Web Design: What Is It About?

- **Web Design**: concepts, principles, and methods that are required (needed) to transform an understanding of WHAT the WebApp should do into a representation of HOW the WebApp should do.

- From WebApp Analysis Model to WebApp Design Model
WebApp Design: Art vs Engineering

- Jakob Nielsen states: "There are essentially two basic approaches to design:
  1. the artistic ideal of expressing yourself, and
  2. the engineering ideal of solving a problem for a customer."

- Even today, some proponents of agile software development use WebApps as poster children for the development of applications based on "limited design."

- However --
  - when WebApp content and functions are complex and very complex,
  - when the size of the WebApp encompasses hundreds of content objects, functions, use scenarios, etc.,
  - when multiple (hundreds and thousands) people become involved in the design, and
  - when expected number of users is high and very high (thousands and millions),
  - when the success of the WebApp will have a direct impact on the success of the multi-million dollar business and/or users,

  WebApp design cannot and should not be taken lightly.

Conclusion: Design of complex WebApp is a part of well-thought and well-structured process – software engineering.

WebApp Design: Approaches

WebApp Design

Web Graphic User Interface (GUI) Design = Interaction Design

Web Development (functions, Web services, browsers, PLs, protocols, etc.)

With growing specialization in the information technology field, there is a strong tendency to draw a clear line between
a) Web graphic design, and
b) Web development (Web programming).

Web graphic interface design is a kind of graphic design intended for development and styling of objects of the Internet's information environment to provide them with high-end consumer features and aesthetic qualities.

Web development emphasizes the functional features of WebApp.

Web Design = Web Graphic Interface Design + Web Development
WebApp Design =
WebApp Graphic Design + WebApp Development

Analogy: Computer Game Design Approaches

- Game Design
  - Game Graphic Design (Visual Effects)
  - Game Development (Game Engine)

Web Applications Design: Components
Main Components of WebApp Design
(Website Design Pyramid)

Interaction Design (interface design and aesthetic design = layout) = Interaction Model (GUI)

Information Design (content design and navigation design) = Information Model

Functional Design (overall behavior and functionality) = Functional Model

Technical Design (architectural design and component design) = Architectural or Structural Model

Additional useful components:
*) Design Patterns or Templates
**) Design and development Technologies and Tools

WebEng Framework Activities
and
WebApp Design Process
(increment-based)
Web Applications - Graphic User Interface Design:

Goals

WebApp Design Goals

- Visual appeal. Design characteristics (e.g., the look and feel of content, interface layout, color coordination, the balance of text, graphics and other media, and navigation mechanisms) contribute to visual appeal.
WebApp Design Goals

- **Navigability.** Users should be able to understand how to move about the WebApp without having to search for navigation links or instructions.

- **Identity.** The aesthetic, interface, and navigational design of a WebApp must be consistent with the application domain for which it is to be built.
WebApp Design Goals

- **Consistency.**
  - Content should be constructed consistently
  - Graphic design (aesthetics) should present a consistent look
  - Architectural design should establish templates that lead to a consistent hypermedia navigation
  - Navigation mechanisms should be used consistently

WebApp Design Goals: Do you see a consistency?
WebApp Design Goals: Simplicity

- Simplicity. Rather than feature-bloat, it is better to strive for moderation and simplicity (simple interface, simple navigation, simple structure, simple hierarchical model, relatively simple functions, etc.).

Web Applications – Functionality Development:

Goals
WebApp Development Goals

- **Main functions to be implemented** (search, retrieve, calculate, translate, convert, links to other WebApps, voice, Web services, Web security, etc.)

- **Web technologies/platforms to be used** (AJAX, Mash-Up, Web2.0, semantic Web, etc.)

- **Programming Languages to be used** (Java, PHP, XML, DHTML, etc.)

- **Web services to be used and implemented. Web services or Application**

WebApp Design Goals

- **Robustness.** The user expects robust content and functions that are relevant to the user’s needs.
WebApp Development Goals

- **Security.** Web application frameworks may come with authentication and authorization frameworks, that enable the web server to identify the users of the application, and restrict access to functions based on some defined criteria.

- **Database access and mapping.** Many web application frameworks create a unified API to a database backend, enabling web applications to work with a variety of databases with no code changes, and allowing programmers to work with higher-level concepts.

- **Cashing.** Web caching is the caching of web documents in order to reduce bandwidth usage, server load, and perceived "lag".

WebApp Design Goals

- **Compatibility.** Most WebApps will be used in a variety of environments (e.g., different hardware, Internet connection types, operating systems, and browsers) and must be designed to be compatible with each of them.
Web Applications Development and Quality Issues

From end user’s point of view:

- Performance
- Response time and latency (delay)
- Content
- WebApp versions (upgrades? How quickly?)
- Structural (how well parts of WebApp hold together?)
Software Design: Fundamental Concepts (in CS590)

1. Abstraction still a model of reality (data, procedure, control, etc.); errors or loss of details may occur

2. Architecture the overall structure of the software

3. Patterns "conveys the essence" of a proven design solution (best cases, re-usable designs)

4. Modularity compartmentalization of data and function

5. Information Hiding controlled interfaces

6. Functional Independence single-minded function and low coupling

7. Refinement elaboration of detail for all abstractions

8. Re-factoring a reorganization technique that simplifies the design
In the next several lectures:

Main Components of WebApp Design = Design Models

- Interaction Design (interface design and aesthetic design = layout) = Interaction Model (GUI)
- Information Design (content design and navigation design) = Information Model
- Functional Design (overall behavior and functionality) = Functional Model
- Technical Design (architectural design and component design) = Architectural or Structural Model

Additional useful components:
- *) Design Patterns or Templates
- **) Design and development Technologies and Tools

Web Applications Design: Components, Goals, Concepts

Additional information.
Software Design: Fundamental Concepts

1. Abstraction  
data, procedure, control
2. Architecture  
the overall structure of the software
3. Patterns  
"conveys the essence" of a proven design solution (best cases, re-usable designs))
4. Modularity  
compartmentalization of data and function
5. Information Hiding  
controlled interfaces
6. Functional Independence  
single-minded function and low coupling
7. Refinement  
elaboration of detail for all abstractions
8. Re-factoring  
a reorganization technique that simplifies the design

1.1. Data Abstraction

Abstraction - allows designers to simplify a problem and focus on solving a problem without being concerned about irrelevant lower level details
(an example: a class – a named collection of data objects)
1.2. Procedural Abstraction

Abstraction - allows designers to simplify a problem and focus on solving a problem without being concerned about irrelevant lower level details

(Example: procedural abstraction: a subroutine – a named sequence of events)

2. Architecture

"The overall structure of the software and the ways in which that structure provides conceptual integrity for a system."

- **Structural properties.** This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

- **Families of related systems (Reusable architectural building blocks).** The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.
System Engineering: A Structural View

System Level
(World View)

Level of Subsystems (Domains)
(Subsystem or Domain View)

Level of Elements or Components
(Element or Component View)

Level of Sub-elements, Details (for ex., attributes)
(Detail View)

Conceptual (Logic) Architecture of the SafeHomeAssured.com WebApp
3. Patterns

Design Pattern Template:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>In Design Patterns</th>
<th>In Code Examples</th>
<th>In Production</th>
<th>In Publishable Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural patterns</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract Factory</td>
<td>Provides an interface for creating families of related or dependent objects without specifying their concrete classes.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Builder</td>
<td>Bases the construction of a complex object on an existing object. Intends that its construction be done in a flexible manner.</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Composite</td>
<td>Enables the composition of a new object to that one object which has the same interface as the original object.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Decorator</td>
<td>Adds additional responsibilities to an object dynamically,without affecting the behavior of other objects on which it interacts.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Facade</td>
<td>Provides a unified access to a group of interfaces in a subsystem without the client having to know the details of its implementation.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Flyweight</td>
<td>Use messages to support the composition of objects that collaborate to achieve a common goal.</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Proxy</td>
<td>Provides a surrogate or placeholder for another object to control access to it.</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

3a. Patterns (details)

A design pattern is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.
3a. Patterns (details)

<table>
<thead>
<tr>
<th>3a. Patterns (details)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Behavioral patterns</strong>:</td>
</tr>
<tr>
<td><strong>Asyncanism</strong></td>
</tr>
<tr>
<td><strong>Actors</strong></td>
</tr>
<tr>
<td>- Types:</td>
</tr>
<tr>
<td>- Female</td>
</tr>
<tr>
<td>- Male</td>
</tr>
<tr>
<td>- Other</td>
</tr>
<tr>
<td><strong>Problem</strong></td>
</tr>
<tr>
<td>- Various users want to update their status online.</td>
</tr>
<tr>
<td><strong>Solution</strong></td>
</tr>
<tr>
<td>- Use a status update service to notify users of new updates.</td>
</tr>
</tbody>
</table>

**Asyncanism**
- Acts as a delegate to handle requests from clients and update the status of an object's status.
- Provides a way to handle requests from clients and update the status of an object's status.
4. Modular Design
(easier to build, easier to change, easier to fix…)

*easier to build, easier to change, easier to fix.*

*) 1969, Boeing 747
*) modern cars (standard modules)
*) reusable learning objects (modules of online courses)

Microsoft Design Center:
Industrialization and modular object-oriented software development has theoretically made it easy to tear things down and replace them.

Sizing Modules: Two Views

One important feature of a module is its length. Longer is not better.

Compilation time is expensive; the longer the module, the longer it takes to compile. Most program changes that require a recompilation of a module are for minor changes, not involving lots of code lines.

Try to restrict your modules to 400 lines of code; optimal length: 60-300 LOC per module

If the module needs to be longer, split it into multiple modules, all with a related name.

Source: http://www.stf.com/ASM/3015Modules
Software Modularity: Trade-offs

What is the "right" number of modules for a specific software design?

- Optimal number of modules
- Number of modules
- Cost of software
- Module development cost
- Module integration cost

---

Modular Approach to Learning Content Design:

- System Level
- Level of Subsystems (Domains)
- Level of Components
- Level of elements
Pedagogical Patterns are high-level patterns that have been recognized in many areas of training and pedagogy such as group work, software design, human computer interaction, education and others.

The concept is an extension of pattern languages. In both cases, the patterns seek to foster best practices of teaching.

You may think about patterns as some kind of metadata – a top-level description of data about data.

5. Information Hiding (or, Encapsulation in Programming)

- algorithm
- data structure
- details of external interface
- resource allocation policy

Module controlled interface

Clients

"secret"

A specific design decision
Why Information Hiding?

- leads to encapsulation—an attribute of high quality design
- results in higher quality software
- limits the global impact of local design decisions
- emphasizes communication through well-structured and controlled interfaces
- reduces the likelihood of “side effects”
- discourages the use of global data

Advantages of Encapsulation in Programming

- Large software programs may be split into manageable modules
- Implementation details are hidden (isolated); as a result, no need to waste your time
- Subprograms and programs become more portable, and, probably, re-usable by external users
- Development time is shortened due to well-written, well-structured, well-tested sub-programs (functions, lists, files, etc.)

6. Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" (disliking) to excessive interaction with other modules.

- **Cohesion** is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

- **Coupling** is an indication of the relative interdependence among modules.
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
System Concepts

Once we have recognized something as a system, how do we understand the system?

Important system concepts include:

**Modularity** is dividing a system into parts/chunks/modules of relatively uniform size.

**Decomposition** is the process of breaking down a system into its component parts.

**Coupling** is the extent to which subsystems are dependent on each other.

**Cohesion** is the extent to which a system or a subsystem performs a single function.

**Open system:** a system that interacts freely with its environment, taking input and returning output.

**Closed system:** a system that is cut off from its environment and does not interact with it.

---

7. Stepwise Refinement

**Object**

- **open**

**Activity Diagram (textual form)**

walk to door; reach for knob; open door; walk through; close door.

repeat until door opens

- turn knob clockwise;
- if knob doesn't turn, then take key out;
- find correct key;
- insert in lock;
- endif
- pull/push door
- move out of way;
- end repeat

**Instructions (code)**

open

walk to door; reach for knob;
open door;
walk through;
close door.

repeat until door opens

- turn knob clockwise;
- if knob doesn't turn, then take key out;
- find correct key;
- insert in lock;
- endif
- pull/push door
- move out of way;
- end repeat
WebApp Design: Main Components

- The WebApp Design Model encompasses
  1. content,
  2. aesthetics (artistic features),
  3. interface,
  4. architecture (structural components),
  5. navigation (functions), and
  6. component-level (objects) design issues.

- The design model provides sufficient information for the WebE team to construct the final WebApp

- Alternative solutions are considered, and the degree to which the current WebApp Design Model will lead to an effective implementation should be continuously assessed

Conceptual Architecture

- Provides an overall structure for the WebApp design

- Affects all later increments – so important for it to be developed in the context of the full set of likely increments

- Represents the major functional and information components for the WebApp and describes how these will fit together

- Depends on the nature of the WebApp, but in every case, it should ensure a sound integration between the WebApp information and the WebApp functionality.
Developing the architecture of WebApp

- How do we achieve an effective balance between information and functionality in the conceptual architecture?

- A good place to start is with workflows or functional scenarios (which are an expression of the system functionality) and information flows

- As a simple example, consider the following set of key functionalities for SafeHomeAssured.com
  - Provide product quotation
  - Process security system order
  - Process user data
  - Create user profile
  - Draw user space layout
  - Recommend security system for layout
  - Process monitoring order
  - Get and display account info
  - Get and display monitoring info
  - Customer service functions (to be defined later)
  - Tech support functions (to be defined later)

From these key functionalities we can identify the following partial list of functional subsystems:

- UserManagement. Manages all user functions, including user registration, authentication and profiling, user-specific content, and interface adaptation and customization.
- ProductManagement. Handles all product information, including pricing models and content management.
- OrderHandling. Supports the management of customers' orders.
- AccountAdministration. Manages customers' accounts, including invoicing and payment.
- SecuritySystemMonitoring. Monitors customers’ security systems and handles security events.

And, of course, there are overall management subsystems:

- ClientInterface. Provides the interface between users and the other subsystems, as required to satisfy users needs.
- SystemMaintenance. Provides maintenance functionality, such as database cleaning.
Languages for Logic Modeling of WebApp

- HDM - W2000
- RMM
- OOHDM
- ARANEUS
- STRUDEL
- TIRAMISU
- WebML
- Hera
- UML Web Application Extension
- UML-based Web Engineering (UWE)
- ACE
- WebArchitect
- OO-H

Technical Architecture

- Shows how the conceptual architecture can be mapped into specific technical components

- Any decision made about how one component might map into the technical architecture will affect the decisions about other components

  - For example, the WebE team may choose to design SafeHomeAssured.com in a way that stores product information as XML files. Later, the team discovers that the content management system doesn’t easily support access to XML content, but rather assumes that the content will be stored in a conventional relational database. One component of the technical architecture conflicts with constraints imposed by another component.