COMPONENT-LEVEL DESIGN

- The purpose of component-level design is to define data structures, algorithms, interface characteristics, and communication mechanisms for each software component identified in the architectural design.
- Component-level design occurs after the data and architectural designs are established.
- The component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built.
- The work product produced is a design for each software component, represented using graphical, tabular, or text-based notation.

Component Definitions

- "A *component is a modular building block for computer software.*"
- *"A component is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."*
- Components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built.
- Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.
- There are three important views of what a component is and how it is used as design modeling proceeds

Three views of Component

1. **An Object-Oriented View**:
   - A component contains a set of collaborating classes.
   - Each class within a component has been fully elaborated to include all attributes and operations that are relevant to its implementation.
   - As part of the design elaboration, all interfaces that enable the classes to communicate and collaborate with other design classes must also be defined.
   - 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

2. The Traditional 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

3. **Process-Related view**

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

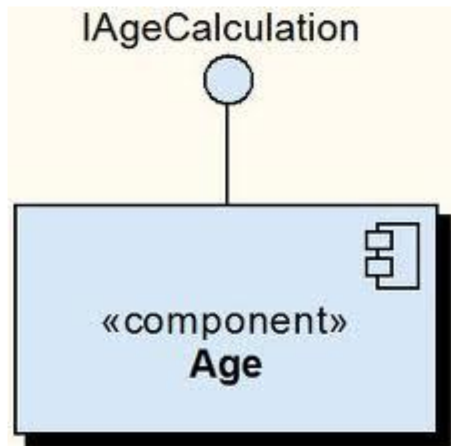
Component-level Design Principles

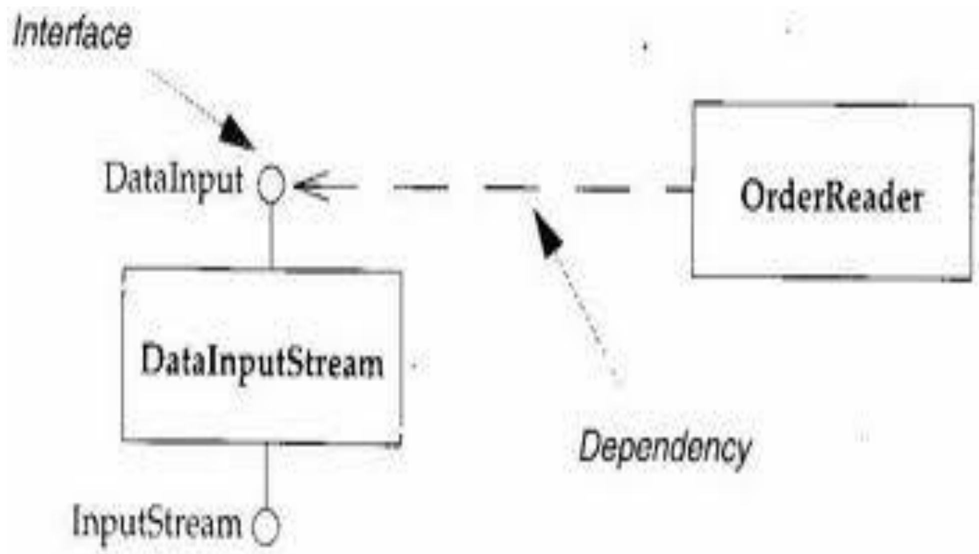
- **Open-closed principle (OCP)**
  - A module or component should be open for extension but closed for modification
  - The designer should specify the component in a way that allows it to be extended without the need to make internal code or design modifications to the existing parts of the component
- **Liskov substitution principle (LSP)**
  - 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 (DIP)**
  - 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 (ISP)**
  - 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

- **Release reuse equivalency principle (REP)**
  - 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 (CCP)**
  - 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 (CRP)**
  - 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)
- Interfaces
  - Interfaces provide important information about communication and collaboration
  - lollipop representation of an interface should be used in lieu of the more formal UML box and dashed arrow approach
  - only those interfaces that are relevant to the component under consideration should be shown, even if other interfaces are available
  - These recommendations are intended to simplify the visual nature of UML component diagrams.


IAgeCalculation
«component»
Age


Interface
DataInput
OrderReader
DataInputStream
Dependency
InputStream

- 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 to lowest
    - Functional
      - A module performs one and only one computation and then returns a result
    - Layer
      - Exhibited by packages, components, and classes, this type of cohesion occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.
    - Communicational
      - All operations that access the same data are defined within one class
  - Classes and components that exhibit functional, layer, and communicational cohesion are relatively easy to implement, test, and maintain.

- 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

- **Content coupling.** Occurs when one component "secretly modifies data that is internal to another component" . This violates information hiding—a basic design concept.
- **Common coupling.** Occurs when a number of components all make use of a global variable. Although this is sometimes necessary (e.g., for establishing default values that are applicable throughout an application), common coupling can lead to uncontrolled error propagation and unforeseen side effects when changes are made.
- **Control coupling.** Occurs when operation *A() invokes operation B() and* passes a control flag to *B. The control flag then "directs" logical flow within B. The problem with this form of coupling is that an unrelated change in B* can result in the necessity to change the meaning of the control flag that *A* passes. If this is overlooked, an error will result.
- **Stamp coupling.** Occurs when ClassB is declared as a type for an argument of an operation of ClassA. Because ClassB is now a part of the definition of ClassA, modifying the system becomes more complex**.**

- **Data coupling.** Occurs when operations pass long strings of data arguments. The "bandwidth" of communication between classes and components grows and the complexity of the interface increases. Testing and maintenance are more difficult.
- **Routine call coupling.** Occurs when one operation invokes another. This level of coupling is common and is often quite necessary.
- **Type use coupling.** Occurs when component A uses a data type defined in component B (e.g., this occurs whenever "a class declares an instance variable or a local variable as having another class for its type"). If the type definition changes, every component that uses the definition must also change.
- **Inclusion or import coupling.** Occurs when component A imports or includes a package or the content of component B.
- **External coupling.** Occurs when a component communicates or collaborates with infrastructure components (e.g., operating system functions, database capability, telecommunication functions). Although this type of coupling is necessary, it should be limited to a small number of components or classes within a system.

Conducting Component-Level Design

- You must transform information from requirements and architectural models into a design representation that provides sufficient detail to guide the construction (coding and testing) activity.
- The following steps represent a typical task set for object-oriented component-level design,

1) Identify all design classes that correspond to the problem domain
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 when classes or components collaborate
    b) Identify appropriate interfaces for each component
    c) Elaborate attributes and define data types and data structures required to implement them
    d) Describe processing flow within each operation in detail by means of pseudocode or UML activity diagrams
4) Describe data sources (databases and files) and identify the classes required to manage them
5) Develop and elaborate behavioral representations for a class or component

6) Elaborate deployment diagrams to provide additional implementation detail
7) Refactor 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

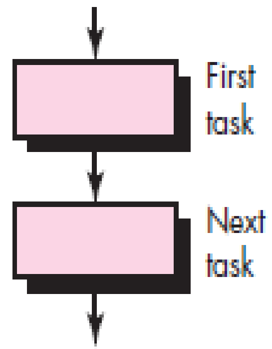
DESIGNING TRADITIONAL COMPONENTS

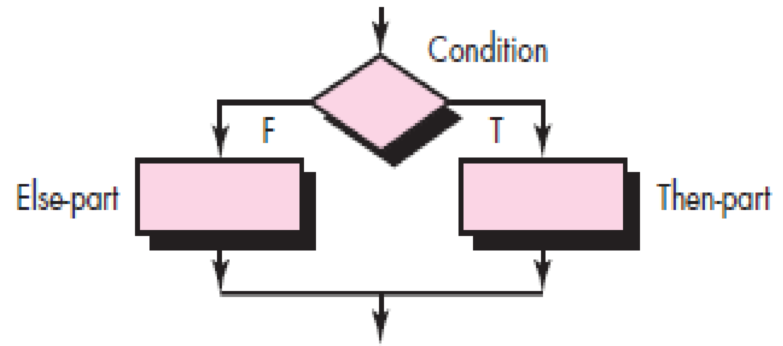
- Traditional components can be defined as a set of constrained logical constructs from which any program could be formed.
- That is, each construct had a predictable logical structure and was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.
- The constructs are sequence, condition, and repetition.
  - Sequence implements processing steps that are essential in the specification of any algorithm.
  - Condition provides the facility for selected processing based on some logical occurrence, and
  - repetition allows for looping.
- These three constructs are fundamental to structured programming—an important component-level design technique.

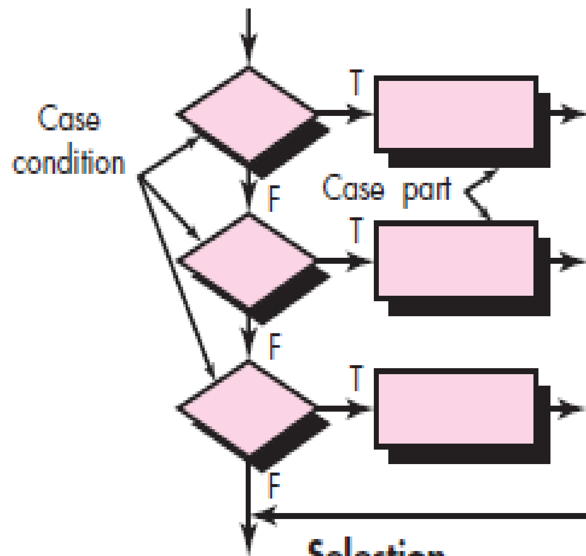
- Various notations depict the use of these constructs
  - Graphical design notation
    - Sequence, if-then-else, selection, repetition (see next slide)
  - Tabular design notation (see upcoming slide)
  - Program design language
    - Similar to a programming language; however, it uses narrative text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

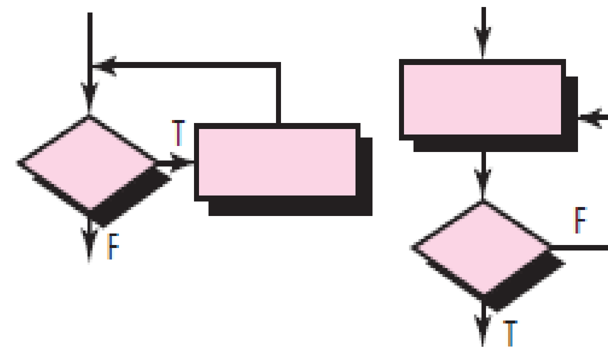
Graphical Design Notation

- ”A picture is worth a thousand words,” but it’s rather important to know which picture and which 1000 words.
- If graphical tools are misused, the wrong picture may lead to the wrong software.
- The activity diagram allows you to represent sequence, condition, and repetition— all elements of structured programming—and is a descendent of an earlier pictorial design representation (still used widely) called a *flowchart*.
- *A flowchart, like* an activity diagram, is quite simple pictorially.
- A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control.


First
task
Next
task
Sequence


Condition
F
T
Else-part
Then-part
If-then-else


Case
condition
T
F
Case part
T
F
T
F
Selection

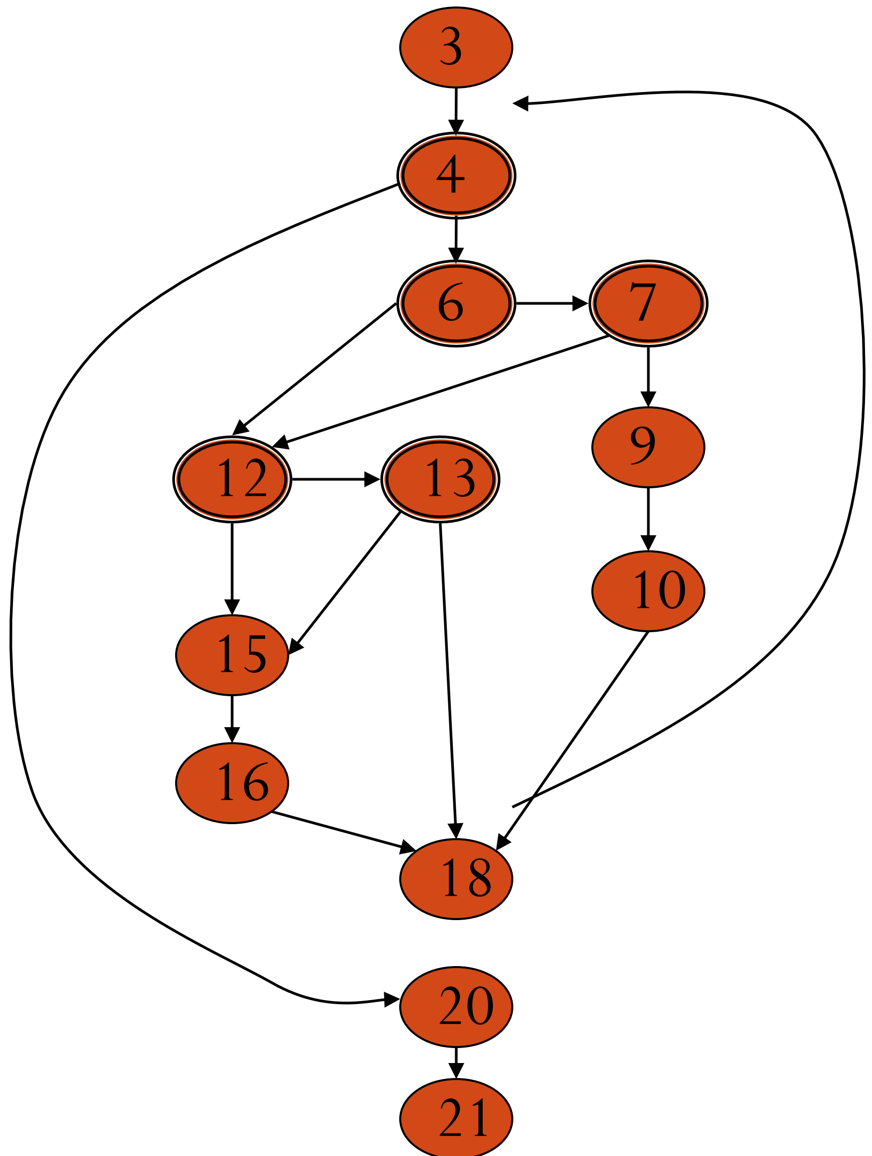

T
F
F
T
Do while
Repeat until
Repetition

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

- **Decision Table:** A **decision table** is an excellent tool to use in both testing and requirements management. Essentially it is a structured exercise to formulate requirements when dealing with complex business rules. **Decision tables** are used to model complicated logic.
- A **decision table** is a good way to deal with combinations of things (e.g. inputs). This technique is sometimes also referred to as a 'cause-effect' table.
- **The four quadrants:**

Conditions	**Condition alternatives**
Actions	Actions Entries

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

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	✓	✓		✓

	Rules					
Conditions	**1**	**2**	**3**	**4**	**5**	**6**
Regular customer	T	T				
Silver customer			T	T		
Gold customer					T	T
Special discount	F	T	F	T	F	T
Actions						
No discount	✓					
Apply 8 percent discount			✓	✓		
Apply 15 percent discount					✓	✓
Apply additional x percent discount		✓		✓		✓

- If you are a new customer and you want to open a credit card account then there are three conditions first you will get a 15% discount on all your purchases today,
- second if you are an existing customer and you hold a loyalty card, you get a 10% discount and
- third if you have a coupon, you can get 20% off today (but it can't be used with the 'new customer' discount). Discount amounts are added, if applicable.

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

Number of Conditions	Number of Columns
1	2
2	4
3	8
4	16
5	32

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

- Scenario: A marketing company wishes to construct a decision table to decide how to treat clients according to three characteristics: Gender, City Dweller, and age group: A (under 30), B (between 30 and 60), C (over 60). The company has four products (W, X, Y and Z) to test market. Product W will appeal to male city dwellers. Product X will appeal to young males. Product Y will appeal to Female middle aged shoppers who do not live in cities. Product Z will appeal to all but older males.

- The three data attributes tested by the conditions in this problem are
  - gender, with values M and F;
  - city dweller, with value Y and N; and
  - age group, with values A, B, and C
- as stated in the problem.

Rules

Proc	1	2	3	4	5	6	7	8	9	10	11	12
Sex	m	f	m	f	m	f	m	f	m	f	m	f
City	y	y	n	n	y	y	n	n	y	Y	n	n
Age	a	a	a	a	b	b	b	b	c	c	c	c

Actions

Market	1	2	3	4	5	6	7	8	9	10	11	12
W	x				x				X			
X	x		X									
Y								X				
Z	x	x	x	x	x	x	x	x		X		X

Chapter 11

User Interface Design

Overview

- Wherever there is flow there is interface.
- The blue print for a house (its architectural design) is not complete without a representation of doors, windows, and utility connections for water, electricity, telephone, cable TV etc. They are the interfaces through which water, electricity, air, telephone signals, TV signals and people flow.
- The "doors, windows and utility connections" for software make up the interface design of a system.
- The interface design focuses on 3 areas:
  - Design of interfaces between software components
  - Design of interface between the software and other non-human producers and consumers of information i.e. other external entities. Sensors, alarms, telephone lines
  - Design of interface between a human and the computer.

- This chapter focuses on user interface design.
- It creates an effective communication medium between a human and a computer.
- User interface design has as much to do with the study of people as it does with technology issue.
- We have already seen what an interface is. Now we will see what are the goals , what you are supposed to do, by doing what and how.

Golden rules

- There are 3 golden rules based on which certain principles have been defined to guide the software engineer in interface design. These 3 rules are
  - Place the user in control. Rather than user feeling that software is controlling the user let the user feel that he/she is in control. Then the user will accept and use the software.
  - Reduce the user's memory load.
  - Make the interface consistent.

- Place the user in control How do you do this? Follow certain principles.
  1. Provide for flexible interaction. Different users have different interaction preferences therefore provide choices. Save icon, ctrl S , File save. software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.
  2. Allow user interaction to be interruptible and undoable. (UPS). Allow user to put something on hold (interrupt) for sometime and come back to it later without losing what he has already done. Allow undo action.
  3. Allow interaction to be customized. Design a macro mechanism for advanced users.
  4. Hide technical internals from the casual user. The user does not need to know about the operating system or file structure. (e.g., a user should never be required to type operating system commands from within application software)
  5. Design for direct interaction with objects that appear on the screen. Allow dragging of corners to increase / decrease size.
  6. Avoid unnecessary actions. Do not force the user to do unnecessary actions i.e. allow him to make corrections through spell check.

- Reduce user's memory load
  1. Reduce demand on short term memory. Give visual cues_to recognize past actions. New windows as you select subfolders in Win NT.
  2. Establish meaningful defaults and allow customization and reset.
  3. Define shortcuts that are intuitive. Ctrl S for saving and Ctrl P for printing. Mnemonics.
  4. The visual layout of the interface should be based on a real world metaphor. For example use exact layout of cheque for making payment. Attendance sheet.
  5. Disclose information in a progressive manner. The interface should be organized hierarchically. Information about a task, object etc. is presented first at a high level of abstraction. More detail should be presented after the user indicates interest in details.

- Make the interface consistent.
    1. Maintain consistency across a family of applications. Appearance, options, mnemonics (shortcuts).
    2. Do not make changes unless it is absolutely necessary. If ctrl s is used for Saving do not change it to Scaling.

User Interface analysis and design

- We need to understand about
  - who is going to interact,
  - what the user will do,
  - how he will do it and
  - on which items (which objects) he will be doing it.
- Basically we need to identify real world objects and actions and create models.
- We have to create 4 types of models –
  - design model,
  - user model,
  - system perception and
  - system image.

- The design model incorporates data, architectural, interface and procedural representation of the software. Made by Software engineer.
- The user model is prepared by a human engineer. Basically we need to establish profile of the end user of the system. This is called user model. We need to get the information about age, gender, physical abilities, education, cultural background, motivation, goals, personality of the users. In addition users can also be categorized as
  - Novices : No syntactic knowledge (syntax) and little semantic knowledge of the application. Syntactic = mechanics of interaction. Semantic means understanding of the underlying sense of the application.
  - Knowledgeable, intermittent users Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.
  - Knowledgeable frequent user. Good semantic and syntactic knowledge. Power user.

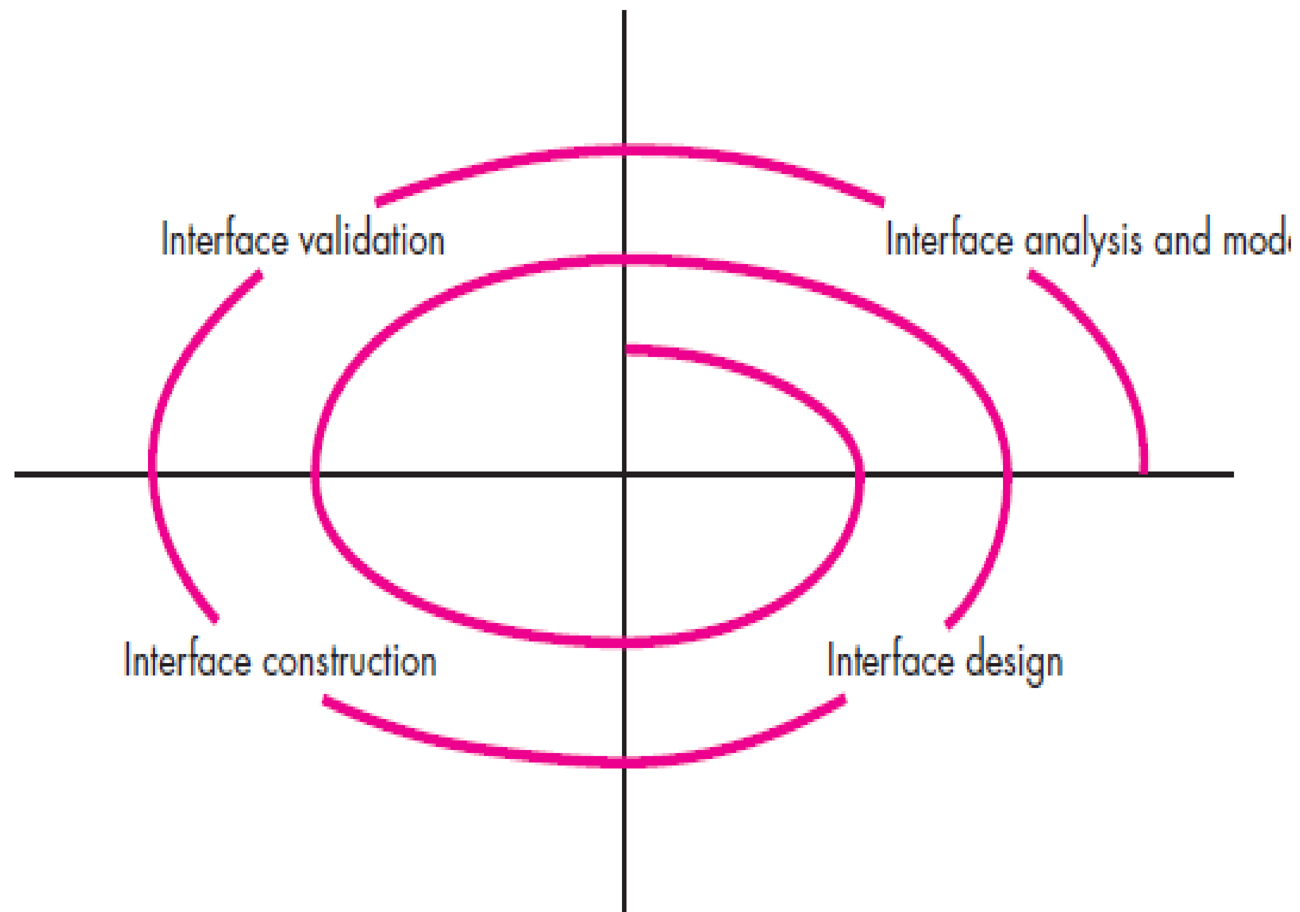
- The system perception is the image of the system that end-users carry in their heads. It will depend on the level of the user. It is also called user's mental model.
- The system image combines the outward manifestation of the computer-based system (look and feel) coupled with all supporting information like manuals, books, help files, video clips.
- Manuals, help files, video clips are all means through which user interface with the system.
- The most important principle that you should remember while designing the user interface is " know the user, know the tasks".

User Interface design process

- Like the overall framework activities of software engineering, there are framework activities for interface design also. They are
  1. Interface analysis and modeling.
  2. Interface design –define set of objects and actions.
  3. Interface construction – create prototype.
  4. Interface validation. To check whether the interface meets the requirements of the user and he/she accepts it.
- You have to carry out these activities in an iterative fashion. They can be shown as a spiral model.

FIGURE 11.1

The user interface design process



- ***Interface analysis*** *focuses on the profile of the users who will interact with the* system.
  - Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, you work 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 Finally, analysis of the user environment focuses on the physical work environment. Among the questions to be asked are
    - Where will the interface be located physically?
    - Will the user be sitting, standing, or performing other tasks unrelated to the interface?
    - Does the interface hardware accommodate space, light, or noise constraints?
    - Are there special human factors considerations driven by environmental factors?
  - The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design action 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.
- ***Interface construction*** normally begins with the creation of a prototype that enables usage scenarios to be evaluated.
- ***Interface 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.

Design Issues

- There are four major issues which surface during interface design. They are
  - response time,
  - help facility,
  - error information handling and
  - menu and command labeling.

- Response time (from hitting of return key or clicking of a mouse to desired output or action)
  - It is the primary complaint for many interactive applications.
  - Response time has 2 characteristics – length and variability.
  - Too slow response time and too fast response time both are bad.
  - Too fast a response time may force the user to take quick actions and make mistakes.
  - Variability (deviation from average response time) is also bad.
  - It is more important than length.
  - If the user knows the length of response time is long he knows when to expect the result. If variability is more the user does not know what is happening. Hour glass / progress bar.

- Help Facility
  - It can be in form of printed manuals or online help.
  - The online help can be integrated or add-on.
  - An integrated help facility is designed into the software from the beginning and context sensitive.
  - This is more user-friendly and preferred over add-on help. Add-on help is added after the system is developed. It is on-line user's manual with limited query facility.
  - Lot of issues that come up regarding help facility are whether it is to be given for all functions, how the user will get help (help menu, key etc), is help flat or hierarchical, how will the user return to the normal interaction, new window or a small part on the same window etc.

- Error information handling
  - Error messages are the bad news delivered to the user when something goes wrong.
  - Error messages which impart useless or misleading information cause frustration e.g. severe system failure – 14A.
  - Error message should be explained in simple language that user can understand, should contain constructive advice for recovering from the error, indicate any negative consequences so that the user can take corrective action, should be accompanied by visual or audible cue, should not place blame on the user.

- **Menu and command labeling.**
- The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type.
- Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction.
- A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:
  - Will every menu option have a corresponding command?
  - What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or 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?

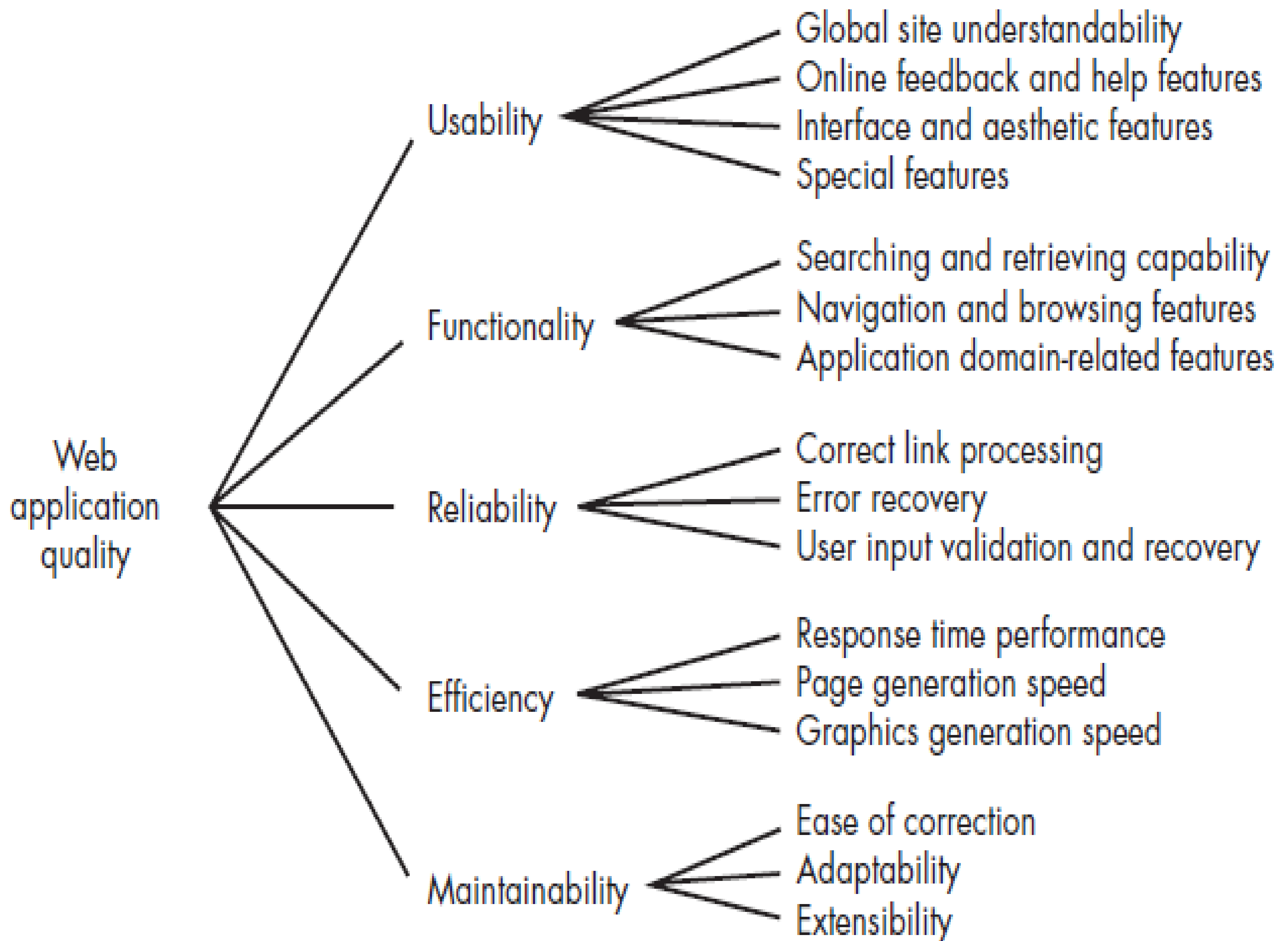
- **Application accessibility**
  - One also needs to remember that there are users with different abilities and hence the accessibility has to be provided for people with different abilities (blind, hard of hearing, physically handicapped etc).
- **Internationalization.**
  - One more aspect that the software engineer needs to remember is that lot of applications these days are used by users all over the world and in different languages. Hence the user interface has to be designed keeping this point in mind.

Chapter 13

WebApp Design

WEBAPP DESIGN QUALITY

- Every person who has surfed the Web or used a corporate Intranet has an opinion about what makes a "good" WebApp. Individual viewpoints vary widely.
  - Some users enjoy flashy graphics; others want simple text. Some demand copious information; others desire an abbreviated presentation.
  - Some like sophisticated analytical tools or database access; others like to keep it simple.
- In fact, the user's perception of "goodness" (and the resultant acceptance or rejection of the WebApp as a consequence) might be more important than any technical discussion of WebApp quality.


Web application quality
Usability
Global site understandability
Online feedback and help features
Interface and aesthetic features
Special features
Functionality
Searching and retrieving capability
Navigation and browsing features
Application domain-related features
Reliability
Correct link processing
Error recovery
User input validation and recovery
Efficiency
Response time performance
Page generation speed
Graphics generation speed
Maintainability
Ease of correction
Adaptability
Extensibility

What are the major attributes of quality for WebApps?

- **Security :** WebApps have become heavily integrated with critical corporate and government databases. E-commerce applications extract and then store sensitive customer information. For these and many other reasons, WebApp security is dominant in many situations. The key measure of security is the ability of the WebApp and its server environment to refuse unauthorized access and/or prevent an outright malicious attack.
- **Availability:** Even the best WebApp will not meet users' needs if it is unavailable. In a technical sense, availability is the measure of the percentage of time that a WebApp is available for use. The typical end user expects WebApps to be available 24/7/365. Anything less is deemed unacceptable. But "up-time" is not the only indicator of availability. Using features available on only one browser or one platform" makes the WebApp unavailable to those with a different browser/platform configuration. The user will invariably go elsewhere.

- **Scalability:** Can the WebApp and its server environment be scaled to handle 100, 1000, 10,000, or 100,000 users? Will the WebApp and the systems with which it is interfaced handle significant variation in volume or will responsiveness drop dramatically (or cease altogether)? It is not enough to build a WebApp that is successful. It is equally important to build a WebApp that can accommodate the burden of success (significantly more end users) and become even more successful.
- **Time-to-market:** Although time-to-market is not a true quality attribute in the technical sense, it is a measure of quality from a business point of view. The first WebApp to address a specific market segment often captures a disproportionate number of end users.

DESIGN GOALS

- **Simplicity:**
  - There is a tendency among some designers to provide the end user with "too much"—exhaustive content, extreme visuals, intrusive animation, enormous Web pages, the list is long. Better to strive for moderation and simplicity.
  - Content should be informative but succinct and should use a delivery mode (e.g., text, graphics, video, audio) that is appropriate to the information that is being delivered.
  - Aesthetics should be pleasing, but not overwhelming (e.g., too many colors tend to distract the user rather than enhancing the interaction).
  - Architecture should achieve WebApp objectives in the simplest possible manner.
  - Navigation should be straightforward and navigation mechanisms should be intuitively obvious to the end user.
  - Functions should be easy to use and easier to understand.

- **Consistency:** This design goal applies to virtually every element of the design model. Content should be constructed consistently (e.g., text formatting and font styles should be the same across all text documents; graphic art should have a consistent look, color scheme, and style). Graphic design (aesthetics) should present a consistent look across all parts of the WebApp. Architectural design should establish templates that lead to a consistent hypermedia structure. Interface design should define consistent modes of interaction, navigation, and content display. Navigation mechanisms should be used consistently across all WebApp elements. As Kaiser [Kai02] notes: "Remember that to a visitor, a Web site is a physical place. It is confusing if pages within a site are not consistent in design."

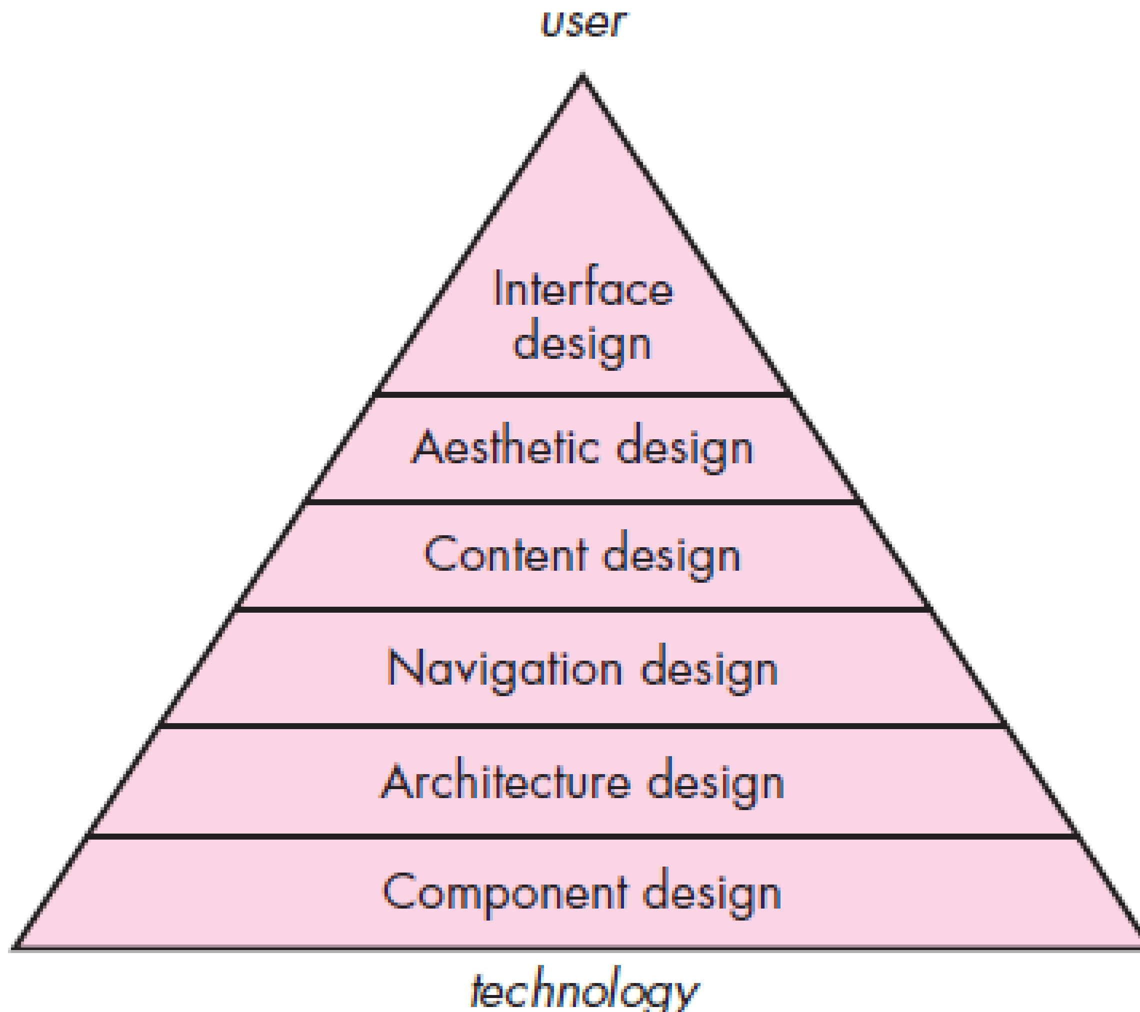
- **Identity:** The aesthetic, interface, and navigational design of a WebApp must be consistent with the application domain for which it is to be built. A website for a hiphop group will undoubtedly have a different look and feel than a WebApp designed for a financial services company. The WebApp architecture will be entirely different, interfaces will be constructed to accommodate different categories of users; navigation will be organized to accomplish different objectives. You (and other design contributors) should work to establish an identity for the WebApp through the design.

- **Robustness:** Based on the identity that has been established, a WebApp often makes an implicit "promise" to a user. The user expects robust content and functions that are relevant to the user's needs. If these elements are missing or insufficient, it is likely that the WebApp will fail.

- **Navigability:** I have already noted that navigation should be simple and consistent. It should also be designed in a manner that is intuitive and predictable. That is, CHAPTER 13 WEBAPP DESIGN 377 **uote:** “Just because you can, doesn’t mean you should.” **Jean Kaiser uote:** “To some, Web design focuses on visual look and feel . . . To others, Web design is about structuring information and navigation through the document space. Others might even consider Web design to be about the technology . . . In reality, design includes all of these things and maybe more.”
- **Thomas Powell** the user should understand how to move about the WebApp without having to search for navigation links or instructions. For example, if a field of graphic icons or images contains selected icons or images that will be used as navigation mechanisms, these must be identified visually. Nothing is more frustrating than trying to find the appropriate live link among many graphical images.
- It is also important to position links to major WebApp content and functions in a predictable location on every Web page. If page scrolling is required (and this is often the case), links at the top and bottom of the page make the user’s navigation tasks easier.

- **Visual Appeal:** Of all software categories, Web applications are unquestionably the most visual, the most dynamic, and the most unapologetically aesthetic. Beauty (visual appeal) is undoubtedly in the eye of the beholder, but many design characteristics (e.g., the look and feel of content; interface layout; color coordination; the balance of text, graphics, and other media; navigation mechanisms) do contribute to visual appeal.
- **Compatibility:** A WebApp will be used in a variety of environments (e.g., different hardware, Internet connection types, operating systems, browsers) and must be designed to be compatible with each.

A DESIGN PYRAMID FOR WEBAPPS



- **WEBAPP INTERFACE DESIGN:** Although WebApps present a few special user interface design challenges.
- The user may enter the WebApp at a "home" location (e.g., the home page) or may be linked into some lower level of the WebApp architecture.
- In some cases, the WebApp can be designed in a way that reroutes the user to a home location, but if this is undesirable, the WebApp design must provide interface navigation features that accompany all content objects and are available regardless of how the user enters the system.

- *Navigation menus*—keyword menus (organized vertically or horizontally) that list key content and/or functionality.
- *Graphic icons*—button, switches, and similar graphical images that enable the user to select some property or specify a decision.
- *Graphic images*—some graphical representation that is selectable by the user and implements a link to a content object or WebApp functionality.

- **AESTHETIC DESIGN:** Aesthetic design, also called *graphic design,* is an artistic endeavor that complements the technical aspects of WebApp design. Without it, a WebApp may be functional, but unappealing.
- But what is aesthetic? There is an old saying, "beauty exists in the eye of the beholder." This is particularly appropriate when aesthetic design for WebApps is considered. To perform effective aesthetic design, return to the user hierarchy developed as part of the requirements model and ask, *Who are the WebApp's users and what "look" do they desire?*
- Layout Issues
  - **Don't be afraid of white space**
  - **Emphasize content**
  - **Organize layout elements from top-left to bottom-right**

- **Group navigation, content, and function geographically within the page.**
- **Don’t extend your real estate with the scrolling bar**
- **Consider resolution and browser window size when designing layout**

- Graphic Design Issues:
  - Graphic design considers every aspect of the look and feel of a WebApp. The graphic design process begins with and proceeds into a consideration of global color schemes; text types, sizes, and styles; the use of supplementary media (e.g., audio, video, animation); and all other aesthetic elements of an application.

- NAVIGATION DESIGN: Once the WebApp architecture has been established and the components (pages, scripts, applets, and other processing functions) of the architecture have been identified, you must define navigation pathways that enable users to access WebApp content and functions. To accomplish this, you should (1) identify the semantics of navigation for different users of the site, and (2) define the mechanics (syntax) of achieving the navigation.
- Navigation Semantics: Each actor may use the WebApp somewhat differently and therefore have different navigation requirements
- Navigation Syntax:
  - *Individual navigation link*
  - *Horizontal navigation bar*
  - *Vertical navigation column*
  - *Tabs*
  - *Site maps*