Unit 4

Design contd...

Architectural Design

- 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
- The choice of a software architecture highlights early design decisions and provides a mechanism for considering the benefits of alternative architectures
- Data design translates the data objects defined in the analysis model into data structures that reside in the software
- 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 architectural design process contains four distinct steps
  - Represent the system in context
  - Identify the component archetypes (the top-level abstractions)
  - Identify and refine components within the context of various architectural styles
  - Formulate a specific instantiation of the architecture
- Once a software architecture has been derived, it is elaborated and then analyzed against quality criteria

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

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

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

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

Data Design

Purpose of Data Design

- Data design translates data objects defined as part of the analysis model into
  - Data structures at the software component level

  - A possible database architecture at the application level
- It focuses on the representation of data structures that are directly accessed by one or more software components
- The challenge is to store and retrieve the data in such way that useful information can be extracted from the data environment
- "Data quality is the difference between a data warehouse and a data garbage dump"

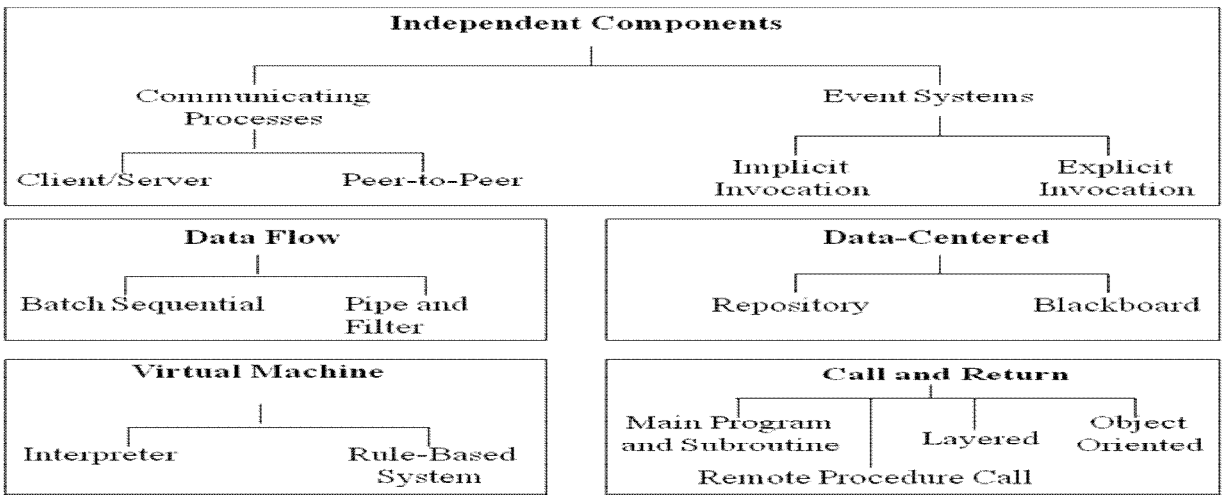
Data Design Principles

- The systematic analysis principles that are applied to function and behavior should also be applied to data
- All data structures and the operations to be performed on each one should be identified
- A mechanism for defining the content of each data object should be established and used to define both data and the operations applied to it
- Low-level data design decisions should be deferred until late in the design process
- The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure
- A library of useful data structures and the operations that may be applied to them should be developed
- A software programming language should support the specification and realization of abstract data types

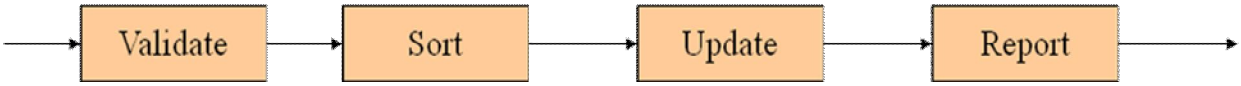
Software Architectural Styles

- 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

A Taxonomy of Architectural Styles

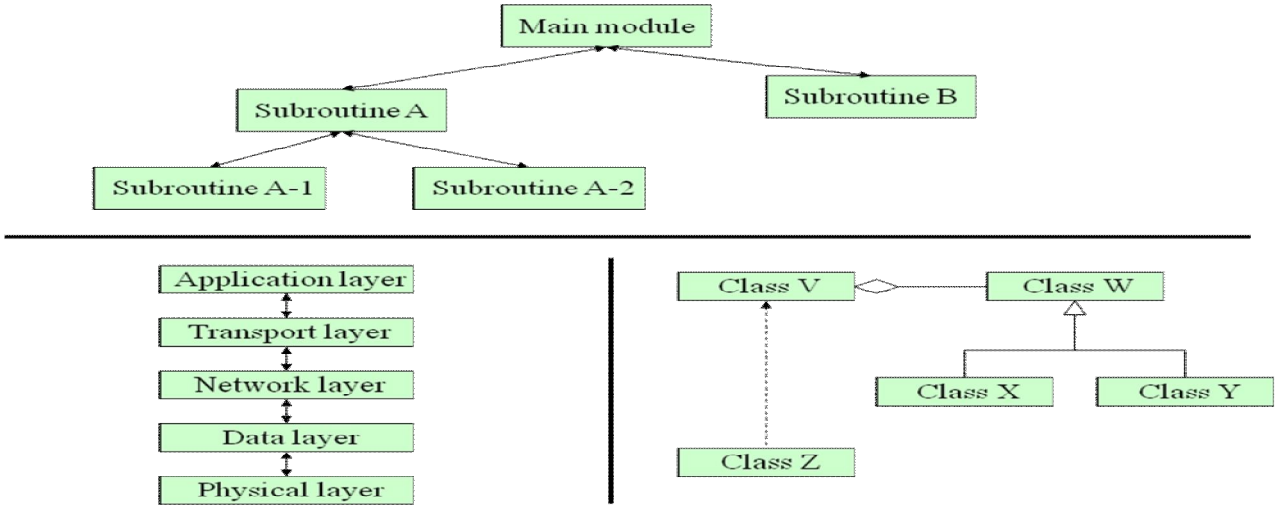


Data Flow Style



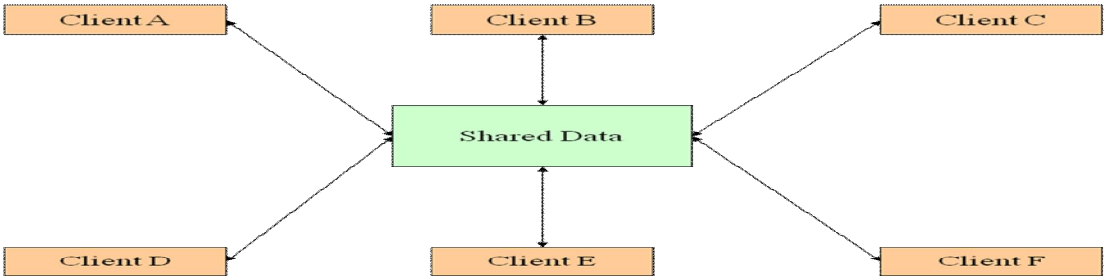
- Has the goal of modifiability
- Characterized by viewing the system as a series of transformations on successive pieces of input data
- Data enters the system and then flows through the components one at a time until they are assigned to output or a data store
- Batch sequential style
  - The processing steps are independent components
  - Each step runs to completion before the next step begins
- Pipe-and-filter style
  - Emphasizes the incremental transformation of data by successive components
  - The filters incrementally transform the data (entering and exiting via streams)
  - The filters use little contextual information and retain no state between instantiations
  - The pipes are stateless and simply exist to move data between filters
- Advantages
  - Has a simplistic design in the limited ways in which the components interact with the environment
  - Consists of no more and no less than the construction of its parts
  - Simplifies reuse and maintenance
  - Is easily made into a parallel or distributed execution in order to enhance system performance
- Disadvantages
  - Implicitly encourages a batch mentality so interactive applications are difficult to create in this style
  - Ordering of filters can be difficult to maintain so the filters cannot cooperatively interact to solve a problem
  - Exhibits poor performance
    - Filters typically force the least common denominator of data representation (usually ASCII stream)
    - Filter may need unlimited buffers if they cannot start producing output until they receive all of the input
    - Each filter operates as a separate process or procedure call, thus incurring overhead in set-up and take-down time
- Use this style when it makes sense to view your system as one that produces a well-defined easily identified output
  - The output should be a direct result of sequentially transforming a well-defined easily identified input in a time-independent fashion

Call-and-Return Style



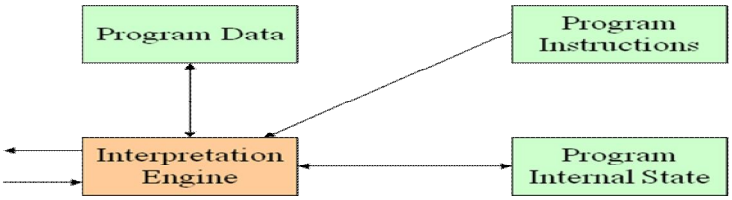
- Has the goal of modifiability and scalability
- Has been the dominant architecture since the start of software development
- Main program and subroutine style
  - Decomposes a program hierarchically into small pieces (i.e., modules)
  - Typically has a single thread of control that travels through various components in the hierarchy
- Remote procedure call style
  - Consists of main program and subroutine style of system that is decomposed into parts that are resident on computers connected via a network
  - Strives to increase performance by distributing the computations and taking advantage of multiple processors
  - Incurs a finite communication time between subroutine call and response
- Object-oriented or abstract data type system
  - Emphasizes the bundling of data and how to manipulate and access data
  - Keeps the internal data representation hidden and allows access to the object only through provided operations
  - Permits inheritance and polymorphism
- Layered system
  - Assigns components to layers in order to control inter-component interaction
  - Only allows a layer to communicate with its immediate neighbor
  - Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer
  - Builds each successive layer on its predecessor, hiding the lower layer and providing services for the upper layer
  - Is compromised by layer bridging that skips one or more layers to improve runtime performance
- Use this style when the order of computation is fixed, when interfaces are specific, and when components can make no useful progress while awaiting the results of request to other components

Data-Centered Style



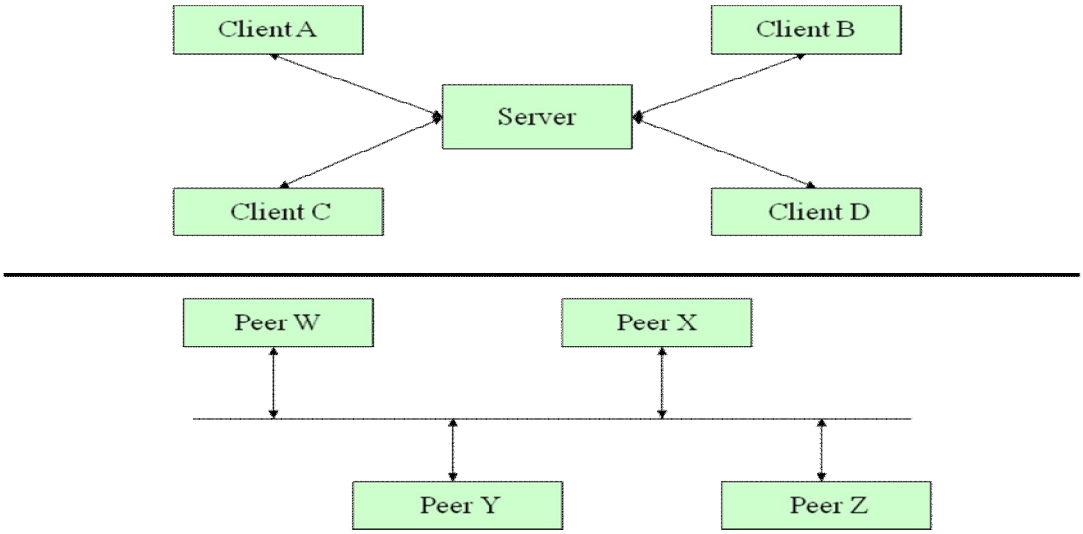
- Has the goal of integrating the data
- Refers to systems in which the access and update of a widely accessed data store occur
- A client runs on an independent thread of control
- The shared data may be a passive repository or an active blackboard
  - A blackboard notifies subscriber clients when changes occur in data of interest
- At its heart is a centralized data store that communicates with a number of clients
- Clients are relatively independent of each other so they can be added, removed, or changed in functionality
- The data store is independent of the clients
- Use this style when a central issue is the storage, representation, management, and retrieval of a large amount of related persistent data
- Note that this style becomes client/server if the clients are modeled as independent processes

Virtual Machine Style



- Has the goal of portability
- Software systems in this style simulate some functionality that is not native to the hardware and/or software on which it is implemented
  - Can simulate and test hardware platforms that have not yet been built
  - Can simulate "disaster modes" as in flight simulators or safety-critical systems that would be too complex, costly, or dangerous to test with the real system
- Examples include interpreters, rule-based systems, and command language processors
- Interpreters
  - Add flexibility through the ability to interrupt and query the program and introduce modifications at runtime
  - Incur a performance cost because of the additional computation involved in execution
- Use this style when you have developed a program or some form of computation but have no make of machine to directly run it on

Independent Component Style



- Consists of a number of independent processes that communicate through messages
- Has the goal of modifiability by decoupling various portions of the computation
- Sends data between processes but the processes do not directly control each other
- Event systems style
  - Individual components announce data that they wish to share (publish) with their environment
  - The other components may register an interest in this class of data (subscribe)
  - Makes use of a message component that manages communication among the other components
  - Components publish information by sending it to the message manager
  - When the data appears, the subscriber is invoked and receives the data
  - Decouples component implementation from knowing the names and locations of other components
- Communicating processes style
  - These are classic multi-processing systems
  - Well-know subtypes are client/server and peer-to-peer
  - The goal is to achieve scalability
  - A server exists to provide data and/or services to one or more clients
  - The client originates a call to the server which services the request
- Use this style when
  - Your system has a graphical user interface
  - Your system runs on a multiprocessor platform
  - Your system can be structured as a set of loosely coupled components
  - Performance tuning by reallocating work among processes is important
  - Message passing is sufficient as an interaction mechanism among components

Heterogeneous Styles

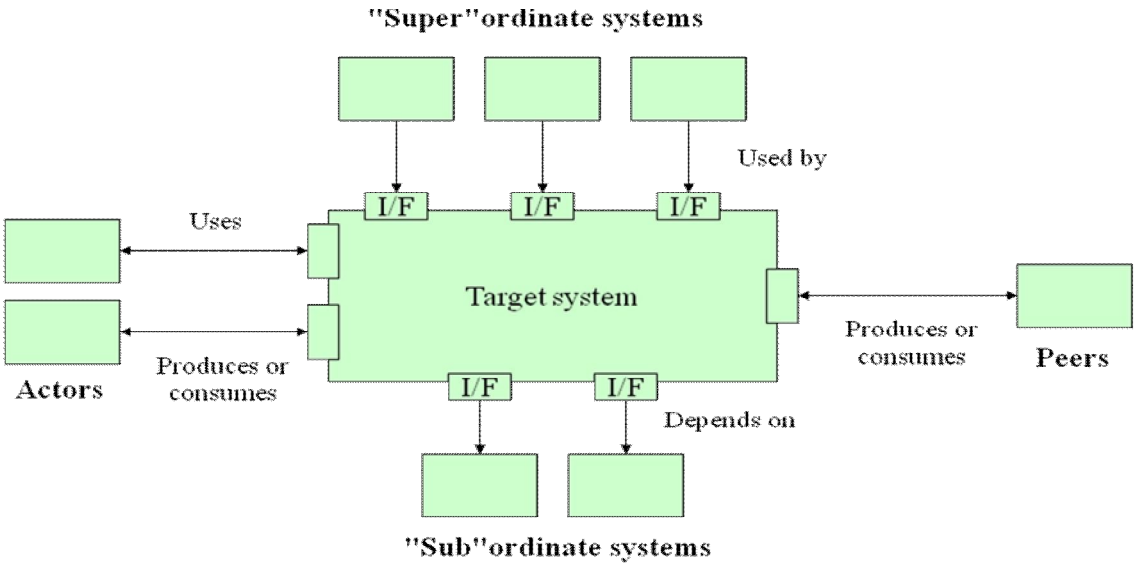
- Systems are seldom built from a single architectural style
- Three kinds of heterogeneity
  - Locationally heterogeneous
    - The drawing of the architecture reveals different styles in different areas (e.g., a branch of a call-and-return system may have a shared repository)
  - Hierarchically heterogeneous
    - A component of one style, when decomposed, is structured according to the rules of a different style
  - Simultaneously heterogeneous
    - Two or more architectural styles may both be appropriate descriptions for the style used by a computer-based system

Architectural Design Process

Architectural Design Steps

1) Represent the system in context
2) Define archetypes
3) Refine the architecture into components
4) Describe instantiations of the system

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

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 Humanity

- Addict/Gambler
- Amateur
- Beggar
- Clown
- Companion
- Damsel in distress
- Destroyer
- Detective
- Don Juan
- Drunk
- Engineer
- Father
- Gossip
- Guide
- Healer
- Hero
- Judge
- King
- Knight
- Liberator/Rescuer
- Lover/Devotee
- Martyr
- Mediator
- Mentor/Teacher
- Messiah/Savior
- Monk/Nun
- Mother
- Mystic/Hermit
- Networker
- Pioneer
- Poet
- Priest/Minister
- Prince
- Prostitute
- Queen
- Rebel/Pirate
- Saboteur
- Samaritan
- Scribe/Journalist
- Seeker/Wanderer
- Servant/Slave
- Storyteller
- Student
- Trickster/Thief
- Vampire
- Victim
- Virgin
- Visionary/Prophet
- Warrior/Soldier

Example Archetypes in Software Architecture

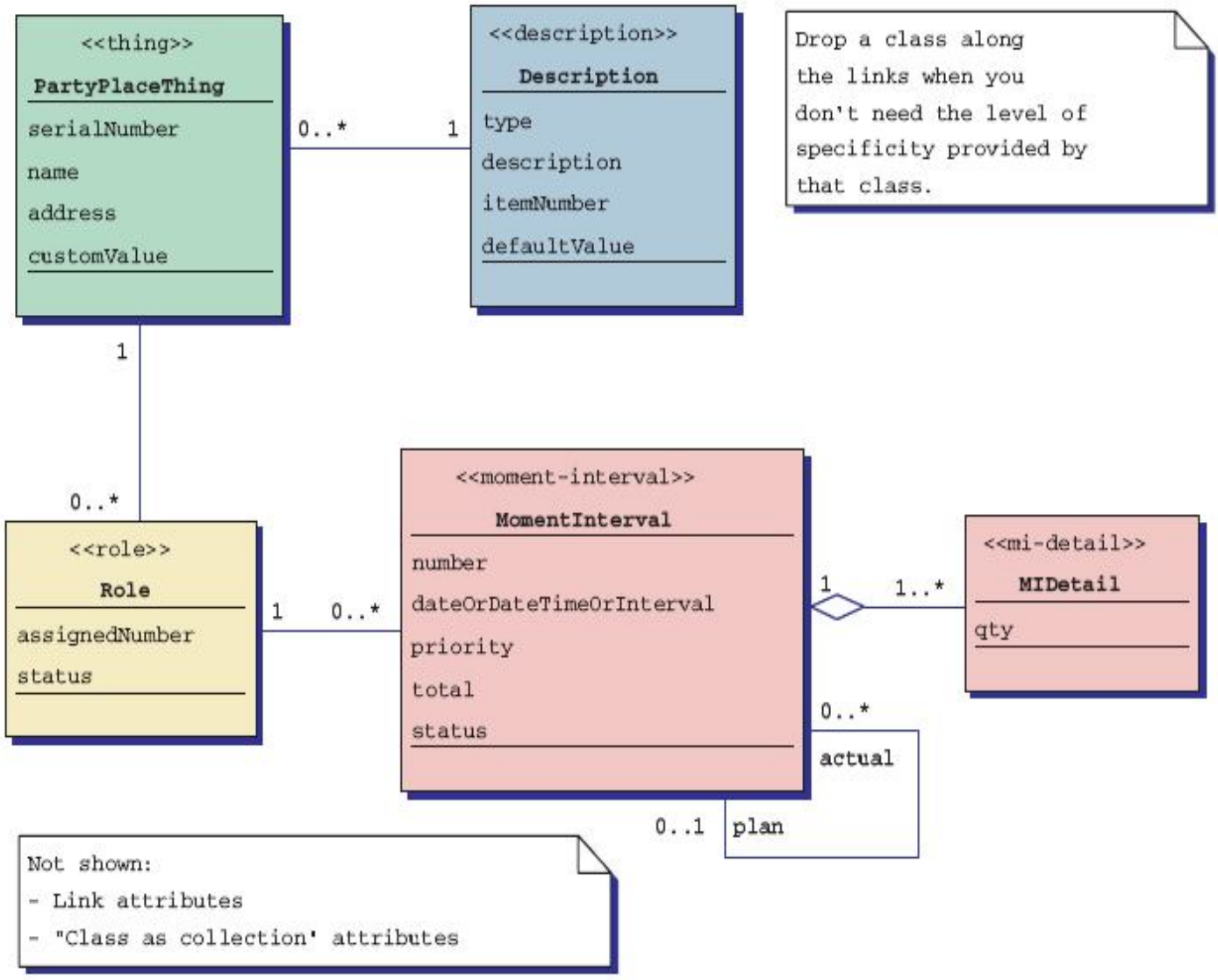
- Node
- Detector/Sensor
- Indicator
- Controller
- Manager

(Source: Pressman)

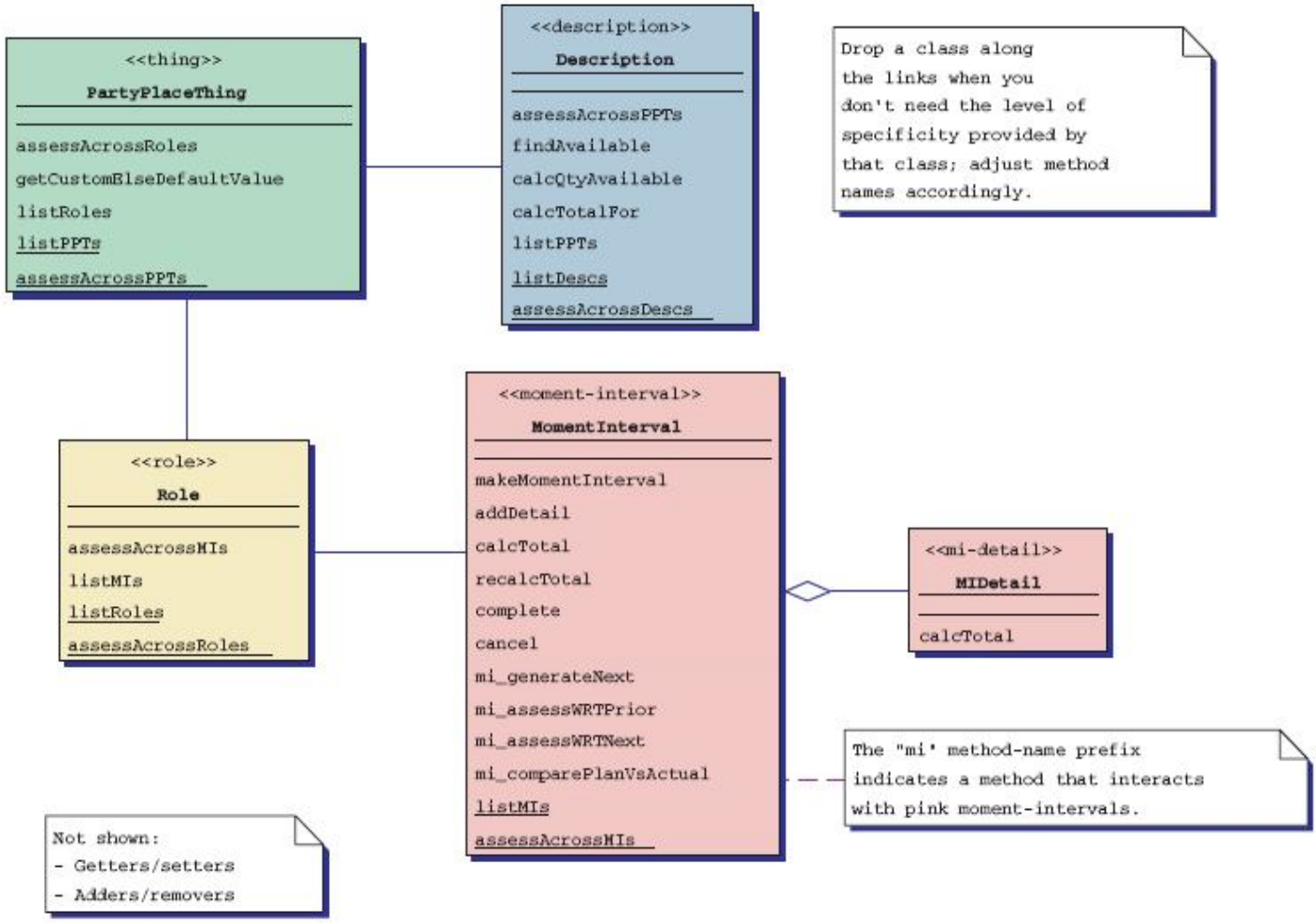
- Moment-Interval
- Role
- Description
- Party, Place, or Thing

(Source: Archetypes, Color, and the Domain Neutral Component)

Archetypes – their attributes



Archetypes – their methods



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

Assessing Alternative Architectural Designs

Various Assessment Approaches

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?
- 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

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 ← → □ ← → V**
    - Represents a dependency relationship among consumers who use the same source or producer
  - Flow dependency **→ U → V →**
    - Represents a dependency relationship between producers and consumers of resources
  - Constrained dependency **U "XOR" V**
    - Represents constraints on the relative flow of control among a set of activities such as mutual exclusion between two components

- - - - - - - - - - - - - - -

Component-Level Design

- Component-level design occurs after the first iteration of the architectural design
- It strives to create a design model from the analysis and architectural models
  - The translation can open the door to subtle errors that are difficult to find and correct later
  - "Effective programmers should not waste their time debugging – they should not introduce bugs to start with." Edsgar Dijkstra
- A component-level design can be represented using some intermediate representation (e.g. graphical, tabular, or text-based) that can be translated into source code
- The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors

The Software Component

- A software component is a modular building block for computer software
  - It is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces
- A component communicates and collaborates with
  - Other components
  - Entities outside the boundaries of the system

- Three different views of a component
  - An object-oriented view
  - A conventional view
  - A process-related view

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
  - Provide further elaboration of each attribute, operation, and interface
  - Specify the data structure appropriate for each attribute
  - Design the algorithmic detail required to implement the processing logic associated with each operation
  - 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 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
  - Define the interface for the transform (the order, number and types of the parameters)
  - Define the data structures used internally by the transform
  - 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

Designing Class-Based Components

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

Component-Level Design Guidelines

- Components
  - Establish naming conventions for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model
  - Obtain architectural component names from the problem domain and ensure that they have meaning to all stakeholders who view the architectural model (e.g., Calculator)
  - Use infrastructure component names that reflect their implementation-specific meaning (e.g., Stack)
- Dependencies and inheritance in UML
  - Model any dependencies from left to right and inheritance from top (base class) to bottom (derived classes)
  - Consider modeling any component dependencies as interfaces rather than representing them as a direct component-to-component dependency

Cohesion

- Cohesion is the “single-mindedness’ of a component
- It implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component itself
- The objective is to keep cohesion as high as possible
- The kinds of cohesion can be ranked in order from highest (best) to lowest (worst)
  - Functional
    - A module performs one and only one computation and then returns a result
  - Layer

        - A higher layer component accesses the services of a lower layer component
    - Communicational
        - All operations that access the same data are defined within one class
- Kinds of cohesion (continued)
    - Sequential
        - Components or operations are grouped in a manner that allows the first to provide input to the next and so on in order to implement a sequence of operations
    - Procedural
        - Components or operations are grouped in a manner that allows one to be invoked immediately after the preceding one was invoked, even when no data passed between them
    - Temporal
        - Operations are grouped to perform a specific behavior or establish a certain state such as program start-up or when an error is detected
    - Utility
        - Components, classes, or operations are grouped within the same category because of similar general functions but are otherwise unrelated to each other

Coupling

- As the amount of communication and collaboration increases between operations and classes, the complexity of the computer-based system also increases
- As complexity rises, the difficulty of implementing, testing, and maintaining software also increases
- Coupling is a qualitative measure of the degree to which operations and classes are connected to one another
- The objective is to keep coupling as low as possible
- The kinds of coupling can be ranked in order from lowest (best) to highest (worst)
    - Data coupling
        - Operation A() passes one or more atomic data operands to operation B(); the less the number of operands, the lower the level of coupling
    - Stamp coupling
        - A whole data structure or class instantiation is passed as a parameter to an operation
    - Control coupling
        - Operation A() invokes operation B() and passes a control flag to B that directs logical flow within B()
        - Consequently, a change in B() can require a change to be made to the meaning of the control flag passed by A(), otherwise an error may result
    - Common coupling

        - A number of components all make use of a global variable, which can lead to uncontrolled error propagation and unforeseen side effects
    - Content coupling
        - One component secretly modifies data that is stored internally in another component
- Other kinds of coupling (unranked)
    - Subroutine call coupling
        - When one operation is invoked it invokes another operation within side of it
    - Type use coupling
        - Component A uses a data type defined in component B, such as for an instance variable or a local variable declaration
        - If/when the type definition changes, every component that declares a variable of that data type must also change
    - Inclusion or import coupling
        - Component A imports or includes the contents of component B
    - External coupling
        - A component communicates or collaborates with infrastructure components that are entities external to the software (e.g., operating system functions, database functions, networking functions)

Conducting Component-Level Design

1) Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
2) Identify all design classes that correspond to the infrastructure domain
    - These classes are usually not present in the analysis or architectural models
    - These classes include GUI components, operating system components, data management components, networking components, etc.
3) Elaborate all design classes that are not acquired as reusable components
    - Specify message details (i.e., structure) when classes or components collaborate
    - Identify appropriate interfaces (e.g., abstract classes) for each component
    - Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
    - Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams
4) Describe persistent data sources (databases and files) and identify the classes required to manage them
5) Develop and elaborate behavioral representations for a class or component
    - This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class

6) Elaborate deployment diagrams to provide additional implementation detail

   - Illustrate the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environments

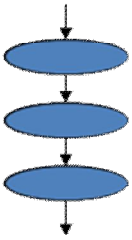
7) Factor every component-level design representation and always consider alternatives

   - Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
   - The final decision can be made by using established design principles and guidelines

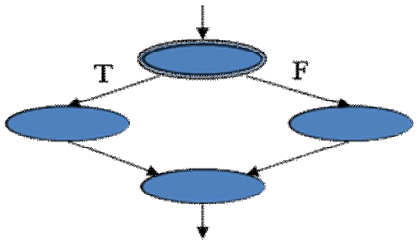
Designing Conventional Components

- Conventional design constructs emphasize the maintainability of a functional/procedural program
  - Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow
- Various notations depict the use of these constructs
  - Graphical design notation
    - Sequence, if-then-else, selection, repetition (see next slide)
  - Tabular design notation (see upcoming slide)
  - Program design language
    - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

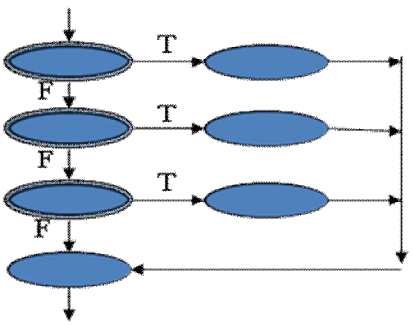
Graphical Design Notation



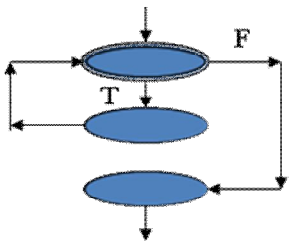
Sequence



If-then-else



Selection



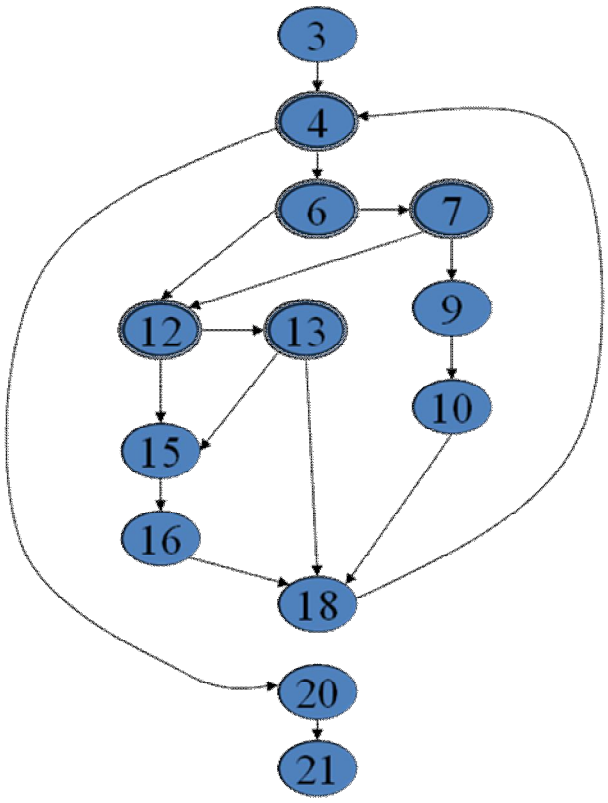
Repetition

Graphical Example used for Algorithm Analysis

```
 1  int functionZ(int y)
 2  {
 3  int x = 0;

 4  while (x <= (y * y))
 5     {
 6     if ((x % 11 == 0) &&
 7          (x % y == 0))
 8        {
 9        printf("%d", x);
10        x++;
11        } // End if
12     else if ((x % 7 == 0) ||
13               (x % y == 1))
14        {
15        printf("%d", y);
16        x = x + 2;
17        } // End else
18     printf("\n");
19     } // End while

20  printf("End of list\n");
21  return 0;
22  } // End functionZ
```



Tabular Design Notation

1) List all actions that can be associated with a specific procedure (or module)
2) List all conditions (or decisions made) during execution of the procedure
3) Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions
4) Define rules by indicating what action(s) occurs for a set of conditions

	Rules			
Conditions	1	2	3	4
Condition A	T	T		F
Condition B		F	T	
Condition C	T			T
Actions				
Action X	✓		✓	
Action Y				✓
Action Z	✓	✓		✓

- - - - - - - - - - - - - - -

User Interface Analysis and Design

- Interface design focuses on the following
  - The design of interfaces between software components
  - The design of interfaces between the software and other nonhuman producers and consumers of information
  - The design of the interface between a human and the computer
- Graphical user interfaces (GUIs) have helped to eliminate many of the most horrific interface problems
- However, some are still difficult to learn, hard to use, confusing, counterintuitive, unforgiving, and frustrating
- User interface analysis and design has to do with the study of people and how they relate to technology

A Spiral Process

- User interface development follows a spiral process
  - Interface analysis (user, task, and environment analysis)
    - Focuses on the profile of the users who will interact with the system
    - Concentrates on users, tasks, content and work environment
    - Studies different models of system function (as perceived from the outside)
    - Delineates the human- and computer-oriented tasks that are required to achieve system function
  - Interface design
    - Defines a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system
  - Interface construction
    - Begins with a prototype that enables usage scenarios to be evaluated
    - Continues with development tools to complete the construction
  - Interface validation, focuses on
    - The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements
    - The degree to which the interface is easy to use and easy to learn
    - The users' acceptance of the interface as a useful tool in their work

The Golden Rules of User Interface Design

Place the User in Control

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

  - The user shall be able to enter and exit a mode with little or no effort (e.g., spell check → edit text → spell check)

- Provide for flexible interaction
  - The user shall be able to perform the same action via keyboard commands, mouse movement, or voice recognition
- Allow user interaction to be interruptible and "undo"able
  - The user shall be able to easily interrupt a sequence of actions to do something else (without losing the work that has been done so far)
  - The user shall be able to "undo" any action
- Streamline interaction as skill levels advance and allow the interaction to be customized
  - The user shall be able to use a macro mechanism to perform a sequence of repeated interactions and to customize the interface
- Hide technical internals from the casual user
  - The user shall not be required to directly use operating system, file management, networking. etc., commands to perform any actions. Instead, these operations shall be hidden from the user and performed "behind the scenes" in the form of a real-world abstraction
- Design for direct interaction with objects that appear on the screen
  - The user shall be able to manipulate objects on the screen in a manner similar to what would occur if the object were a physical thing (e.g., stretch a rectangle, press a button, move a slider)

Reduce the User's Memory Load

- Reduce demand on short-term memory
  - The interface shall reduce the user's requirement to remember past actions and results by providing visual cues of such actions
- Establish meaningful defaults
  - The system shall provide the user with default values that make sense to the average user but allow the user to change these defaults
  - The user shall be able to easily reset any value to its original default value
- Define shortcuts that are intuitive
  - The user shall be provided mnemonics (i.e., control or alt combinations) that tie easily to the action in a way that is easy to remember such as the first letter
- The visual layout of the interface should be based on a real world metaphor
  - The screen layout of the user interface shall contain well-understood visual cues that the user can relate to real-world actions
- Disclose information in a progressive fashion
  - When interacting with a task, an object or some behavior, the interface shall be organized hierarchically by moving the user progressively in a step-wise fashion from an abstract concept to a concrete action (e.g., text format options → format dialog box)

Make the Interface Consistent

- The interface should present and acquire information in a consistent fashion
  - All visual information shall be organized according to a design standard that is maintained throughout all screen displays
  - Input mechanisms shall be constrained to a limited set that is used consistently throughout the application
  - Mechanisms for navigating from task to task shall be consistently defined and implemented
- Allow the user to put the current task into a meaningful context
  - The interface shall provide indicators (e.g., window titles, consistent color coding) that enable the user to know the context of the work at hand
  - The user shall be able to determine where he has come from and what alternatives exist for a transition to a new task
- Maintain consistency across a family of applications
  - A set of applications performing complimentary functionality shall all implement the same design rules so that consistency is maintained for all interaction
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so
  - Once a particular interactive sequence has become a de facto standard (e.g., alt-S to save a file), the application shall continue this expectation in every part of its funtionality

Reconciling Four Different Models

- Four different models come into play when a user interface is analyzed and designed
  - User profile model – Established by a human engineer or software engineer
  - Design model – Created by a software engineer
  - Implementation model – Created by the software implementers
  - User's mental model – Developed by the user when interacting with the application
- The role of the interface designer is to reconcile these differences and derive a consistent representation of the interface

User Profile Model

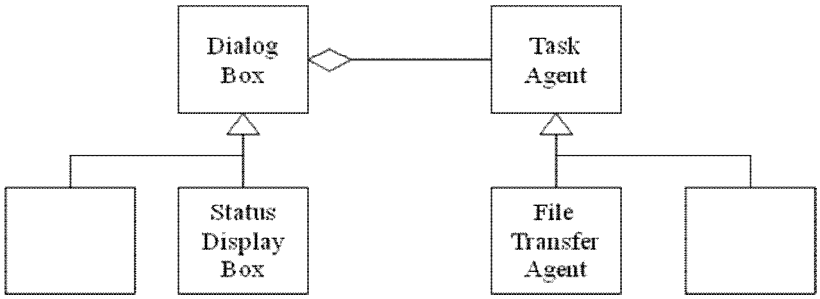
- Establishes the profile of the end-users of the system
  - Based on age, gender, physical abilities, education, cultural or ethnic background, motivation, goals, and personality
- Considers syntactic knowledge of the user
  - The mechanics of interaction that are required to use the interface effectively
- Considers semantic knowledge of the user

- The underlying sense of the application; an understanding of the functions that are performed, the meaning of input and output, and the objectives of the system

- Categorizes users as
  - Novices
    - No syntactic knowledge of the system, little semantic knowledge of the application, only general computer usage
  - Knowledgeable, intermittent users
    - Reasonable semantic knowledge of the system, low recall of syntactic information to use the interface
  - Knowledgeable, frequent users
    - Good semantic and syntactic knowledge (i.e., power user), look for shortcuts and abbreviated modes of operation

Design Model

- Derived from the analysis model of the requirements
- Incorporates data, architectural, interface, and procedural representations of the software
- Constrained by information in the requirements specification that helps define the user of the system
- Normally is incidental to other parts of the design model
  - But in many cases it is as important as the other parts



Implementation Model

- Consists of the look and feel of the interface combined with all supporting information (books, videos, help files) that describe system syntax and semantics
- Strives to agree with the user's mental model; users then feel comfortable with the software and use it effectively
- Serves as a translation of the design model by providing a realization of the information contained in the user profile model and the user’s mental model

User's Mental Model

- Often called the user's system perception
- Consists of the image of the system that users carry in their heads

- Accuracy of the description depends upon the user's profile and overall familiarity with the software in the application domain

User Interface Analysis

Elements of the User Interface

- To perform user interface analysis, the practitioner needs to study and understand four elements
  - The users who will interact with the system through the interface
  - The tasks that end users must perform to do their work
  - The content that is presented as part of the interface
  - The work environment in which these tasks will be conducted

User Analysis

- The analyst strives to get the end user's mental model and the design model to converge by understanding
  - The users themselves
  - How these people use the system
- Information can be obtained from
  - User interviews with the end users
  - Sales input from the sales people who interact with customers and users on a regular basis
  - Marketing input based on a market analysis to understand how different population segments might use the software
  - Support input from the support staff who are aware of what works and what doesn't, what users like and dislike, what features generate questions, and what features are easy to use
- A set of questions should be answered during user analysis (see next slide)

User Analysis Questions

1) Are the users trained professionals, technicians, clerical or manufacturing workers?
2) What level of formal education does the average user have?
3) Are the users capable of learning on their own from written materials or have they expressed a desire for classroom training?
4) Are the users expert typists or are they keyboard phobic?
5) What is the age range of the user community?
6) Will the users be represented predominately by one gender?
7) How are users compensated for the work they perform or are they volunteers?
8) Do users work normal office hours, or do they work whenever the job is required?

9) Is the software to be an integral part of the work users do, or will it be used only occasionally?
10) What is the primary spoken language among users?
11) What are the consequences if a user makes a mistake using the system?
12) Are users experts in the subject matter that is addressed by the system?
13) Do users want to know about the technology that sits behind the interface?

Task Analysis and Modeling

- Task analysis strives to know and understand
  - The work the user performs in specific circumstances
  - The tasks and subtasks that will be performed as the user does the work
  - The specific problem domain objects that the user manipulates as work is performed
  - The sequence of work tasks (i.e., the workflow)
  - The hierarchy of tasks
- Use cases
  - Show how an end user performs some specific work-related task
  - Enable the software engineer to extract tasks, objects, and overall workflow of the interaction
  - Helps the software engineer to identify additional helpful features

Content Analysis

- The display content may range from character-based reports, to graphical displays, to multimedia information
- Display content may be
  - Generated by components in other parts of the application
  - Acquired from data stored in a database that is accessible from the application
  - Transmitted from systems external to the application in question
- The format and aesthetics of the content (as it is displayed by the interface) needs to be considered
- A set of questions should be answered during content analysis (see next slide)

Content Analysis Guidelines

1) Are various types of data assigned to consistent locations on the screen (e.g., photos always in upper right corner)?
2) Are users able to customize the screen location for content?
3) Is proper on-screen identification assigned to all content?
4) Can large reports be partitioned for ease of understanding?

5) Are mechanisms available for moving directly to summary information for large collections of data?

6) Is graphical output scaled to fit within the bounds of the display device that is used?

7) How is color used to enhance understanding?

8) How are error messages and warnings presented in order to make them quick and easy to see and understand?

Work Environment Analysis

- Software products need to be designed to fit into the work environment, otherwise they may be difficult or frustrating to use
- Factors to consider include
  - Type of lighting
  - Display size and height
  - Keyboard size, height and ease of use
  - Mouse type and ease of use
  - Surrounding noise
  - Space limitations for computer and/or user
  - Weather or other atmospheric conditions
  - Temperature or pressure restrictions
  - Time restrictions (when, how fast, and for how long)

User Interface Design

- User interface design is an iterative process, where each iteration elaborate and refines the information developed in the preceding step
- General steps for user interface design
  1) Using information developed during user interface analysis, define user interface objects and actions (operations)
  2) Define events (user actions) that will cause the state of the user interface to change; model this behavior
  3) Depict each interface state as it will actually look to the end user
  4) Indicate how the user interprets the state of the system from information provided through the interface
- During all of these steps, the designer must
  1) Always follow the three golden rules of user interfaces
  2) Model how the interface will be implemented
  3) Consider the computing environment (e.g., display technology, operating system, development tools) that will be used

Interface Objects and Actions

- Interface objects and actions are obtained from a grammatical parse of the use cases and the software problem statement
- Interface objects are categorized into types: source, target, and application
  - A source object is dragged and dropped into a target object such as to create a hardcopy of a report
  - An application object represents application-specific data that are not directly manipulated as part of screen interaction such as a list
- After identifying objects and their actions, an interface designer performs screen layout which involves
  - Graphical design and placement of icons
  - Definition of descriptive screen text
  - Specification and titling for windows
  - Definition of major and minor menu items
  - Specification of a real-world metaphor to follow

Design Issues to Consider

- Four common design issues usually surface in any user interface
  - System response time (both length and variability)
  - User help facilities
    - When is it available, how is it accessed, how is it represented to the user, how is it structured, what happens when help is exited
  - Error information handling (more on next slide)
    - How meaningful to the user, how descriptive of the problem
  - Menu and command labeling (more on upcoming slide)
    - Consistent, easy to learn, accessibility, internationalization
- Many software engineers do not address these issues until late in the design or construction process
  - This results in unnecessary iteration, project delays, and customer frustration

Guidelines for Error Messages

- The message should describe the problem in plain language that a typical user can understand
- The message should provide constructive advice for recovering from the error
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have)
- The message should be accompanied by an audible or visual cue such as a beep, momentary flashing, or a special error color
- The message should be non-judgmental (the message should never place blame on the user)

Questions for Menu Labeling and Typed Commands

- Will every menu option have a corresponding command?
- What form will a command take? A control sequence? A function key? A typed word?
- How difficult will it be to learn and remember the commands?
- What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

User Interface Evaluation

Design and Prototype Evaluation

- Before prototyping occurs, a number of evaluation criteria can be applied during design reviews to the design model itself
  - The amount of learning required by the users
    - Derived from the length and complexity of the written specification and its interfaces
  - The interaction time and overall efficiency
    - Derived from the number of user tasks specified and the average number of actions per task
  - The memory load on users
    - Derived from the number of actions, tasks, and system states
  - The complexity of the interface and the degree to which it will be accepted by the user
    - Derived from the interface style, help facilities, and error handling procedures
- Prototype evaluation can range from an informal test drive to a formally designed study using statistical methods and questionnaires
- The prototype evaluation cycle consists of prototype creation followed by user evaluation and back to prototype modification until all user issues are resolved
- The prototype is evaluated for
  - Satisfaction of user requirements
  - Conformance to the three golden rules of user interface design
  - Reconciliation of the four models of a user interface