UNIT-IV

Creating Architectural Design: Software architecture, Data design, Architectural Styles and Patterns, Architectural Design, Assessing alternative Architectural Designs, Mapping data flow into software Architecture.

Modeling Component-Level Design: What is a Component, Designing Class-Based components, Conducting Component–level Design, Object Constraint Language, Designing Conventional Components.

Performing User Interface Design: The Golden Rules, User Interface Analysis and Design, Interface Analysis, Interface Design Steps, Design Evaluation**.**

Architectural Design

- Introduction
- Data design
- Software architectural styles
- Architectural design process
- Assessing alternative architectural designs

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

Why Architecture?\

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to:

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively easy, and

(3) reduce the risks associated with the construction of the software.

Importance of Architecture

- Communication between all parties (stakeholders) interested in the development of a computer-based system.
- Highlights early design decisions
  - as important, on the ultimate success of the system as an operational entity.
- “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together”.

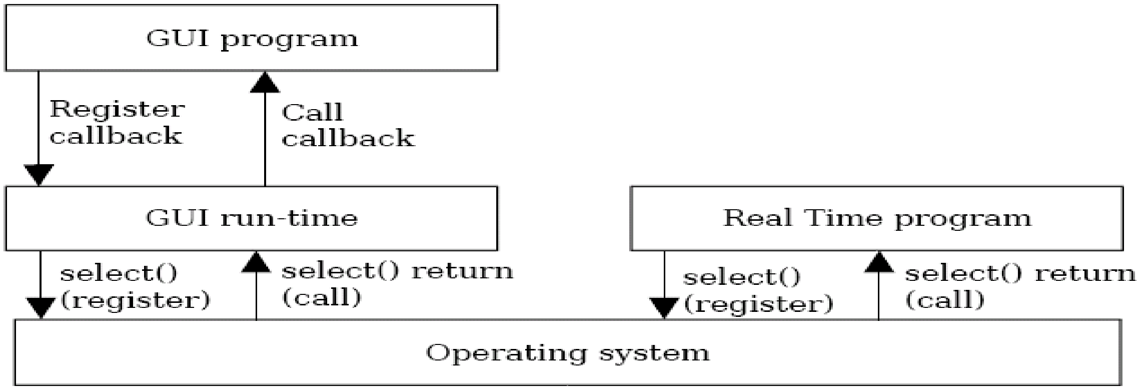
Architectural Design Process

- 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

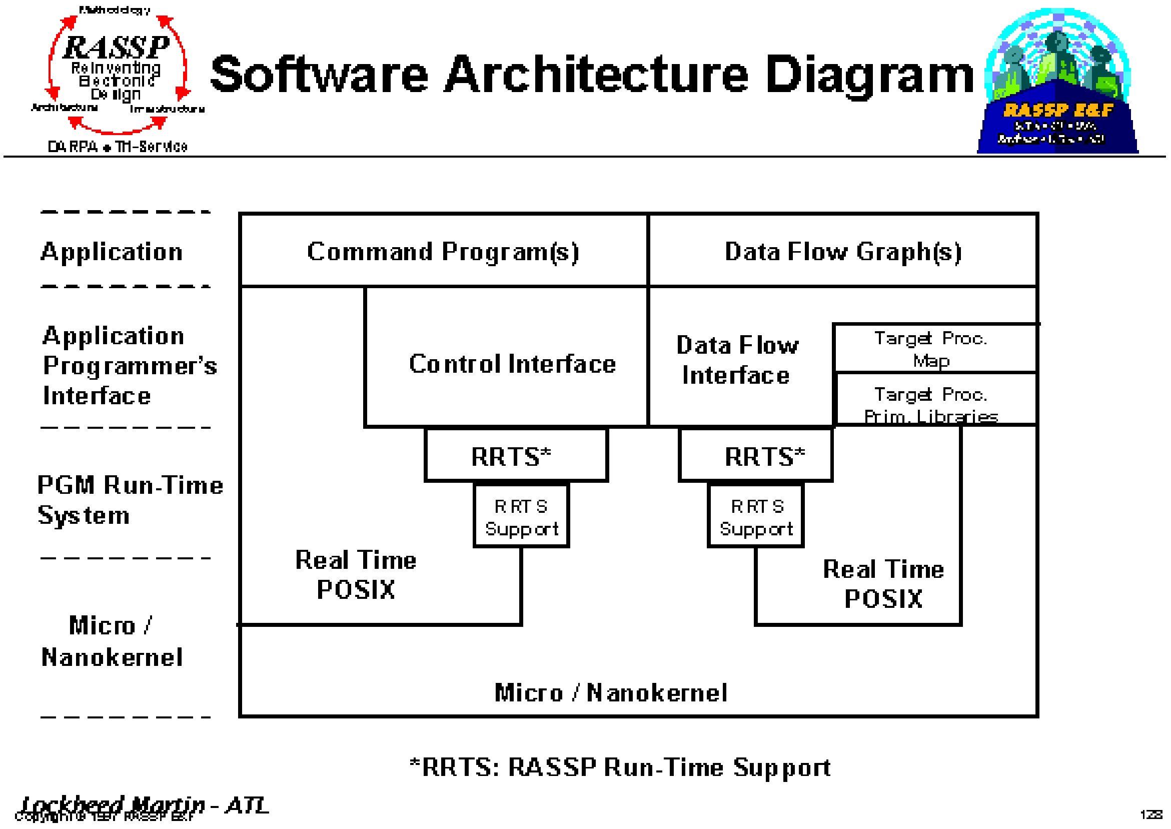
Emphasis on Software Components

- 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

Examples: (1)



(2)



Software Architecture Diagram

Application

Application Programmer's Interface

PGM Run-Time System

Micro / Nanokernel

Command Program(s)

Data Flow Graph(s)

Control Interface

Data Flow Interface

Target Proc. Map

Target Proc. Prim. Libraries

RRTS*

RRTS Support

Real Time POSIX

Micro / Nanokernel

*RRTS: RASSP Run-Time Support

Lockheed Martin - ATL

Copyright © 1997 RASSP E&F

123

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 – At the Component Level

1. The systematic analysis principles that are applied to function and behavior should also be applied to data
2. All data structures and the operations to be performed on each one should be identified
3. 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
4. Low-level data design decisions should be deferred until late in the design process – Stepwise refinement
5. 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 – information hiding and coupling
6. A library of useful data structures and the operations that may be applied to them should be developed – development of class library
7. 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

(1) Data-centered architectures

(2) Data flow architectures

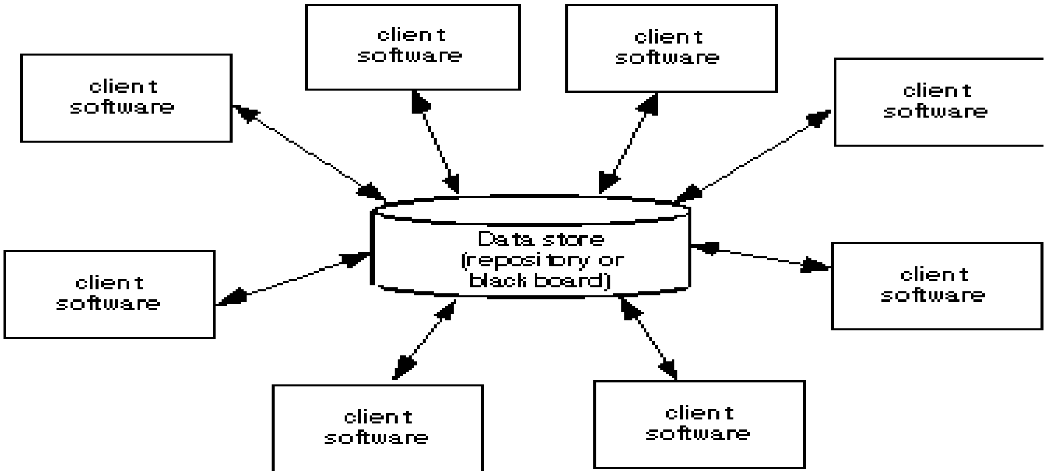
(3) Call and return architectures

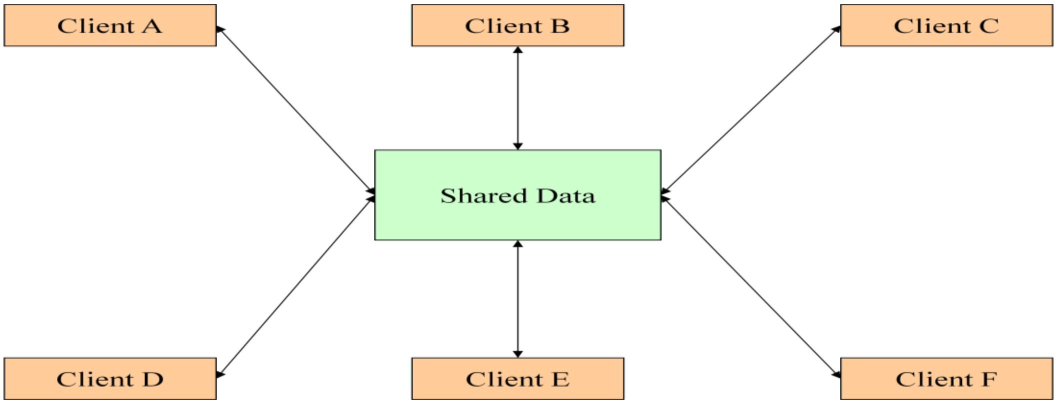
(4) Object-oriented architectures

(5) Layered architectures

(1) Data-Centered Architecture

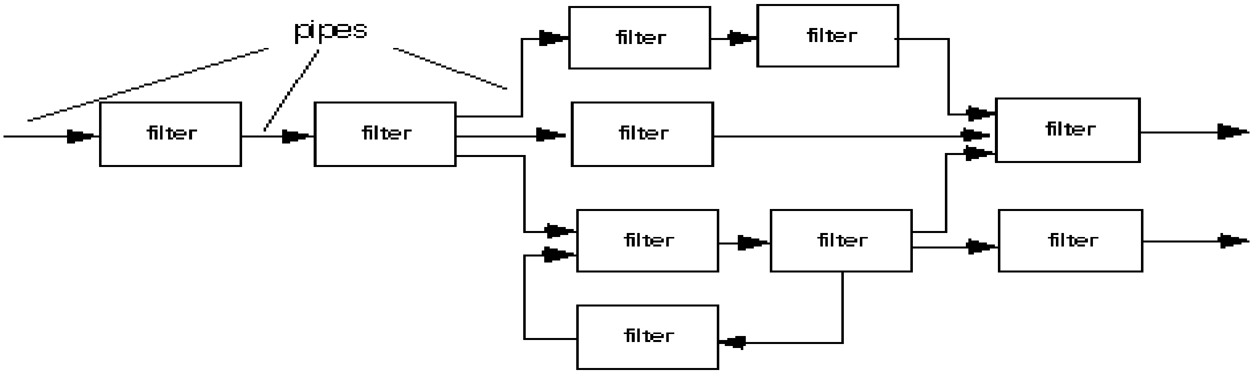
- A data store resides at the center, is accessed frequently by other components.
- Increase integratability (independent components)
- Blackboard: sends notifications to client software when data of its interest is changed.



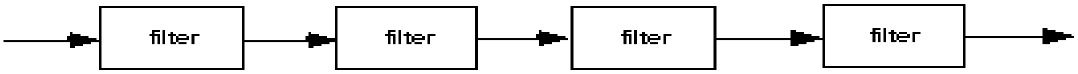


- Has the <u>goal</u> of integrating the data
- Refers to systems in which the access and update of a widely accessed data store occur
- A client runs on an <u>independent</u> thread of control
- The shared data may be a <u>passive</u> repository or an <u>active</u> blackboard
  - A blackboard notifies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

(2) Data Flow Architecture

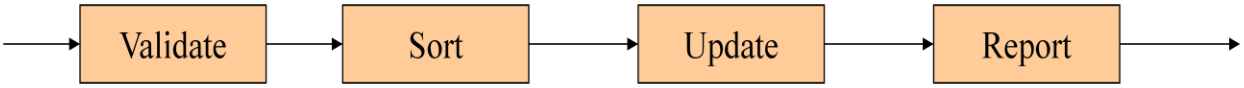


(a) pipes and filters



(b) batch sequential

When input data are to be transformed through a series of computational or manipulative components into output data. Filters (components) work independently.



- 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

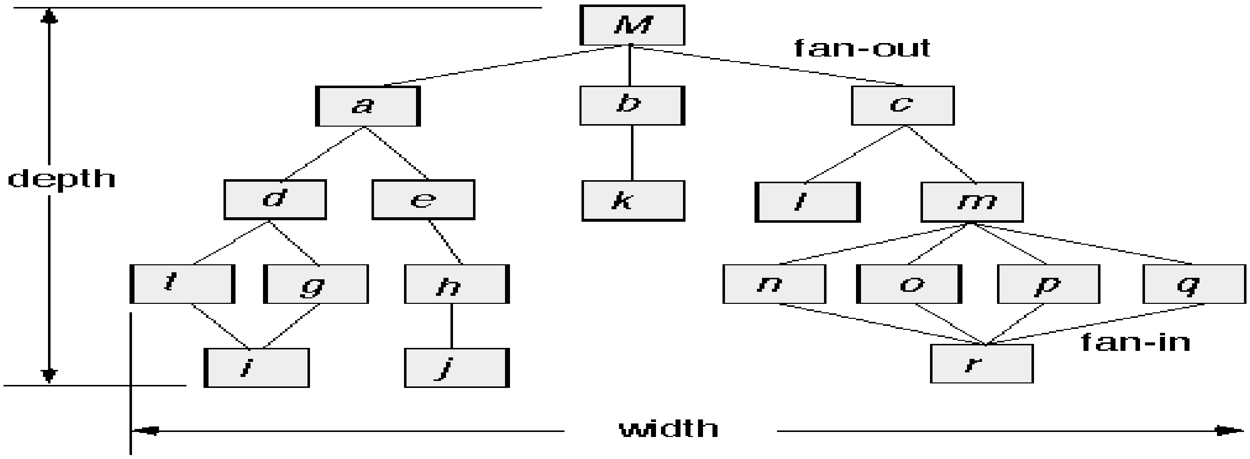
- Implicitly encourages a batch mentality so interactive applications are difficult to create in this style
- Ordering of filters can be difficult to maintain so the filters cannot cooperatively interact to solve a problem
- Exhibits poor performance
- Filters typically force the least common denominator 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

(3) Call and Return Architecture

- Main program/subprogram (classic program structure)
- Remote procedure call (components are distributed across multiple computers)
- 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

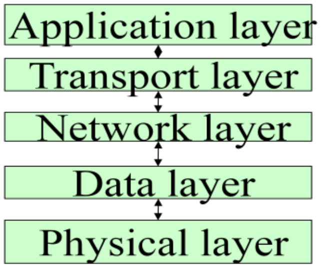
Object-oriented or abstract data type system

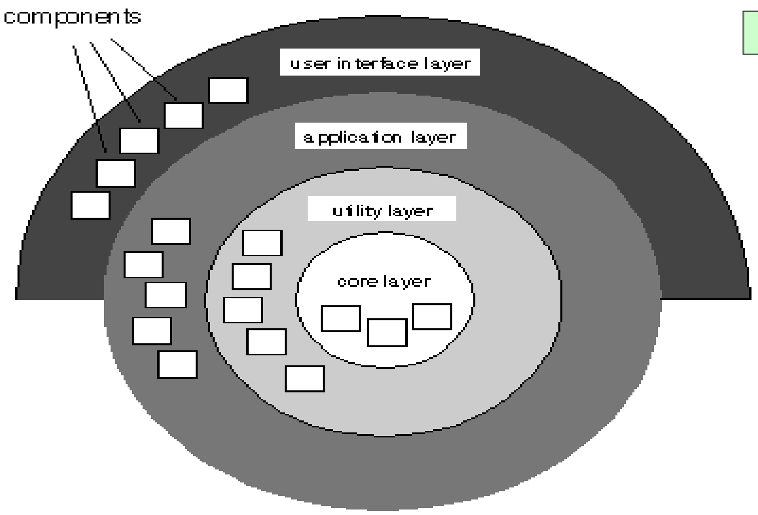
- Emphasizes the bundling of data and how to manipulate and access data
- Keeps the internal data representation hidden and allows access to the object only through provided operations
- Permits inheritance and polymorphism



(4)Layered Architecture

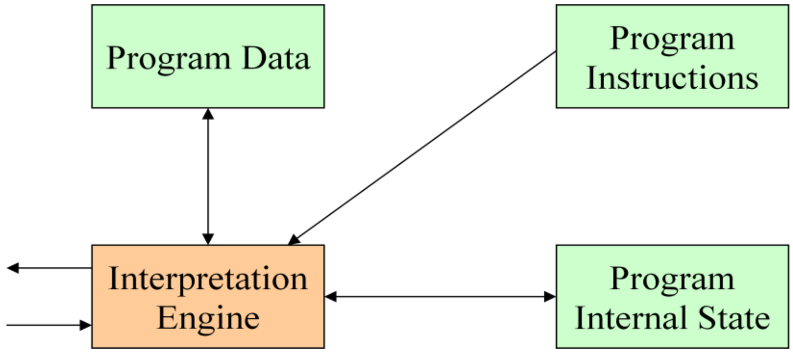
A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.





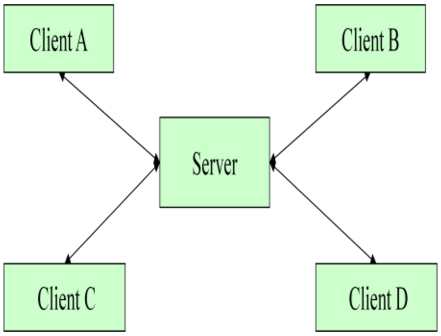
- Layered system
  - Assigns components to layers in order to control inter-component interaction
  - Only allows a layer to communicate with its immediate neighbor
  - Assigns core functionality such as hardware interfacing or system kernel operations to the lowest layer
  - Builds each successive layer 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

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

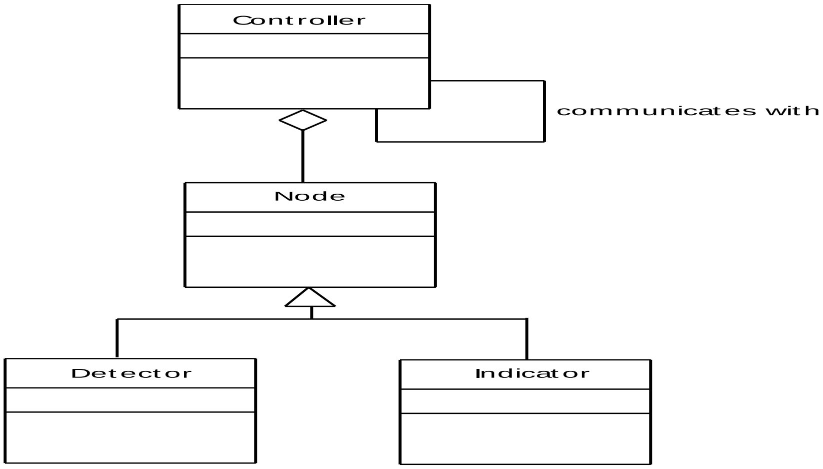
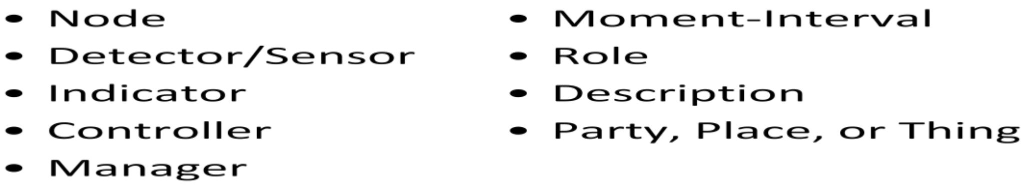


Figure 10.7 UML relationships for SafeHome security function archetypes
(adapted from [BOS00])

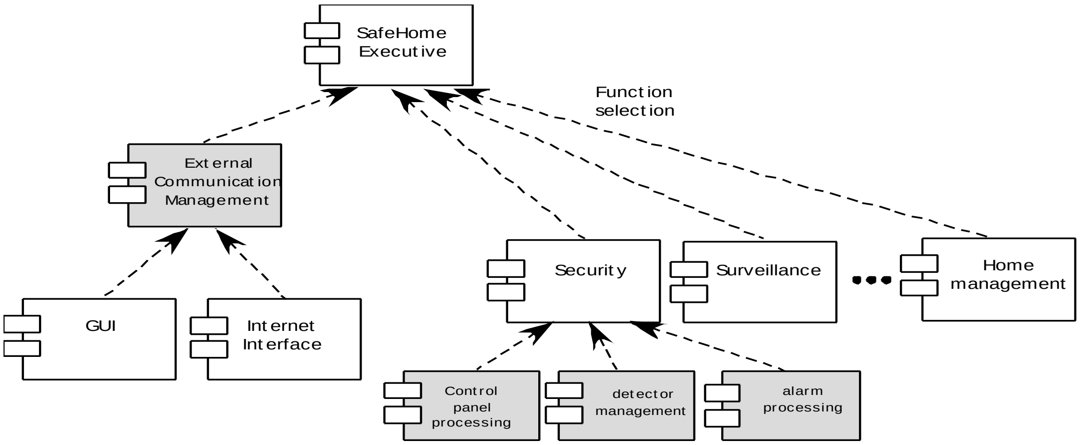


- Node
- Detector/Sensor
- Indicator
- Controller
- Manager
- Moment-Interval
- Role
- Description
- Party, Place, or Thing

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

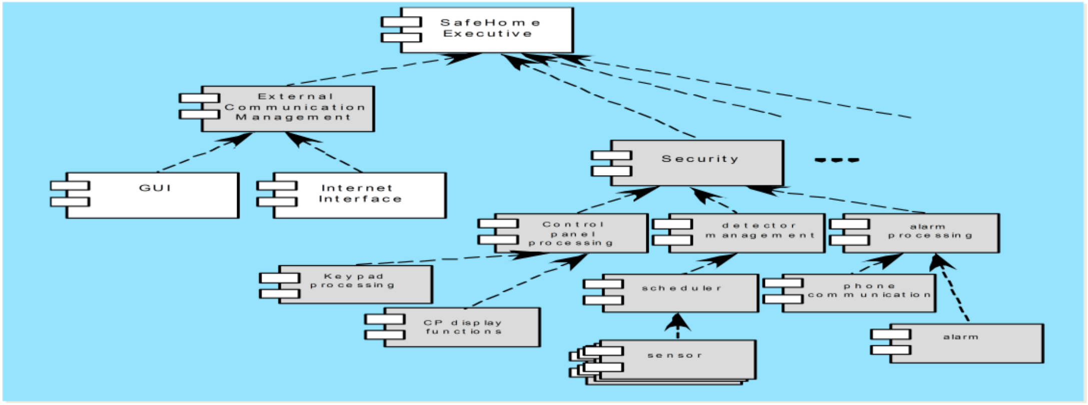
Components Example



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
-

Refined Components Example



Assessing Alternative Architectural Designs

Various Assessment Approaches

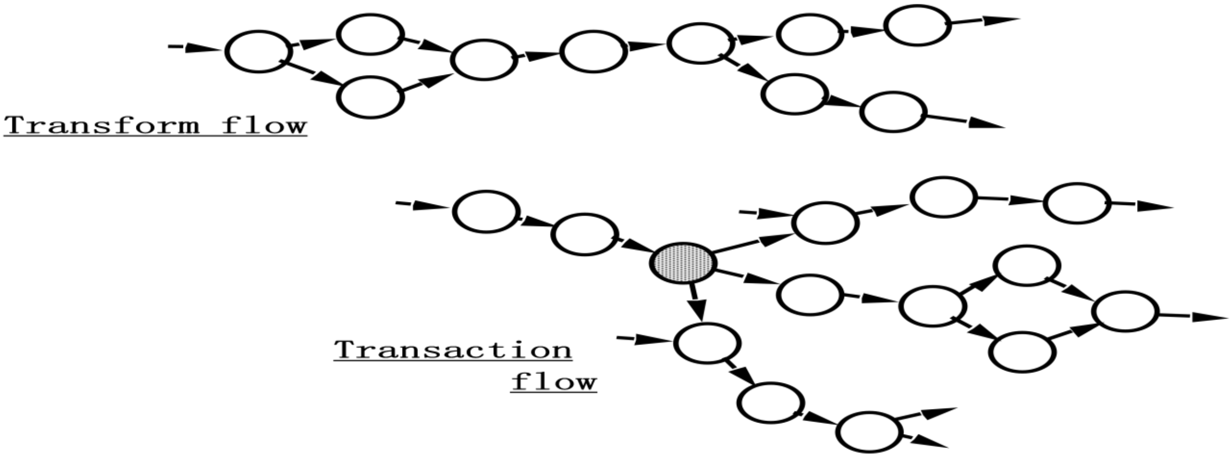
A. Ask a set of questions that provide the designer with an early assessment of design quality and lay the foundation for more detailed analysis of the architecture
   - Assess the control in an architectural design (see next slide)
   - Assess the data in an architectural design (see upcoming slide)

B. Apply the architecture trade-off analysis method
C. Assess the architectural complexity
D. Architectural Description Language

Mapping Data Flow to Architecture

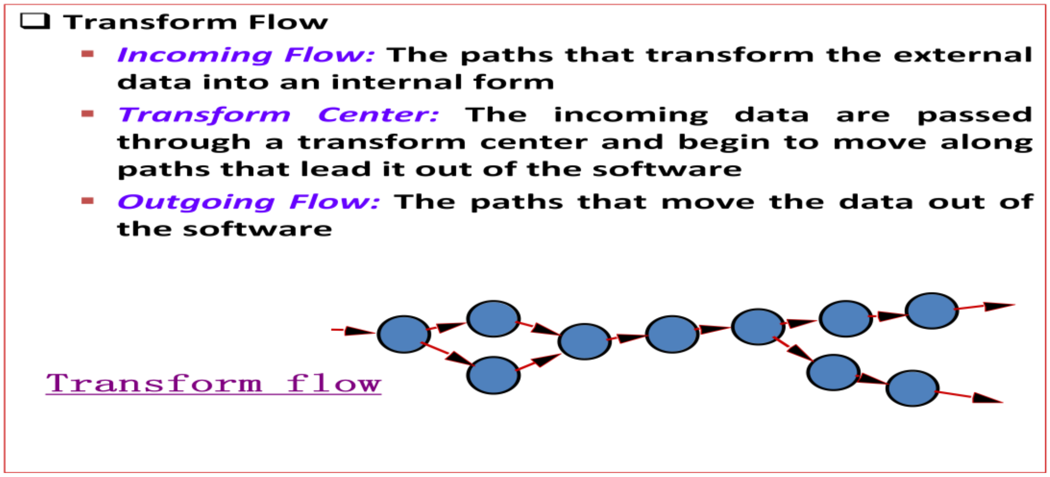
- Transform Mapping
    1. Review the fundamental system model.
    2. Review and refine data flow diagrams for the software
    3. Determine whether the DFD has transform or transaction flow characteristics.
    4. Isolate the transform center by specifying incoming and outgoing flow boundaries.
    5. Perform “first-level factoring”
    6. Perform “second-level factoring”
    7. Refine the first-iteration architecture using design heuristics for improved software quality.

Flow Characteristics



Transform Flow

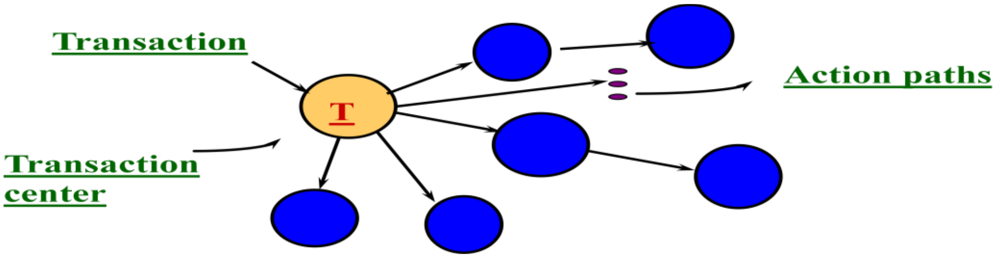
- Incoming Flow: The paths that transform the external data into an internal form
- Transform Center: The incoming data are passed through a transform center and begin to move along paths that lead it out of the software
- Outgoing Flow: The paths that move the data out of the software



❑ Transform Flow

- *Incoming Flow:* The paths that transform the external data into an internal form
- *Transform Center:* The incoming data are passed through a transform center and begin to move along paths that lead it out of the software
- *Outgoing Flow:* The paths that move the data out of the software

Transform flow



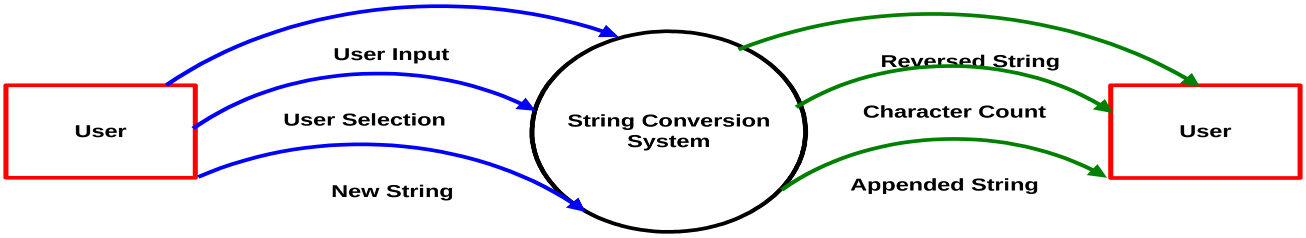
Transform Mapping

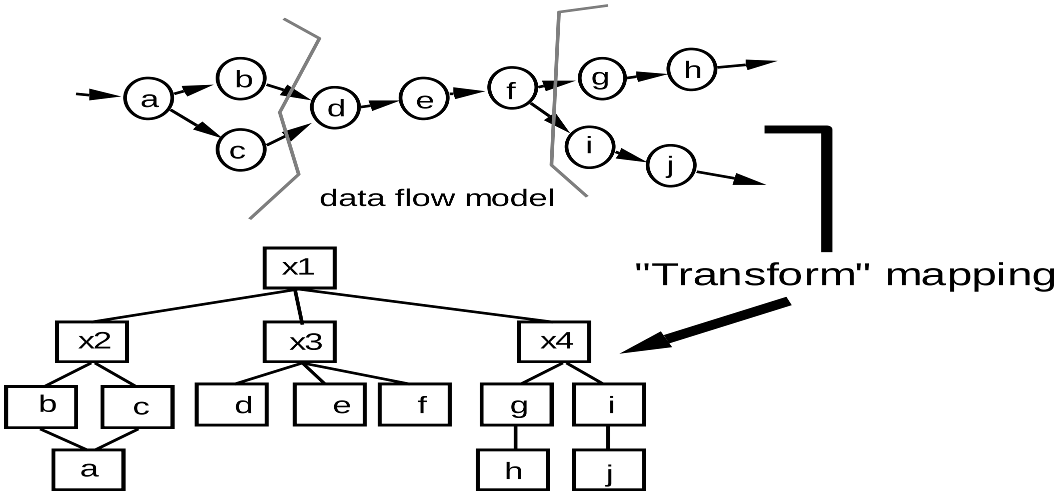
Design steps

Step 1. Review the fundamental system model.
Step 2. Review and refine data flow diagrams for the software.
Step 3. Determine whether DFD has transform or transaction flow characteristics.

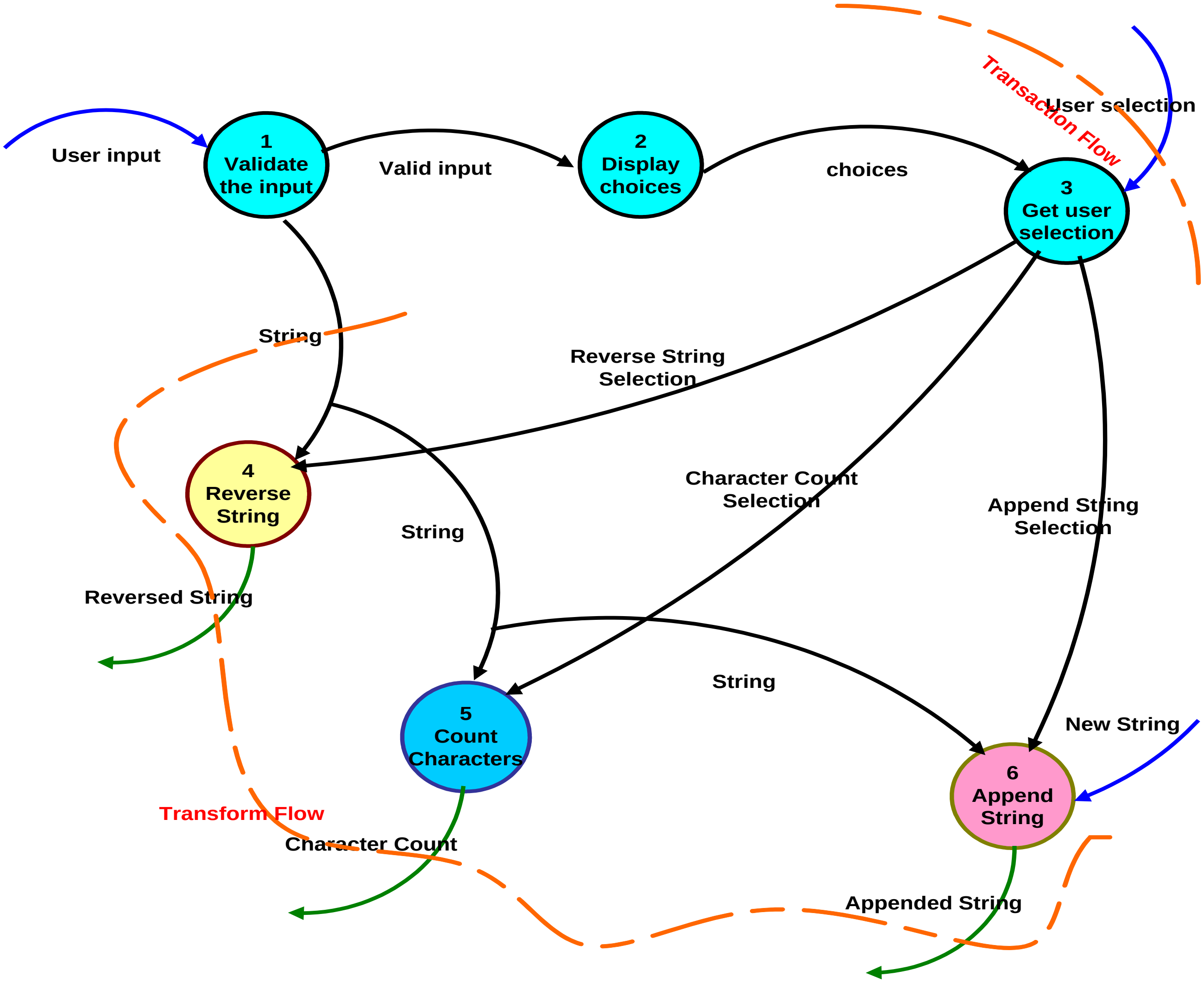
- in general---transform flow
- special case---transaction flow

- **Context Level DFD**

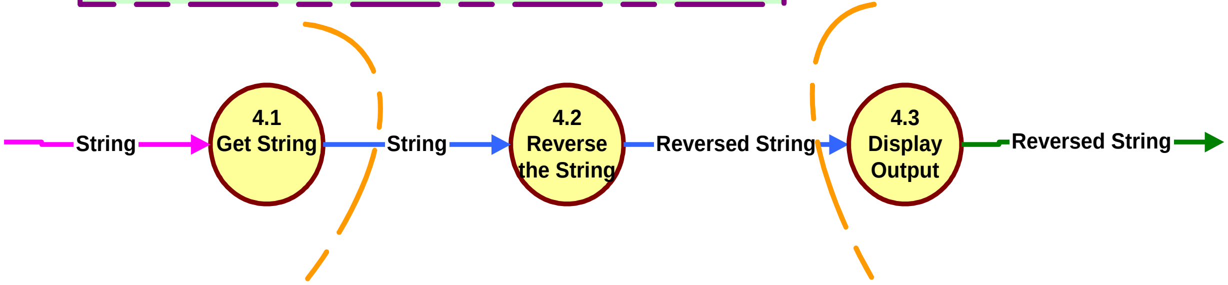



a
b
c
d
e
f
g
h
i
j
data flow model
"Transform" mapping
x1
x2
x3
x4
b
c
a
d
e
f
g
i
h
j

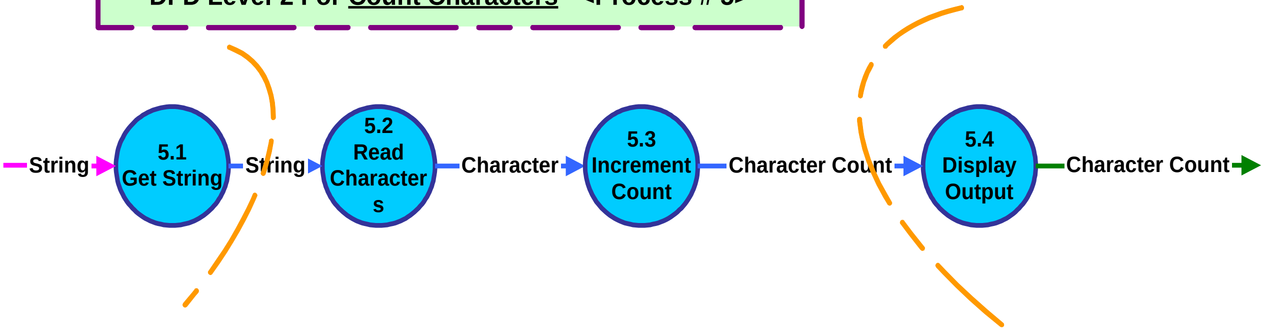
LEVEL 1 DFD:


User input
1 Validate the input
Valid input
2 Display choices
choices
3 Get user selection
User selection
Transaction Flow
String
Reverse String Selection
4 Reverse String
String
Character Count Selection
Append String Selection
Reversed String
String
5 Count Characters
New String
6 Append String
Transform Flow
Character Count
Appended String

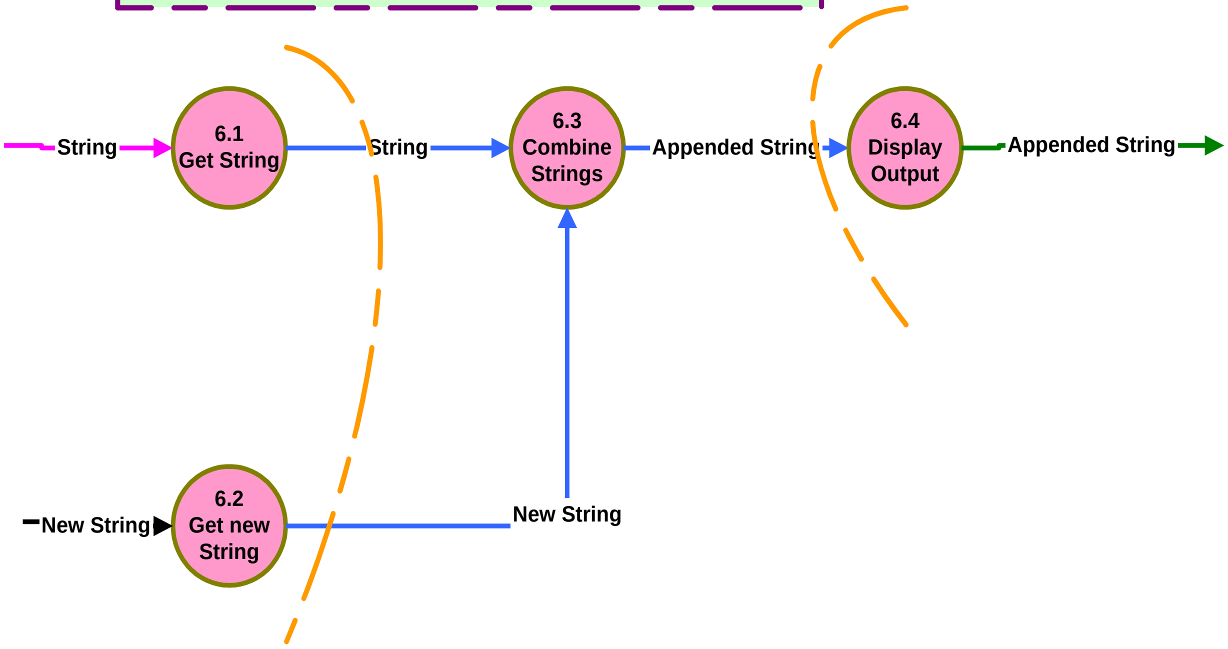
DFD Level-2 For REVERSE STRING - <Process # 4>



DFD Level-2 For Count Characters - <Process # 5>

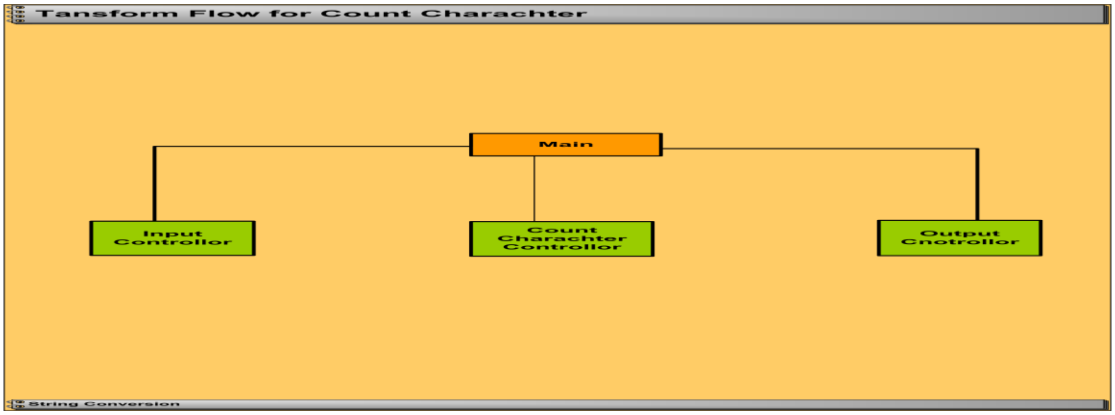


DFD Level-2 For Append STRING - <Process # 6>



- **Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries**
  - different designers may select slightly differently
  - transform center can contain more than one bubble.
- **Step 5. Perform "first-level factoring"**
  - program structure represent a top-down distribution control.
  - factoring results in a program structure(top-level, middle-level, low-level)
  - number of modules limited to minimum.

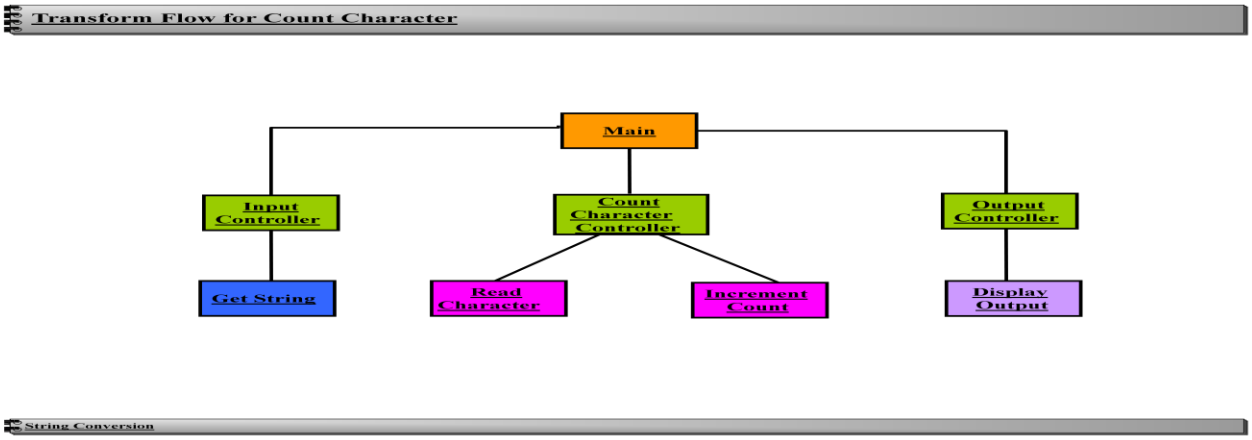
First Level Factoring


Tansform Flow for Count Charachter
Main
Input Controllor
Count Charachter Controllor
Output Cnotrollor
String Conversion

Step 6. Perform "second-level factoring"

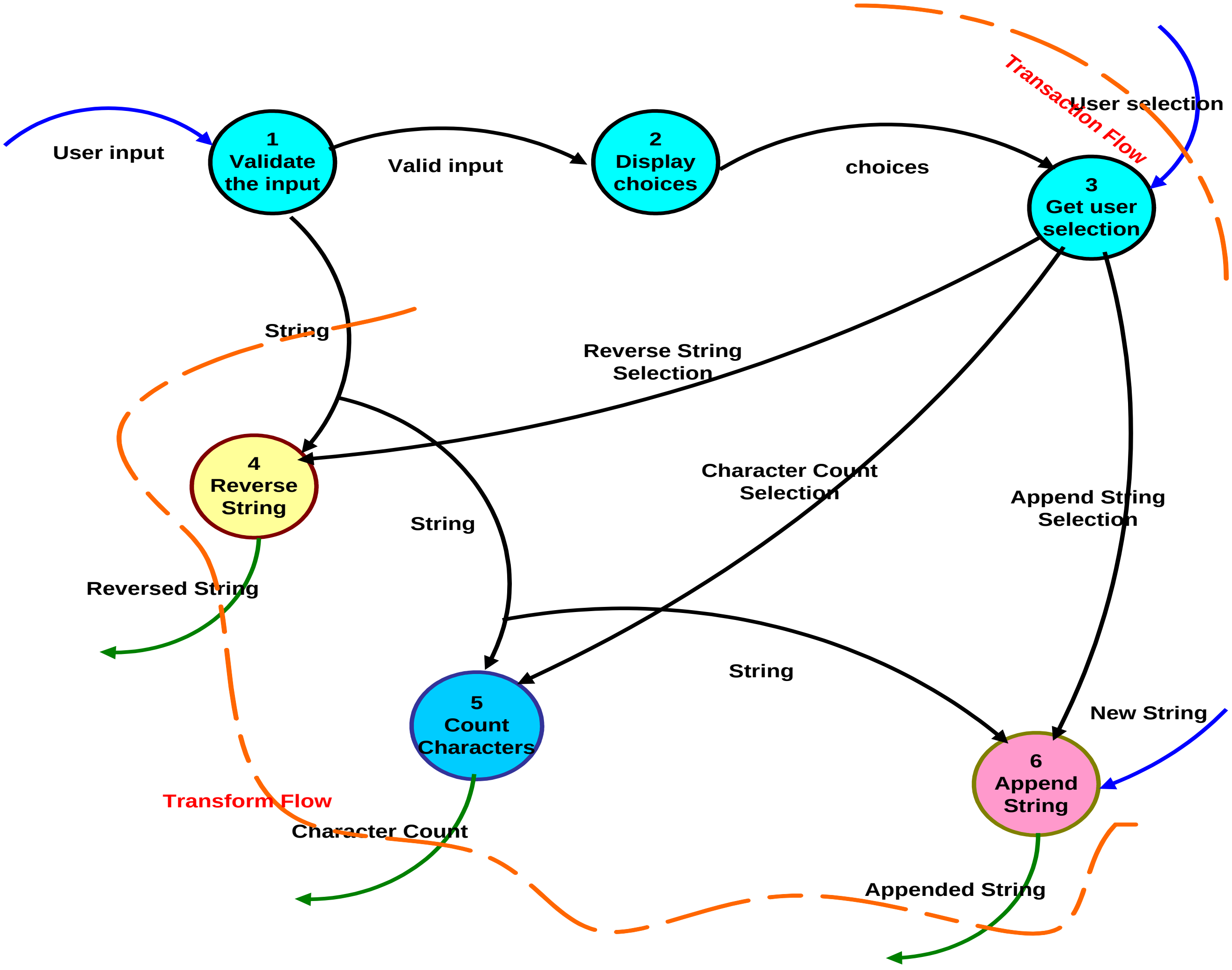
- mapping individual transforms(bubbles) to appropriate modules.
- factoring accomplished by moving outwards from transform center boundary.

Step 7. Refine the first iteration program structure using design heuristics for improved software quality.



Transaction Mapping

A single data item triggers one or more information flows



Transaction Mapping Design

- **Step 1.Review the fundamental system model.**
- **Step 2.Review and refine DFD for the software**
- **Step 3.Determine whether the DFD has transform or transaction flow characteristics**
- **Step 4. Identify the transaction center and flow characteristics along each of the action paths**
  - isolate incoming path and all action paths
  - each action path evaluated for its flow characteristic.

- **Step 5. Map the DFD in a program structure amenable to transaction processing**
  - *incoming branch*
    - bubbles along this path map to modules
  - *dispatch branch*
    - dispatcher module controls all subordinate action modules
    - each action path mapped to corresponding structure

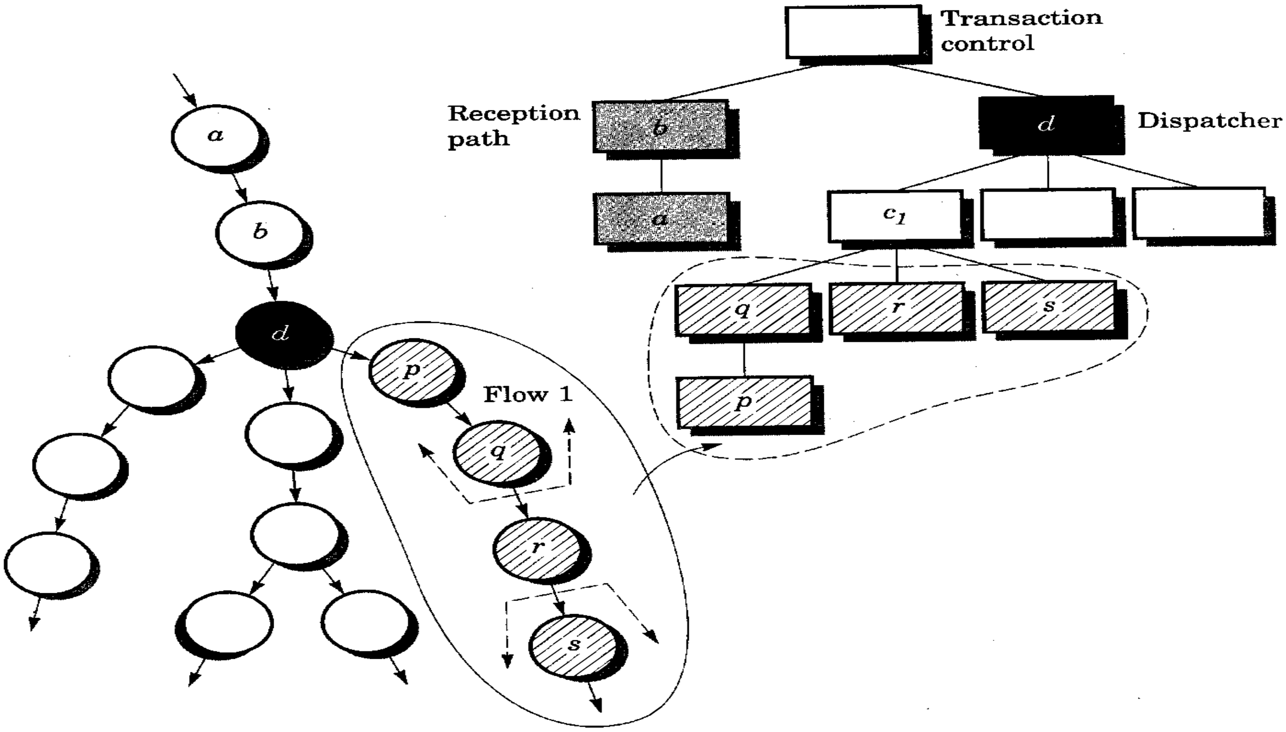
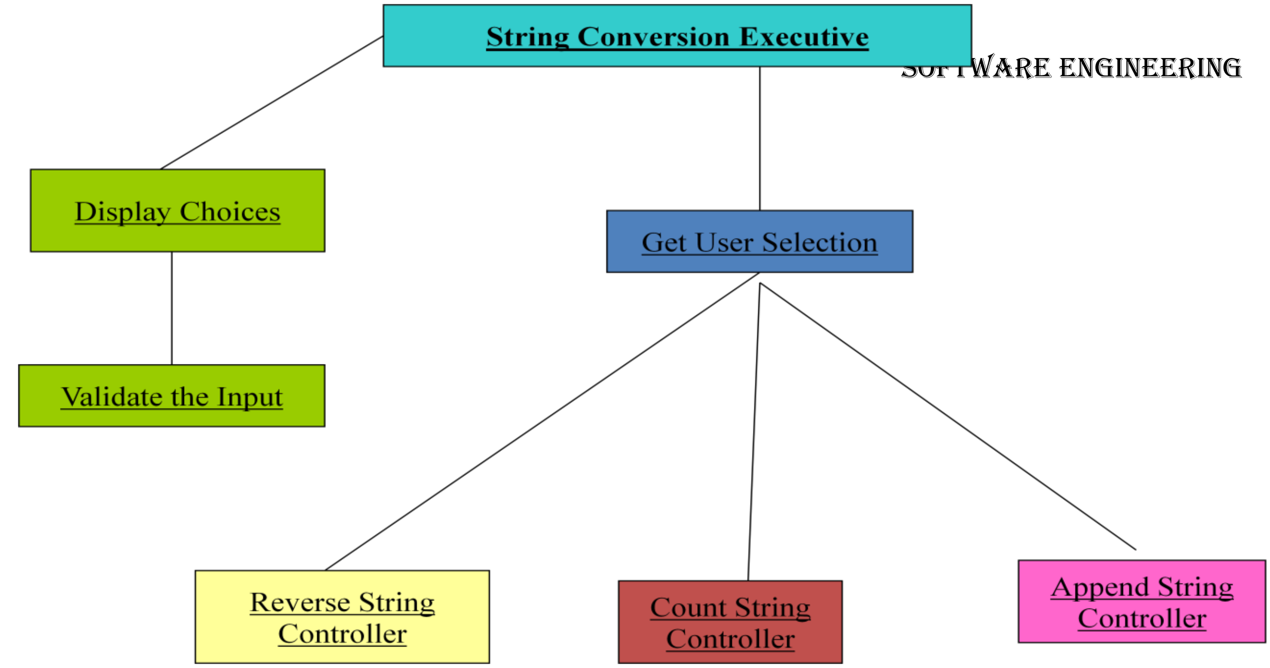
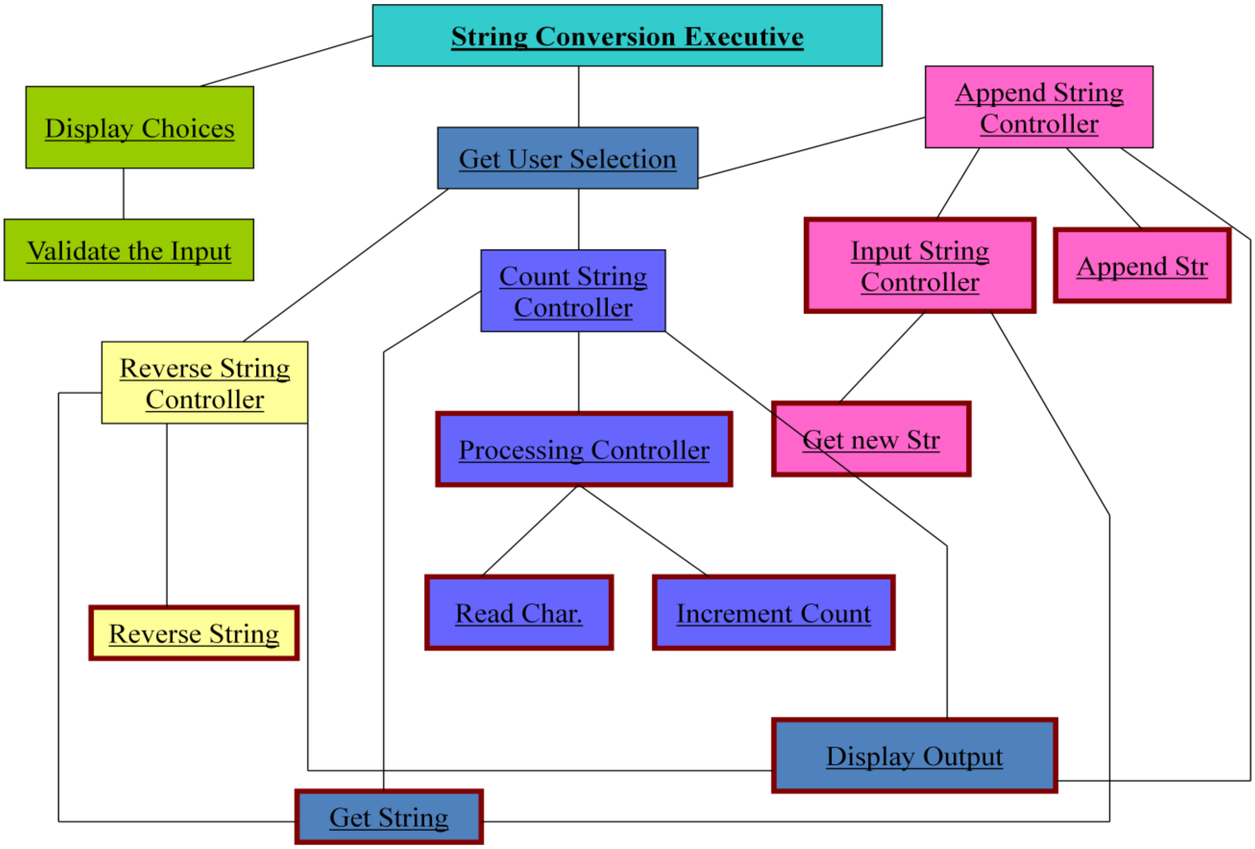


FIGURE 14.12. Transaction mapping

First Level Factoring





- **Step 6. Factor and refine the transaction structure and the structure of each action path**
- **Step 7. Refine the first iteration program structure using design heuristics for improved software quality**

Modeling Component-Level Design:

Introduction:

- A complete set of software components is defined during architectural design
- But the internal data structures and processing details of each component are not represented at a level of abstraction that is close to code
- Component-level design defines the data structures, algorithms, interface characteristics, and communication mechanisms allocated to each component

Background:

- 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
- 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

Component:

- "A modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
- 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**
    1. An object-oriented view
    2. A conventional view
    3. A process-related view

1. **Object-oriented View**

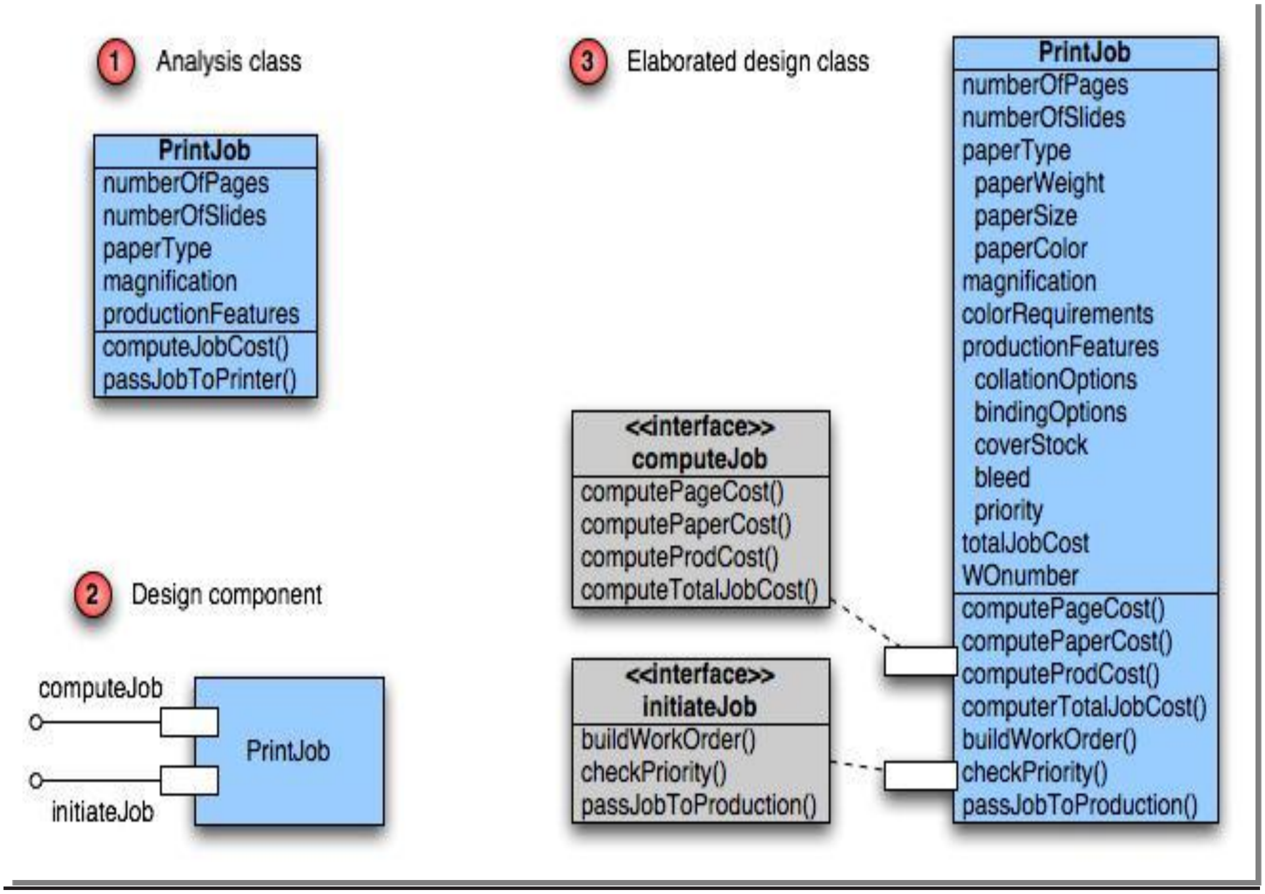
- A component is viewed as a set of one or more collaborating classes
- Each problem domain (i.e., analysis) class and infrastructure (i.e., design) class is elaborated to identify all attributes and operations that apply to its implementation
    - This also involves defining the interfaces that enable classes to communicate and collaborate
- This elaboration activity is applied to every component defined as part of the architectural design

- Once this is completed, the following steps are performed

1) Provide further elaboration of each attribute, operation, and interface
2) Specify the data structure appropriate for each attribute
3) Design the algorithmic detail required to implement the processing logic associated with each operation
4) Design the mechanisms required to implement the interface to include the messaging that occurs between objects

..

Eg: Class Elaboration

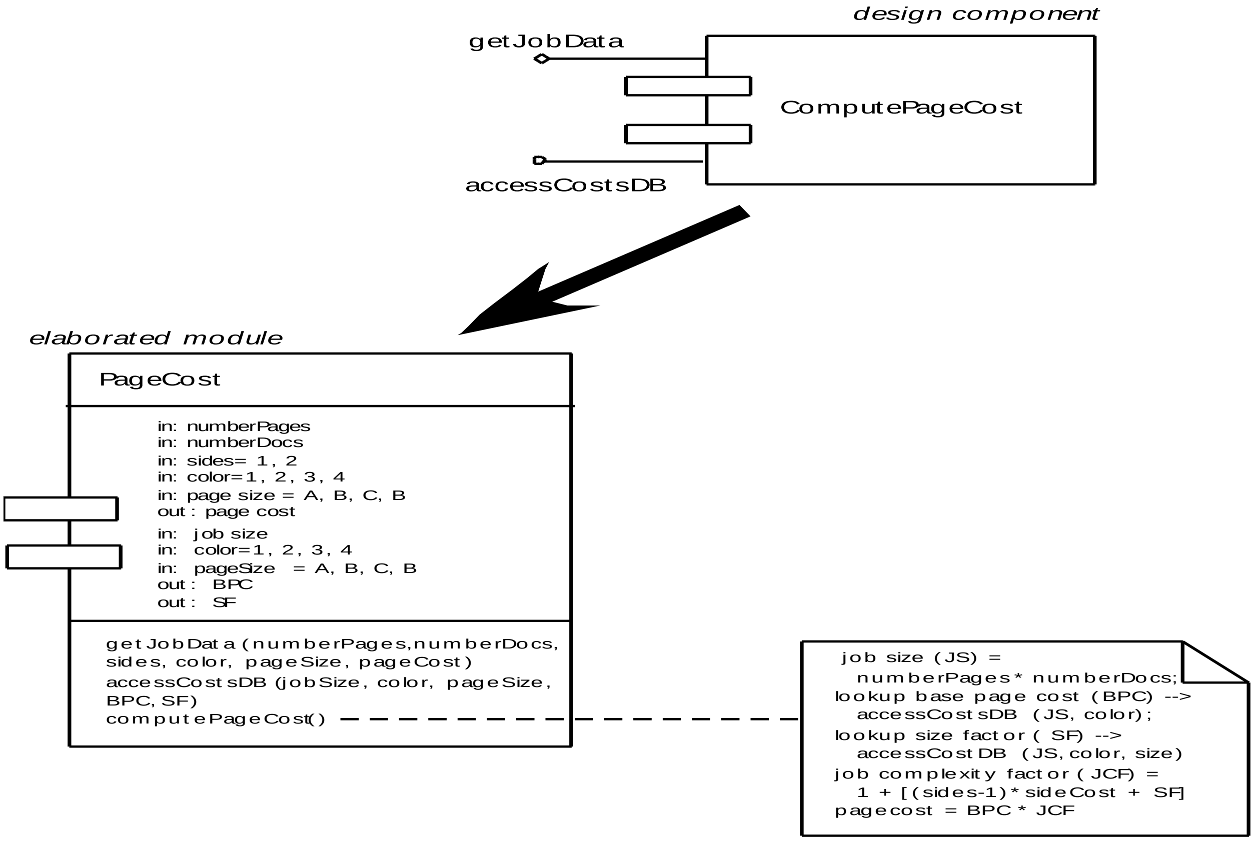


Conventional View

- 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)

Example: Conventional Component


design component
getJobData
ComputePageCost
accessCostsDB
elaborated module
PageCost
in: numberPages
in: numberDocs
in: sides= 1, 2
in: color=1, 2, 3, 4
in: page size = A, B, C, B
out : page cost
in: job size
in: color=1, 2, 3, 4
in: pageSize = A, B, C, B
out : BPC
out : SF
getJobData (numberPages,numberDocs, sides, color, pageSize, pageCost)
accessCostsDB (jobSize, color, pageSize, BPC, SF)
computePageCost()
job size (JS) =
numberPages* numberDocs;
lookup base page cost (BPC) -->
accessCostsDB (JS, color);
lookup size factor (SF) -->
accessCostDB (JS, color, size)
job complexity factor (JCF) =
1 + [(sides-1)*sideCost + SF]
pagecost = BPC * JCF

Process-related View

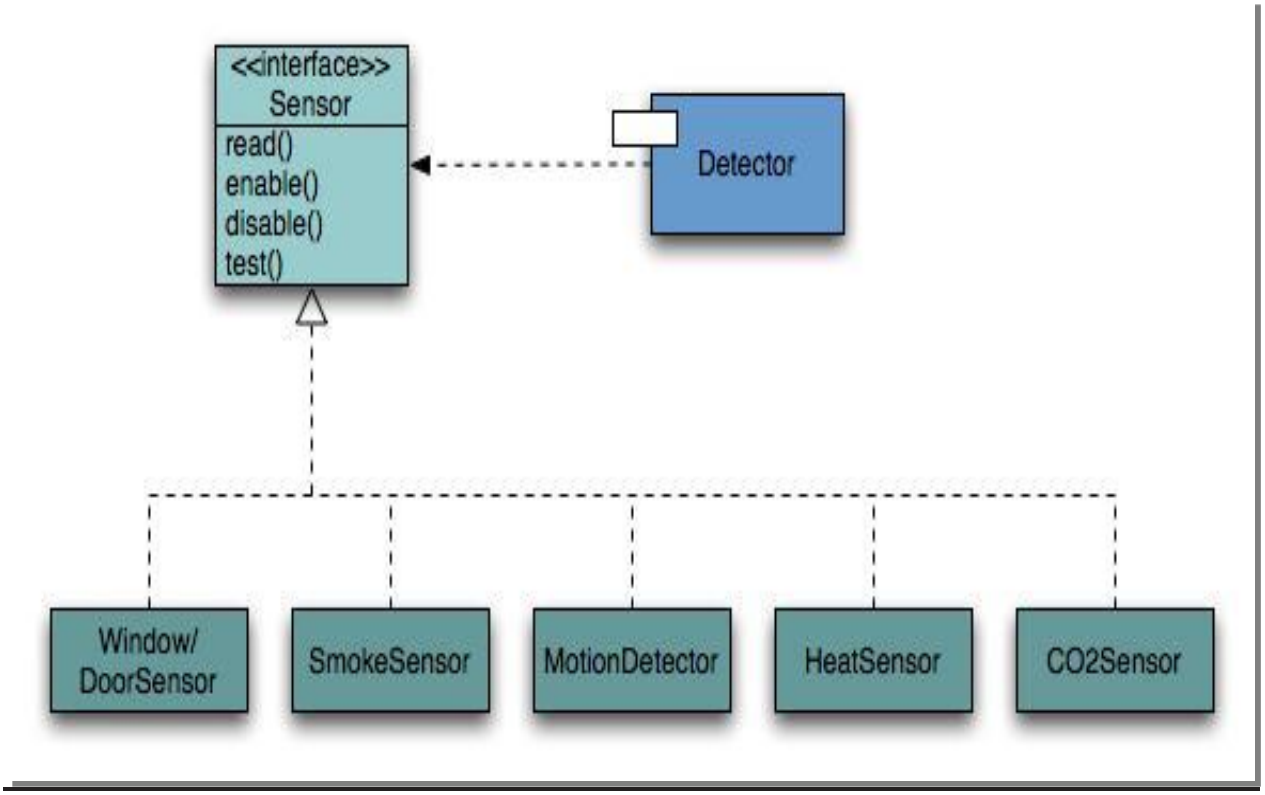
- Emphasis is placed on building systems from existing components maintained in a library rather than creating each component from scratch
- As the software architecture is formulated, components are selected from the library and used to populate the architecture
- Because the components in the library 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

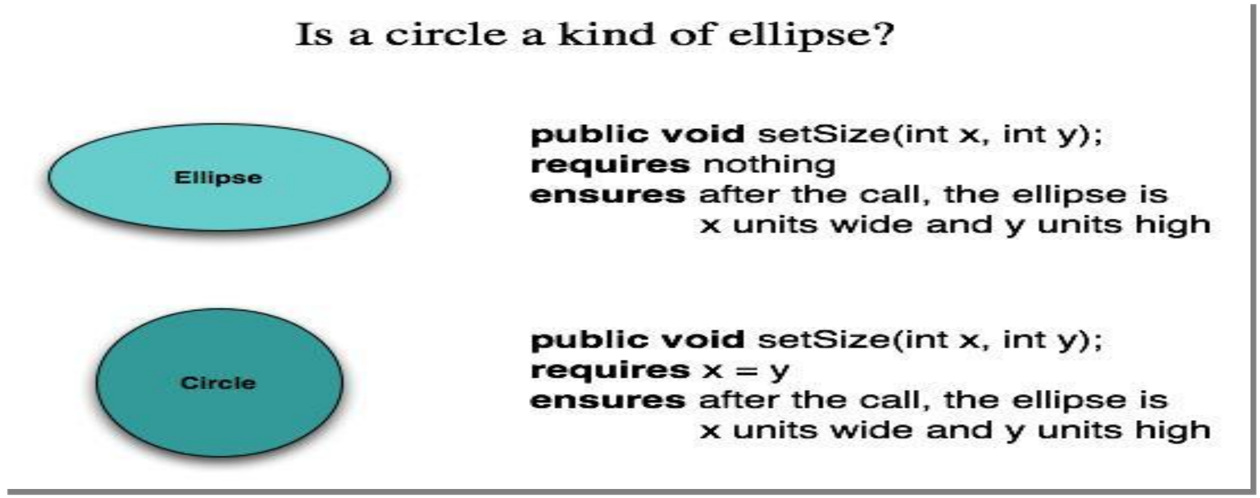
Component-level Design Principles:

- **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

A module should be open for extension but closed for modification.



- **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

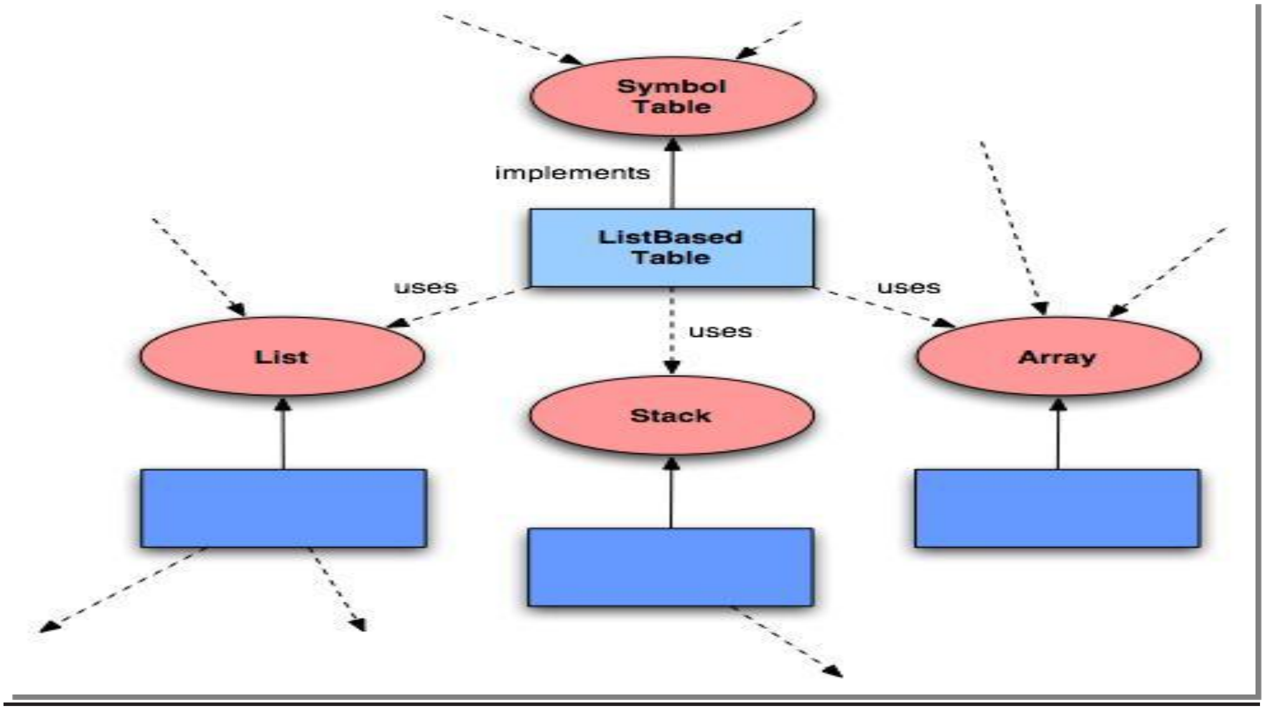


Is a circle a kind of ellipse?

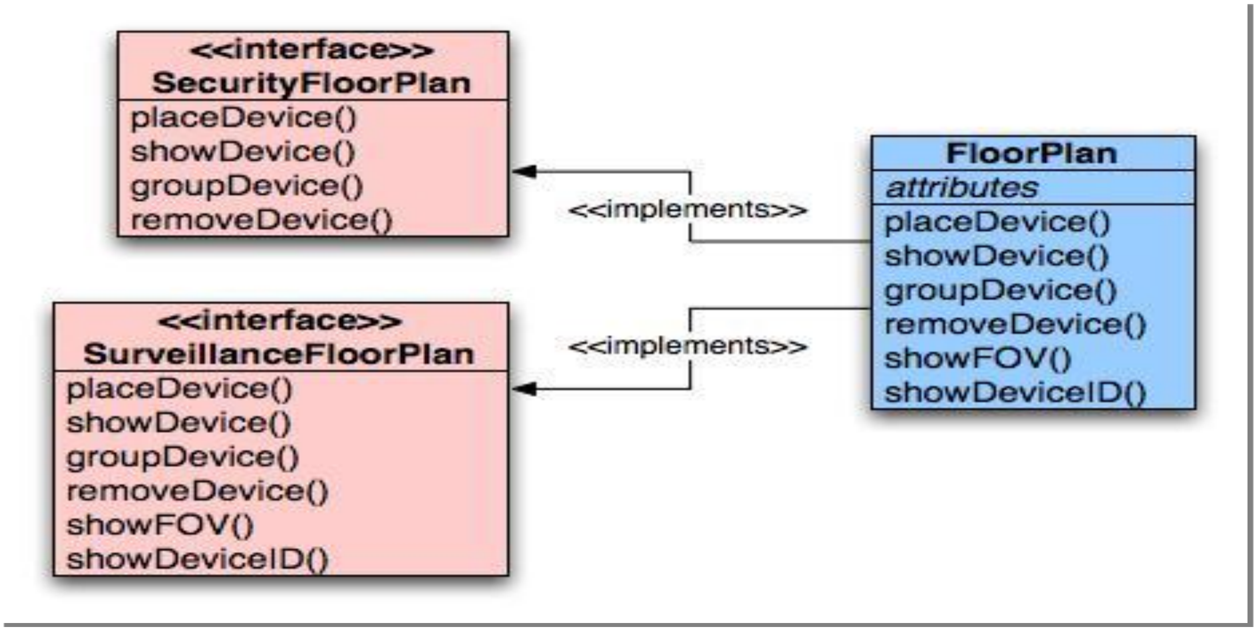
```
public void setSize(int x, int y);
requires nothing
ensures after the call, the ellipse is
        x units wide and y units high
```

```
public void setSize(int x, int y);
requires x = y
ensures after the call, the ellipse is
        x units wide and y units high
```

- **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

Conducting Component-Level Design

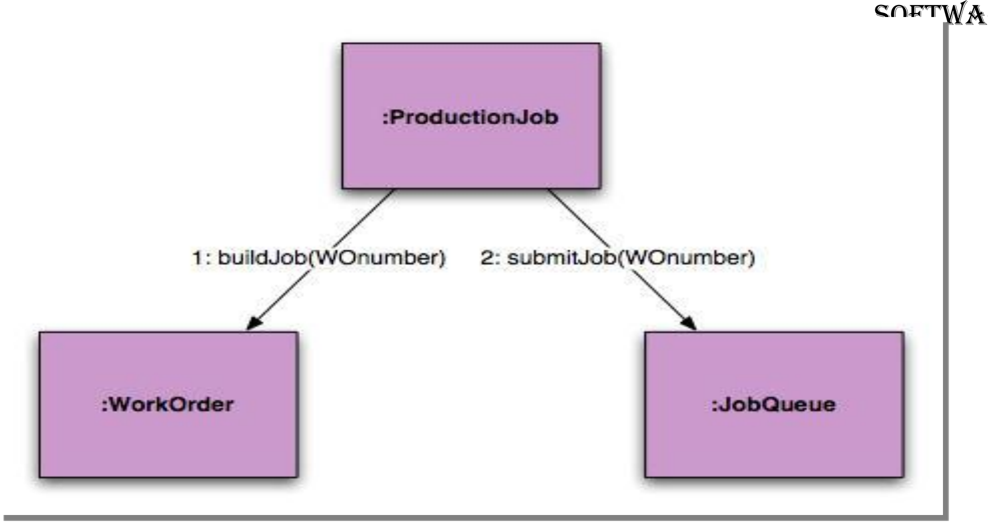
1) Identify all design classes that correspond to the problem domain as defined in the analysis model and architectural model
2) Identify all design classes that correspond to the infrastructure domain
   - These classes are usually not present in the analysis or architectural models
   - These classes include GUI components, operating system components, data management components, networking components, etc.

3) Elaborate all design classes that are not acquired as reusable components

   a) Specify message details (i.e., structure) when classes or components collaborate
   b) Identify appropriate interfaces (e.g., abstract classes) for each component
   c) Elaborate attributes and define data types and data structures required to implement them (usually in the planned implementation language)
   d) Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams

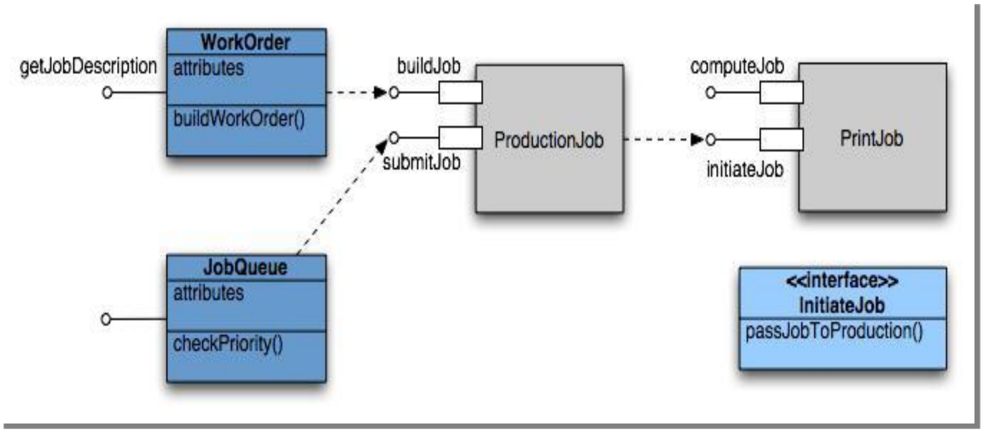
   3a. Collaboration Details

   - Messages can be elaborated by expanding their syntax in the following manner:
     - [guard condition] sequence expression (return value) := message name (argument list)



3b. Appropriate Interfaces

- Pressman argues that the PrintJob interface “initiateJob” in slide 5 does not exhibit sufficient cohesion because it performs three different subfunctions. He suggests this refactoring.



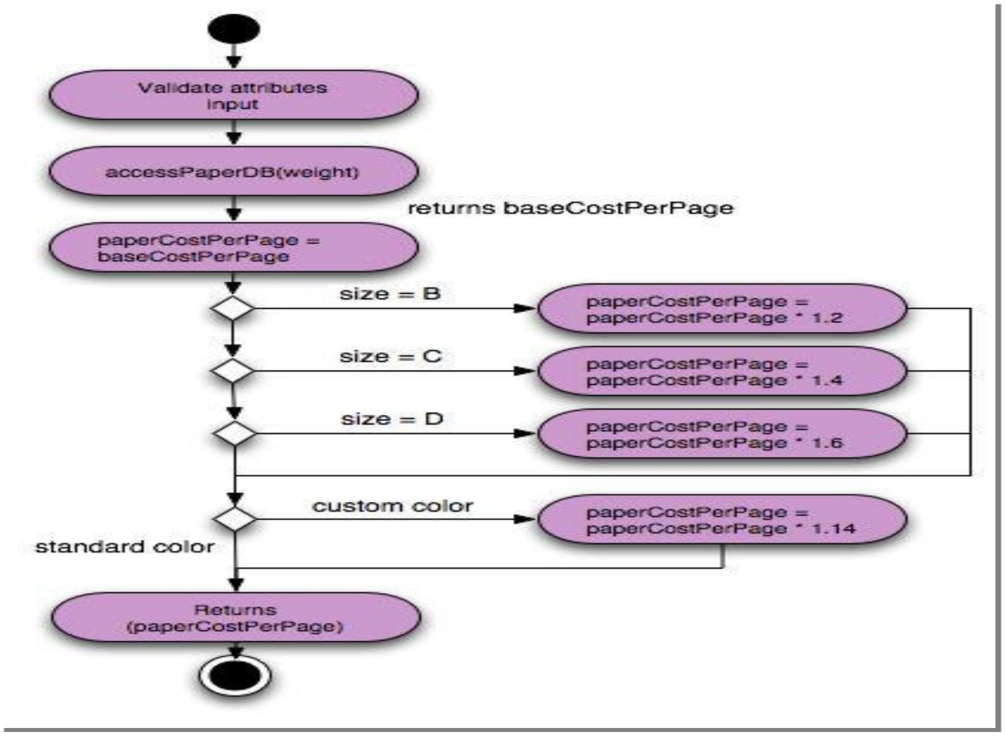
3c. Elaborate Attributes

- Analysis classes will typically only list names of general attributes (ex. paperType).
- List all attributes during component design.
- UML syntax:
  - name : type-expression = initial-value { property string }
- For example, paperType can be broken into weight, size, and color. The weight attribute would be:
  - paperType-weight: string =

“A” { contains 1 of 4 values – A, B, C, or D }

3d. Describe Processing Flow

Activity diagram for computePaperCost( )


Validate attributes input
accessPaperDB(weight)
returns baseCostPerPage
paperCostPerPage = baseCostPerPage
size = B
paperCostPerPage = paperCostPerPage * 1.2
size = C
paperCostPerPage = paperCostPerPage * 1.4
size = D
paperCostPerPage = paperCostPerPage * 1.6
custom color
paperCostPerPage = paperCostPerPage * 1.14
standard color
Returns (paperCostPerPage)

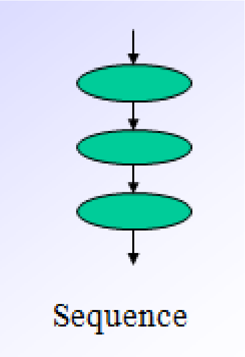
4) Describe persistent data sources (databases and files) and identify the classes required to manage them
5) Develop and elaborate behavioral representations for a class or component
    1) This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class
6) Elaborate deployment diagrams to provide additional implementation detail
    1) 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
    1) Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model
    2) The final decision can be made by using established design principles and guidelines

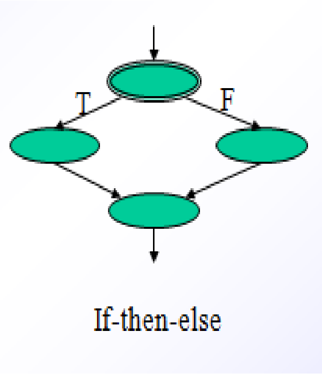
Designing Conventional Components

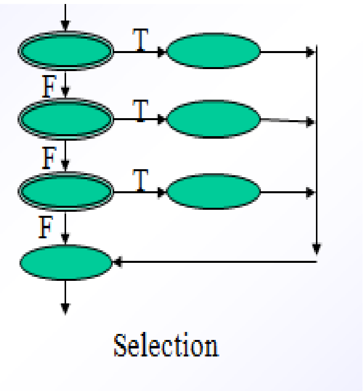
- Conventional design constructs emphasize the maintainability of a functional/procedural program
    - Sequence, condition, and repetition
- Each construct has a predictable logical structure where control enters at the top and exits at the bottom, enabling a maintainer to easily follow the procedural flow

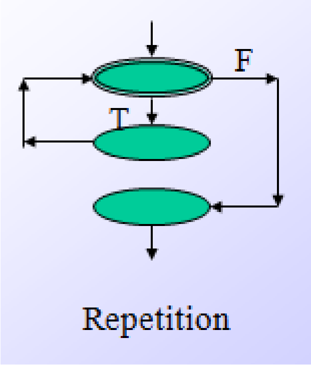
- Various notations depict the use of these constructs
  - Graphical design notation
    - Sequence, if-then-else, selection, repetition (see next slide)
  - Tabular design notation (see upcoming slide)
  - Program design language
    - Similar to a programming language; however, it uses narrative text embedded directly within the program statements

Graphical Design Notation

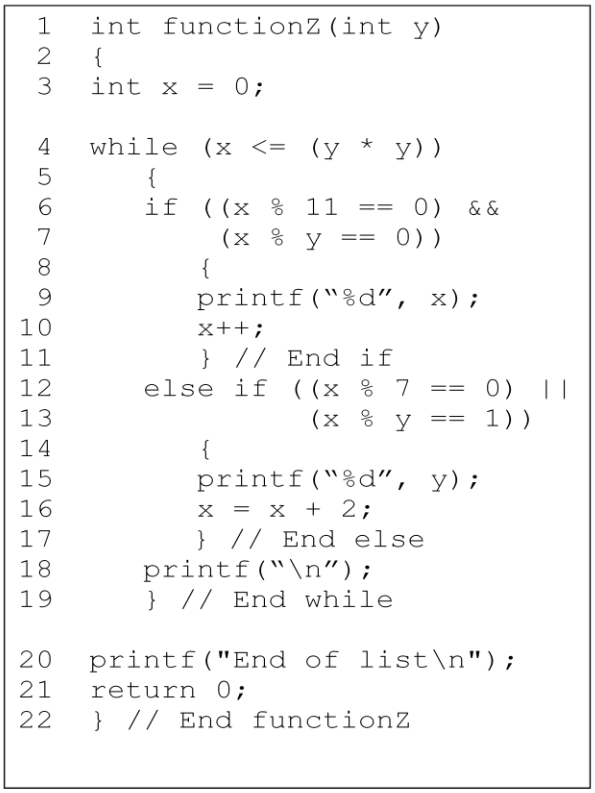




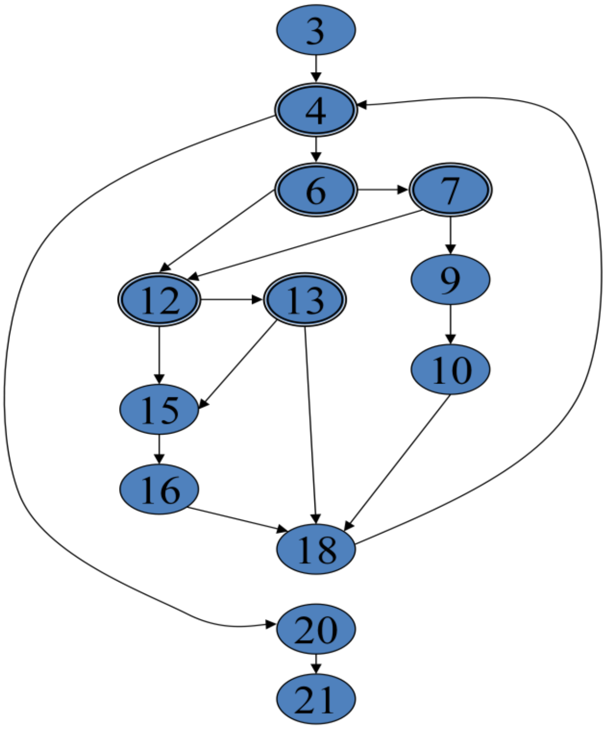




Graphical Example used for Algorithm Analysis

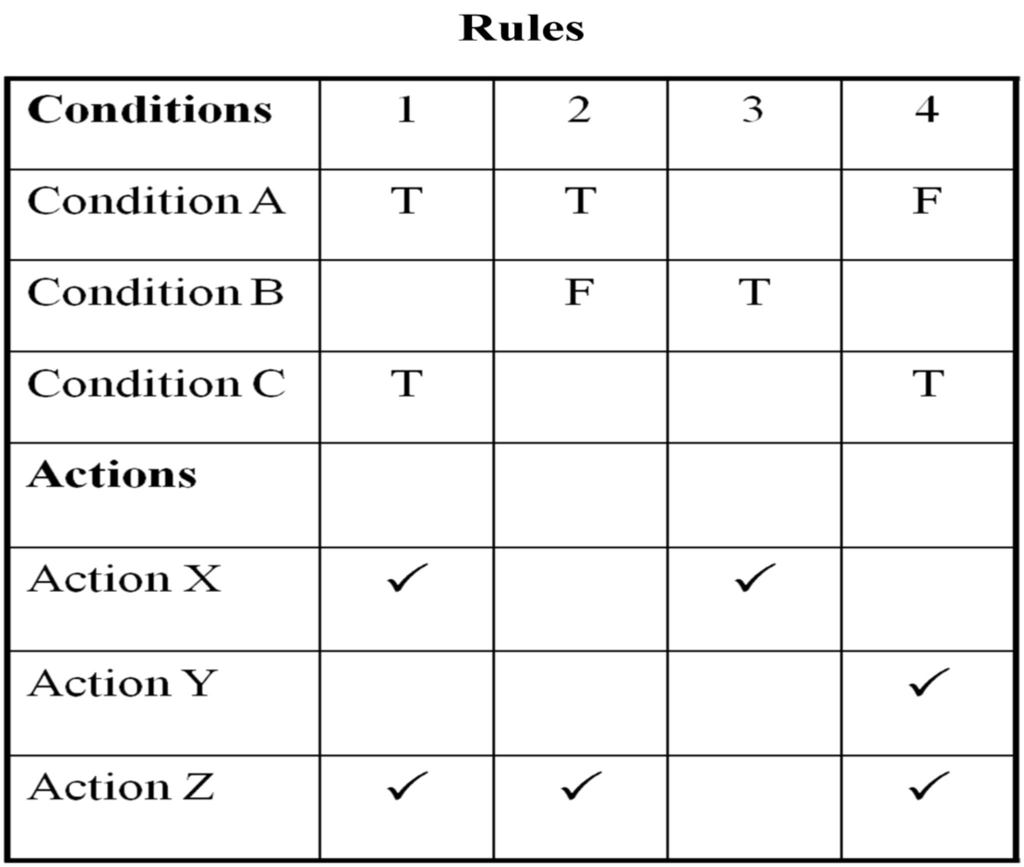


```
 1  int functionZ(int y)
 2  {
 3  int x = 0;

 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	✓	✓		✓

Performing User Interface Design

Three "golden rules":
1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important software design activity.

1. Place the User in Control

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface.
Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction

A number of design principles that allow the user to maintain control:

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

An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell checking mode. The user should be able to enter and exit the mode with little or no effort.

(b) Provide for flexible interaction

Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, or voice recognition commands. But every action is not amenable to every interaction mechanism.

[c] Allow user interaction to be interruptible and undoable

Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else. The user should also be able to “undo” any action.

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

Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a “macro” mechanism that enables an advanced user to customize the interface to facilitate interaction.

(e) Hide technical internals from the casual user

The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions etc.

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

The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing.

2. Reduce the User's Memory Load

The more a user has to remember, the more error-prone will be the interaction with the system. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user

Design principles that enable an interface to reduce the user's memory load:

(a) Reduce demand on short-term memory

When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

(b) Establish meaningful defaults.

The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

[c] Define shortcuts that are intuitive

When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember

(d) The visual layout of the interface should be based on a real world metaphor

For example, a bill payment system should use a check book and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

(e) Disclose information in a progressive fashion

The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.
An example, common to many word-processing applications, is the underlining function.

3. Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to a design standard that is maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that are used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented.

A set of design principles that help make the interface consistent:

(a) Allow the user to put the current task into a meaningful context

Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

(b) Maintain consistency across a family of applications

A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

[c] 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., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

USER INTERFACE ANALYSIS AND DESIGN

The overall process for designing a user interface begins with the creation of different models of system function. The human- and computer-oriented tasks that are required to achieve system function are then delineated; design issues that apply to all interface designs are considered; tools are used to prototype and ultimately implement the design model; and the result is evaluated for quality.

Interface Analysis and Design Models

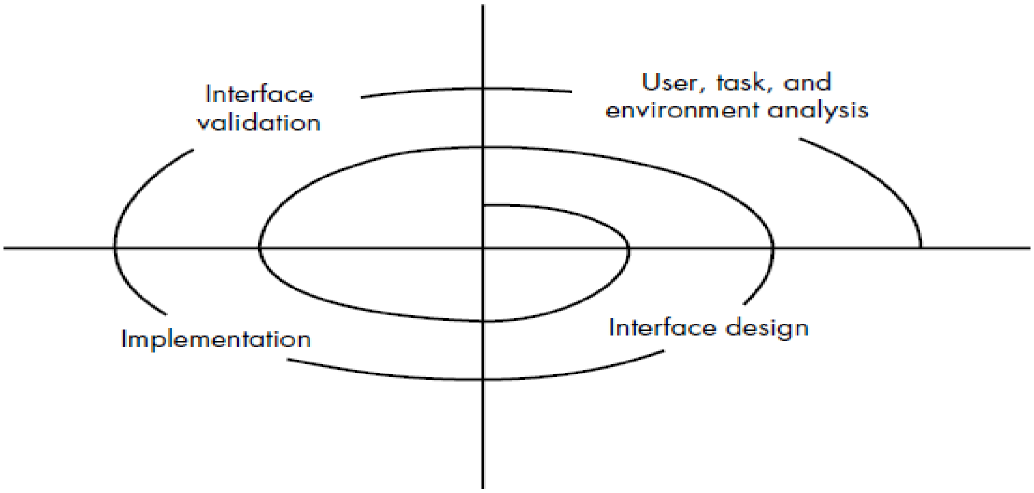
Four different models come into play when a user interface is to be designed. The software engineer creates a design model, a human engineer (or the software engineer) establishes a user model. The role of interface designer is to reconcile these differences and derive a consistent representation of the interface.

To build an effective user interface, "all design should begin with an understanding of the intended users.

(Classification of user)

Users can be categorized as
• Novices.
No syntactic knowledge2 of the system and little semantic knowledge application or computer usage in general.
• Knowledgeable, intermittent users.
Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.
• Knowledgeable, frequent users.
Good semantic and syntactic knowledge that often leads to the "power-user syndrome"; that is, individuals who look for shortcuts and abbreviated modes of interaction.

The Process



The design process for user interfaces is iterative and can be represented using a spiral Model and encompasses four distinct framework activities:

1. User, task, and environment analysis and modeling
2. Interface design
3. Interface construction
4. Interface validation

The spiral shown in the above figure implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. The initial analysis activity focuses on the profile of the users who will interact with the system. The software engineer attempts to understand the system perception for each class of users.

Once general **requirements** have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral).

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences. The goal of **interface design** is to define 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.

The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit

Validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn; and (3) the users' acceptance of the interface as a useful tool in their work. may be used to complete the construction of the 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
  - Are the users trained professionals, technicians, clerical or manufacturing workers?

- What level of formal education does the average user have?
- Are the users capable of learning on their own from written materials or have they expressed a desire for classroom training?
- Are the users expert typists or are they keyboard phobic? Etc

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
  - Are various types of data assigned to consistent locations on the screen (e.g., photos always in upper right corner)?
  - Are users able to customize the screen location for content?
  - Is proper on-screen identification assigned to all content?
  - Can large reports be partitioned for ease of understanding?
  - Are mechanisms available for moving directly to summary information for large collections of data?
  - Is graphical output scaled to fit within the bounds of the display device that is used?
  - How is color used to enhance understanding?

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 <u>objects</u> and <u>actions</u> (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

<u>Interface Objects and Actions</u>

- Interface objects and actions are obtained from a <u>grammatical parse</u> of the use cases and the software problem statement
- Interface objects are categorized into types: source, target, and application
  - A <u>source</u> object is dragged and dropped into a <u>target</u> object such as to create a hardcopy of a report
  - An <u>application</u> 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 <u>screen layout</u> 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
- 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

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