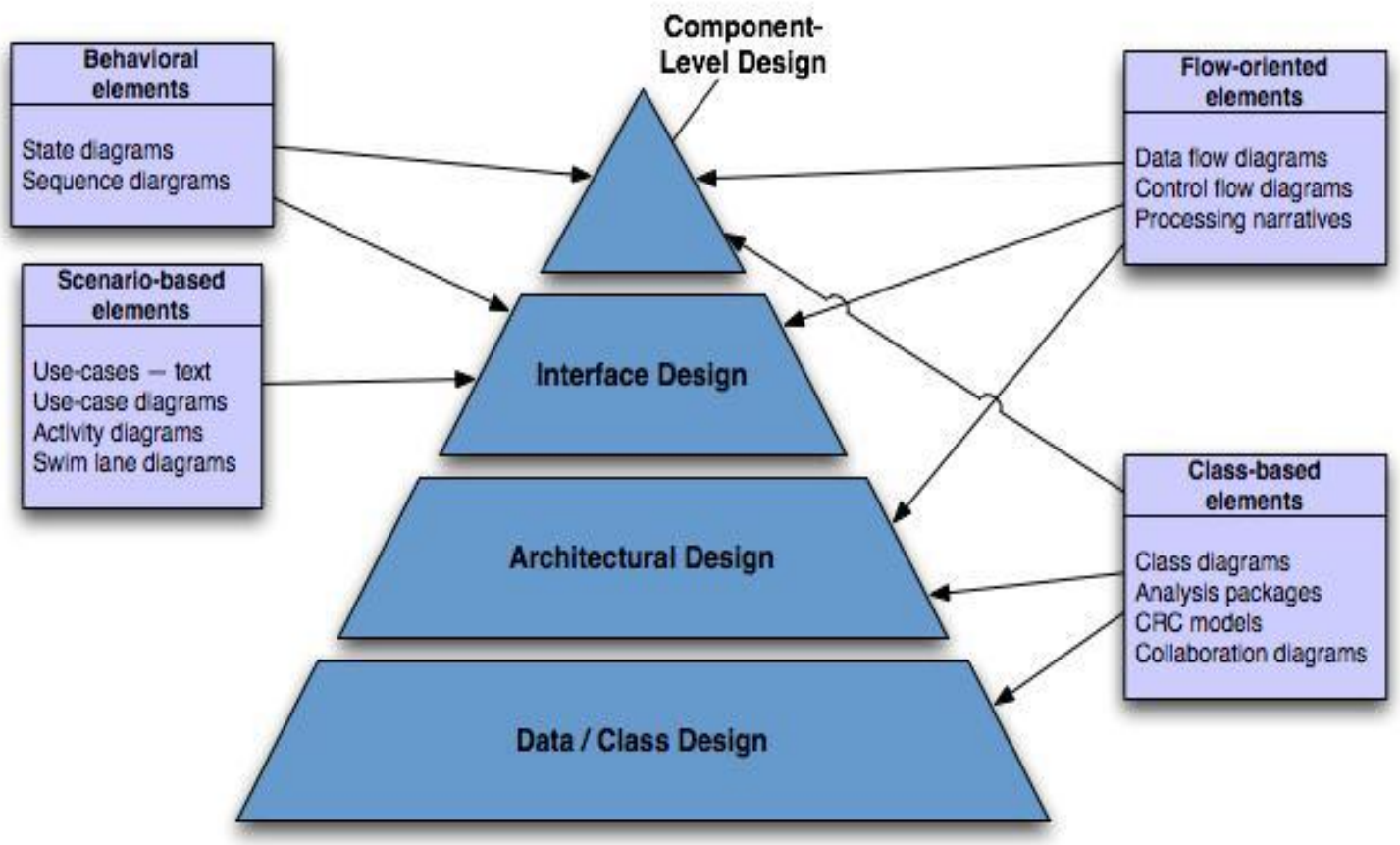
Component-Level Design

Analysis Model to Design Model
(Review)

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

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

Analysis → Design
(Review)



What is a Component?

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

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

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

Object-oriented View

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

Conventional View

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

(More on next slide)

Conventional View (continued)

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

Process-related View

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

Component-level Design Principles

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

Component Packaging Principles

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

Component-Level Design

Step 1. Identify all design classes that correspond to the problem domain.

Step 2. Identify all design classes that correspond to the infrastructure domain.

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

Step 3a. Specify message details when classes or component collaborate.

Step 3b. Identify appropriate interfaces for each component.

Step 3c. Elaborate attributes and define data types and data structures required to implement them.

Step 3d. Describe processing flow within each operation in detail.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.

Step 5. Develop and elaborate behavioral representations for a class or component.

Step 6. Elaborate deployment diagrams to provide additional implementation detail.

Step 7. Refactor every component-level design representation and always consider alternatives.

Component Design for WebApps

WebApp component is

(1) a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or

(2) a cohesive package of content and functionality that provides end-user with some required capability.

Component-level design for WebApps often has content design besides functional design.

Content Design focuses on content objects and the packaging for presentation to a WebApp enduser e.g. Web-based video surveillance capability within **SafeHomeAssured.com** potential content components can be defined for the video surveillance capability:

(1) the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras

(2) the collection of thumbnail video captures (each an separate data object)

(3) the streaming video window for a specific camera. Each of these components can be separately named and manipulated as a package.

Designing Conventional Components

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

- Represent the algorithm at a level of detail that can be reviewed for quality
- options:
  - graphical (e.g. flowchart, box diagram)
  - pseudocode (PDL or program design language or structured english)
  - decision table
- use structured programming to implement procedural logic
- use 'formal methods' to prove logic

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

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

Component-Based Development

When faced with the possibility of reuse, the software team must check:

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

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

Impediments to Reuse

• Few companies and organizations have anything that even slightly resembles a comprehensive software reusability plan.

• Although an increasing number of software vendors currently sell tools or components that provide direct assistance for software reuse, the majority of software developers do not use them.

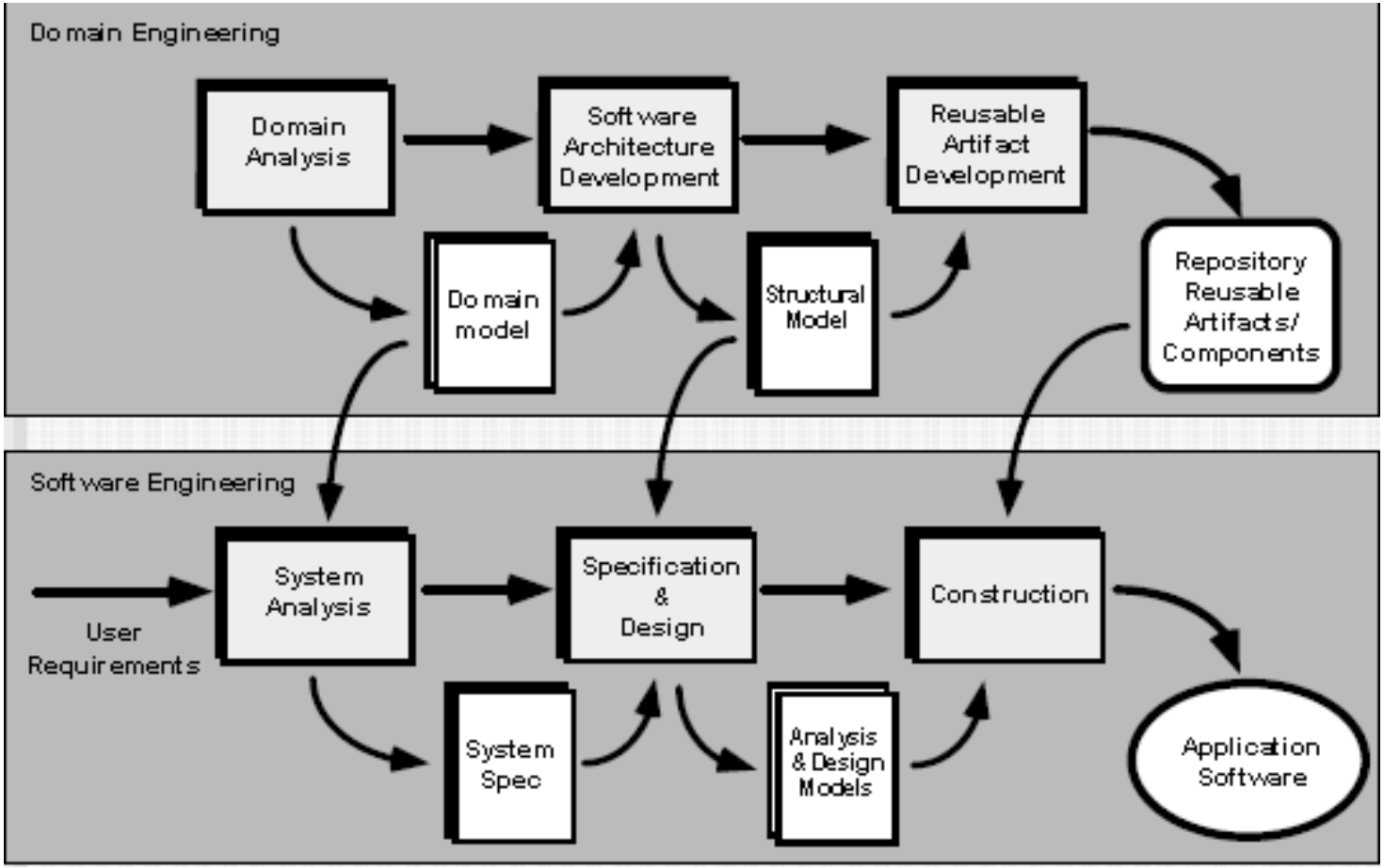
• Relatively little training is available to help software engineers and managers understand and apply reuse.

• Many software practitioners continue to believe that reuse is “more trouble than it’s worth.”

• Many companies continue to encourage of software development methodologies which do not facilitate reuse

• Few companies provide an incentives to produce reusable program components.

Component-Based Software Engineering



Component-Based Software Engineering
(contd…)

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

CBSE Activities

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

Component Qualification

Before a component can be used, you must consider:

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

Component Adaptation

The implication of “easy integration” is:

(1) that consistent methods of resource management have been implemented for all components in the library;

(2) that common activities such as data management exist for all components, and

(3) that interfaces within the architecture and with the external environment have been implemented in a consistent manner.

Adaptation techniques

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

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

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

Component Composition

An infrastructure must be established to bind components together

Architectural ingredients for composition include:

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

Component development for reuse

- Components for reuse may be specially constructed by generalising existing components.

- Component reusability
  - Should reflect stable domain abstractions;
  - Should hide state representation;
  - Should be as independent as possible;
  - Should publish exceptions through the component interface.

- There is a trade-off between reusability and usability
  - The more general the interface, the greater the reusability but it is then more complex and hence less usable.

Changes for reusability

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

Component validation

- Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests.
  - The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests.

- As well as testing that a component for reuse does what you require, you may also have to check that the component does not include any malicious code or functionality that you don't need.

Ariane launcher failure – validation failure?

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

Web services

- A web service is an instance of a more general notion of a service:

  "an act or performance offered by one party to another. Although the process may be tied to a physical product, the performance is essentially intangible and does not normally result in ownership of any of the factors of production".

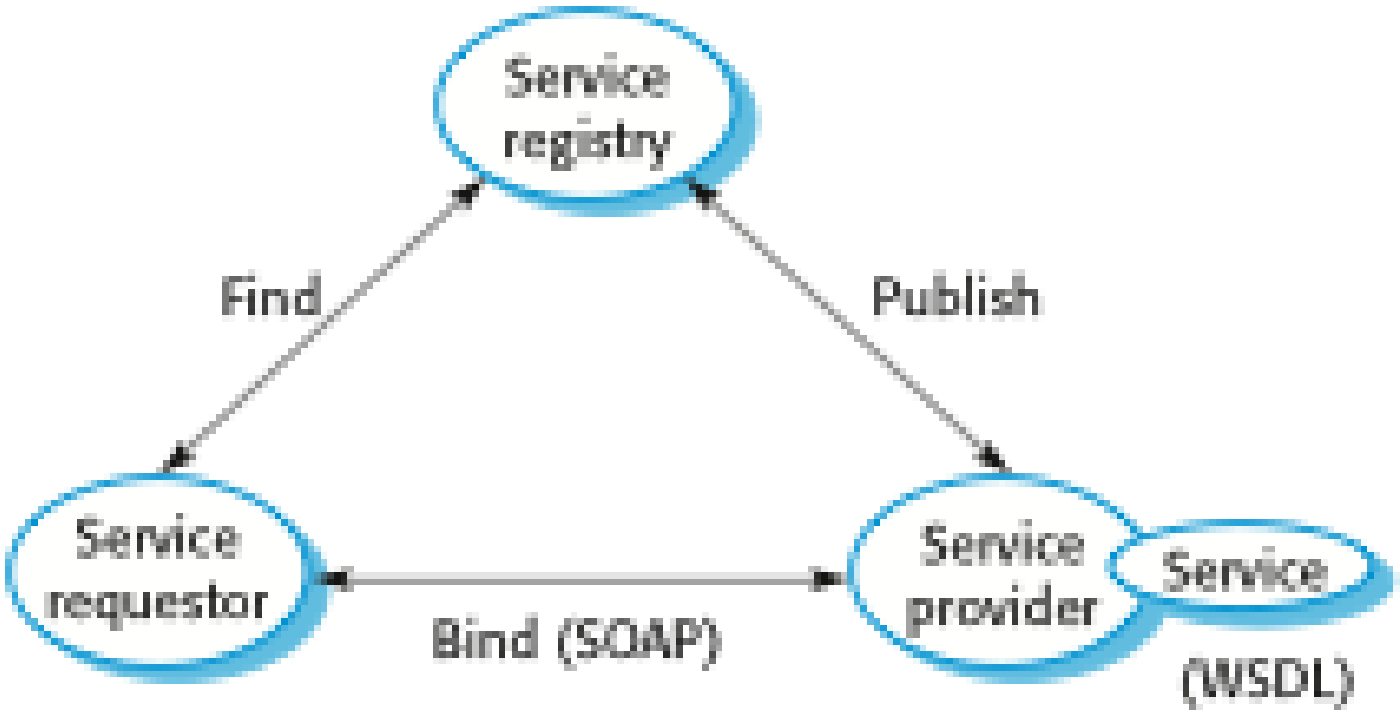
- The essence of a service, therefore, is that the provision of the service is independent of the application using the service.

- Service providers can develop specialized services and offer these to a range of service users from different organizations.

Service-oriented architectures

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

Service-oriented architecture



Benefits of SOA

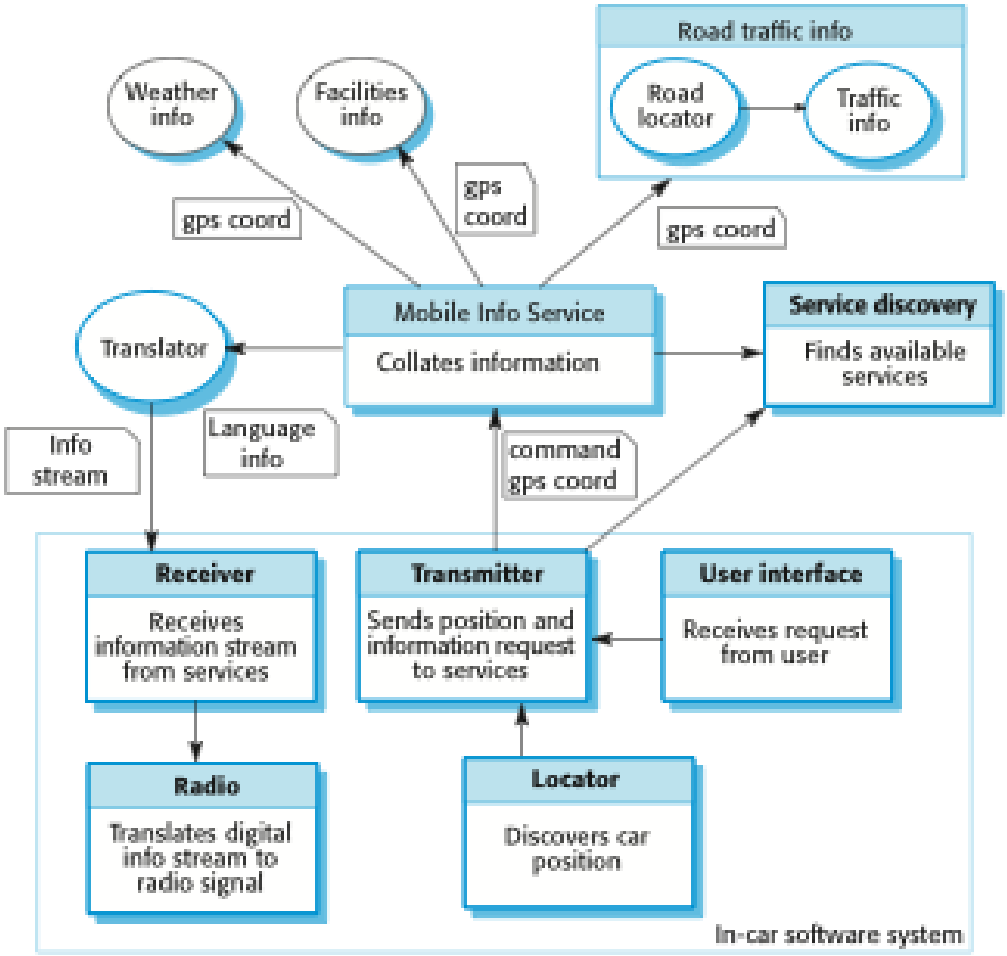
- Services can be provided locally or outsourced to external providers
- Services are language-independent
- Investment in legacy systems can be preserved
- Inter-organisational computing is facilitated through simplified information exchange

Services Example

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

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

A service-based, in-car information system



User Interface Design

Where Do Design Guidelines Come From?

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

Interface Design

What do users expect?

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

What do they get?

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

Golden Rules of UI Design

- Place the user in control
- Reduce the user's memory load
- Make the interface consistent

– Theo Mandel[97] , PhD in Cognitive Psychology
(Consultant, Author)

Place the User in Control

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

Provide for flexible interaction.

Allow user interaction to be interruptible and undoable.

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

Hide technical internals from the casual user.

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

Reduce the User's Memory Load

Reduce demand on short-term memory.

Establish meaningful defaults.

Define shortcuts that are intuitive.

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

Disclose information in a progressive fashion.

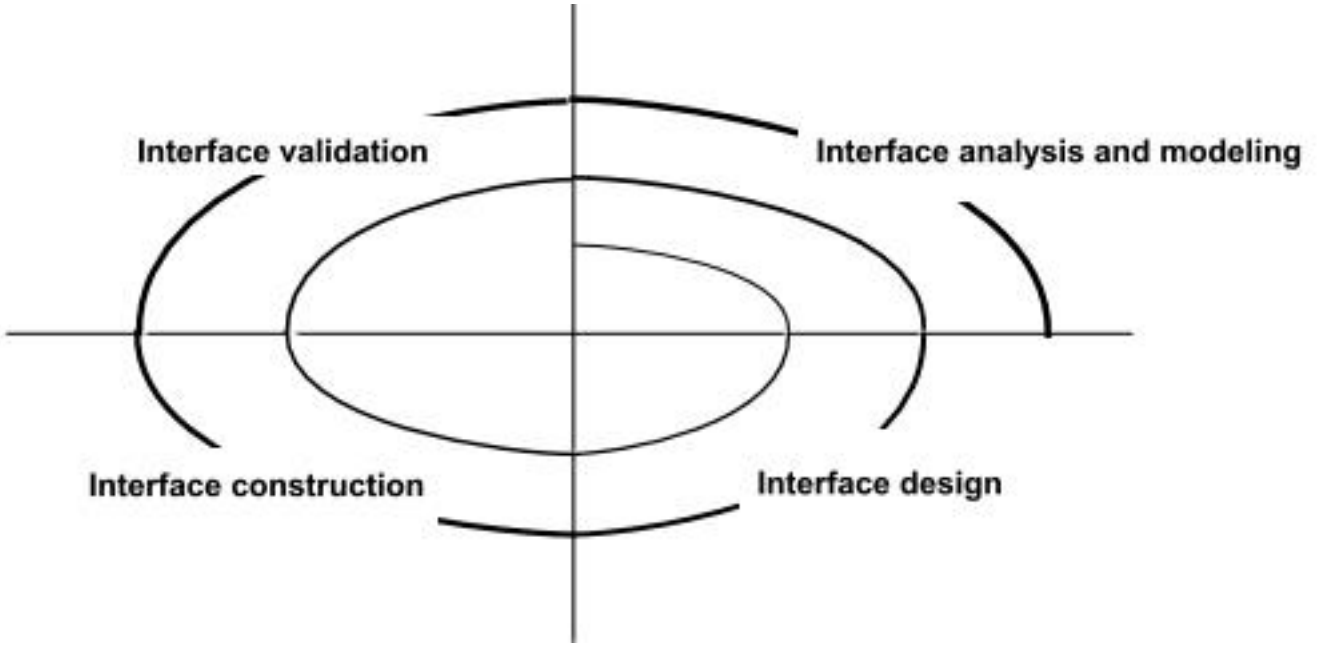
Make the Interface Consistent

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

Maintain consistency across a family of applications.

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

User Interface Design Process



Interface Analysis

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

User Analysis

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

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

Task Analysis and Modeling

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

Analysis of Display Content

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

Interface Design Steps

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

Design Issues

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

WebApp Interface Design

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

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

WebApp Interface Design Principles-I (as per Tognozzi)

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

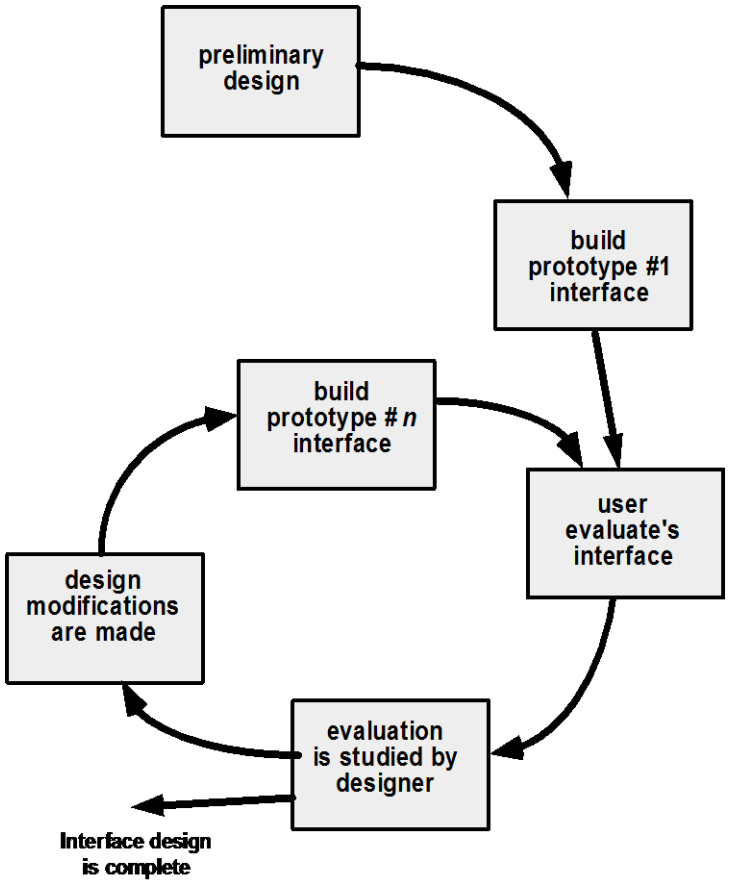
WebApp Interface Design Principles-II (as per Tognozzi)

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

Aesthetic Design

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

Design Evaluation Cycle



Heuristic Evaluation of UI

- Heuristic evaluation is a type of user interface (UI) or usability inspection where an individual, or a team of individuals, evaluates a specification, prototype, or product against a brief list of succinct usability or user experience (UX) principles or areas of concern
- A heuristic is a commonsense rule or a simplified principle. A list of heuristics is meant as an aid for the evaluators.
- Below list of heuristics from Nielsen (1994) might be used by evaluators to identify potential problem areas
    - Visibility of system status
    - Match between system and the real world
    - User control and freedom
    - Consistency and standards
    - Error prevention
    - Recognition rather than recall
    - Flexibility and efficiency of use
    - Aesthetic and minimalist design

User Interface (UI) Patterns

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

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

Component Design Summary

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

Component-level design occurs after the data and architectural designs are established.

Component-level design represents the software in a way that allows the designer to review it for correctness and consistency, before it is built.

The work product is a design for each software component, represented using graphical, tabular, or text-based notation.

Design walkthroughs are conducted to determine correctness of the data transformation or control transformation allocated to each component during earlier design steps

Interface Design Summary

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

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

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

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

Prototyping is a common approach to user interface design.

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

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