SEN 307: Software Design II

What is a design?

A meaningful engineering representation of something that is to be built.

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction. UI can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

Attractive

Simple to use

Responsive in short time

Clear to understand

Consistent on all interfacing screens

UI is broadly divided into two categories:

Command Line Interface Command Line Interface

Graphical User Interface Graphical User Interface

Command Line Interface (CLI)

CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users. CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate. CLI uses less amount of computer resource as compared to GUI User Interface Design

User interface design is an essential part of the software design process. User interface design should ensure that interaction between the human and the machine provides for effective operation and control of the machine. For software to achieve its full potential, the user interface should be designed to match the skills, experience, and expectations of its anticipated users.

General User Interface Design Principles

Learnability. The software should be easy to learn so that the user can rapidly start working with the software.

User familiarity. The interface should use terms and concepts drawn from the experiences of the people who will use the software.

Consistency. The interface should be consistent so that comparable operations are activated in the same way.

Minimal surprise. The behavior of software should not surprise users.

Recoverability. The interface should provide mechanisms allowing users to recover from errors.

User guidance. The interface should give meaningful feedback when errors occur and provide context-related help to users.

User diversity. The interface should provide appropriate interaction mechanisms for diverse types of users and for users with different capabilities (blind, poor eyesight, deaf, colorblind, etc.)

User Interface Design Issues

User interface design should solve two key issues: How should the user interact with the software? How should information from the software be presented to the user? User interface design must integrate user interaction and information presentation. User interface design should consider a compromise between the most appropriate styles of interaction and presentation for the software, the background and experience of the software users, and the available devices.

The Design of User Interaction Modalities

User interaction involves issuing commands and providing associated data to the software. User interaction styles can be classified into the following primary styles:

Question-answer. The interaction is essentially restricted to a single question-answer exchange between the user and the software. The user issues a question to the software, and the software returns the answer to the question.

Direct manipulation. Users interact with objects on the computer screen. Direct manipulation often includes a pointing device (such as a mouse, trackball, or a finger on touch screens) that manipulates an object and invokes actions that specify what is to be done with that object.

Menu selection. The user selects a command from a menu list of commands.

Form fill-in. The user fills in the fields of a form. Sometimes fields include menus, in which case the form has action buttons for the user to initiate action.

Command language. The user issues a command and provides related parameters to direct the software what to do.

Natural language. The user issues a command in natural language. That is, the natural language is a front end to a command language and is parsed and translated into software commands.

8 Characteristics of Successful User Interfaces

There is a lot of information out there about various interface design techniques and patterns you can use when crafting your user interfaces and websites, solutions to common problems and general usability recommendations. Following guidelines from experts will likely lead you towards creating a good user interface – but what exactly is a good interface? What are the characteristics of an effective user interface? Here are 8 things to be considered in a good user interface:
1. Clear 2. Concise 3. Familiar 4. Responsive 5. Consistent 6. Attractive 7. Efficient and 8. Forgiving

Clear
Clarity is the most important element of user interface design. Indeed, the whole purpose of user interface design is to enable people to interact with your system by communicating meaning and function. If people can't figure out how your application works or where to go on your website they'll get confused and frustrated.

What does that do? Hover over buttons in WordPress and a tooltip will pop up explaining their functions.

2. Concise

Clarity in a user interface is great; however, one should be careful not to fall into the trap of over-clarifying. It is easy to add definitions and explanations, but every time one does that you add mass. Your interface grows. Adding too many explanations will make users to spend too much time reading through them. Keep things clear but also keep things concise. When you can explain a feature in one sentence instead of three, do it. When you can label an item with one word instead of two, do it. Save the valuable time of your users by keeping things concise. Keeping things clear and concise at the same time isn't easy and takes time and effort to achieve, but the rewards are great.[

The volume controls in OS X use little icons to show each side of the scale from low to high.

3. Familiar

Many designers strive to make their interfaces 'intuitive'. But what does intuitive really mean? It means something that can be naturally and instinctively understood and comprehended. But how can you make something intuitive? You do it by making it 'familiar'.

Familiar is just that: something which appears like something else you've encountered before. When you're familiar with something, you know how it behaves – you know what to expect. Identify things that are familiar to your users and integrate them into your user interface.

4. Responsive

Responsive means a couple of things. First of all, responsive means fast. The interface, if not the software behind it, should work fast. Waiting for things to load and using laggy and slow interfaces is frustrating. Seeing things load quickly, or at the very least, an interface that loads quickly (even if the content is yet to catch up) improves the user experience.

Responsive also means the interface provides some form of feedback. The interface should talk back to the user to inform them about what's happening. Have you pressed that button successfully? How would you know? The button should display a 'pressed' state to give that feedback. Perhaps the button text could change to "Loading…" and it's state disabled. Is the software stuck or is the content loading? Play a spinning wheel or show a progress bar to keep the user in the loop.

5. Consistent

Now, I've talked before about the importance of context and how it should guide your design decisions. I think that adapting to any given context is smart, however, there is still a level of consistency that an interface should maintain throughout.

Consistent interfaces allow users to develop usage patterns – they'll learn what the different buttons, tabs, icons and other interface elements look like and will recognize them and realize what they do in different contexts. They'll also learn how certain things work, and will be able to work out how to operate new features quicker, extrapolating from those previous experiences.

6. Attractive

This one may be a little controversial but, a good interface should be attractive so as to make its usage enjoyable. Yes, you can make your UI simple, easy to use, efficient and responsive, and it will do its job well – but if you can go that extra step further and make it attractive, then you will make the experience of using that interface truly satisfying. When your software is pleasant to use, your customers or staff will not simply be using it – they'll look forward to using it.

There are of course many different types of software and websites, all produced for different markets and audiences. What looks 'good' for any one particular audience may vary. This means that you should fashion the look and feel of your interface for your audience. Also, aesthetics should be used in moderation and to reinforce function. Adding a level of polish to the interface is different to loading it with superfluous eye-candy.

7. Efficient

A user interface is the vehicle that takes you places. Those places are the different functions of the software application or website. A good interface should allow you to perform those functions faster and with less effort. Now, 'efficient' sounds like a fairly vague attribute – if you combine all of the other things on this list, surely the interface will end up being efficient? Almost, but not quite.

What you really need to do to make an interface efficient is to figure out what exactly the user is trying to achieve, and then let them do exactly that without any fuss. You have to identify how your application should 'work' – what functions does it need to have, what are the goals you're trying to achieve? Implement an interface that lets people easily accomplish what they want instead of simply implementing access to a list of features.

8. Forgiving

Nobody is perfect, and people are bound to make mistakes when using your software or website. How well you can handle those mistakes will be an important indicator of your software's quality. Don't punish the user – build a forgiving interface to remedy issues that come up.

A forgiving interface is one that can save your users from costly mistakes. For example, if someone deletes an important piece of information, can they easily retrieve it or undo this action? When someone navigates to a broken or nonexistent page on your website, what do they see? Are they greeted with a cryptic error or do they get a helpful list of alternative destinations?

The Design of Information Presentation

Information presentation may be textual or graphical in nature. A good design keeps the information presentation separate from the information itself. The MVC (Model-View-Controller) approach is an effective way to keep information presentation separating from the information being presented. Software Design Software engineers also consider software response time and feedback in the design of information presentation. Response time is generally measured from the point at which a user executes a certain control action until the software responds with a response. An indication of progress is desirable while the software is preparing the response. Feedback can be provided by restating the user's input while processing is being completed. Abstract visualizations can be used when large amounts of information are to be presented. According to the style of information presentation, designers can also use color to enhance the interface. There are several important guidelines: Limit the number of colors used. Use color change to show the change of software status. Use color-coding to support the user's task. Use color-coding in a thoughtful and consistent way. Use colors to facilitate access for people with color blindness or color deficiency (e.g., use the change of color saturation and color brightness, try to avoid blue and red combinations). Don't depend on color alone to convey important information to users with different capabilities (blindness, poor eyesight, colorblindness, etc.).

User Interface Design Process

User interface design is an iterative process; interface prototypes are often used to determine the features, organization, and look of the software user interface. This process includes three core activities: ***User analysis***. In this phase, the designer analyzes the users' tasks, the working environment, other software, and how users interact with other people. ***Software prototyping***; Developing prototype software help users to guide the evolution of the interface. ***Interface evaluation***; Designers can observe users' experiences with the evolving interface.

Localization and Internationalization

User interface design often needs to consider internationalization and localization, which are means of adapting software to the different languages, regional differences, and the technical requirements of target market. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without major engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating the text. Localization and

internationalization should consider factors such as symbols, numbers, currency, time, and measurement units.

Metaphors and Conceptual Models

User interface designers can use metaphors and conceptual models to set up mappings between the software and some reference system known to the users in the real world, which can help the users to more readily learn and use the interface. For example, the operation “delete file” can be made into a metaphor using the icon of a trash can. When designing a user interface, software engineers should be careful to not use more than one metaphor for each concept. Metaphors also present potential problems with respect to internationalization, since not all metaphors are meaningful or are applied in the same way within all cultures.

Software Design Quality Analysis and Evaluation

Quality Attributes

Various attributes contribute to the quality of a software design, including various “-ilities” (maintainability, portability, testability, usability) and “-nesses” (correctness, robustness). There is an interesting distinction between quality attributes discernible at runtime (for example, performance, security, availability, functionality, usability), those not discernible at runtime (for example, modifiability, portability, reusability, testability), and those related to the architecture’s intrinsic qualities (for example, conceptual integrity, correctness, completeness).

Quality Analysis and Evaluation Techniques

Various tools and techniques can help in analyzing and evaluating software design quality. Software design reviews: informal and formalized techniques to determine the quality of design artifacts (for example, architecture reviews, design reviews, and inspections; scenario-based techniques; requirements tracing). Software design reviews can also evaluate security. Aids for installation, operation, and usage (for example, manuals and help files) can be reviewed. Static analysis: formal or semiformal static (nonexecutable) analysis that can be used to evaluate a design (for example, fault-tree analysis or automated cross-checking). Design vulnerability analysis (for example, static analysis for security weaknesses) can be performed if security is a concern. Formal design analysis uses mathematical models that allow designers to predicate the behavior and validate the performance of the software instead of having to rely entirely on testing. Formal design analysis can be used to detect residual specification and design errors (perhaps caused by imprecision, ambiguity, and sometimes other kinds of mistakes). (See also the Software Engineering Models and Methods KA.) Simulation and prototyping: dynamic techniques to evaluate a design (for example, performance simulation or feasibility prototypes).

Measures

Measures can be used to assess or to quantitatively estimate various aspects of a software design such as size, structure, or quality. Most measures that have been proposed depend on the approach used for producing the design. These measures are classified in two broad categories: *Function-based (structured) design measures*: measures obtained by analyzing functional decomposition; generally represented using a structure chart (sometimes called a hierarchical diagram) on which various measures can be computed. *Object-oriented design measures*: the design structure is typically represented as a class diagram, on which various measures can be computed. Measures on the properties of the internal content of each class can also be computed.

Software Design Notations

Many notations exist to represent software design artifacts. Some are used to describe the structural organization of a design, others to represent software behavior. Certain notations are used mostly during architectural design and others mainly during detailed design, although some notations can be used for both purposes. In addition, some notations are used mostly in the context of specific design methods. Please note that software design is often accomplished using multiple notations. Here, they are categorized into notations for describing the structural (static) view vs. the behavioral (dynamic) view.

Structural Descriptions (Static View)

The following notations, mostly but not always graphical, describe and represent the structural aspects of a software design—that is, they are used to describe the major components and how they are interconnected (static view):

Architecture description languages (ADLs): textual, often formal, languages used to describe software architecture in terms of components and connectors.

Class and object diagrams: used to represent a set of classes (and objects) and their interrelationships.

Component diagrams: used to represent a set of components ("physical and replaceable part[s] of a system that [conform] to and [provide] the realization of a set of interfaces") and their interrelationships. Class responsibility collaborator cards (CRCs): used to denote the names of components (class), their responsibilities, and their collaborating components' names.

Deployment diagrams: used to represent a set of (physical) nodes and their interrelationships, and, thus, to model the physical aspects of software.

Entity-relationship diagrams (ERDs): used to represent conceptual models of data stored in information repositories.

Interface description languages (IDLs): programming-like languages used to define the interfaces (names and types of ex ported operations) of software components.

Structure charts: used to describe the calling structure of programs (which modules call, and are called by, which other modules).

Behavioral Descriptions (Dynamic View)

The following notations and languages, some graphical and some textual, are used to describe the dynamic behavior of software systems and components. Many of these notations are useful mostly, but not exclusively, during detailed design. Moreover, behavioral descriptions can include a rationale for design decision such as how a design will meet security requirements.

Activity diagrams: used to show control flow from activity to activity which can be used to represent concurrent activities.

Communication diagrams: used to show the interactions that occur among a group of objects; emphasis is on the objects, their links, and the messages they exchange on those links.

Data flow diagrams (DFDs): used to show data flow among elements. A data flow diagram provides "a description based on modeling the flow of information around a network of operational elements, with each element making use of or modifying the information flowing into that element". Data flows (and therefore data flow diagrams) can be used for security analysis, as they offer identification of possible paths for attack and disclosure of confidential information.

Decision tables and diagrams: used to represent complex combinations of conditions and actions.

Flowcharts: used to represent the flow of control and the associated actions to be performed.

Sequence diagrams: used to show the interactions among a group of objects, with emphasis on the time ordering of messages passed between objects.

State transition and state chart diagrams: used to show the control flow from state to state and how the behavior of a component changes based on its current state in a state machine.

Formal specification languages: textual languages that use basic notions from mathematics (for example, logic, set, sequence) to rigorously and abstractly define software component interfaces and behavior, often in terms of pre- and post-conditions.

Pseudo code and program design languages (PDLs): structured programming-like languages used to describe, generally at the detailed design stage, the behavior of a procedure or method.

Software Design Strategies and Methods

There exist various general strategies to help guide the design process. In contrast with general strategies, methods are more specific in that they generally provide a set of notations to be used with the method, a description of the process to be used when following the method, and a set of guidelines for using the method. Such methods are useful as a common framework for teams of software engineers.

General Strategies

Some often-cited examples of general strategies useful in the design process include the divide and conquer and stepwise refinement strategies, top-down vs. bottom-up strategies, and strategies making use of heuristics, use of patterns and pattern languages, and use of an iterative and incremental approach.

Function-Oriented (Structured) Design

This is one of the classical methods of software design, where decomposition centers on identifying the major software functions and then elaborating and refining them in a hierarchical top-down manner. Structured design is generally used after structured analysis, thus producing (among other things) data flow diagrams and associated process descriptions. Researchers have proposed various strategies (for example, transformation analysis, transaction analysis) and heuristics (for example, fan-in/fan-out, scope of effect vs. scope of control) to transform a DFD into a software architecture generally represented as a structure chart.

Object-Oriented Design

Numerous software design methods based on objects have been proposed. The field has evolved from the early object-oriented (OO) design of the mid-1980s (noun = object; verb = method; adjective = attribute), where inheritance and polymorphism play a key role, to the field of component-based design, where meta information can be defined and accessed (through reflection, for example). Although OO design's roots stem from the concept of data abstraction, responsibility-driven design has been proposed as an alternative approach to OO design.

Data Structure-Centered Design

Data structure-centered design starts from the data structures a program manipulates rather than from the function it performs. The software engineer first describes the input and output data structures and then develops the program’s control structure based on these data structure diagrams. Various heuristics have been proposed to deal with special cases—for example, when there is a mismatch between the input and output structures.

Component-Based Design (CBD)

A software component is an independent unit, having well-defined interfaces and dependencies that can be composed and deployed independently. Component-based design addresses issues related to providing, developing, and integrating such components in order to improve reuse. Reused and off-the-shelf software components should meet the same security requirements as new software. Trust management is a design concern; components created as having a certain degree of trustworthiness should not depend on less trustworthy components or services.

Other Methods

Other interesting approaches also exist; such as iterative and adaptive methods that implement software increments and reduce emphasis on rigorous software requirement and design. Aspect-oriented design is a method by which software is constructed using aspects to implement the crosscutting concerns and extensions that are identified during the software requirements process. Service-oriented architecture is a way to build distributed software using web services executed on distributed computers. Software systems are often constructed by using services from different providers because standard protocols (such as HTTP, HTTPS, SOAP) have been designed to support service communication and service information exchange.