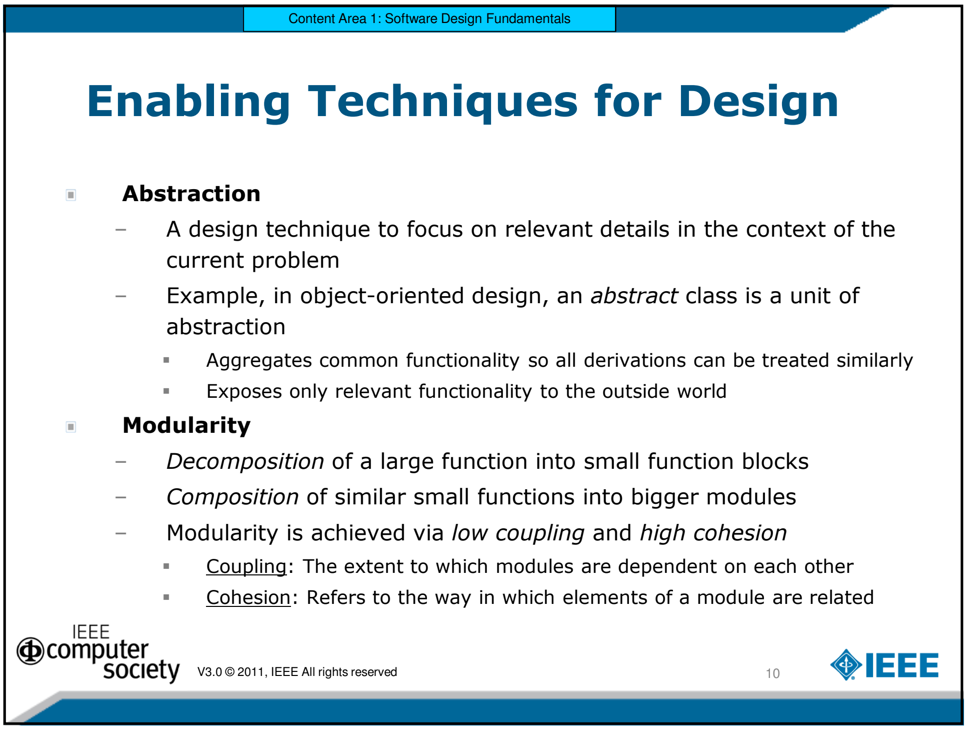
8



Discussion Question

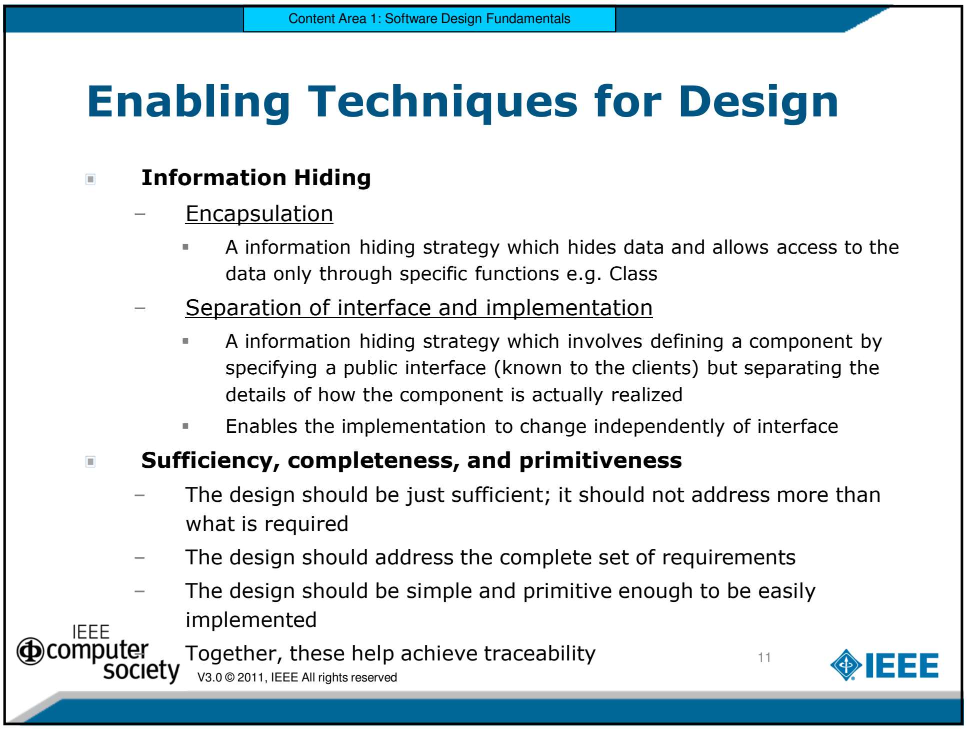
- Which of the following activities will most-likely be done during architectural design
  a. Identifying the data structures and algorithms
  b. Identifying the high-level components
  c. Designing the the user interface
  d. Identifying the behavioral design patterns to be used

Answer: b



Enabling Techniques for Design

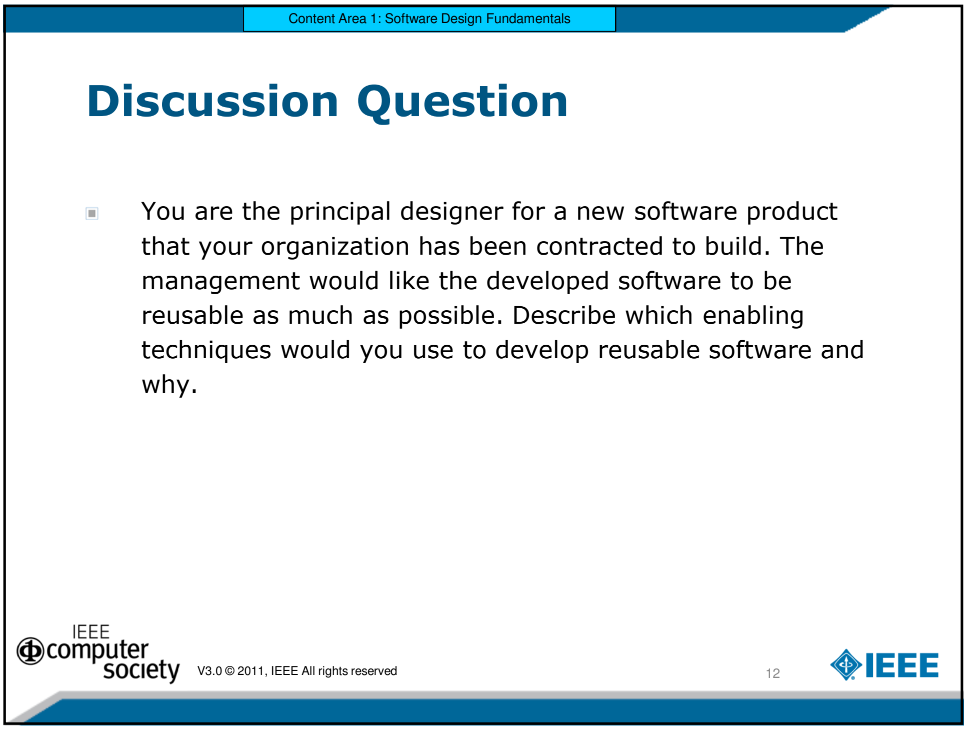
- **Abstraction**
  - A design technique to focus on relevant details in the context of the current problem
  - Example, in object-oriented design, an *abstract* class is a unit of abstraction
    - Aggregates common functionality so all derivations can be treated similarly
    - Exposes only relevant functionality to the outside world
- **Modularity**
  - *Decomposition* of a large function into small function blocks
  - *Composition* of similar small functions into bigger modules
  - Modularity is achieved via *low coupling* and *high cohesion*
    - Coupling: The extent to which modules are dependent on each other
    - Cohesion: Refers to the way in which elements of a module are related



Content Area 1: Software Design Fundamentals

Enabling Techniques for Design

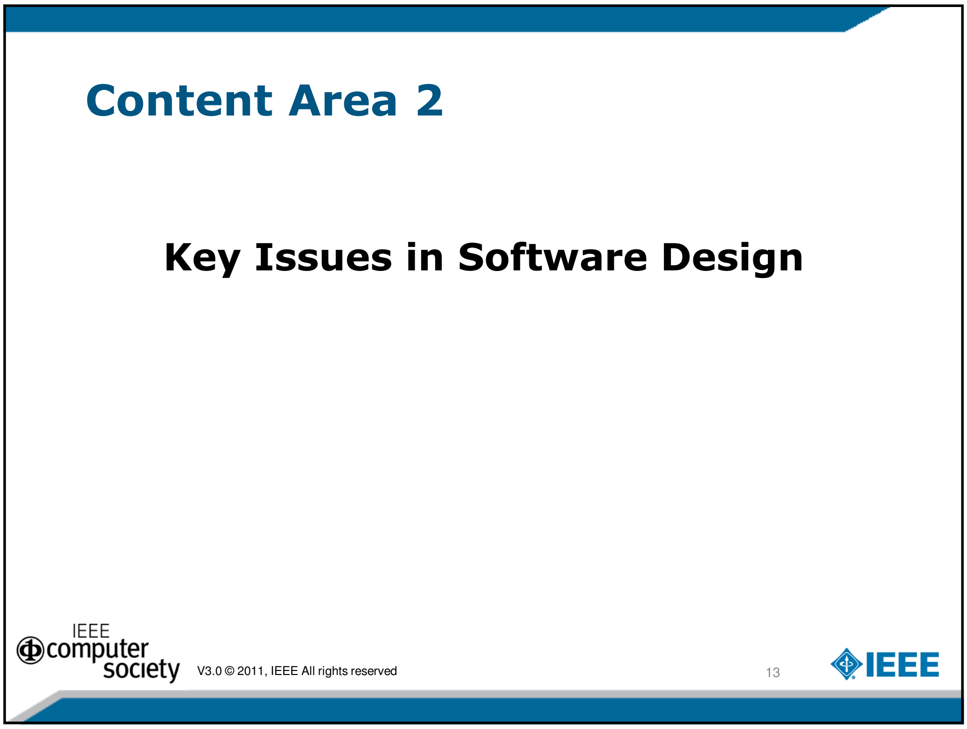
- **Information Hiding**
  - Encapsulation
    - A information hiding strategy which hides data and allows access to the data only through specific functions e.g. Class
  - Separation of interface and implementation
    - A information hiding strategy which involves defining a component by specifying a public interface (known to the clients) but separating the details of how the component is actually realized
    - Enables the implementation to change independently of interface
- **Sufficiency, completeness, and primitiveness**
  - The design should be just sufficient; it should not address more than what is required
  - The design should address the complete set of requirements
  - The design should be simple and primitive enough to be easily implemented
  - Together, these help achieve traceability



Content Area 1: Software Design Fundamentals

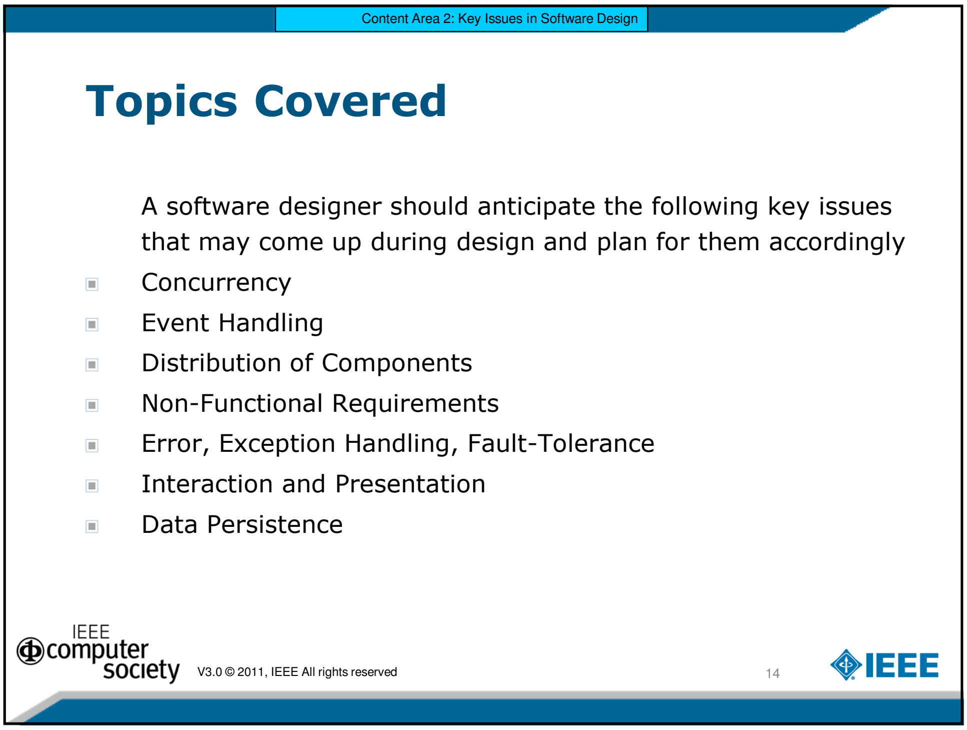
Discussion Question

- You are the principal designer for a new software product that your organization has been contracted to build. The management would like the developed software to be reusable as much as possible. Describe which enabling techniques would you use to develop reusable software and why.



Content Area 2

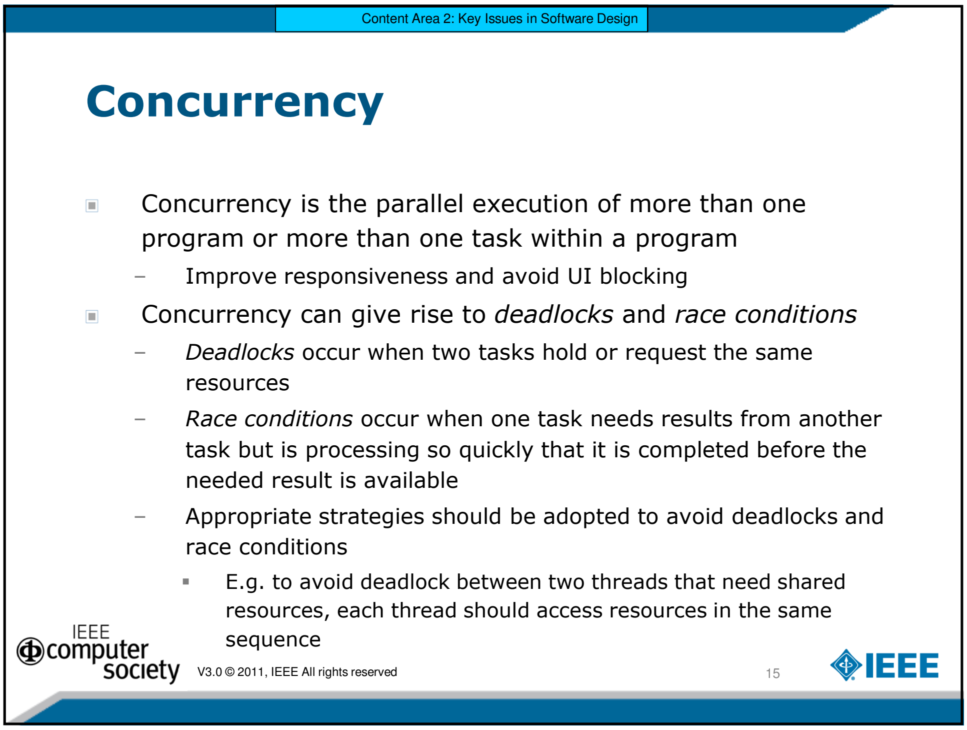
Key Issues in Software Design



Topics Covered

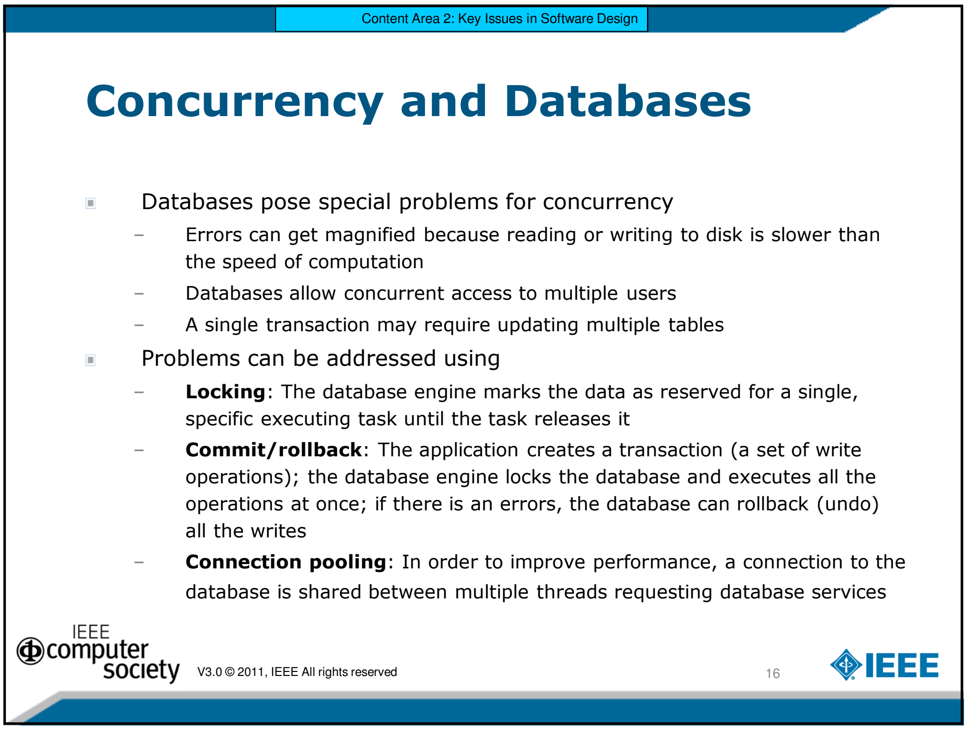
A software designer should anticipate the following key issues that may come up during design and plan for them accordingly

- Concurrency
- Event Handling
- Distribution of Components
- Non-Functional Requirements
- Error, Exception Handling, Fault-Tolerance
- Interaction and Presentation
- Data Persistence



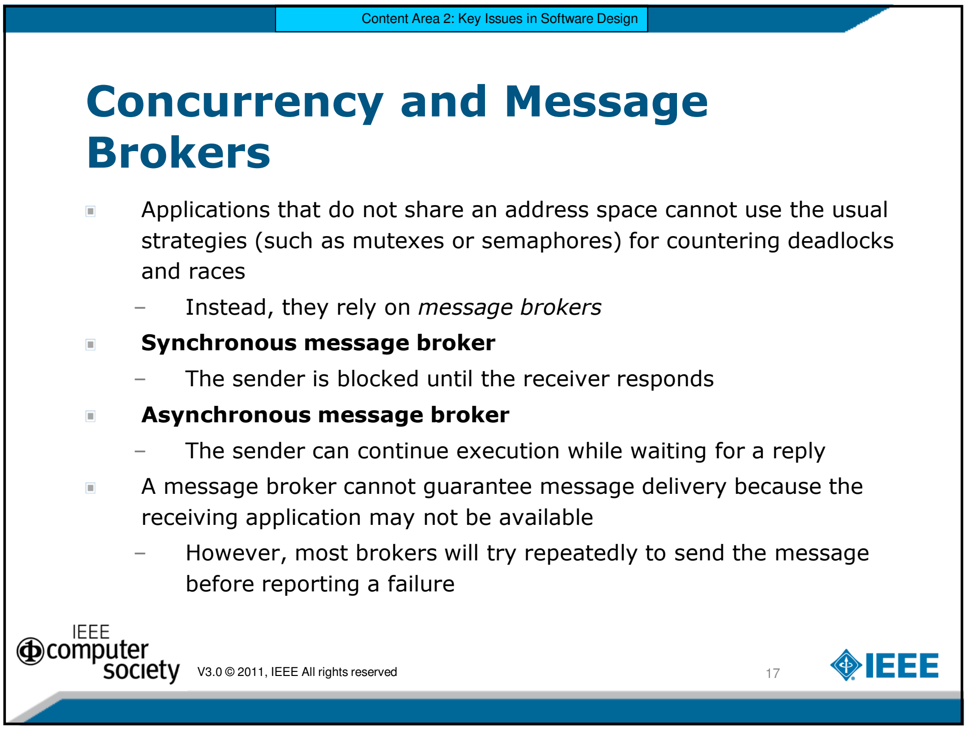
Concurrency

- Concurrency is the parallel execution of more than one program or more than one task within a program
  - Improve responsiveness and avoid UI blocking
- Concurrency can give rise to *deadlocks* and *race conditions*
  - *Deadlocks* occur when two tasks hold or request the same resources
  - *Race conditions* occur when one task needs results from another task but is processing so quickly that it is completed before the needed result is available
  - Appropriate strategies should be adopted to avoid deadlocks and race conditions
    - E.g. to avoid deadlock between two threads that need shared resources, each thread should access resources in the same sequence



Concurrency and Databases

- Databases pose special problems for concurrency
  - Errors can get magnified because reading or writing to disk is slower than the speed of computation
  - Databases allow concurrent access to multiple users
  - A single transaction may require updating multiple tables
- Problems can be addressed using
  - **Locking**: The database engine marks the data as reserved for a single, specific executing task until the task releases it
  - **Commit/rollback**: The application creates a transaction (a set of write operations); the database engine locks the database and executes all the operations at once; if there is an errors, the database can rollback (undo) all the writes
  - **Connection pooling**: In order to improve performance, a connection to the database is shared between multiple threads requesting database services

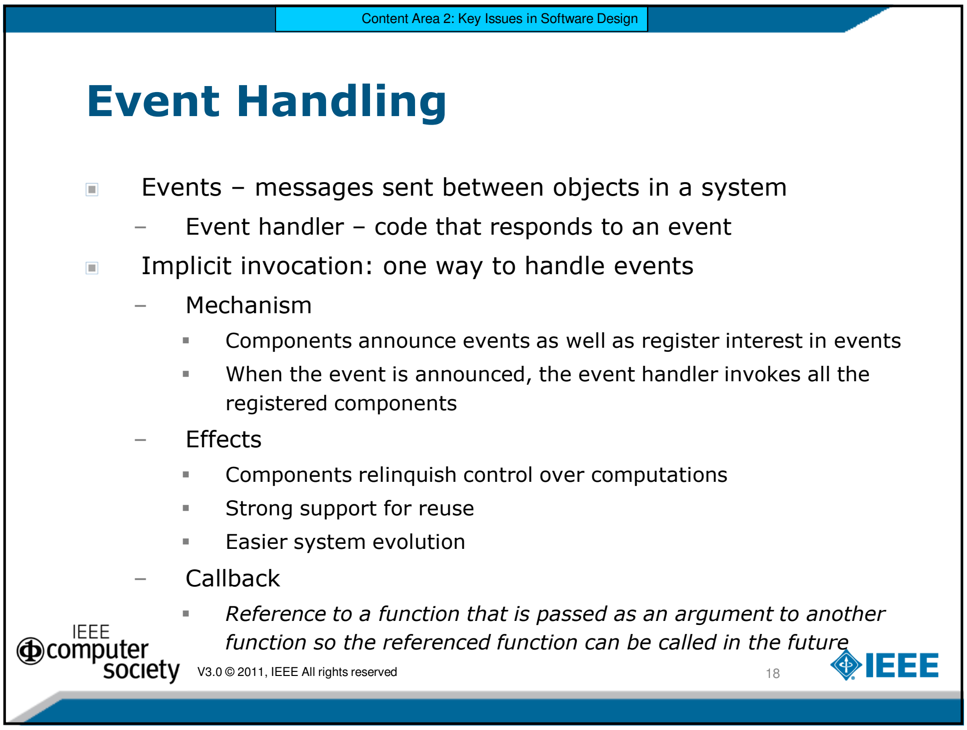


Content Area 2: Key Issues in Software Design

Concurrency and Message Brokers

- Applications that do not share an address space cannot use the usual strategies (such as mutexes or semaphores) for countering deadlocks and races
  - Instead, they rely on *message brokers*
- **Synchronous message broker**
  - The sender is blocked until the receiver responds
- **Asynchronous message broker**
  - The sender can continue execution while waiting for a reply
- A message broker cannot guarantee message delivery because the receiving application may not be available
  - However, most brokers will try repeatedly to send the message before reporting a failure

IEEE computer society V3.0 © 2011, IEEE All rights reserved 17 IEEE

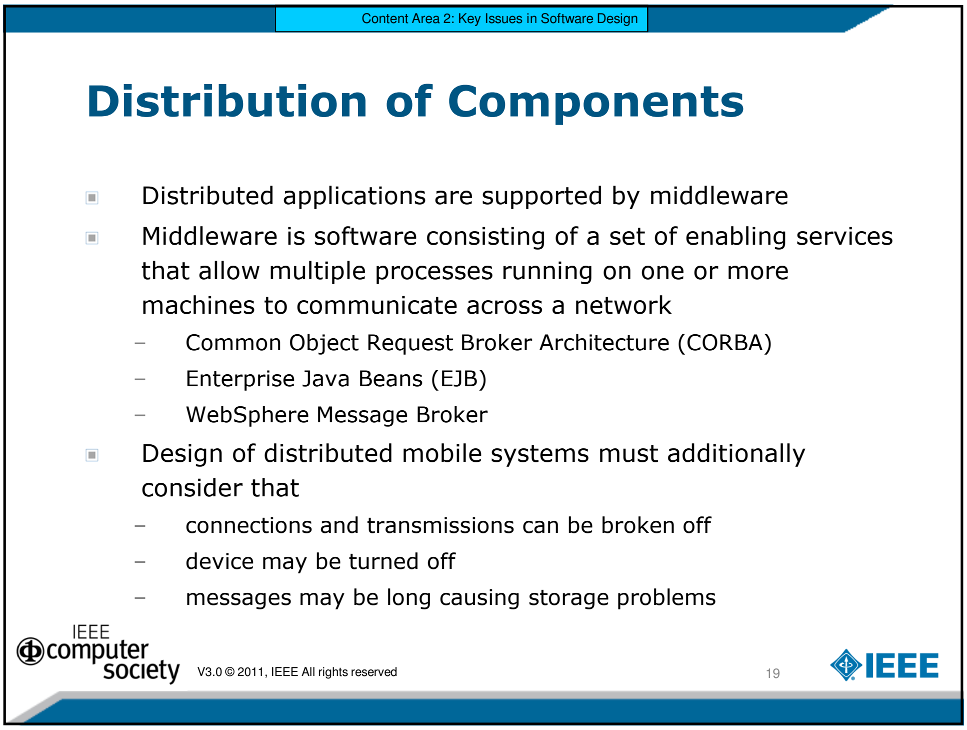


Content Area 2: Key Issues in Software Design

Event Handling

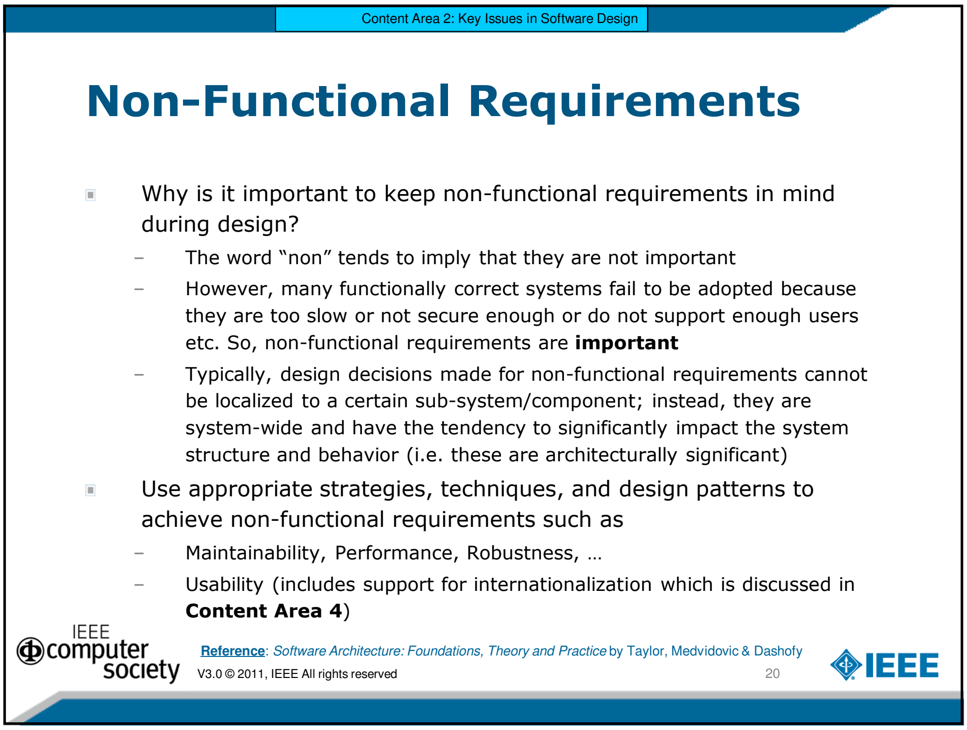
- Events – messages sent between objects in a system
  - Event handler – code that responds to an event
- Implicit invocation: one way to handle events
  - Mechanism
    - Components announce events as well as register interest in events
    - When the event is announced, the event handler invokes all the registered components
  - Effects
    - Components relinquish control over computations
    - Strong support for reuse
    - Easier system evolution
  - Callback
    - *Reference to a function that is passed as an argument to another function so the referenced function can be called in the future*

IEEE computer society V3.0 © 2011, IEEE All rights reserved 18 IEEE



Distribution of Components

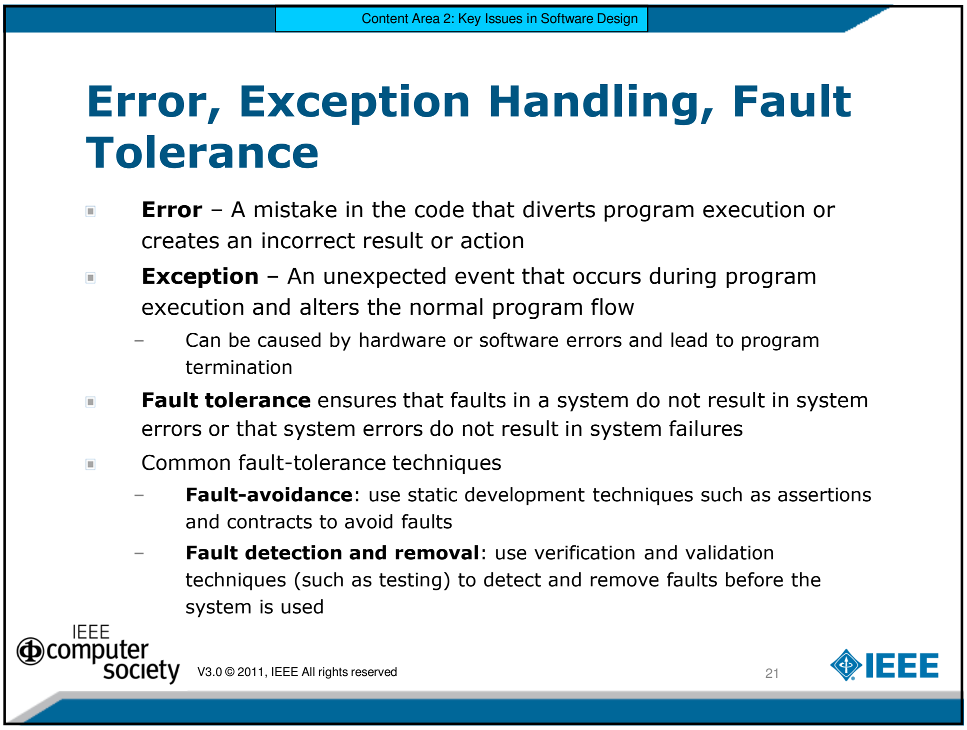
- Distributed applications are supported by middleware
- Middleware is software consisting of a set of enabling services that allow multiple processes running on one or more machines to communicate across a network
  - Common Object Request Broker Architecture (CORBA)
  - Enterprise Java Beans (EJB)
  - WebSphere Message Broker
- Design of distributed mobile systems must additionally consider that
  - connections and transmissions can be broken off
  - device may be turned off
  - messages may be long causing storage problems



Non-Functional Requirements

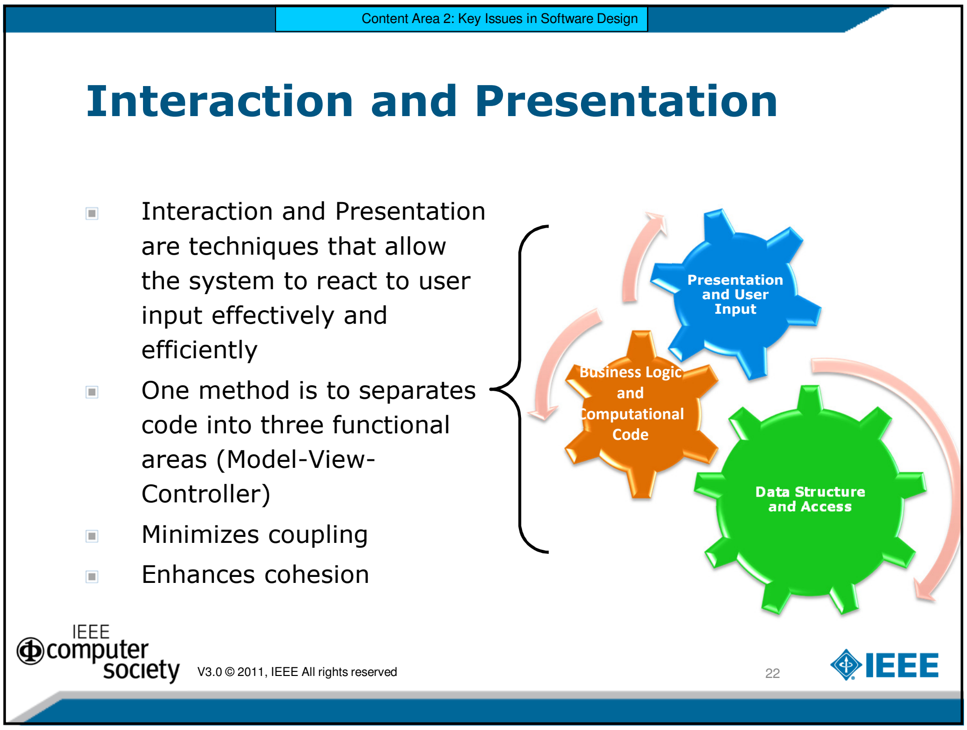
- Why is it important to keep non-functional requirements in mind during design?
  - The word "non" tends to imply that they are not important
  - However, many functionally correct systems fail to be adopted because they are too slow or not secure enough or do not support enough users etc. So, non-functional requirements are **important**
  - Typically, design decisions made for non-functional requirements cannot be localized to a certain sub-system/component; instead, they are system-wide and have the tendency to significantly impact the system structure and behavior (i.e. these are architecturally significant)
- Use appropriate strategies, techniques, and design patterns to achieve non-functional requirements such as
  - Maintainability, Performance, Robustness, …
  - Usability (includes support for internationalization which is discussed in **Content Area 4**)

Reference: *Software Architecture: Foundations, Theory and Practice* by Taylor, Medvidovic & Dashofy



Error, Exception Handling, Fault Tolerance

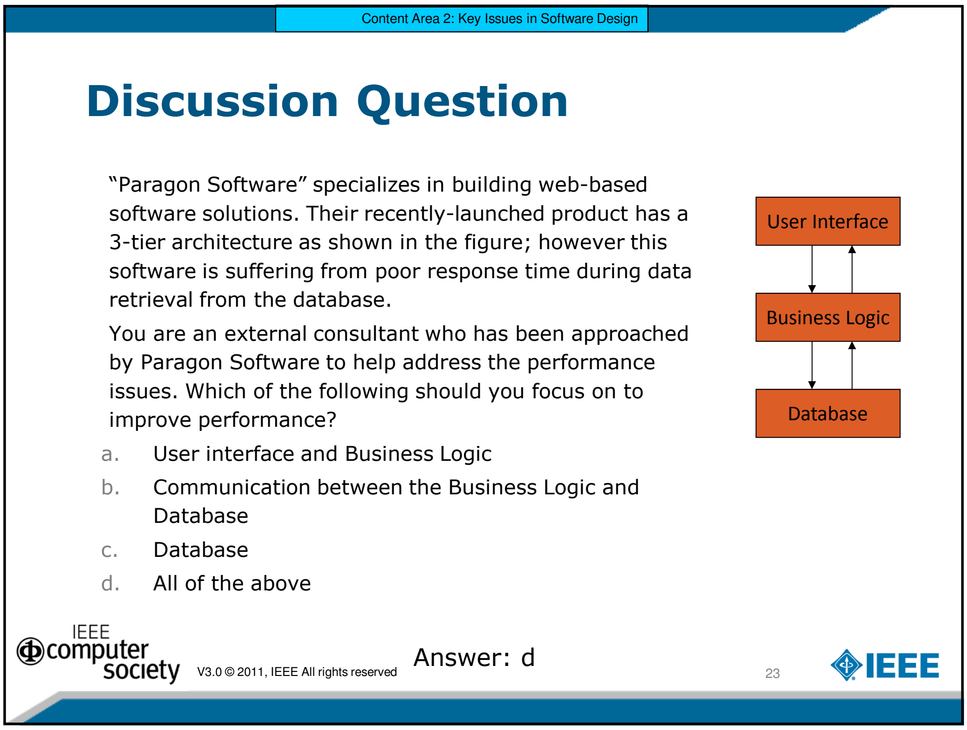
- **Error** – A mistake in the code that diverts program execution or creates an incorrect result or action
- **Exception** – An unexpected event that occurs during program execution and alters the normal program flow
  - Can be caused by hardware or software errors and lead to program termination
- **Fault tolerance** ensures that faults in a system do not result in system errors or that system errors do not result in system failures
- Common fault-tolerance techniques
  - **Fault-avoidance**: use static development techniques such as assertions and contracts to avoid faults
  - **Fault detection and removal**: use verification and validation techniques (such as testing) to detect and remove faults before the system is used



Interaction and Presentation

- Interaction and Presentation are techniques that allow the system to react to user input effectively and efficiently
- One method is to separates code into three functional areas (Model-View-Controller)
- Minimizes coupling
- Enhances cohesion

Presentation and User Input
Business Logic and Computational Code
Data Structure and Access



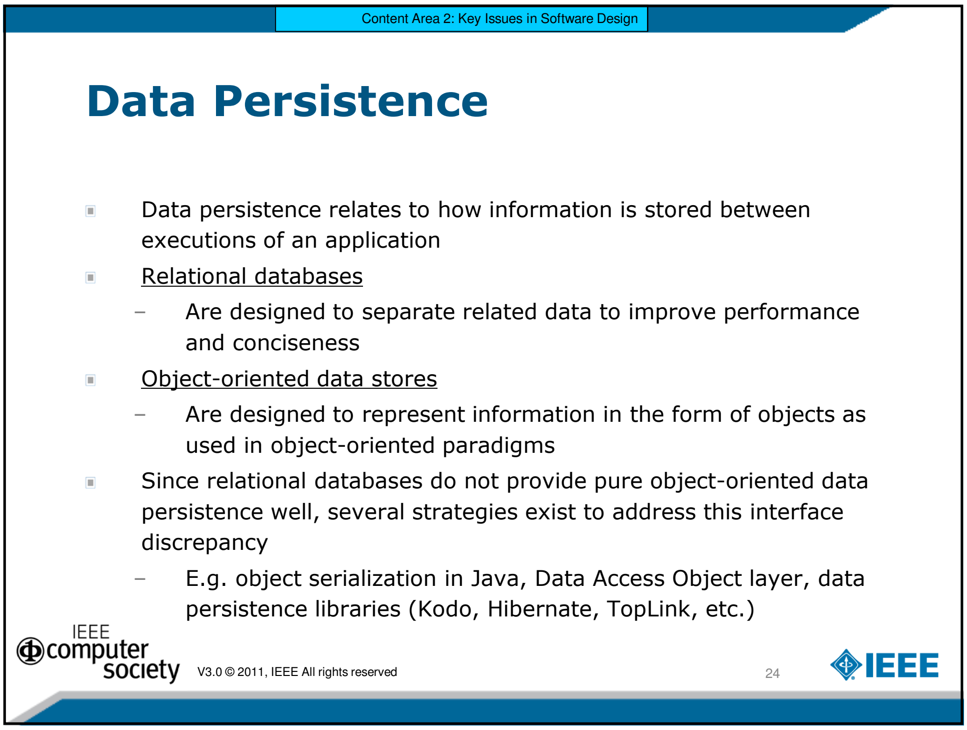
Discussion Question

"Paragon Software" specializes in building web-based software solutions. Their recently-launched product has a 3-tier architecture as shown in the figure; however this software is suffering from poor response time during data retrieval from the database.

You are an external consultant who has been approached by Paragon Software to help address the performance issues. Which of the following should you focus on to improve performance?

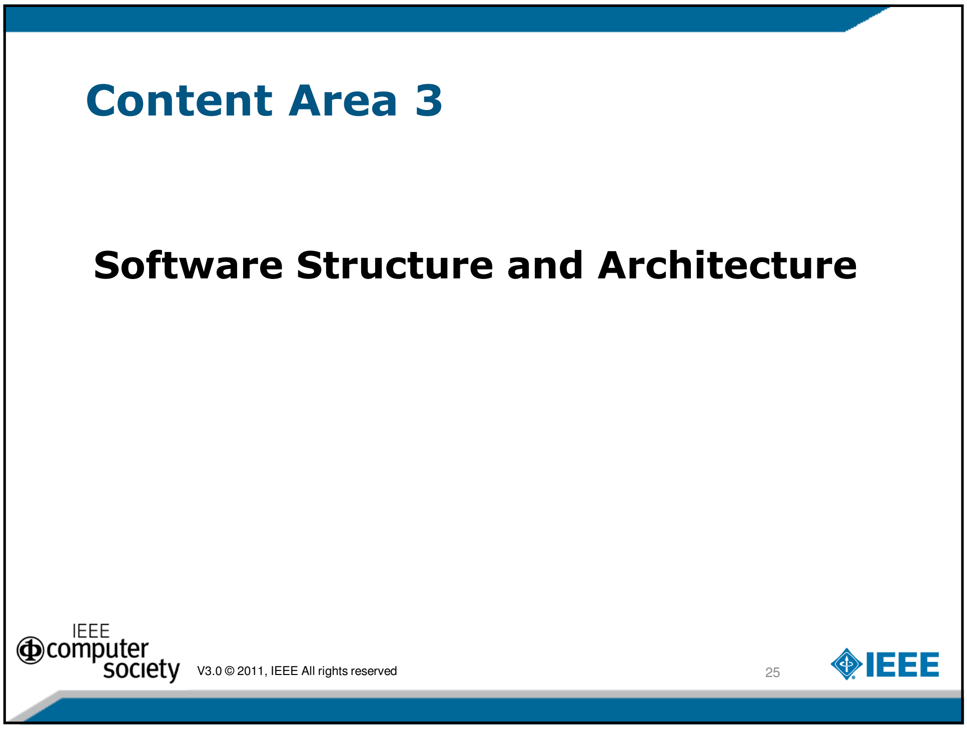
a. User interface and Business Logic
b. Communication between the Business Logic and Database
c. Database
d. All of the above

Answer: d



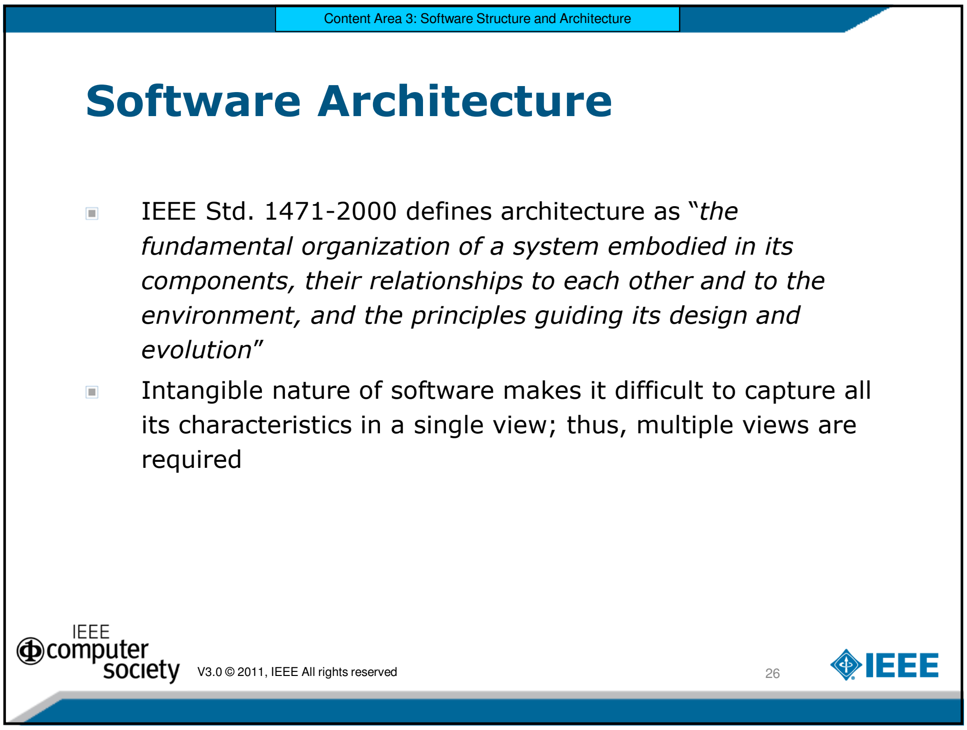
Data Persistence

- Data persistence relates to how information is stored between executions of an application
- Relational databases
  - Are designed to separate related data to improve performance and conciseness
- Object-oriented data stores
  - Are designed to represent information in the form of objects as used in object-oriented paradigms
- Since relational databases do not provide pure object-oriented data persistence well, several strategies exist to address this interface discrepancy
  - E.g. object serialization in Java, Data Access Object layer, data persistence libraries (Kodo, Hibernate, TopLink, etc.)



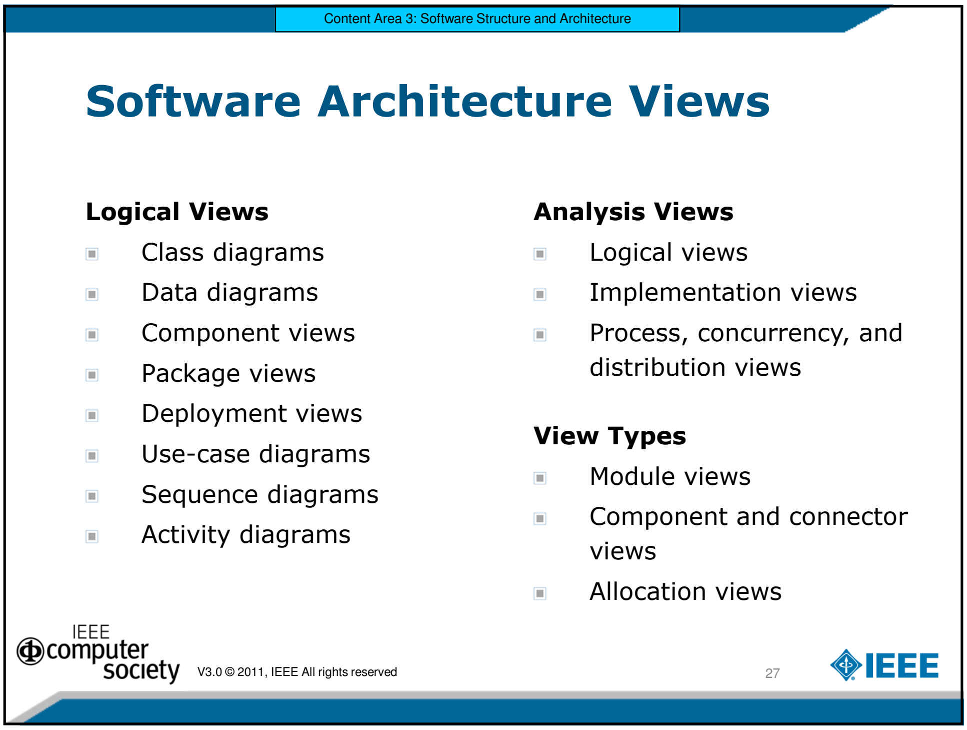
Content Area 3

Software Structure and Architecture



Software Architecture

- IEEE Std. 1471-2000 defines architecture as "*the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution*"
- Intangible nature of software makes it difficult to capture all its characteristics in a single view; thus, multiple views are required



Software Architecture Views

Logical Views

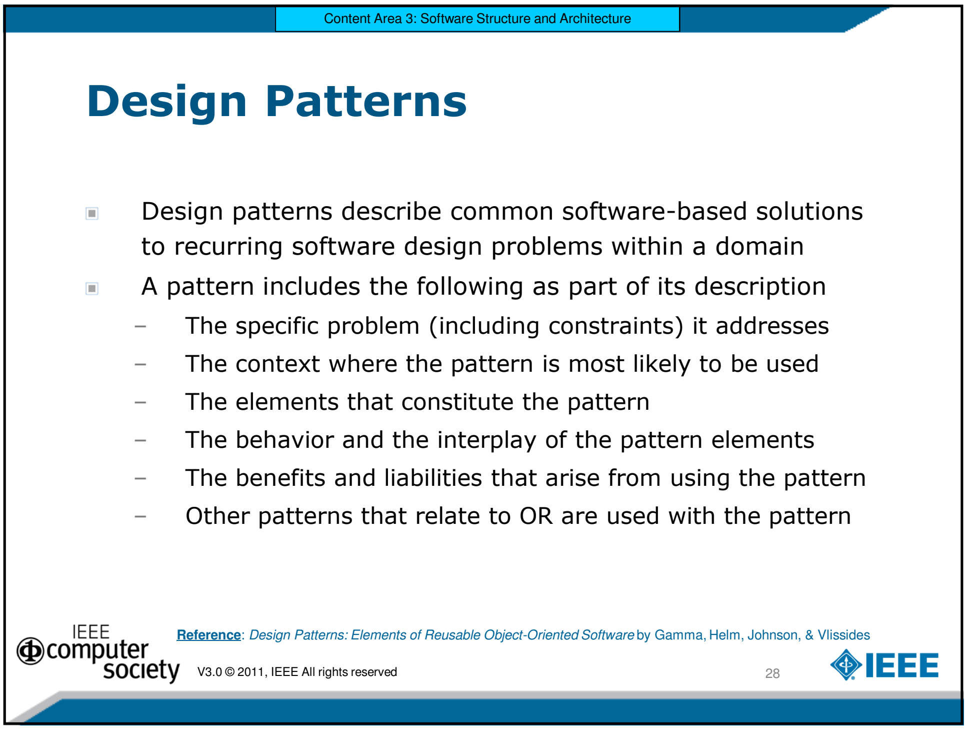
- Class diagrams
- Data diagrams
- Component views
- Package views
- Deployment views
- Use-case diagrams
- Sequence diagrams
- Activity diagrams

Analysis Views

- Logical views
- Implementation views
- Process, concurrency, and distribution views

View Types

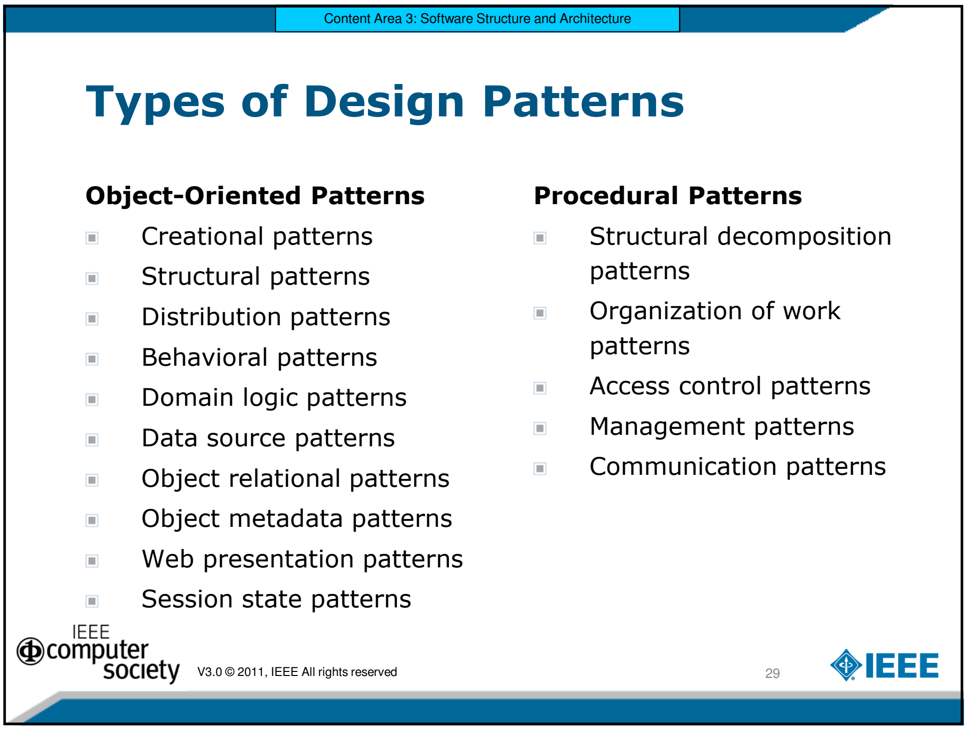
- Module views
- Component and connector views
- Allocation views



Design Patterns

- Design patterns describe common software-based solutions to recurring software design problems within a domain
- A pattern includes the following as part of its description
  - The specific problem (including constraints) it addresses
  - The context where the pattern is most likely to be used
  - The elements that constitute the pattern
  - The behavior and the interplay of the pattern elements
  - The benefits and liabilities that arise from using the pattern
  - Other patterns that relate to OR are used with the pattern

Reference: *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, & Vlissides



Content Area 3: Software Structure and Architecture

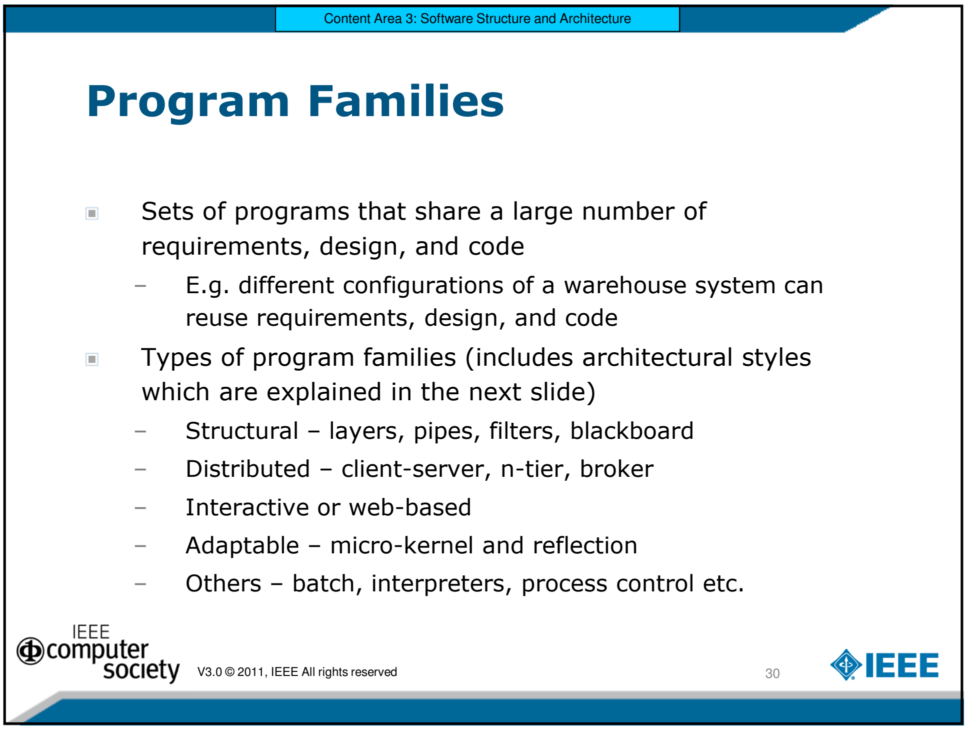
Types of Design Patterns

Object-Oriented Patterns

- Creational patterns
- Structural patterns
- Distribution patterns
- Behavioral patterns
- Domain logic patterns
- Data source patterns
- Object relational patterns
- Object metadata patterns
- Web presentation patterns
- Session state patterns

Procedural Patterns

- Structural decomposition patterns
- Organization of work patterns
- Access control patterns
- Management patterns
- Communication patterns



Content Area 3: Software Structure and Architecture

Program Families

- Sets of programs that share a large number of requirements, design, and code
  - E.g. different configurations of a warehouse system can reuse requirements, design, and code
- Types of program families (includes architectural styles which are explained in the next slide)
  - Structural – layers, pipes, filters, blackboard
  - Distributed – client-server, n-tier, broker
  - Interactive or web-based
  - Adaptable – micro-kernel and reflection
  - Others – batch, interpreters, process control etc.

Software Architectural Styles

- SWEBOK defines an architectural style as *a set of constraints on an architecture [that] defines a set or family of architectures that satisfies them*
  - Constraints affect both the structure and behavior of the system
- *An architectural style is a named collection of architectural design decisions that are applicable in a given development context, constrain architectural design decisions that are specific to a particular system within that context, and elicit beneficial qualities in each resulting system* [See reference below]
- Benefits of architectural styles
  - Reuse of design and code components
  - Ease of understanding the architecture
  - Increased interoperability

Reference: *Software Architecture: Foundations, Theory and Practice* by Taylor, Medvidovic & Dashofy

Frameworks

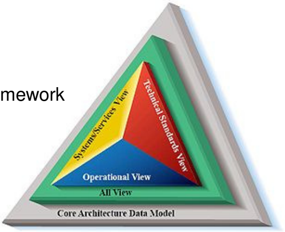
A framework helps organize how the subsystems are related

- Used when multiple vendors each make part of a system

Software Architecture Frameworks

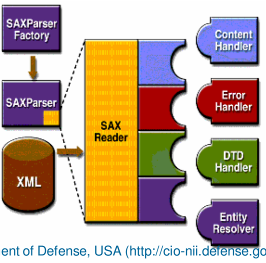
- Provide a blueprint to organize the information and interactions among system components e.g. DODAF

DoDAF Framework



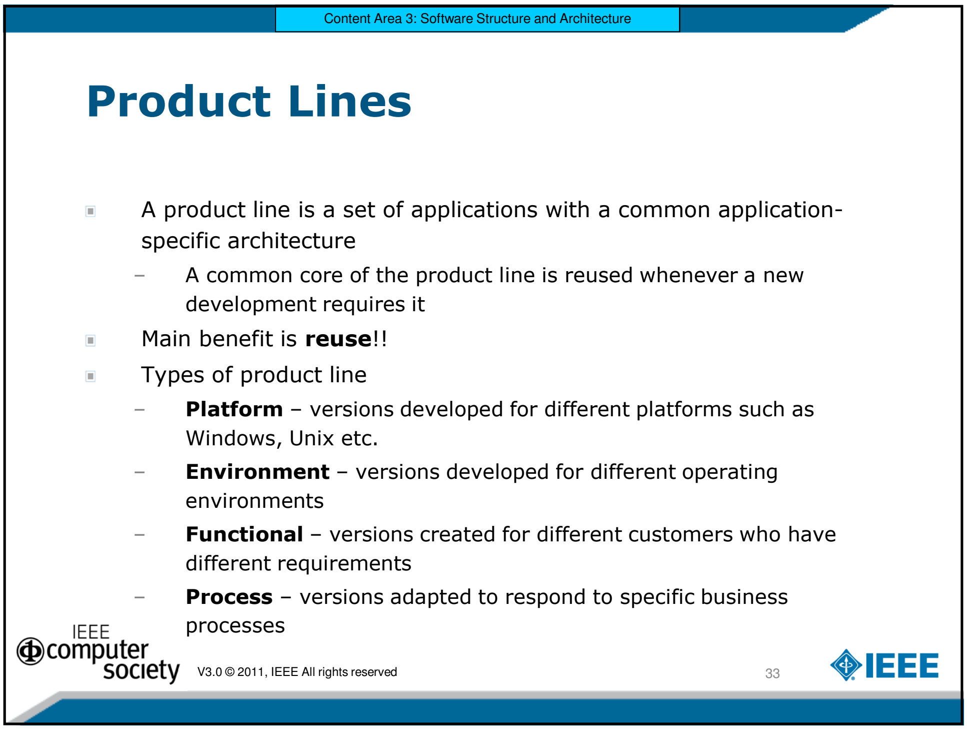
Implementation Frameworks

- Provide the technology organizations need to build and run various types of systems e.g. APIs such as J2EE, .NET



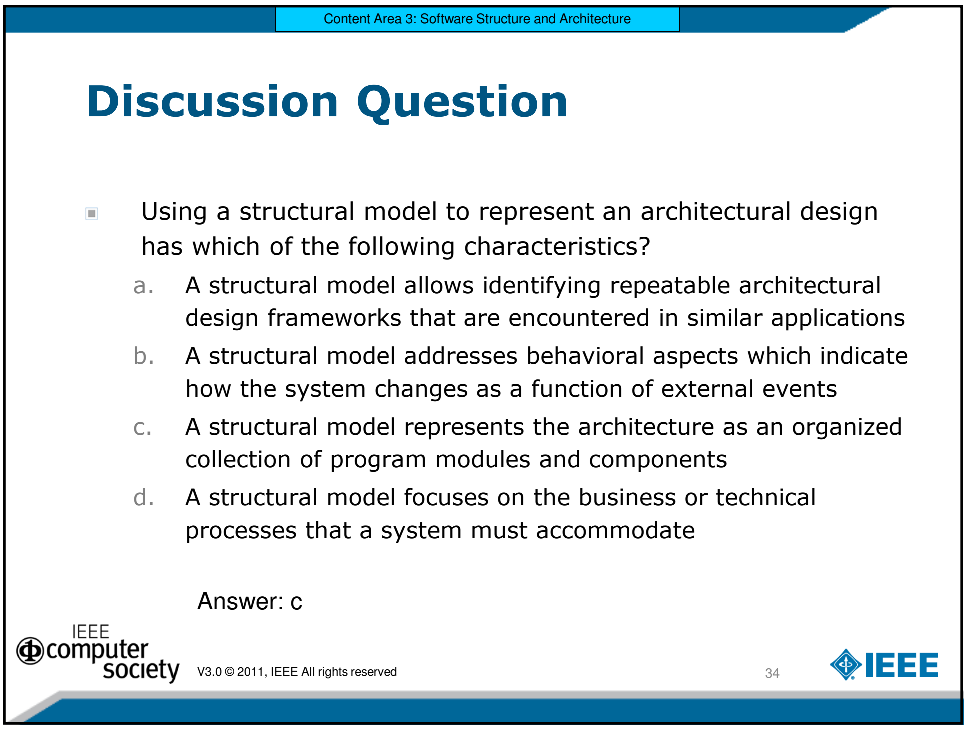
Sun's SAX API

References: 1. *DoD Architecture Framework V 1.5*, Department of Defense, USA (http://cio-nii.defense.gov)
2. *Java & XML*, by Brett McLaughlin



Product Lines

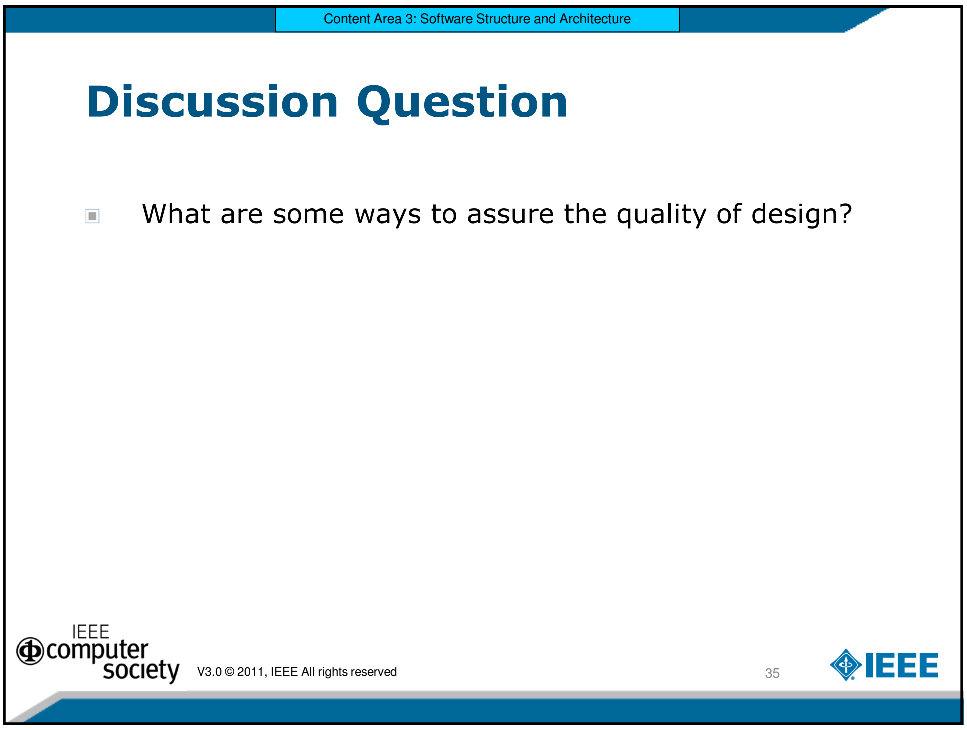
- A product line is a set of applications with a common application-specific architecture
  - A common core of the product line is reused whenever a new development requires it
- Main benefit is **reuse**!!
- Types of product line
  - **Platform** – versions developed for different platforms such as Windows, Unix etc.
  - **Environment** – versions developed for different operating environments
  - **Functional** – versions created for different customers who have different requirements
  - **Process** – versions adapted to respond to specific business processes



Discussion Question

- Using a structural model to represent an architectural design has which of the following characteristics?
  a. A structural model allows identifying repeatable architectural design frameworks that are encountered in similar applications
  b. A structural model addresses behavioral aspects which indicate how the system changes as a function of external events
  c. A structural model represents the architecture as an organized collection of program modules and components
  d. A structural model focuses on the business or technical processes that a system must accommodate

Answer: c

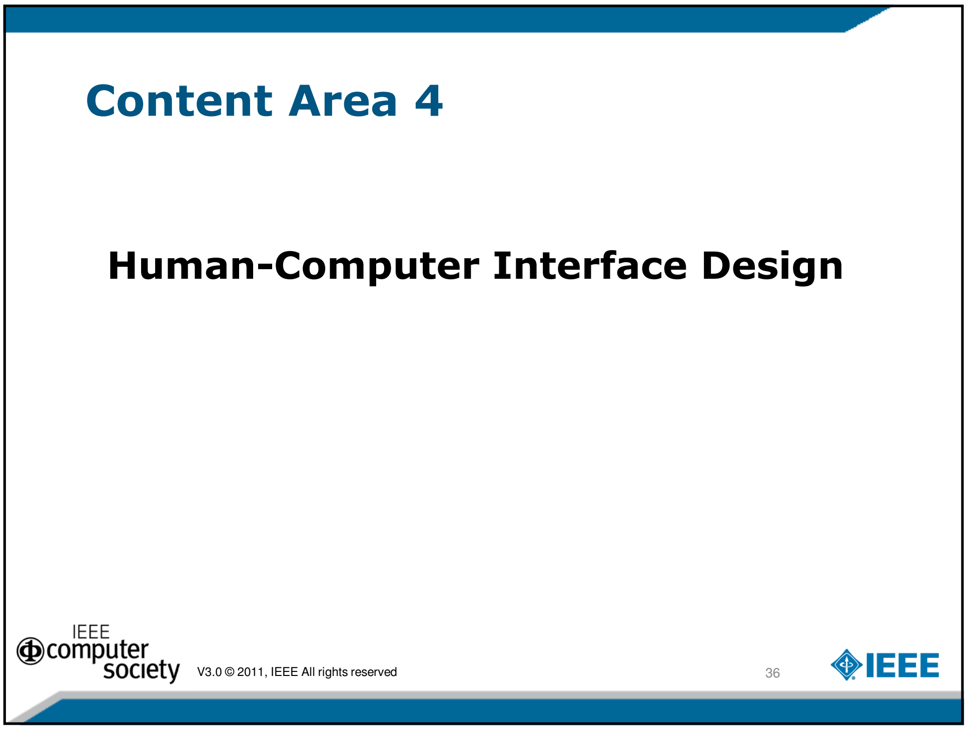


Content Area 3: Software Structure and Architecture

Discussion Question

- What are some ways to assure the quality of design?

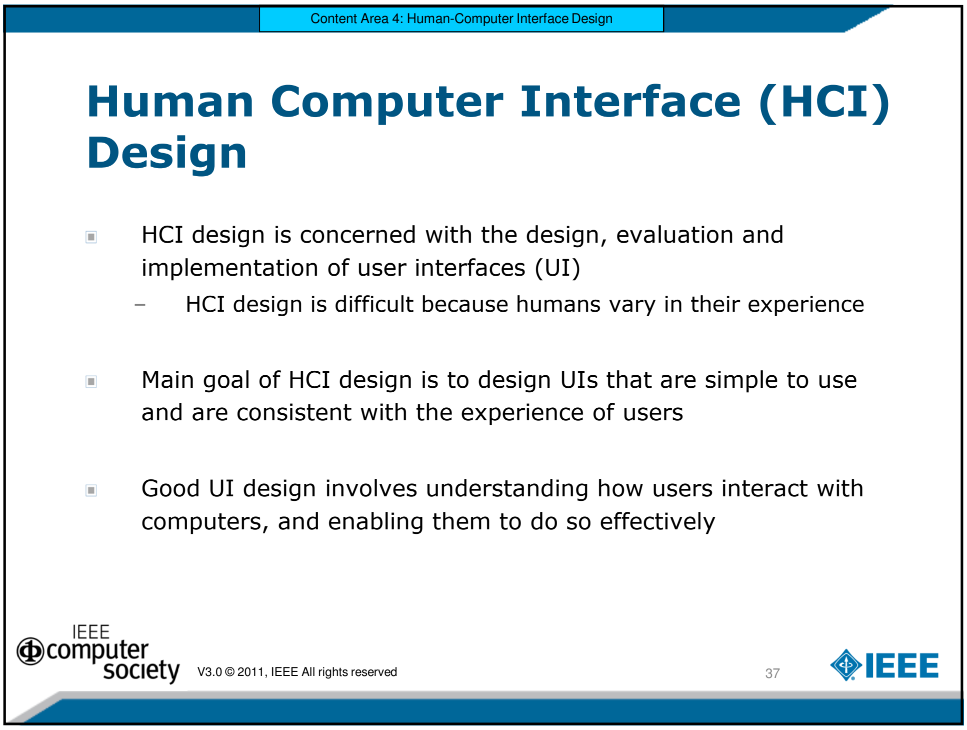
IEEE computer society V3.0 © 2011, IEEE All rights reserved 35



Content Area 4

Human-Computer Interface Design

IEEE computer society V3.0 © 2011, IEEE All rights reserved 36



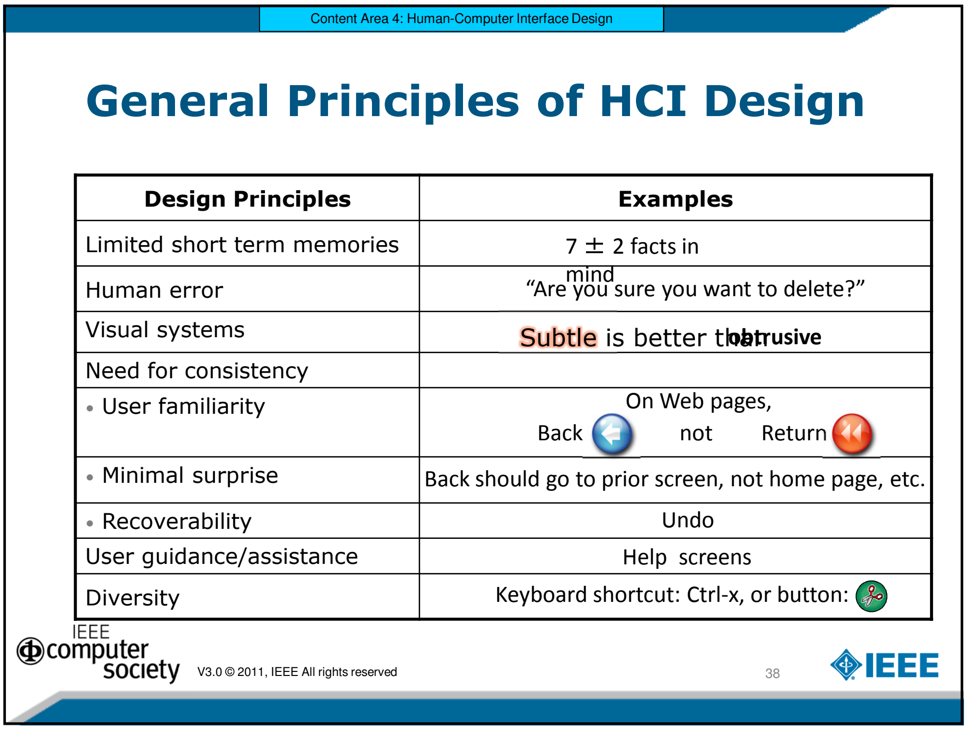
Content Area 4: Human-Computer Interface Design

Human Computer Interface (HCI) Design

- HCI design is concerned with the design, evaluation and implementation of user interfaces (UI)
  - HCI design is difficult because humans vary in their experience
- Main goal of HCI design is to design UIs that are simple to use and are consistent with the experience of users
- Good UI design involves understanding how users interact with computers, and enabling them to do so effectively

IEEE computer society

37 IEEE



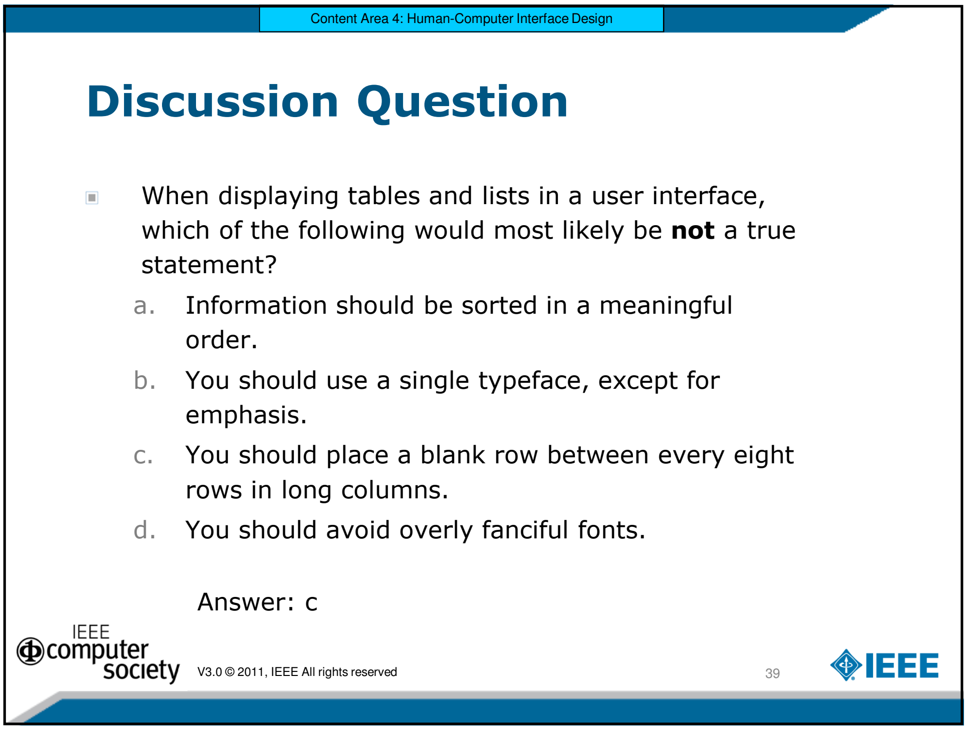
Content Area 4: Human-Computer Interface Design

General Principles of HCI Design

Design Principles	Examples
Limited short term memories	7 ± 2 facts in mind
Human error	"Are you sure you want to delete?"
Visual systems	Subtle is better than obtrusive
Need for consistency	
• User familiarity	On Web pages, Back ← not Return ⏪
• Minimal surprise	Back should go to prior screen, not home page, etc.
• Recoverability	Undo
User guidance/assistance	Help screens
Diversity	Keyboard shortcut: Ctrl-x, or button: ✂

IEEE computer society

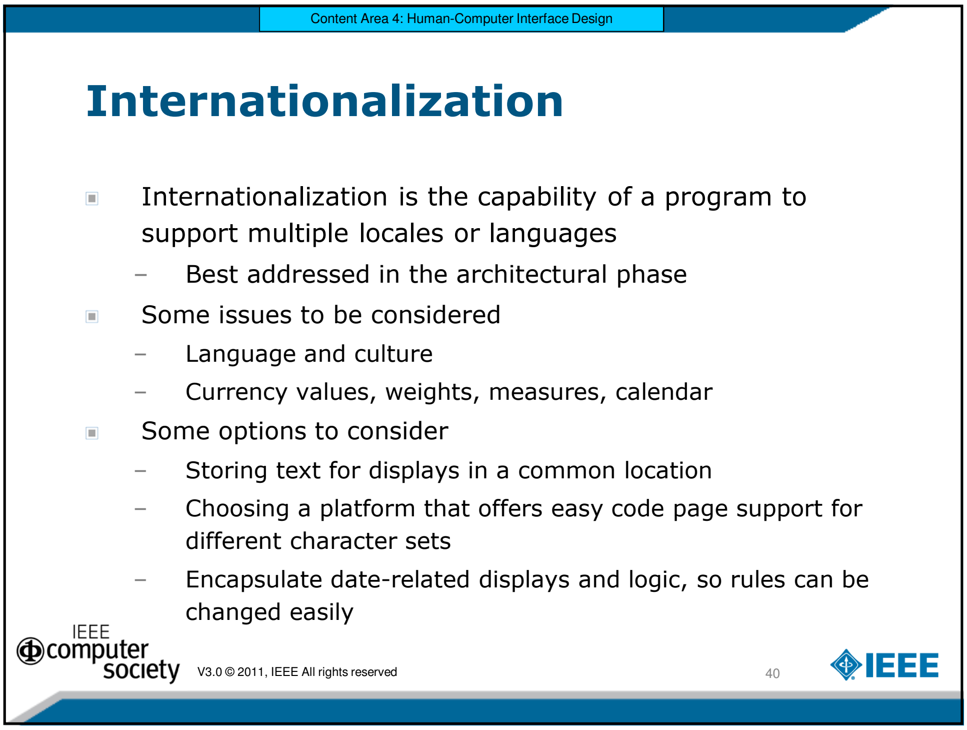
38 IEEE



Discussion Question

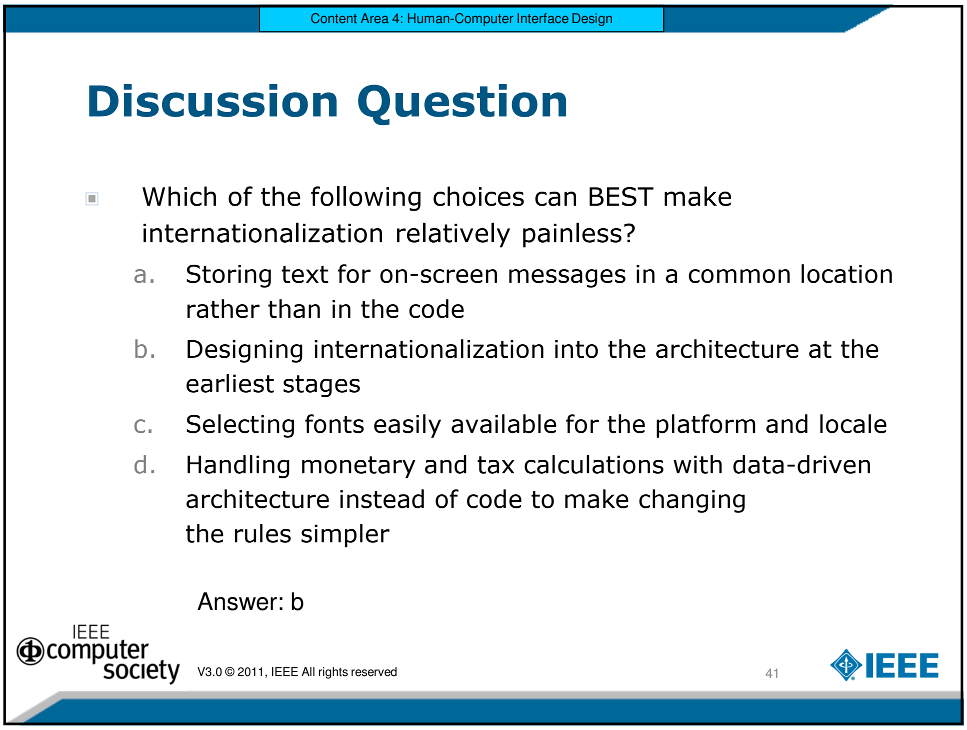
- When displaying tables and lists in a user interface, which of the following would most likely be **not** a true statement?
  a. Information should be sorted in a meaningful order.
  b. You should use a single typeface, except for emphasis.
  c. You should place a blank row between every eight rows in long columns.
  d. You should avoid overly fanciful fonts.

Answer: c



Internationalization

- Internationalization is the capability of a program to support multiple locales or languages
  - Best addressed in the architectural phase
- Some issues to be considered
  - Language and culture
  - Currency values, weights, measures, calendar
- Some options to consider
  - Storing text for displays in a common location
  - Choosing a platform that offers easy code page support for different character sets
  - Encapsulate date-related displays and logic, so rules can be changed easily



Content Area 4: Human-Computer Interface Design

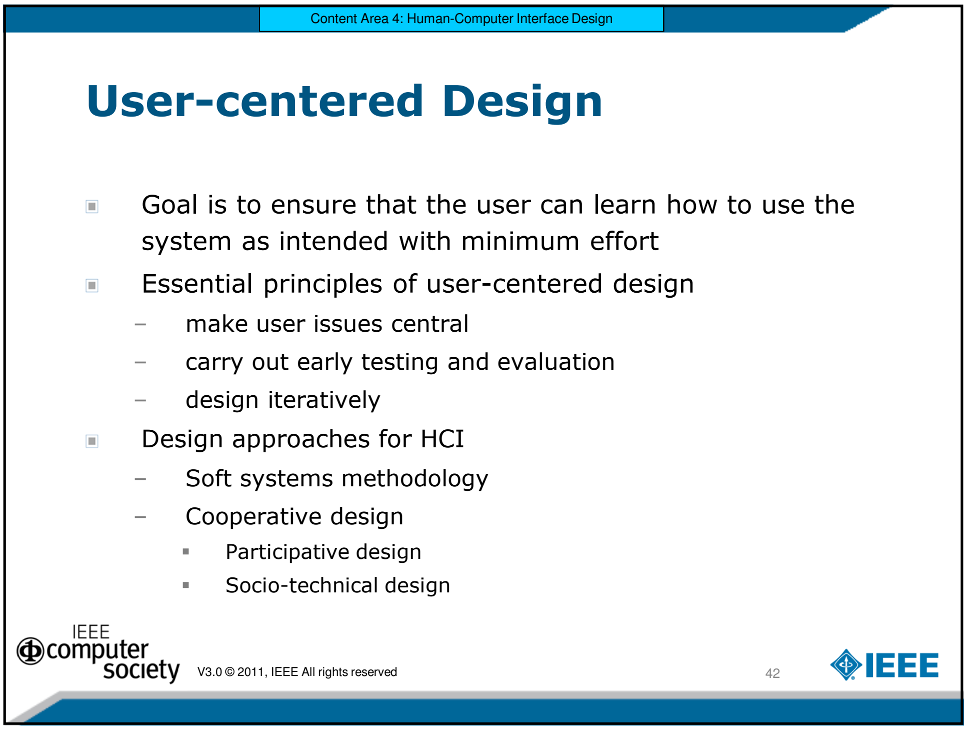
Discussion Question

- Which of the following choices can BEST make internationalization relatively painless?
  a. Storing text for on-screen messages in a common location rather than in the code
  b. Designing internationalization into the architecture at the earliest stages
  c. Selecting fonts easily available for the platform and locale
  d. Handling monetary and tax calculations with data-driven architecture instead of code to make changing the rules simpler

Answer: b

IEEE computer society

41



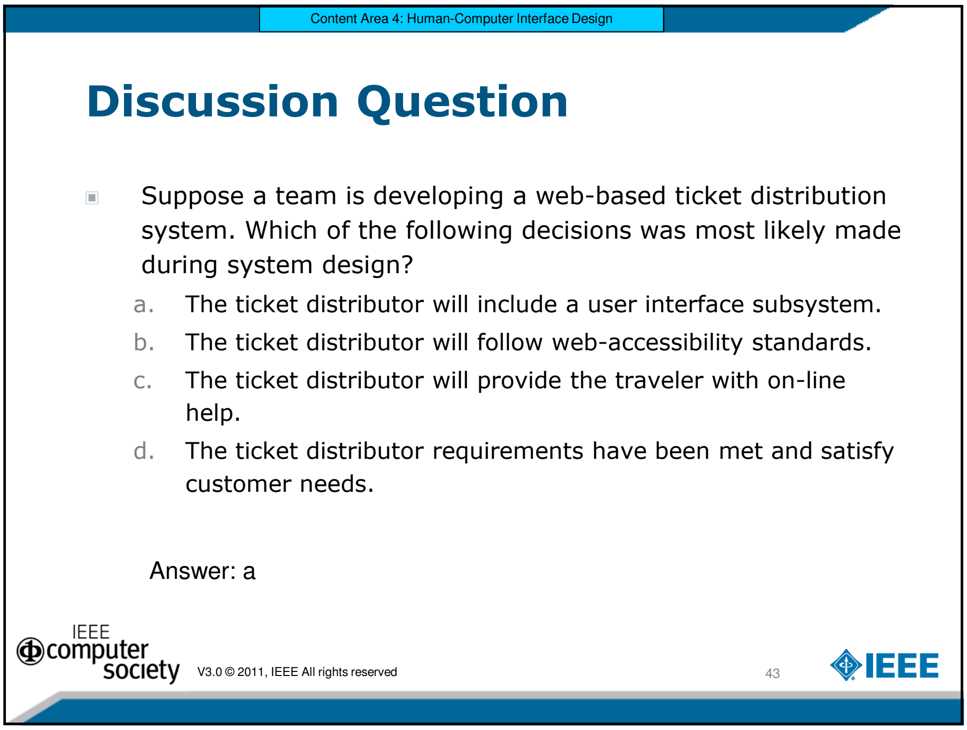
Content Area 4: Human-Computer Interface Design

User-centered Design

- Goal is to ensure that the user can learn how to use the system as intended with minimum effort
- Essential principles of user-centered design
  - make user issues central
  - carry out early testing and evaluation
  - design iteratively
- Design approaches for HCI
  - Soft systems methodology
  - Cooperative design
    - Participative design
    - Socio-technical design

IEEE computer society

42



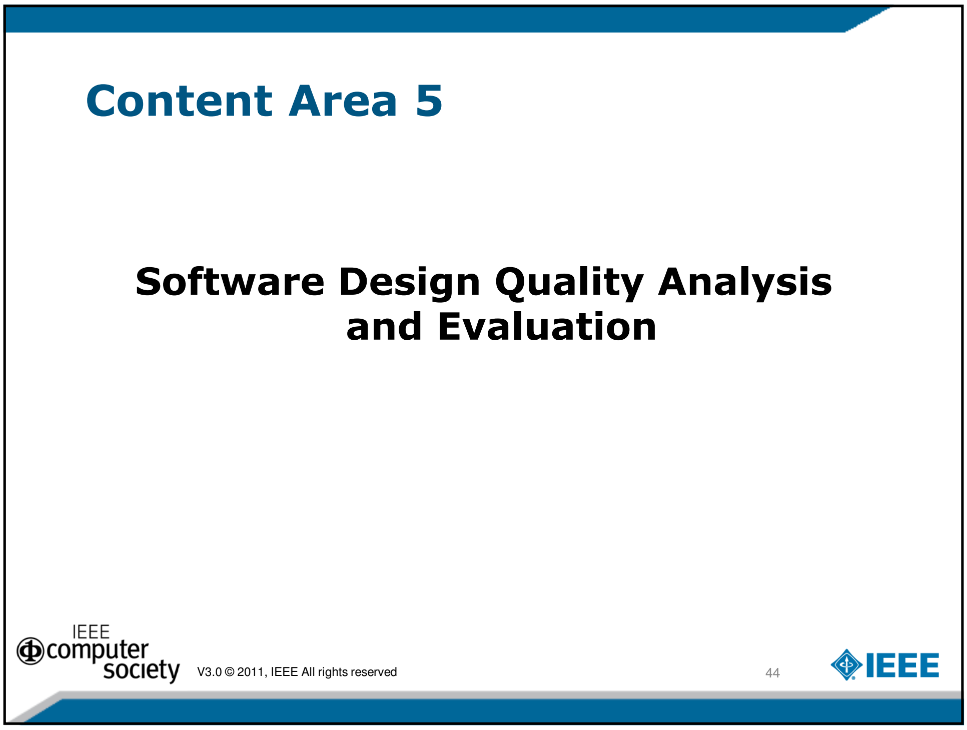
Content Area 4: Human-Computer Interface Design

Discussion Question

- Suppose a team is developing a web-based ticket distribution system. Which of the following decisions was most likely made during system design?
  a. The ticket distributor will include a user interface subsystem.
  b. The ticket distributor will follow web-accessibility standards.
  c. The ticket distributor will provide the traveler with on-line help.
  d. The ticket distributor requirements have been met and satisfy customer needs.

Answer: a

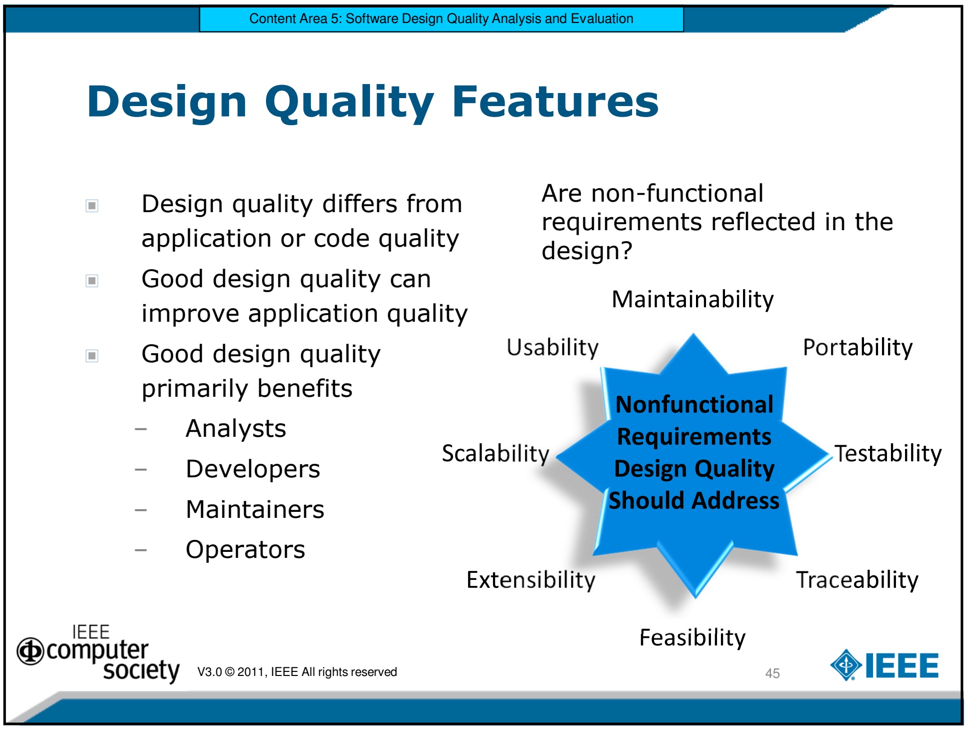
IEEE computer society



Content Area 5

Software Design Quality Analysis and Evaluation

IEEE computer society



Content Area 5: Software Design Quality Analysis and Evaluation

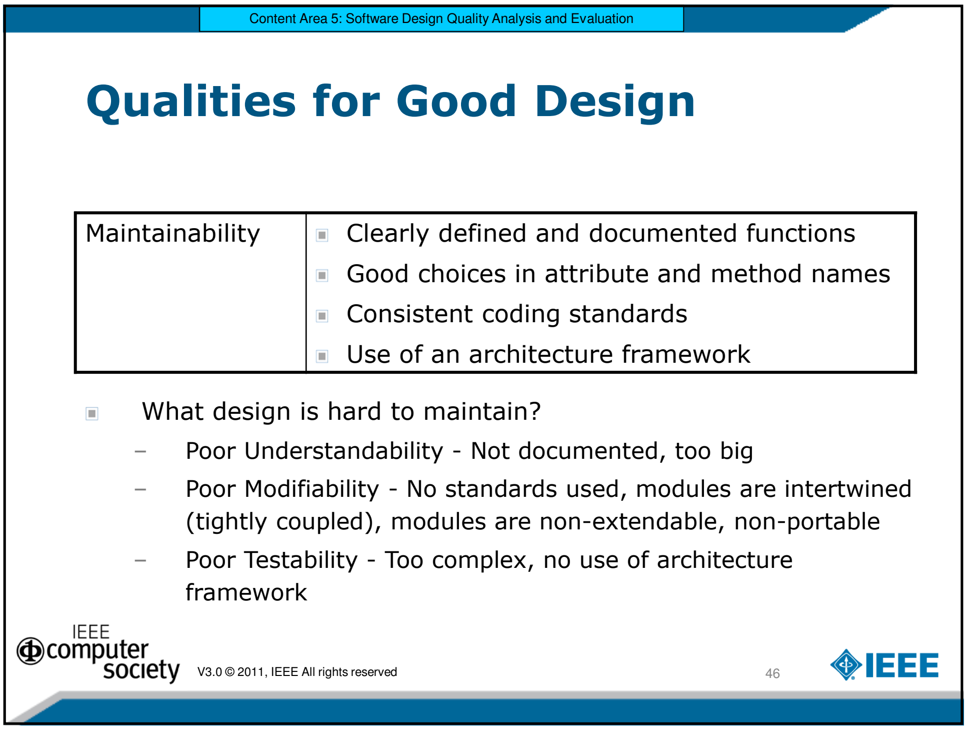
Design Quality Features

- Design quality differs from application or code quality
- Good design quality can improve application quality
- Good design quality primarily benefits
  - Analysts
  - Developers
  - Maintainers
  - Operators

Are non-functional requirements reflected in the design?

Maintainability
Usability
Portability
Nonfunctional Requirements Design Quality Should Address
Scalability
Testability
Extensibility
Traceability
Feasibility

IEEE computer society



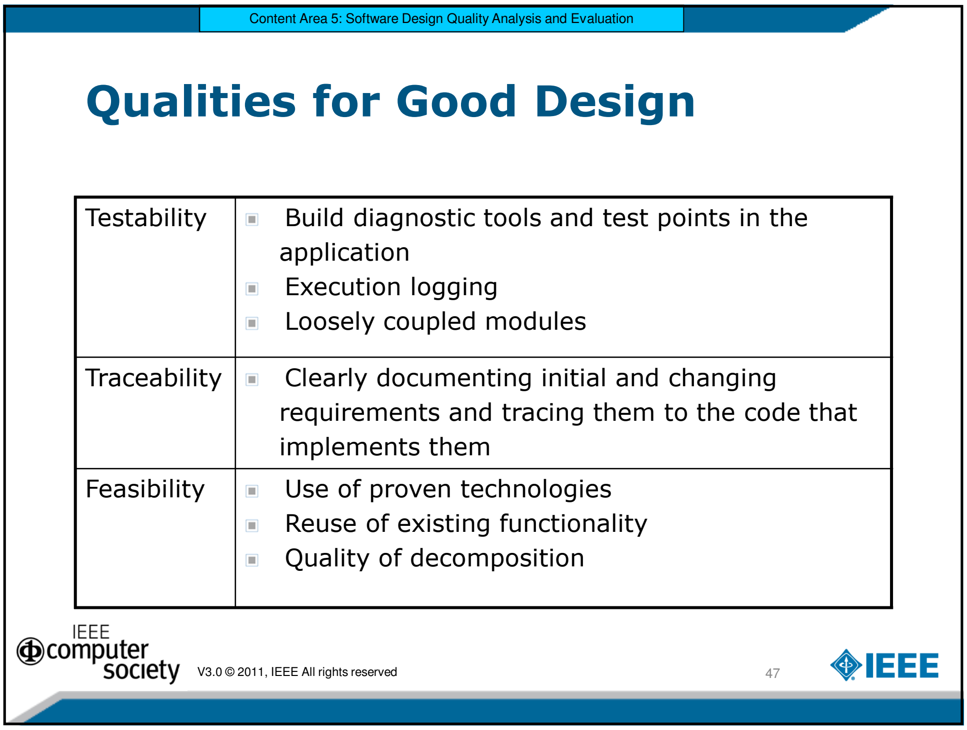
Content Area 5: Software Design Quality Analysis and Evaluation

Qualities for Good Design

Maintainability	Clearly defined and documented functions Good choices in attribute and method names Consistent coding standards Use of an architecture framework

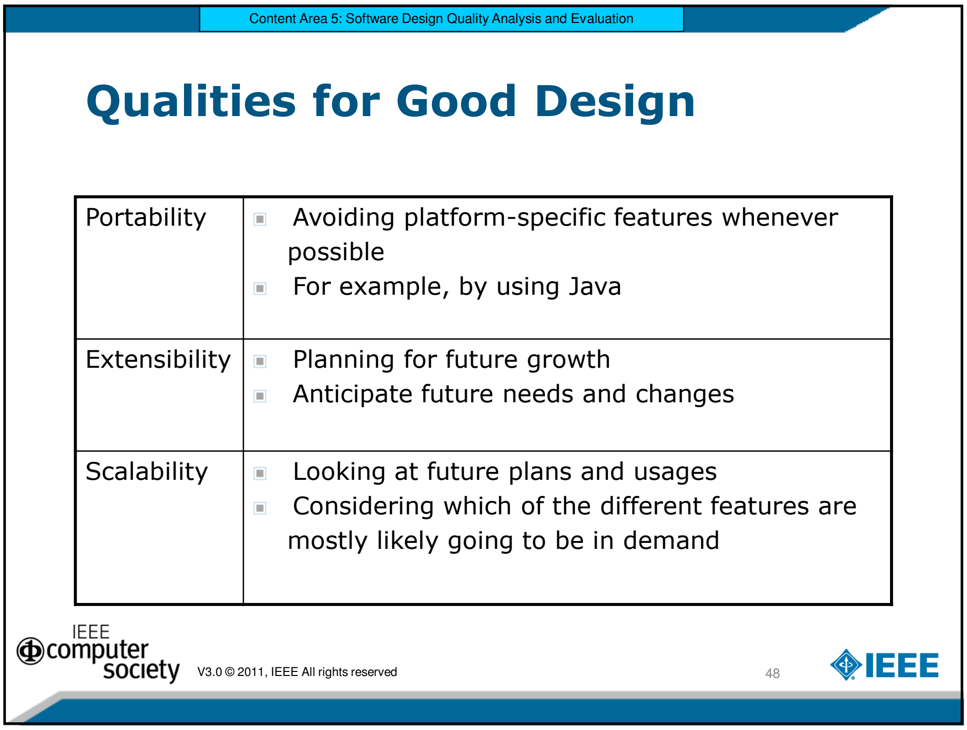
- What design is hard to maintain?
  - Poor Understandability - Not documented, too big
  - Poor Modifiability - No standards used, modules are intertwined (tightly coupled), modules are non-extendable, non-portable
  - Poor Testability - Too complex, no use of architecture framework

IEEE computer society



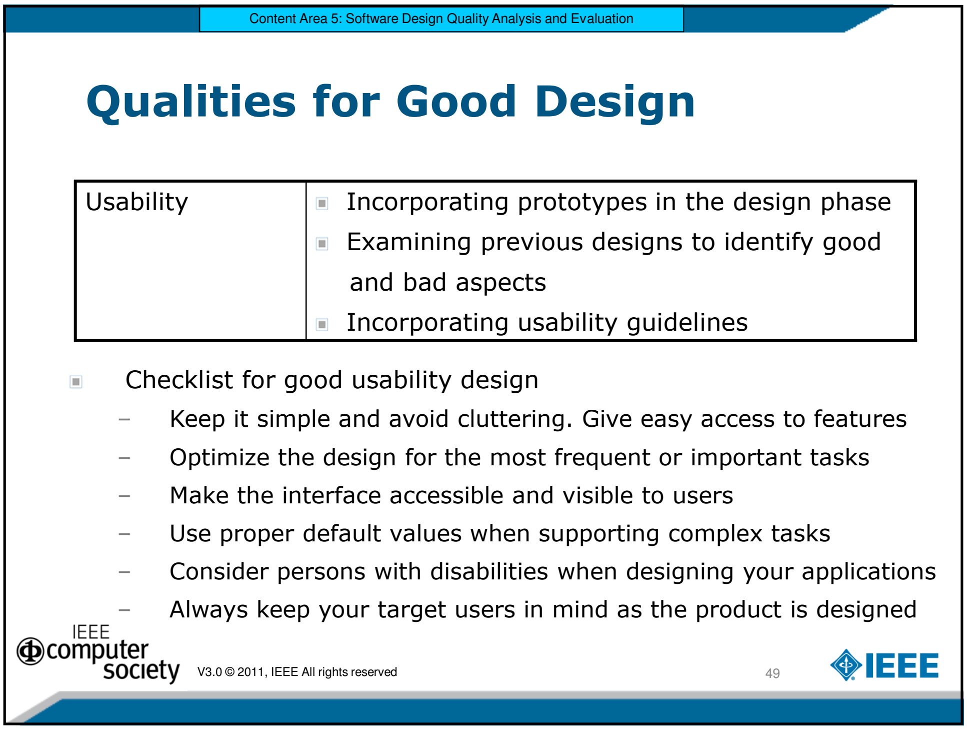
Qualities for Good Design

Testability	▣ Build diagnostic tools and test points in the application ▣ Execution logging ▣ Loosely coupled modules
Traceability	▣ Clearly documenting initial and changing requirements and tracing them to the code that implements them
Feasibility	▣ Use of proven technologies ▣ Reuse of existing functionality ▣ Quality of decomposition



Qualities for Good Design

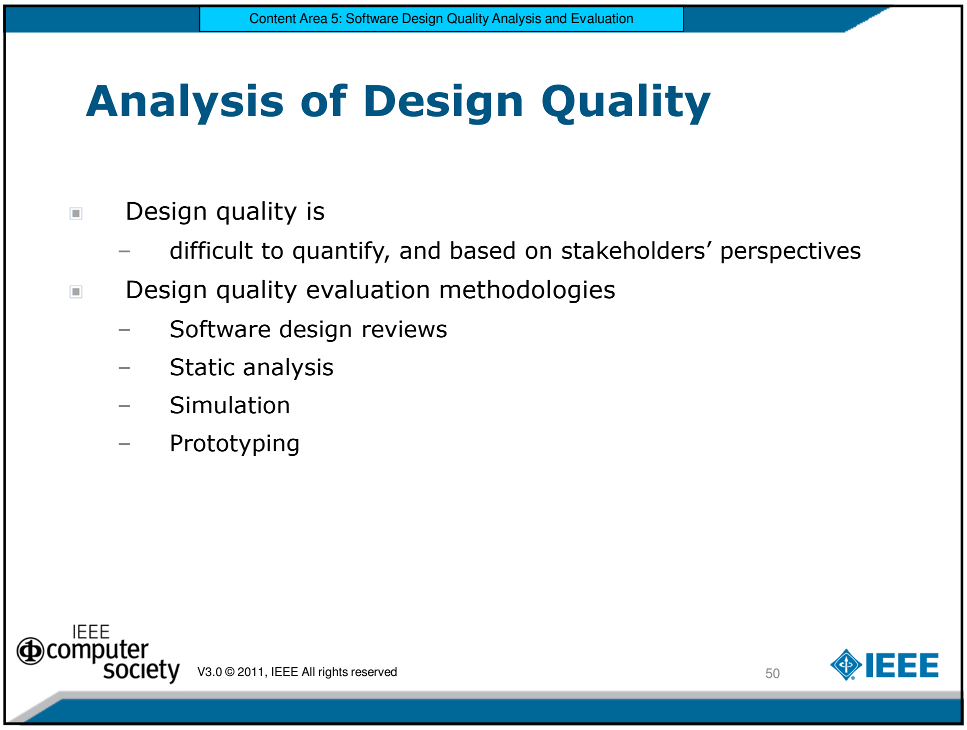
Portability	▣ Avoiding platform-specific features whenever possible ▣ For example, by using Java
Extensibility	▣ Planning for future growth ▣ Anticipate future needs and changes
Scalability	▣ Looking at future plans and usages ▣ Considering which of the different features are mostly likely going to be in demand



Qualities for Good Design

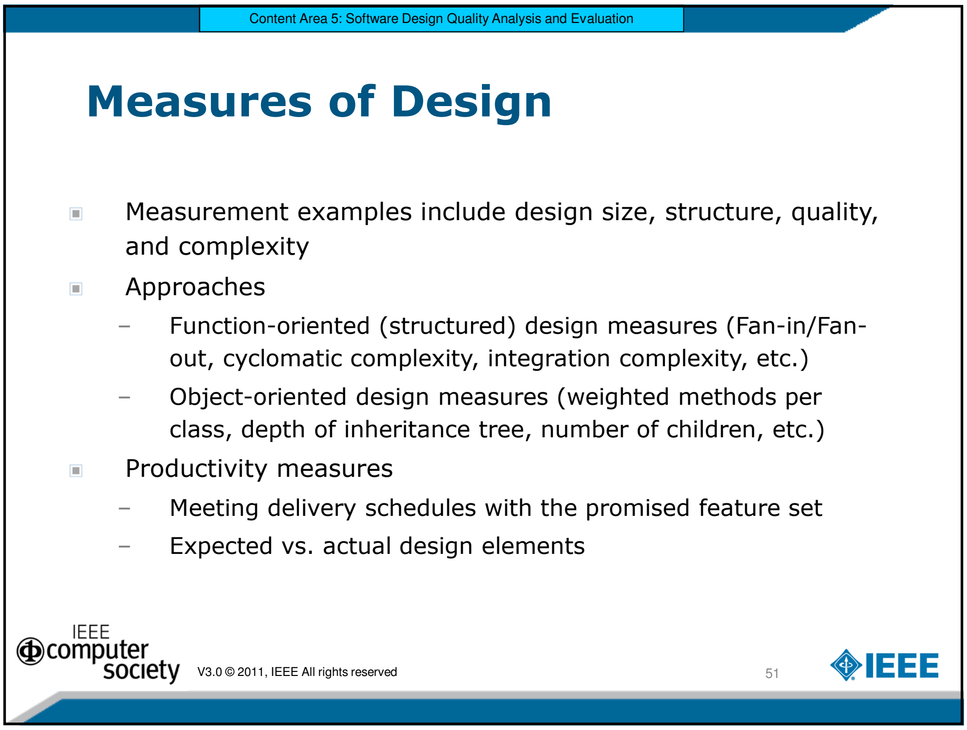
Usability	▣ Incorporating prototypes in the design phase ▣ Examining previous designs to identify good and bad aspects ▣ Incorporating usability guidelines

- Checklist for good usability design
  - Keep it simple and avoid cluttering. Give easy access to features
  - Optimize the design for the most frequent or important tasks
  - Make the interface accessible and visible to users
  - Use proper default values when supporting complex tasks
  - Consider persons with disabilities when designing your applications
  - Always keep your target users in mind as the product is designed



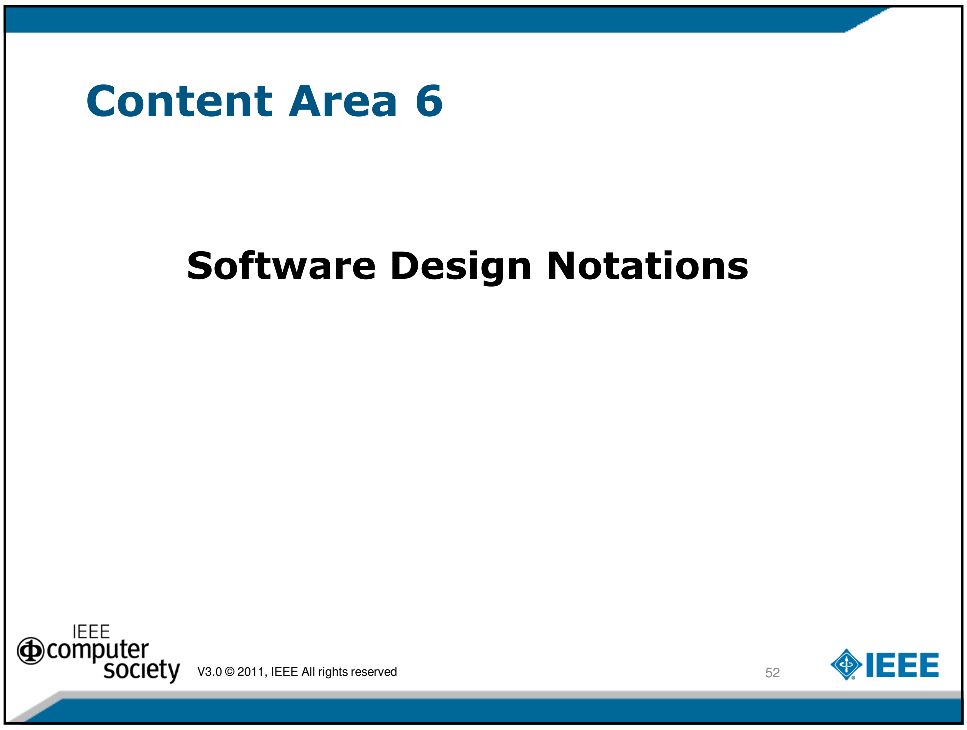
Analysis of Design Quality

- Design quality is
  - difficult to quantify, and based on stakeholders’ perspectives
- Design quality evaluation methodologies
  - Software design reviews
  - Static analysis
  - Simulation
  - Prototyping



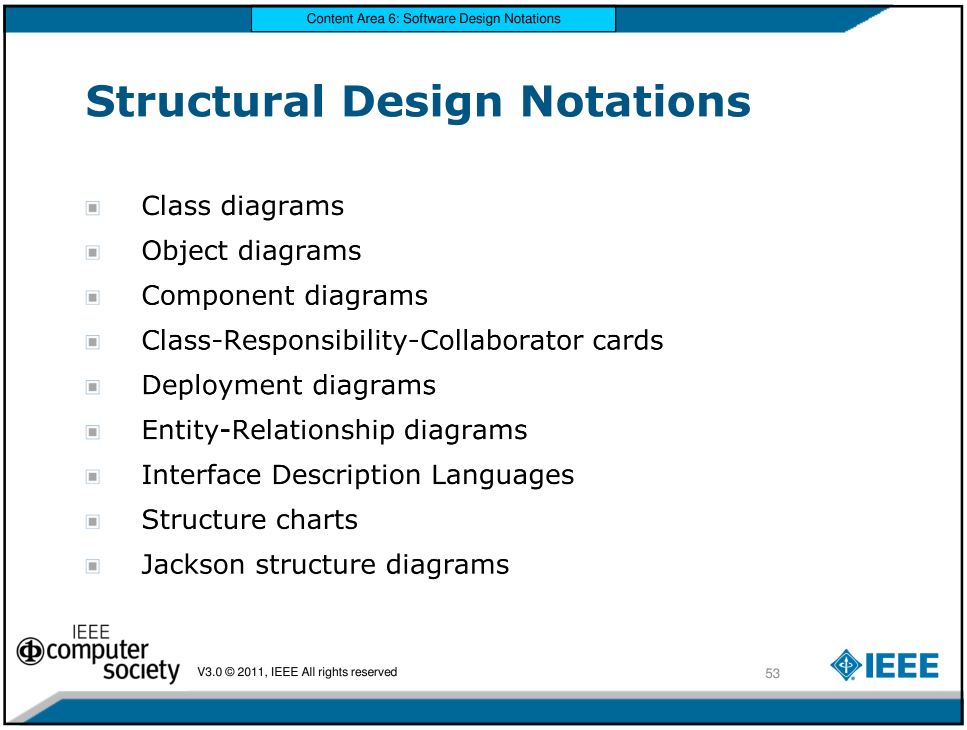
Measures of Design

- Measurement examples include design size, structure, quality, and complexity
- Approaches
  - Function-oriented (structured) design measures (Fan-in/Fan-out, cyclomatic complexity, integration complexity, etc.)
  - Object-oriented design measures (weighted methods per class, depth of inheritance tree, number of children, etc.)
- Productivity measures
  - Meeting delivery schedules with the promised feature set
  - Expected vs. actual design elements



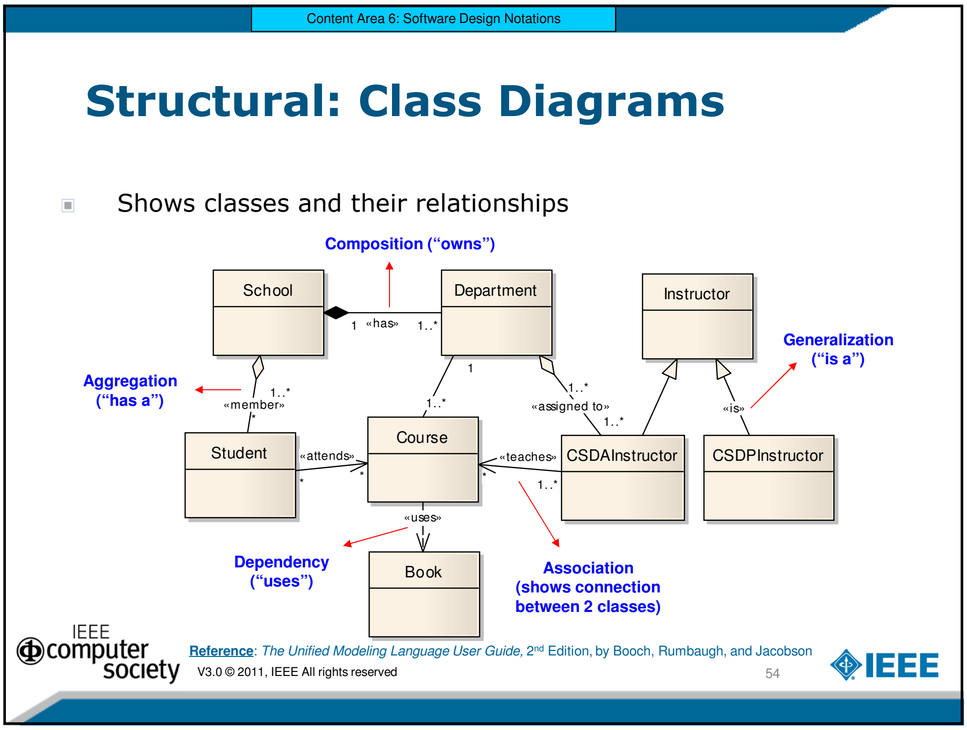
Content Area 6

Software Design Notations



Structural Design Notations

- Class diagrams
- Object diagrams
- Component diagrams
- Class-Responsibility-Collaborator cards
- Deployment diagrams
- Entity-Relationship diagrams
- Interface Description Languages
- Structure charts
- Jackson structure diagrams



Structural: Class Diagrams

- Shows classes and their relationships

Reference: *The Unified Modeling Language User Guide*, 2nd Edition, by Booch, Rumbaugh, and Jacobson

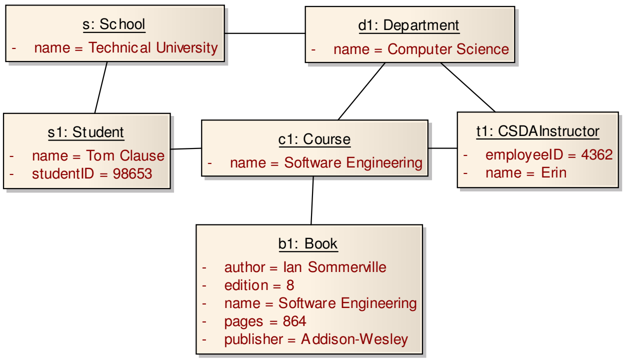
Discussion Question

- Class diagrams in object-oriented design use the following types of relationships:
  a. generalization
  b. association
  c. aggregation
  d. all of the above

Answer: d

Structural: Object Diagrams

- Shows a set of instantiated objects, their state, and their relationships at a point in time



s: School
- name = Technical University

d1: Department
- name = Computer Science

s1: Student
- name = Tom Clause
- studentID = 98653

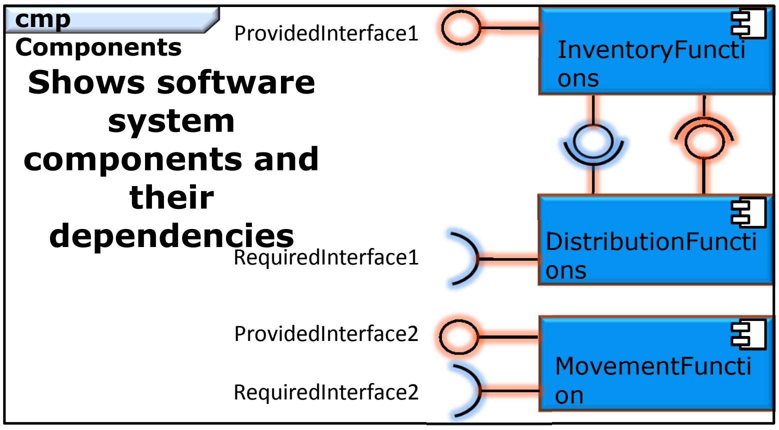
c1: Course
- name = Software Engineering

t1: CSDAInstructor
- employeeID = 4362
- name = Erin

b1: Book
- author = Ian Sommerville
- edition = 8
- name = Software Engineering
- pages = 864
- publisher = Addison-Wesley

Reference: *The Unified Modeling Language User Guide*, 2nd Edition, by Booch, Rumbaugh, and Jacobson

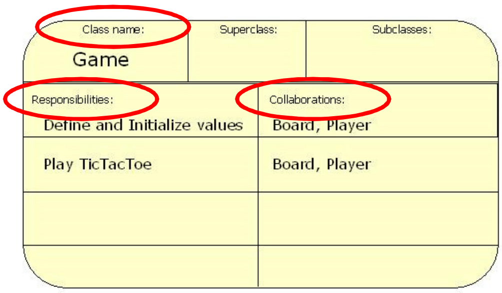
Structural: Component Diagrams



Shows software system components and their dependencies

Structural: CRC Cards

- Class-Responsibility-Collaboration (CRC) cards are 3" x 5" physical cards that contain
  - Class name
  - Class responsibilities
  - Names of other classes that the class will collaborate with to fulfill its responsibilities
- Cards with similar capabilities or responsibilities are moved around and grouped into potential classes



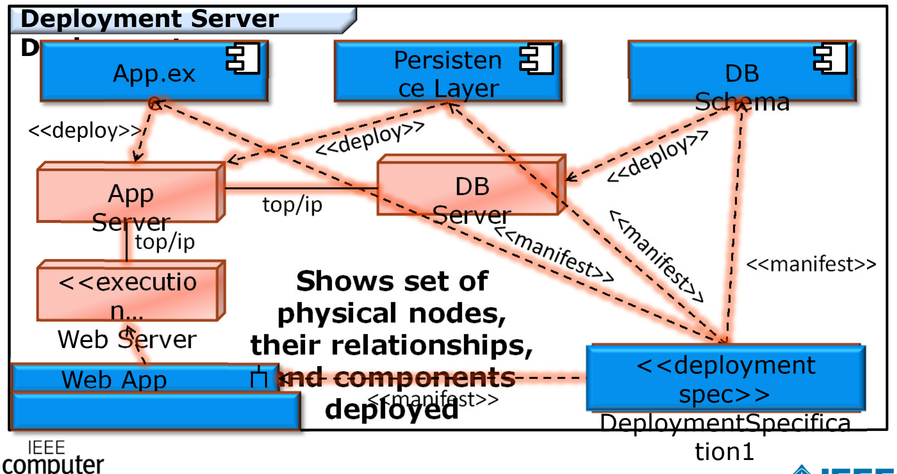
Class name: Game	Superclass:	Subclasses:
Responsibilities:	Collaborations:	
Define and Initialize values	Board, Player	
Play TicTacToe	Board, Player	

Discussion Question

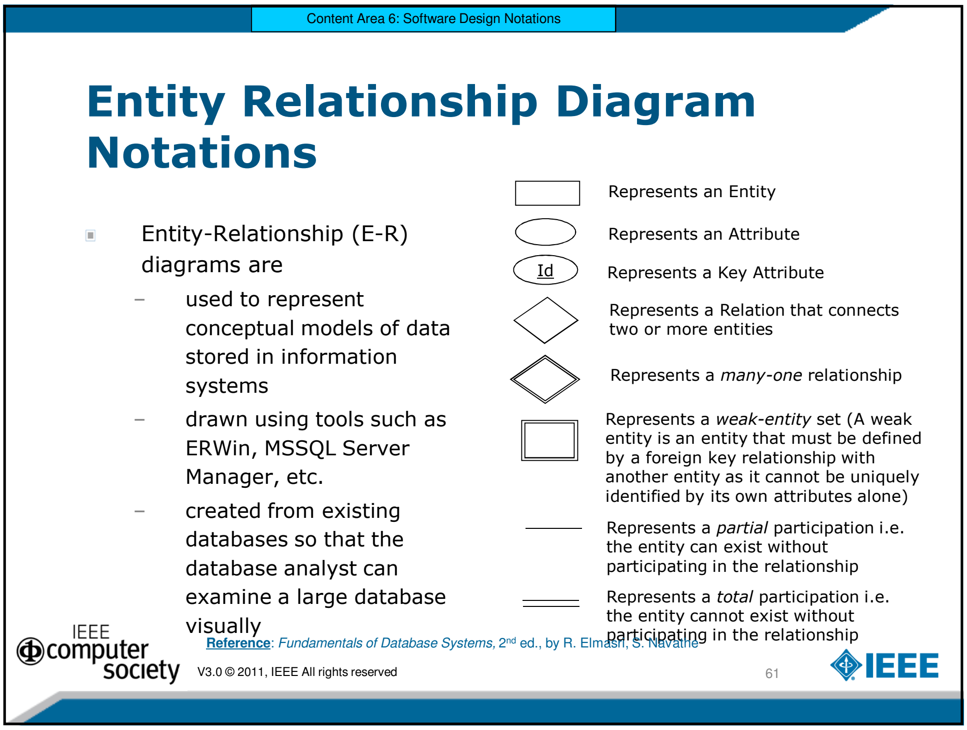
- Which of the following is true of the class responsibility collaborator card (CRC) ?
  a. CRC is the final notation form for classes
  b. CRC cards with similar capabilities or responsibilities may be physically grouped
  c. CRC format allows for lengthy descriptions
  d. CRC is a purely electronic effort

Answer: b

Structural: Deployment Diagrams



Shows set of physical nodes, their relationships, and components deployed

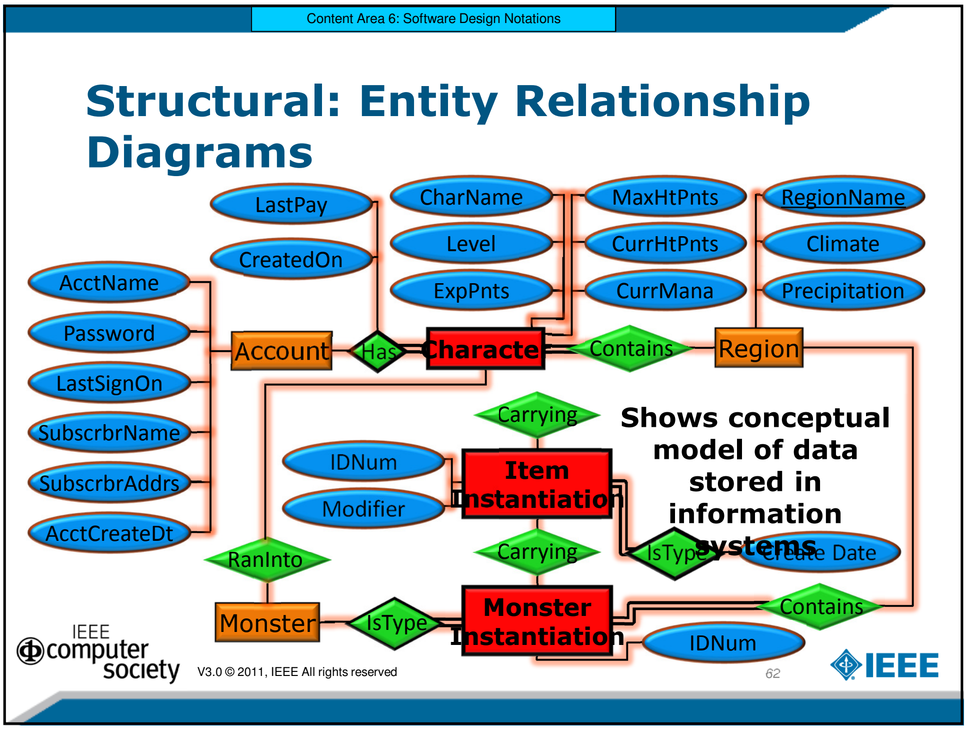


Entity Relationship Diagram Notations

- Entity-Relationship (E-R) diagrams are
  - used to represent conceptual models of data stored in information systems
  - drawn using tools such as ERWin, MSSQL Server Manager, etc.
  - created from existing databases so that the database analyst can examine a large database visually

▭	Represents an Entity
⬭	Represents an Attribute
Id	Represents a Key Attribute
◇	Represents a Relation that connects two or more entities
◈	Represents a *many-one* relationship
▣	Represents a *weak-entity* set (A weak entity is an entity that must be defined by a foreign key relationship with another entity as it cannot be uniquely identified by its own attributes alone)
—	Represents a *partial* participation i.e. the entity can exist without participating in the relationship
═	Represents a *total* participation i.e. the entity cannot exist without participating in the relationship

Reference: *Fundamentals of Database Systems, 2nd ed., by R. Elmasri, S. Navathe*



Structural: Entity Relationship Diagrams

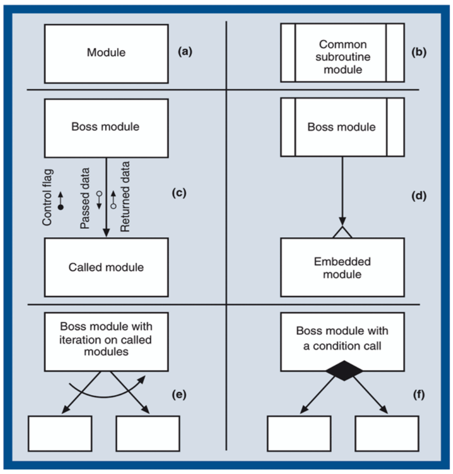
Shows conceptual model of data stored in information systems

Structural: IDLs

- Interface Description Languages (IDL) are programming-like languages used to define the interface of software components
  - The language specifies operations, parameters and parameter ranges, and data types
- Why use IDLs?
  - The interfaces of a component that is going to be exposed outside its original application need to be explicitly defined
  - If the interfaces is expressed in an IDL, the IDL code provides a defined filter, especially for in-bound requests and data, stopping out of bounds data from entering the component and causing mysterious failures/crashes
- **Example**: Object Management Group (OMG) format IDL

```
Interface inventory_qty{
      int get_qty(in int qty_on_hand);
}
```

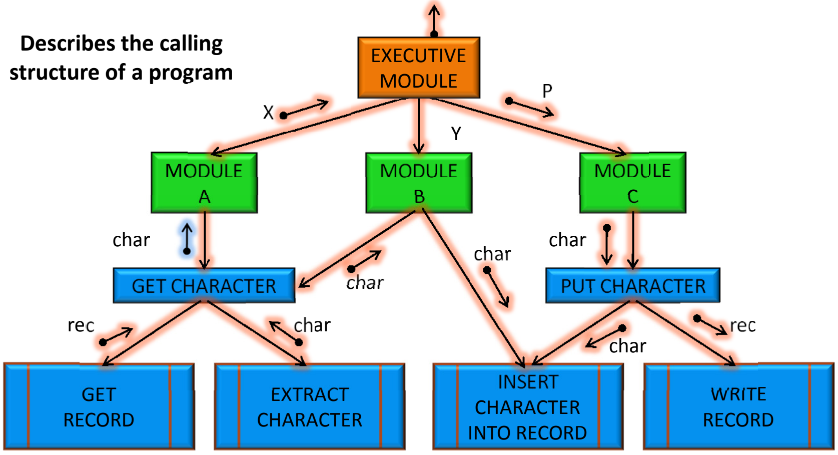
Structure Chart Symbols



Reference: *Systems Analysis and Design in a Changing World,* 3rd Edition, by Satzinger, Jackson and Burd

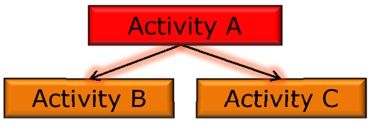
Structural: Structure Charts

Describes the calling structure of a program



Structural: Jackson Structure Diagrams

- Describes the data structures manipulated by a program in terms of
  - sequence (sequence of activities)
  - selection (a conditional activity)
  - iteration (a repetitive activity)
- First introduced in Jackson Structured Programming (JSP)
- Process of design is as follows
  - Identify the data structures and data flows
  - Next, derive the system functions required to maintain the data and move it around

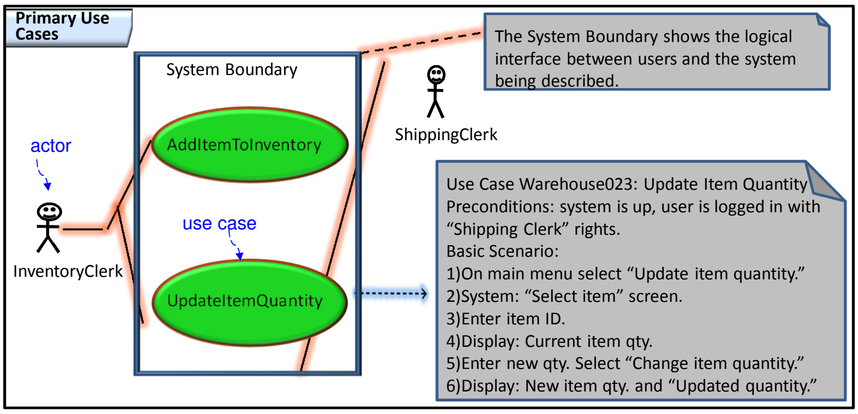


Behavioral Design Notations

- Use-case diagrams
- Activity diagrams
- Collaboration diagrams
- Data-flow diagrams
- Decision tables
- Flowcharts
- Sequence (event-tracing) diagrams
- State diagrams
- Formal specification languages
- Pseudocode

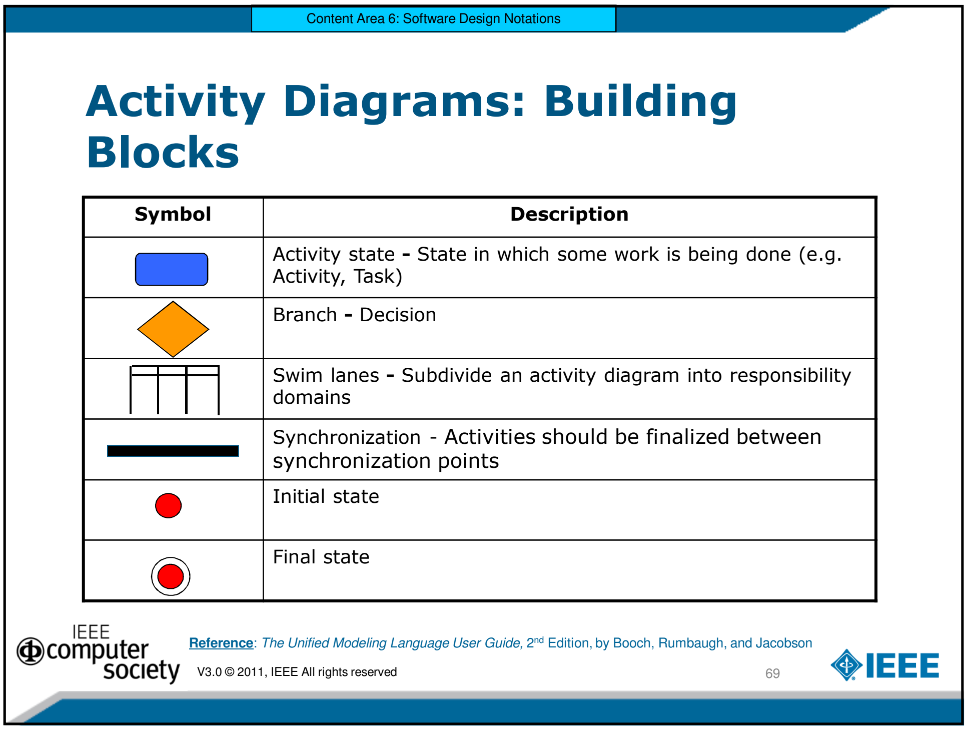
Behavioral: Use-case Diagrams

- Shows who (users/other systems) interacts with the system and how



The System Boundary shows the logical interface between users and the system being described.

Use Case Warehouse023: Update Item Quantity
Preconditions: system is up, user is logged in with “Shipping Clerk” rights.
Basic Scenario:
1)On main menu select “Update item quantity.”
2)System: “Select item” screen.
3)Enter item ID.
4)Display: Current item qty.
5)Enter new qty. Select “Change item quantity.”
6)Display: New item qty. and “Updated quantity.”



Content Area 6: Software Design Notations

Activity Diagrams: Building Blocks

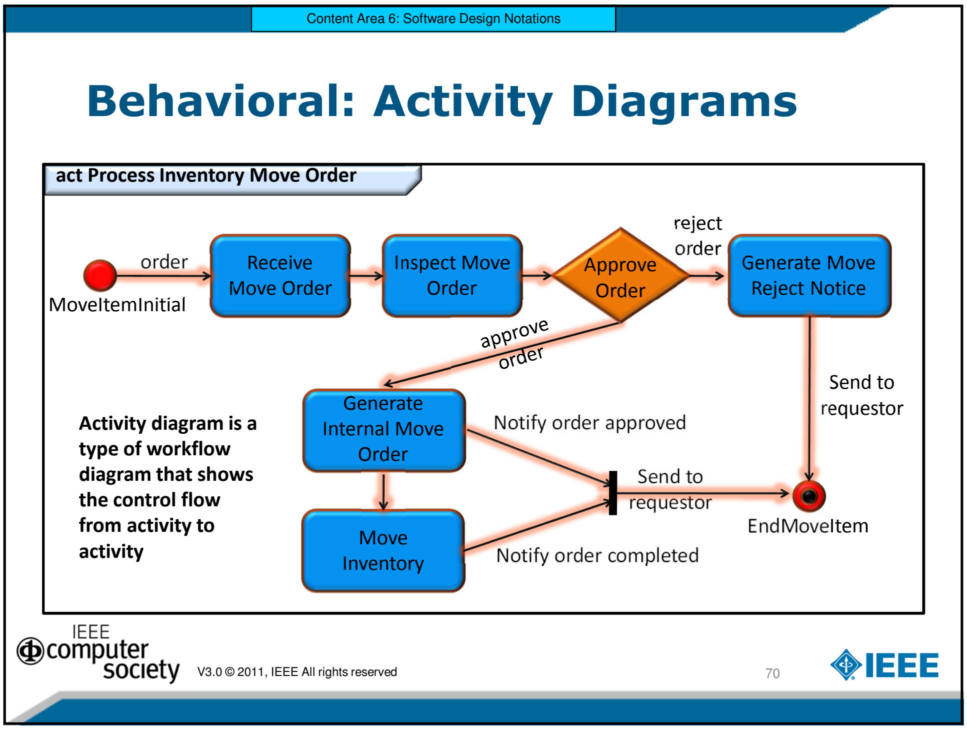
Symbol	Description
▬	Activity state - State in which some work is being done (e.g. Activity, Task)
◆	Branch - Decision
┬┬┬	Swim lanes - Subdivide an activity diagram into responsibility domains
━	Synchronization - Activities should be finalized between synchronization points
●	Initial state
◉	Final state

IEEE computer society

Reference: *The Unified Modeling Language User Guide,* 2nd Edition, by Booch, Rumbaugh, and Jacobson

69

IEEE



Content Area 6: Software Design Notations

Behavioral: Activity Diagrams

Activity diagram is a type of workflow diagram that shows the control flow from activity to activity

IEEE computer society

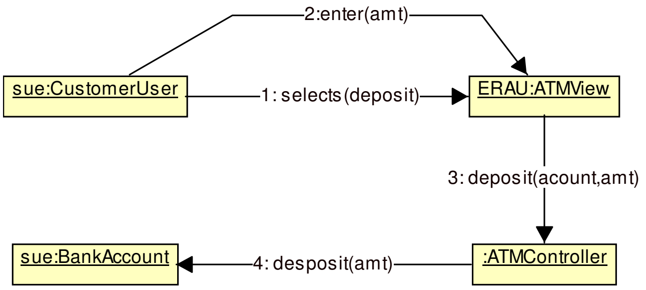
70

IEEE

Content Area 6: Software Design Notations

Behavioral: Collaboration Diagrams

- Shows the interactions that occur among a group of objects, where the emphasis is on the objects, their links, and the messages they exchange on these links

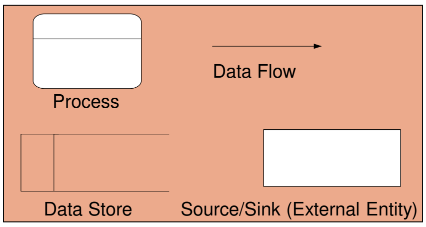


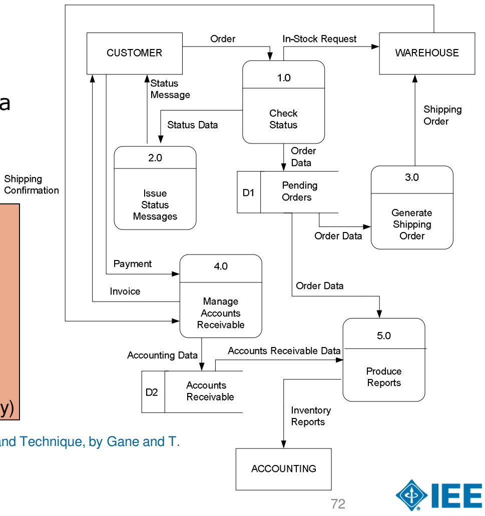
Reference: *The Unified Modeling Language User Guide,* 2nd Edition, by Booch, Rumbaugh, and Jacobson

Content Area 6: Software Design Notations

Behavioral: Data Flow Diagrams

- Data flow diagram (DFD) is a picture of the movement of data between external entities and the processes and data stores within a system





Reference: Structured Systems Analysis: Tools and Technique, by Gane and T. Sarson

DFD Rules

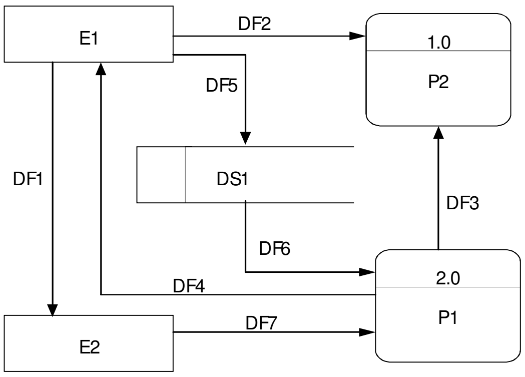
Data Flow that connects

	YES	NO
A process to another process	✓	
A process to an external entity	✓	
A process to a data store	✓	
An external entity to another external entity		✓
An external entity to a data store		✓
A data store to another data store		✓

Additional Rule: A process must have at least one input and one output data flow

Discussion Question

- List the errors of this DFD



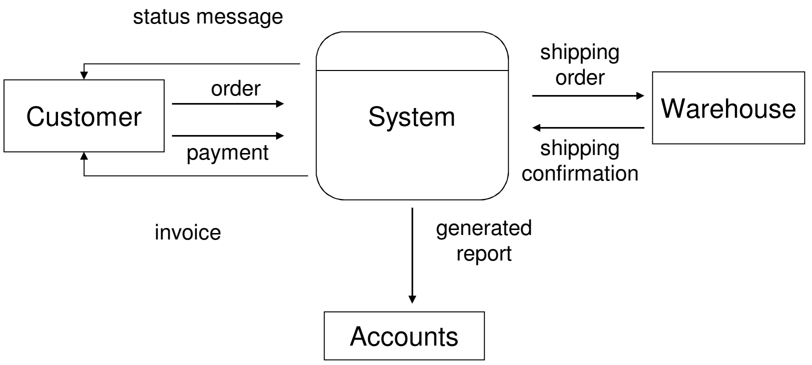
Answer:

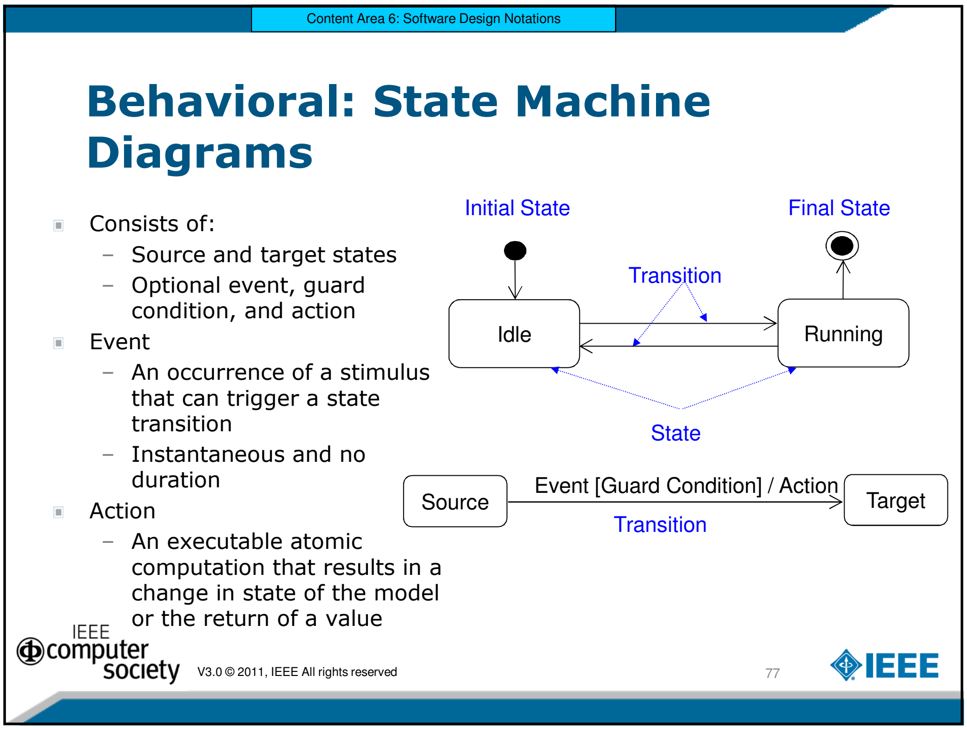
- DF1
- DF5
- P2 has no outputs (the rule says that a process should have at least one input and at least one output)

Discussion Question

- International Motors Ltd., sells a line of vehicles such as cars, vans, and SUVs. When customers place orders on the company's web site, the system
  - checks to see if the items are in stock,
  - issues a status message to the customer, and
  - generates a shipping order to the warehouse, which fills the order.
- When the order is shipped, the customer is billed. The system also produces various reports.
- **Draw DFD diagram Level 0 for the order system**

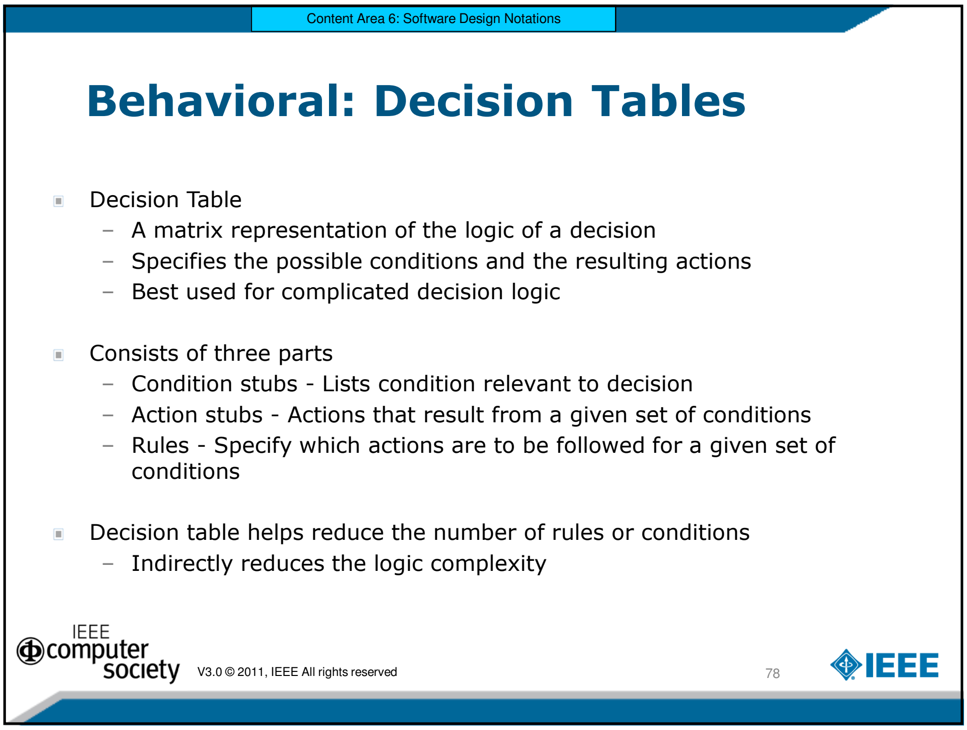
Answer for Level 0





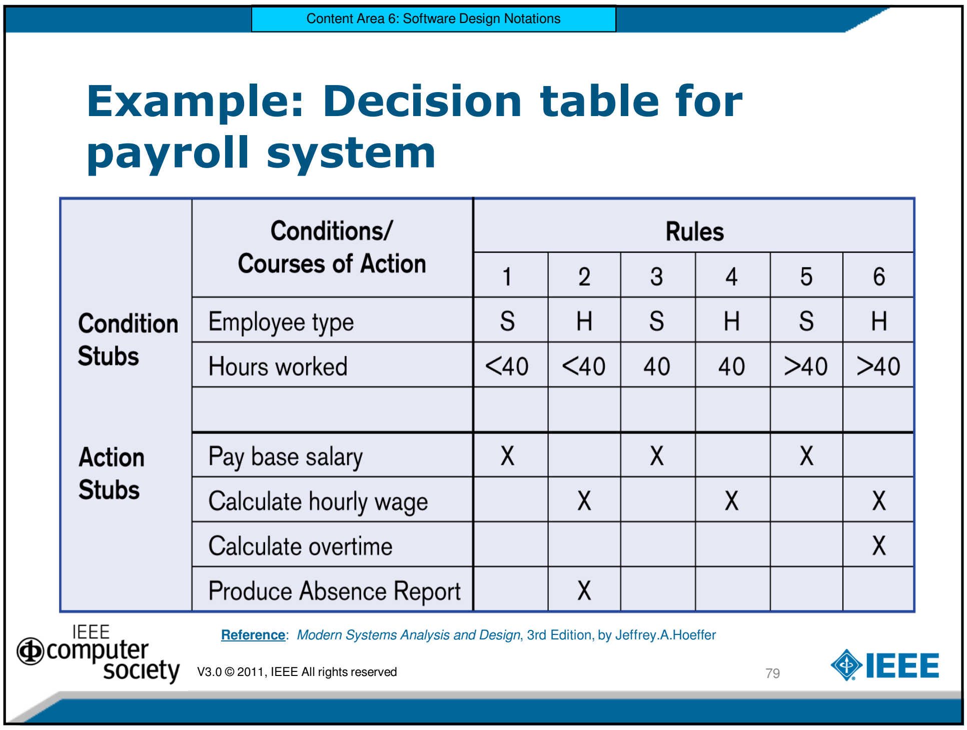
Behavioral: State Machine Diagrams

- Consists of:
  - Source and target states
  - Optional event, guard condition, and action
- Event
  - An occurrence of a stimulus that can trigger a state transition
  - Instantaneous and no duration
- Action
  - An executable atomic computation that results in a change in state of the model or the return of a value



Behavioral: Decision Tables

- Decision Table
  - A matrix representation of the logic of a decision
  - Specifies the possible conditions and the resulting actions
  - Best used for complicated decision logic
- Consists of three parts
  - Condition stubs - Lists condition relevant to decision
  - Action stubs - Actions that result from a given set of conditions
  - Rules - Specify which actions are to be followed for a given set of conditions
- Decision table helps reduce the number of rules or conditions
  - Indirectly reduces the logic complexity



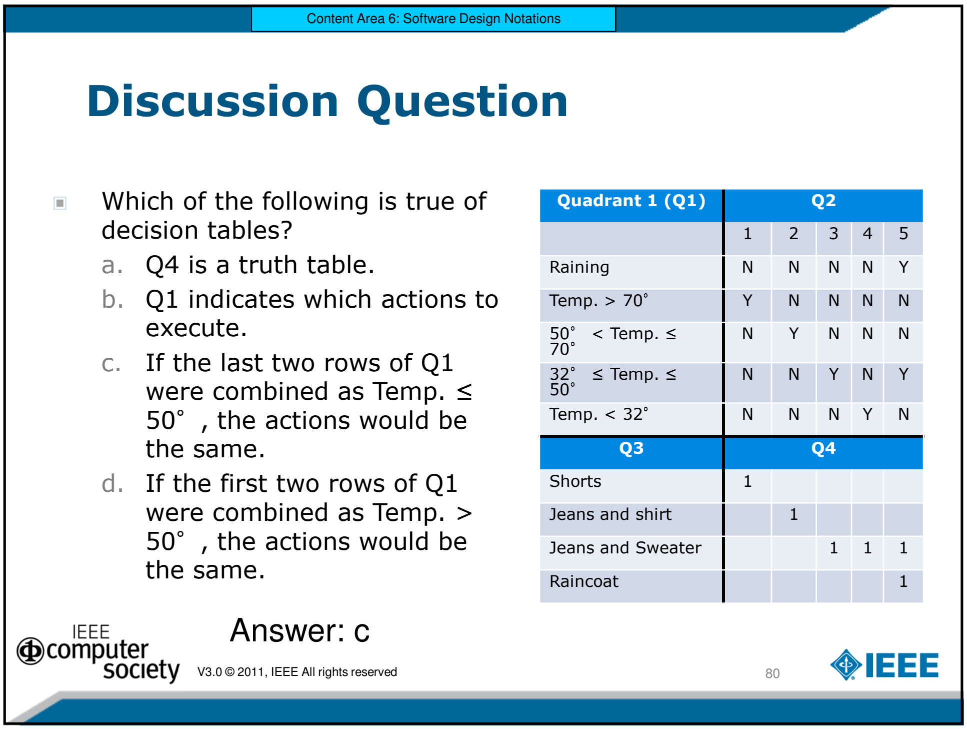
Content Area 6: Software Design Notations

Example: Decision table for payroll system

	Conditions/ Courses of Action	Rules					
		1	2	3	4	5	6
Condition Stubs	Employee type	S	H	S	H	S	H
	Hours worked	<40	<40	40	40	>40	>40
Action Stubs	Pay base salary	X		X		X	
	Calculate hourly wage		X		X		X
	Calculate overtime						X
	Produce Absence Report		X				

Reference: *Modern Systems Analysis and Design*, 3rd Edition, by Jeffrey.A.Hoeffer

IEEE computer society



Content Area 6: Software Design Notations

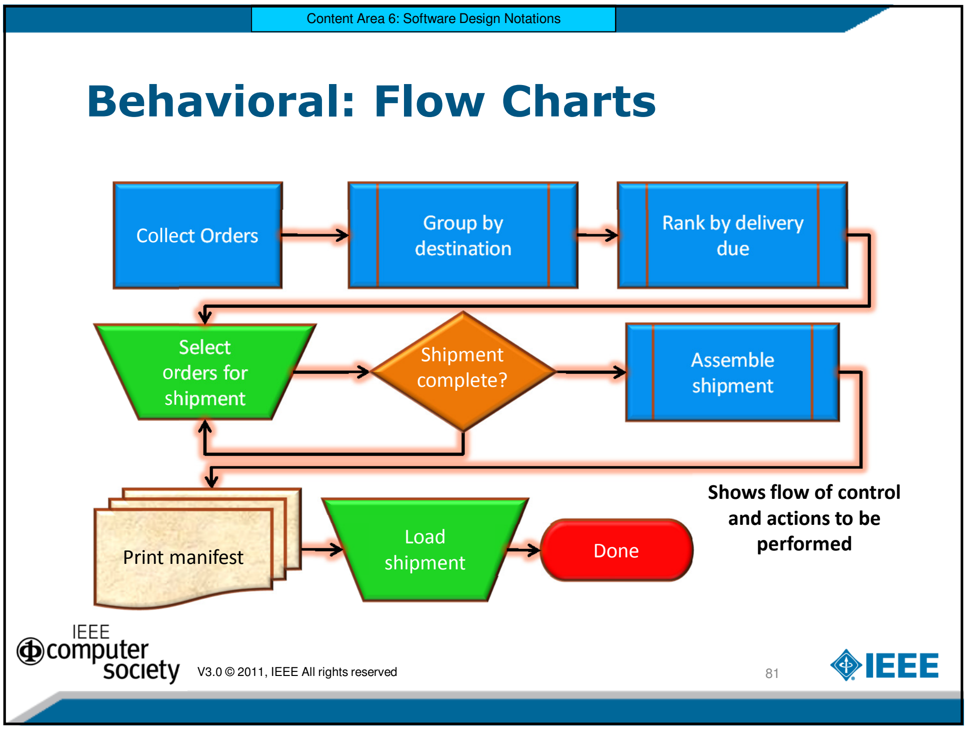
Discussion Question

- Which of the following is true of decision tables?
  a. Q4 is a truth table.
  b. Q1 indicates which actions to execute.
  c. If the last two rows of Q1 were combined as Temp. ≤ 50° , the actions would be the same.
  d. If the first two rows of Q1 were combined as Temp. > 50° , the actions would be the same.

Quadrant 1 (Q1)	Q2				
	1	2	3	4	5
Raining	N	N	N	N	Y
Temp. > 70°	Y	N	N	N	N
50° < Temp. ≤ 70°	N	Y	N	N	N
32° ≤ Temp. ≤ 50°	N	N	Y	N	Y
Temp. < 32°	N	N	N	Y	N
Q3	**Q4**				
Shorts	1				
Jeans and shirt		1			
Jeans and Sweater			1	1	1
Raincoat					1

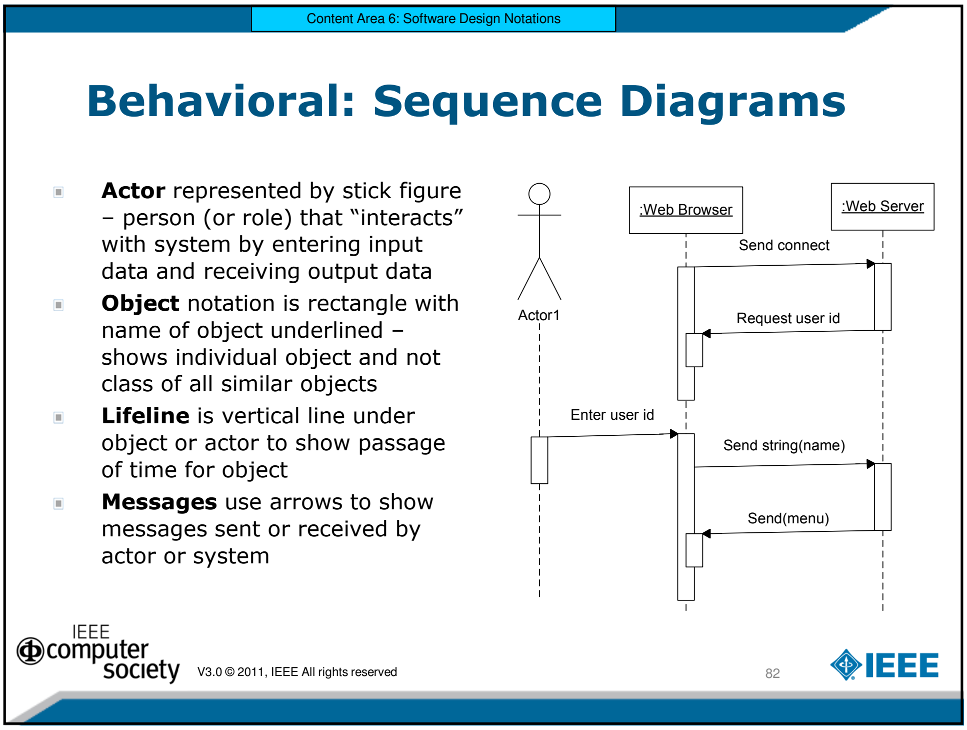
Answer: c

IEEE computer society



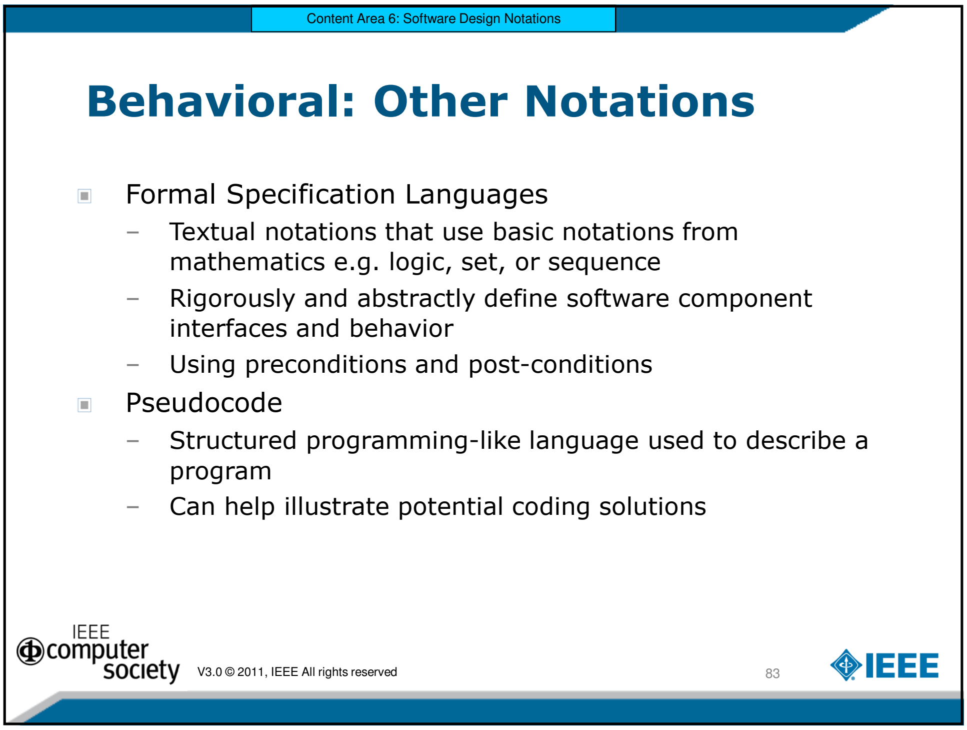
Behavioral: Flow Charts

Shows flow of control and actions to be performed



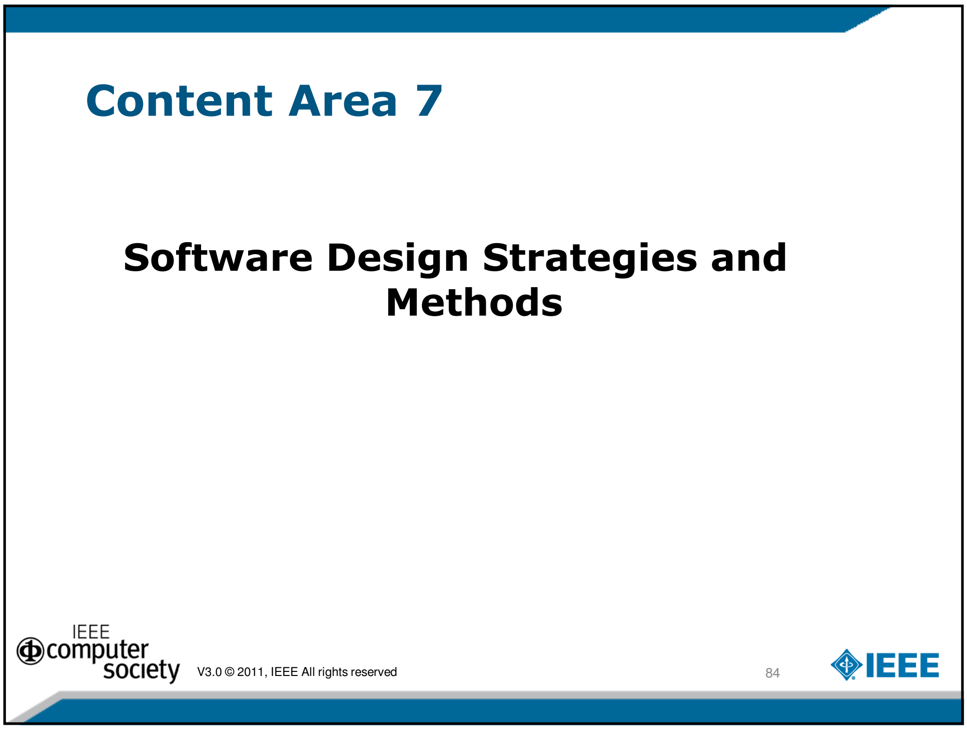
Behavioral: Sequence Diagrams

- **Actor** represented by stick figure – person (or role) that "interacts" with system by entering input data and receiving output data
- **Object** notation is rectangle with name of object underlined – shows individual object and not class of all similar objects
- **Lifeline** is vertical line under object or actor to show passage of time for object
- **Messages** use arrows to show messages sent or received by actor or system



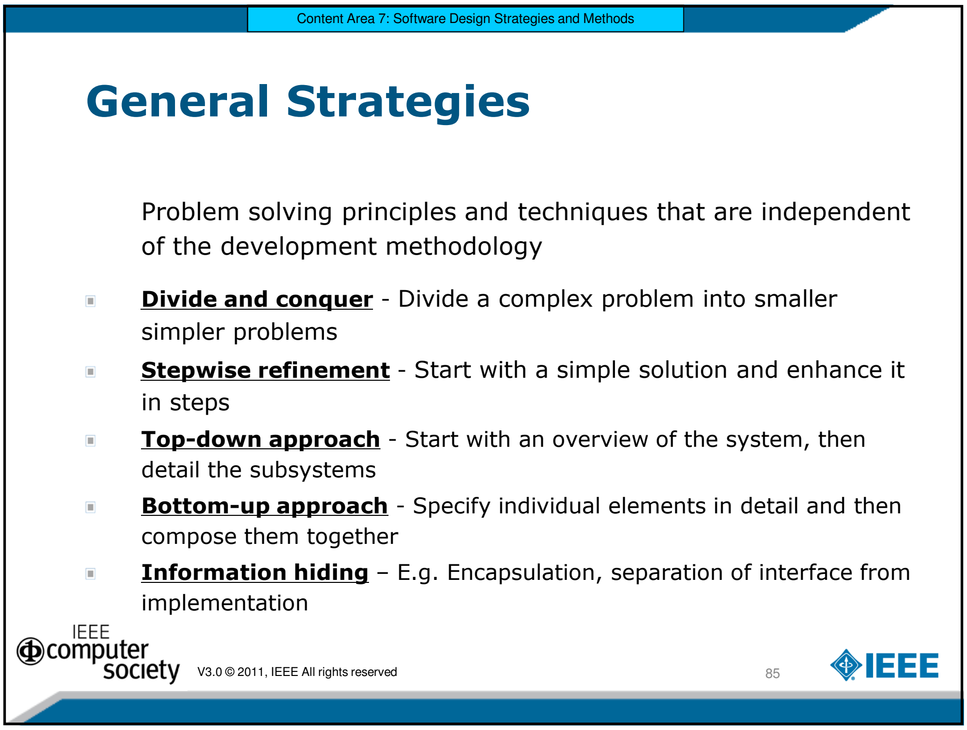
Behavioral: Other Notations

- Formal Specification Languages
  - Textual notations that use basic notations from mathematics e.g. logic, set, or sequence
  - Rigorously and abstractly define software component interfaces and behavior
  - Using preconditions and post-conditions
- Pseudocode
  - Structured programming-like language used to describe a program
  - Can help illustrate potential coding solutions



Content Area 7

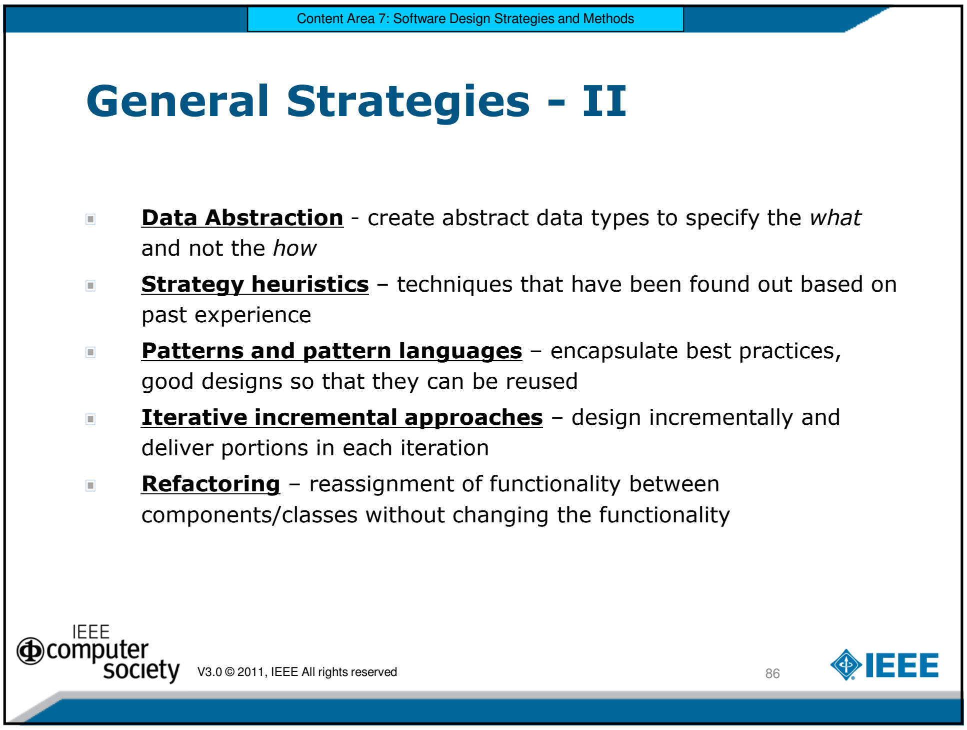
Software Design Strategies and Methods



General Strategies

Problem solving principles and techniques that are independent of the development methodology

- **Divide and conquer** - Divide a complex problem into smaller simpler problems
- **Stepwise refinement** - Start with a simple solution and enhance it in steps
- **Top-down approach** - Start with an overview of the system, then detail the subsystems
- **Bottom-up approach** - Specify individual elements in detail and then compose them together
- **Information hiding** – E.g. Encapsulation, separation of interface from implementation



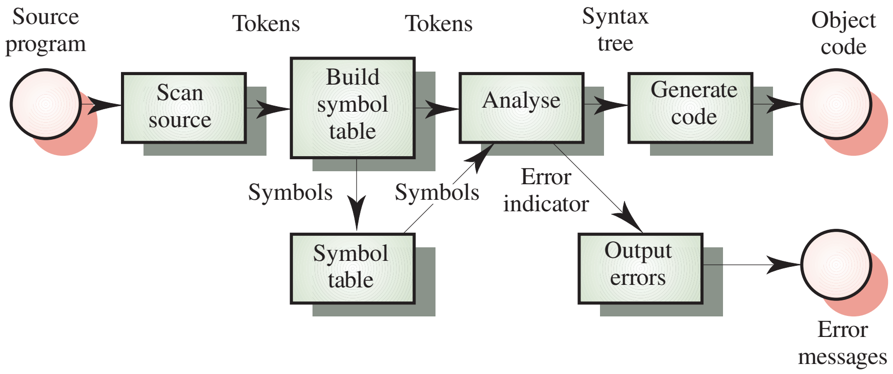
General Strategies - II

- **Data Abstraction** - create abstract data types to specify the *what* and not the *how*
- **Strategy heuristics** – techniques that have been found out based on past experience
- **Patterns and pattern languages** – encapsulate best practices, good designs so that they can be reused
- **Iterative incremental approaches** – design incrementally and deliver portions in each iteration
- **Refactoring** – reassignment of functionality between components/classes without changing the functionality

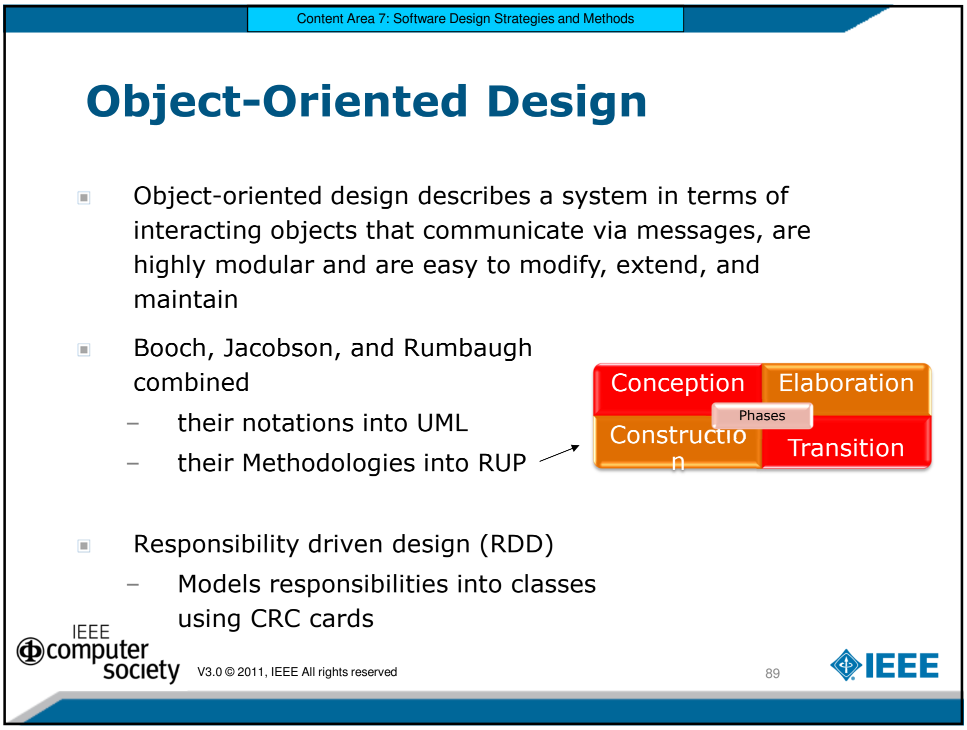
Function-Oriented Design

- Function-oriented design is also called *structured design*
- Structured design follows structural analysis producing data flow diagrams (DFD)
- Structure diagram derived from DFDs using
  - Transaction analysis
    - Identifies key transaction types of a system and uses them as units of design
  - Transformation analysis
    - Identifies the central transform which is the portion of the DFD that includes the essential functions but is independent of the input/output

Compiler: Functional View



Reference: *Software Engineering*, Addison-Wesley, by Ian Sommerville



Content Area 7: Software Design Strategies and Methods

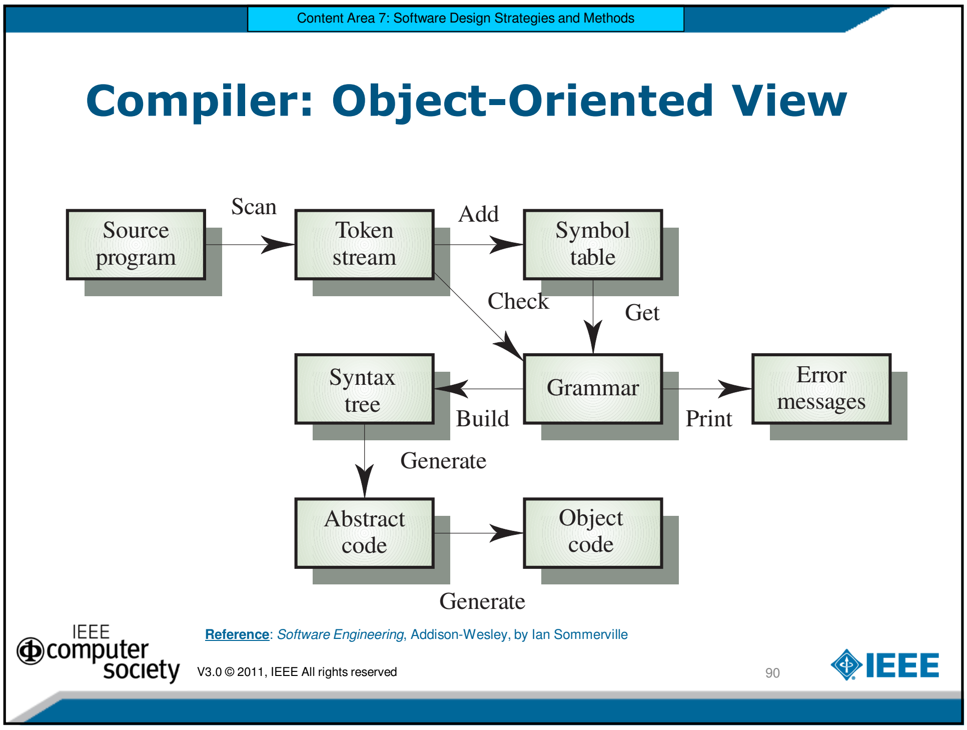
Object-Oriented Design

- Object-oriented design describes a system in terms of interacting objects that communicate via messages, are highly modular and are easy to modify, extend, and maintain
- Booch, Jacobson, and Rumbaugh combined
  - their notations into UML
  - their Methodologies into RUP →

Conception | Elaboration | Phases | Constructio n | Transition

- Responsibility driven design (RDD)
  - Models responsibilities into classes using CRC cards

IEEE computer society V3.0 © 2011, IEEE All rights reserved 89 IEEE



Content Area 7: Software Design Strategies and Methods

Compiler: Object-Oriented View

Source program → Scan → Token stream → Add → Symbol table

Token stream → Check → Grammar

Symbol table → Get → Grammar

Grammar → Build → Syntax tree

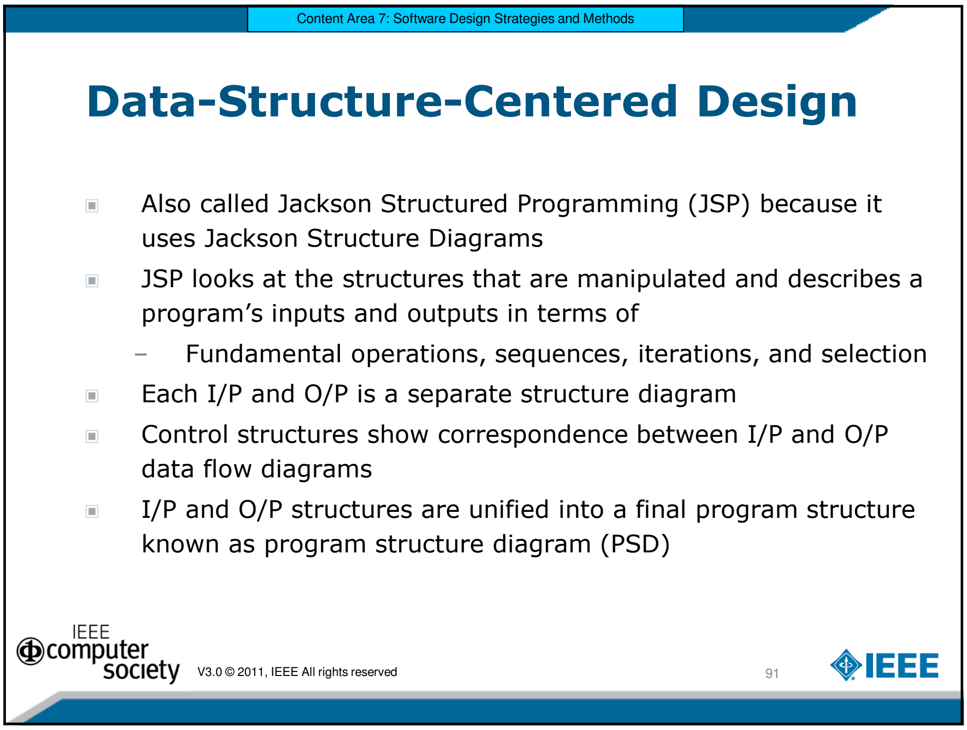
Grammar → Print → Error messages

Syntax tree → Generate → Abstract code

Abstract code → Generate → Object code

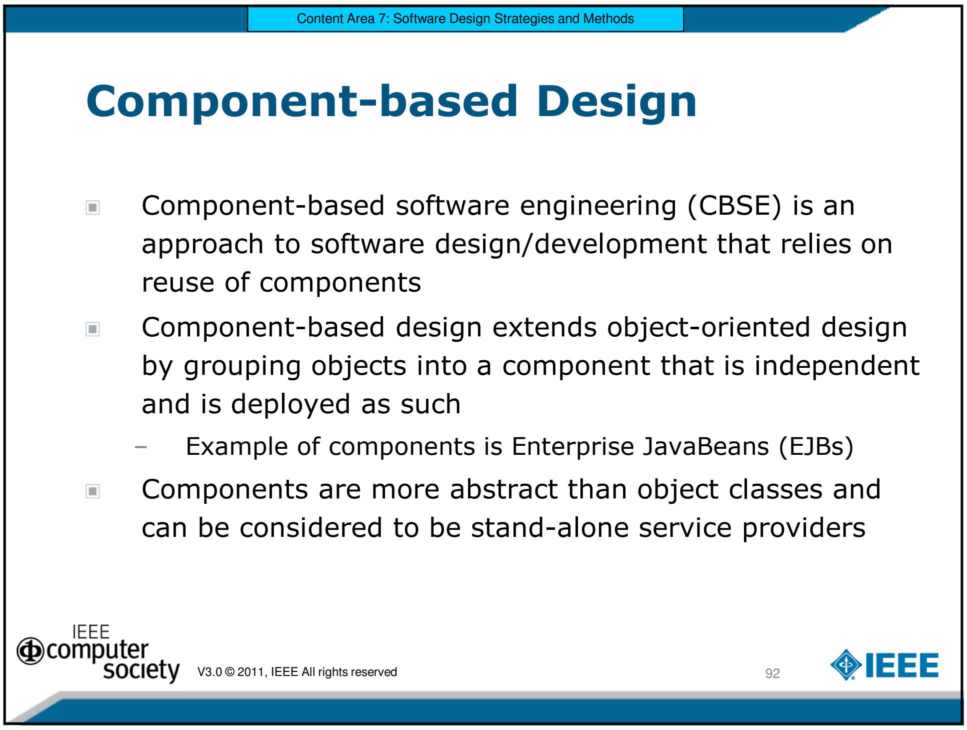
Reference: *Software Engineering*, Addison-Wesley, by Ian Sommerville

IEEE computer society V3.0 © 2011, IEEE All rights reserved 90 IEEE



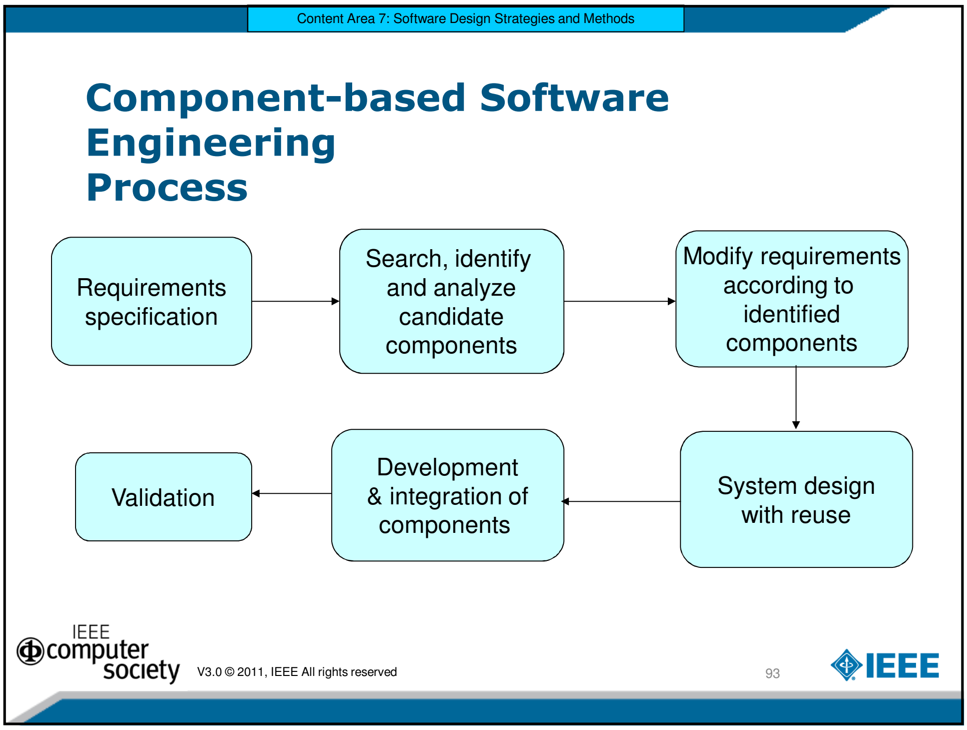
Data-Structure-Centered Design

- Also called Jackson Structured Programming (JSP) because it uses Jackson Structure Diagrams
- JSP looks at the structures that are manipulated and describes a program’s inputs and outputs in terms of
  - Fundamental operations, sequences, iterations, and selection
- Each I/P and O/P is a separate structure diagram
- Control structures show correspondence between I/P and O/P data flow diagrams
- I/P and O/P structures are unified into a final program structure known as program structure diagram (PSD)



Component-based Design

- Component-based software engineering (CBSE) is an approach to software design/development that relies on reuse of components
- Component-based design extends object-oriented design by grouping objects into a component that is independent and is deployed as such
  - Example of components is Enterprise JavaBeans (EJBs)
- Components are more abstract than object classes and can be considered to be stand-alone service providers

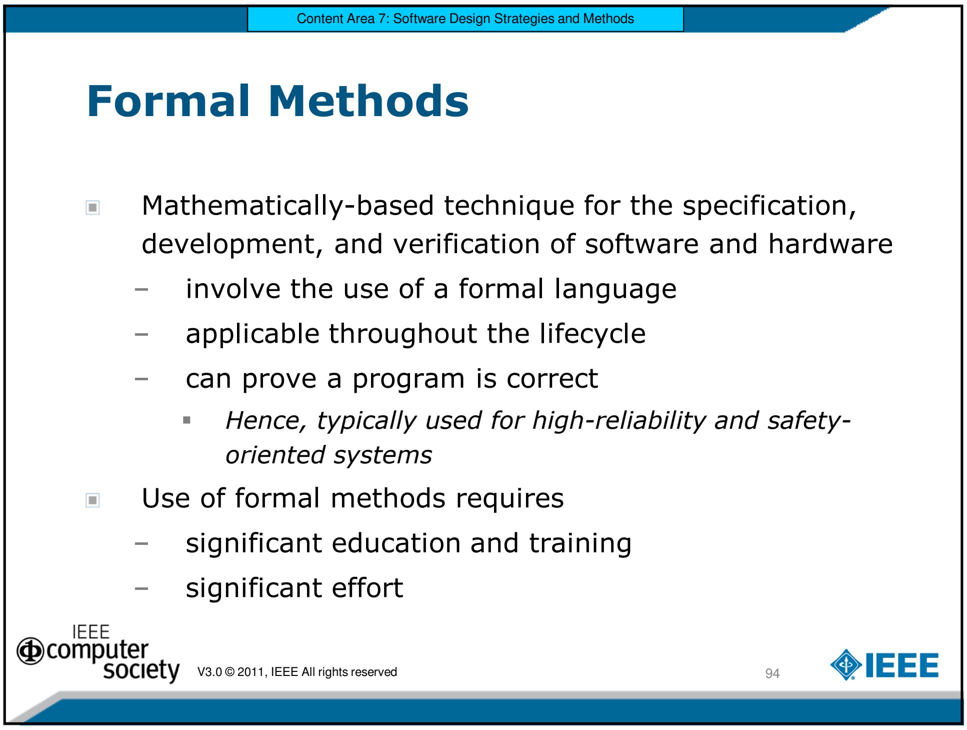


Content Area 7: Software Design Strategies and Methods

Component-based Software Engineering Process

Requirements specification → Search, identify and analyze candidate components → Modify requirements according to identified components → System design with reuse → Development & integration of components → Validation

IEEE computer society V3.0 © 2011, IEEE All rights reserved 93 IEEE

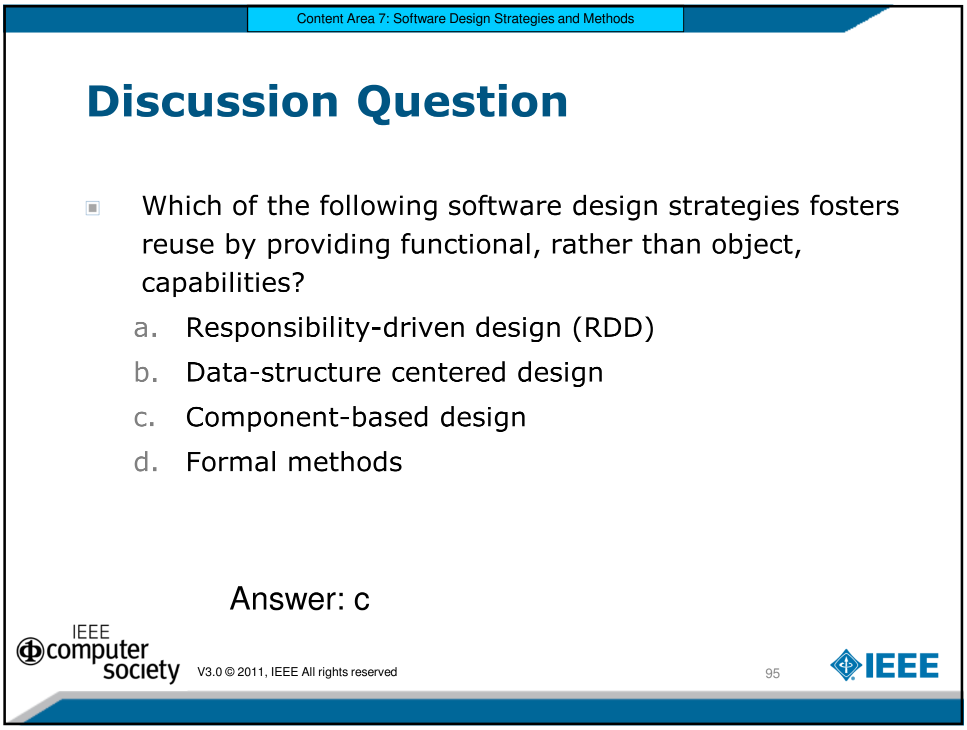


Content Area 7: Software Design Strategies and Methods

Formal Methods

- Mathematically-based technique for the specification, development, and verification of software and hardware
  - involve the use of a formal language
  - applicable throughout the lifecycle
  - can prove a program is correct
    - *Hence, typically used for high-reliability and safety-oriented systems*
- Use of formal methods requires
  - significant education and training
  - significant effort

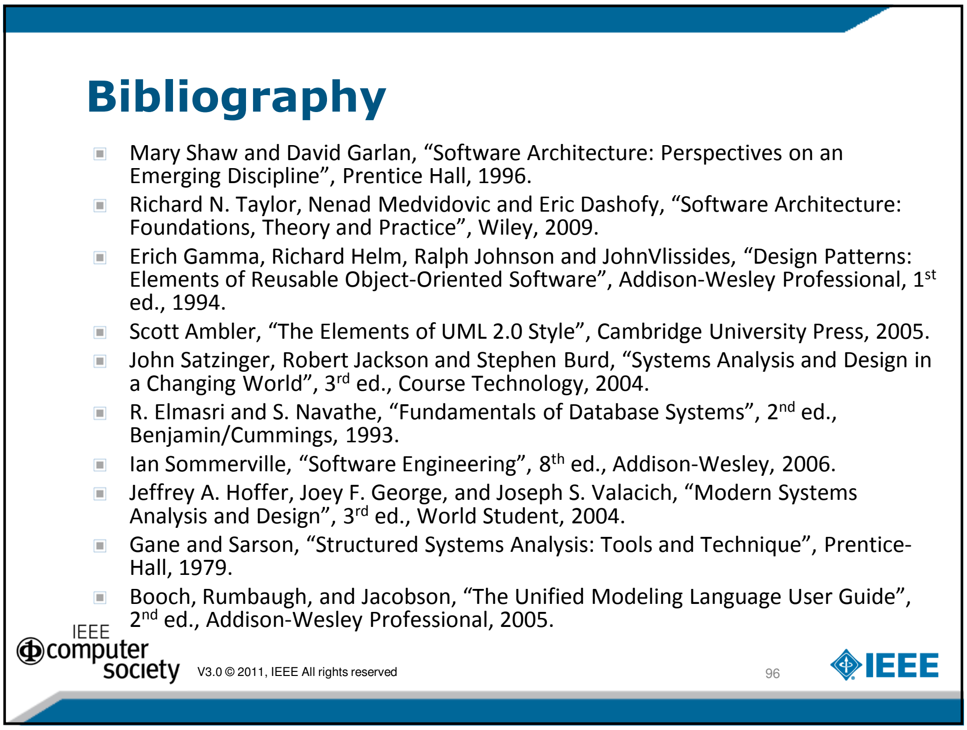
IEEE computer society V3.0 © 2011, IEEE All rights reserved 94 IEEE



Discussion Question

- Which of the following software design strategies fosters reuse by providing functional, rather than object, capabilities?
  a. Responsibility-driven design (RDD)
  b. Data-structure centered design
  c. Component-based design
  d. Formal methods

Answer: c



Bibliography

- Mary Shaw and David Garlan, “Software Architecture: Perspectives on an Emerging Discipline”, Prentice Hall, 1996.
- Richard N. Taylor, Nenad Medvidovic and Eric Dashofy, “Software Architecture: Foundations, Theory and Practice”, Wiley, 2009.
- Erich Gamma, Richard Helm, Ralph Johnson and JohnVlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley Professional, 1st ed., 1994.
- Scott Ambler, “The Elements of UML 2.0 Style”, Cambridge University Press, 2005.
- John Satzinger, Robert Jackson and Stephen Burd, “Systems Analysis and Design in a Changing World”, 3rd ed., Course Technology, 2004.
- R. Elmasri and S. Navathe, “Fundamentals of Database Systems”, 2nd ed., Benjamin/Cummings, 1993.
- Ian Sommerville, “Software Engineering”, 8th ed., Addison-Wesley, 2006.
- Jeffrey A. Hoffer, Joey F. George, and Joseph S. Valacich, “Modern Systems Analysis and Design”, 3rd ed., World Student, 2004.
- Gane and Sarson, “Structured Systems Analysis: Tools and Technique”, Prentice-Hall, 1979.
- Booch, Rumbaugh, and Jacobson, “The Unified Modeling Language User Guide”, 2nd ed., Addison-Wesley Professional, 2005.