User Interaction Design

Interaction Design Overview

Interaction design is concerned with specifying the user experience for a software product. As such, it is concerned with the visual and aural appearance of the program and with the sequence of events that occur as a user and a program exchange data. The quality of a user interface is often referred to as its *usability*, but this term is both vague and incomplete because there are user experience goals that may go beyond usability (such as the aesthetic quality of an interface). Consequently it is better to focus on more specific and complete interaction design goals, including the following.

Effectiveness—Users can access all the features needed to achieve their goals.

Efficiency—Users achieve their goals quickly.

Safety—Neither users, their computers, nor their data are harmed by the product.

Learnability—Users become proficient in a short time.

Memorability—Users regain proficiency quickly after not using the program for some time.

Enjoyability—Users experience positive emotions when using the product.

Beauty—Users find the product is aesthetically pleasing.

Interaction design draws material from many fields, in particular from ergonomics (the study of how to design and arrange things so that people can use them easily and safely), perceptual physiology and psychology (the study of the senses and our perceptual mechanisms), cognitive psychology (the study of memory, learning, and thinking), and graphic design (the study of communication through the use of type, space, color, and image). We cannot do more than the broadest survey of this rich field in this chapter. We will concentrate on the place of interaction design in the software life cycle, models and notations for interaction design, interaction design processes, and a few interaction design principles For a far more detailed treatment, see [1] or [2].

Interaction Design In the Life Cycle

Interaction design is a major part of product design, and so it is part of software requirements specification. In traditional approaches, interaction design has often been done as an after-thought or a mini-phase between requirements specification and engineering design. Designing user interaction after most of the features and behaviors of a product have been specified makes it very difficult to do well because interaction is in large part *about* the features that a product should provide and the behaviors it should exhibit. A better approach is for interaction design to be a driver of requirements specification.

Agile methods tend to omit mention of interaction design altogether, but the fundamental directive that agile teams include all the skills needed to create a product means that interaction design skills should be present in the team and that interaction design should be part of agile processes. Several practitioners have proposed that interaction design be conducted during a sprint for the features the team expects to be implemented in the following sprint. Then the designers have time to prototype and test designs with users before they are incorporated into the product.

Interaction Design Models and Notations

Interaction designers specify both the appearance and the behavior of products, and hence they need both static and dynamic models. A **static interaction design model** depicts the visual and

aural features of products that do not change during execution, such as screen layouts and sounds associated with states or events. A **dynamic interaction design model** depicts product behavior during execution, such as product states and state transitions and the communications between the product and its users. There are several notations for making static and dynamic models.

Static Model Notations

A **use case** is a type of complete interaction between a product and its environment [3]. For example, some use cases for an e-reader are reading a book, browsing an e-bookstore, buying a book, or browsing books on the e-reader. An **actor** is a type of agent that interacts with a product. For example, an e-reader's actors might be a reader and an e-bookstore. A use case diagram is a UML notation for showing the use cases in a product. A **use case diagram** represents a product's use cases and the actors involved in each use case. The diagram in Figure 1 shows an example UML use case diagram.

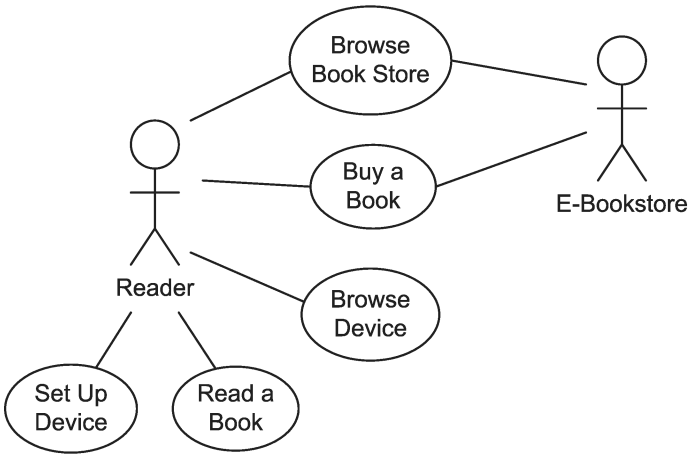

Figure 1: A Use Case Diagram for an E-Book Reader

This diagram shows use cases for an e-book reader. The diagram shows two actors: the Reader is the person operating the device, and the E-Bookstore is a computer accessed over the Internet. Use cases are shown as ovals. A line between a use case and an actor indicates that the actor is involved in that use case. The Browse Book Store use case is an interaction that occurs when the Reader uses the device to look at e-books available in the bookstore—both actors are involved in this interaction. The Reader may also Buy a Book from the E-Bookstore, again involving both actors. The other use cases do not involve the E-Bookstore, only the Reader.

Use case diagrams are an easy way to represent collections of features that a product might have. They are useful in product design because several diagrams representing different collections of features show design alternatives. It is helpful to make several of these diagrams and manipulate them to consider various combinations of product features.

A **screen** (or **page**) **layout diagram** is a drawing of (part of) a product's visual display when it is in a particular state. Screen layout diagrams are easy to sketch and modify and are a simple but essential tool in designing graphical user interfaces. Layout diagrams can be drawn at various levels of abstraction. A good place to start is with a low-fidelity diagram, often called a **wireframe**, that roughly indicates the layout of screen elements without many details. There are many free tools on the Internet for making wireframes. Once a satisfactory overall design is formulated, a high-fidelity diagrams showing colors, fonts, controls, and so forth can be made as the basis for implementation. Low- and high-fidelity screen layout diagrams are easy to draw and manipulate, so they are good

tools for considering many design alternatives. They also are a great help to stakeholders in visualizing what a product will look like.

The following is an example of a wireframe for a screen from a document search system.

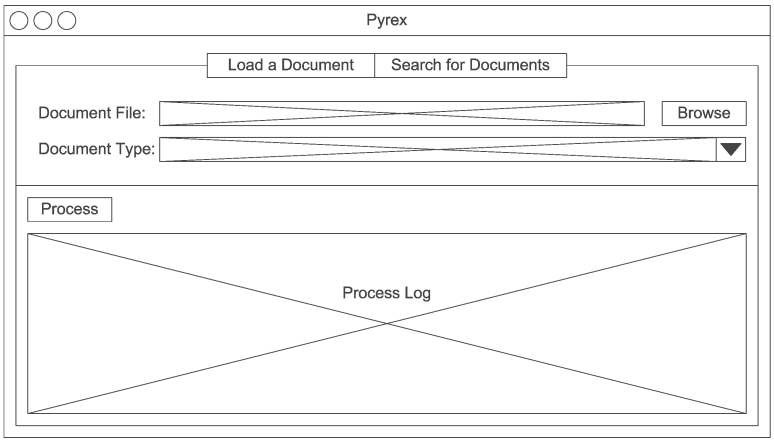

Figure 2: A Wireframe for a Screen in a Document Retrieval System

This window has a title (Pyrex) and an indication of window controls (the circles at the top left). It depicts a frame with two tabs. The rectangles with crossed lines indicate text fields. Labels, buttons, and a combo box are also shown. This drawing could be part of a a preliminary design, but it shows enough detail that it could also be the basis for an implementation.

Dynamic Model Notations

A **use case description** is a specification of the interaction between the product and the actors in a use case. Use case descriptions are text, usually conforming to a template. They provide some context information for the use case, along with a narrative resembling a script that describes the sequence of actions by actors and the product as they interact. Figure 3 below shows a typical use case description template.

Use Case Name: To identify the use case
Actors: The agents participating in the use case
Stakeholders and Needs: What this use case does to meet stakeholder needs
Preconditions: What must be true before this use case begins
Post conditions: What will be true when this use case ends
Trigger: The event that causes this use case to begin
Basic Flow: The steps in a typical successful instance of this use case
Extensions: The steps in alternative instances of this use case occurring either because of variations in the normal flow or because of errors.

Figure 3: A Use Case Description Template

Use case descriptions fill in the details of the use cases named in a use case diagram. They provide the notation for the next step in product design after choosing features, which is to describe the interactions realizing each feature. Use case descriptions can be used to explore alternative

interaction flows. This is important because the order in which interactions occur has a big influence over many interaction design criteria, such as efficiency, safety, learnability, memorability, and enjoyability.

A **storyboard** is a collection of screen layout diagrams linked by arrows depicting events or the passage of time. A storyboard shows how the screen changes over time or in response to various inputs. Storyboards can be made to elaborate use case descriptions by filling in the details of the visual display. They can even be linked to capture the connections between use cases. Storyboard development usually begins with wireframe nodes. As design progresses higher-fidelity nodes can be filled in, or placeholders can be used to refer to higher fidelity screen layout diagrams outside the storyboard.

Like screen layout diagrams, storyboards are an excellent tool for investigating design alternatives and for helping stakeholders visualize designs before their implementation.

Use Case Models and Requirements

Many people have suggested that use case models be used as requirements specifications. Use cases model the interactions between a system and its users and other things with which it exchanges data (like other computer systems). Use case models are valuable for interaction design, and can document some requirements. But in principle they cannot document all requirements because they are not able to model requirements that don't have to do with interactions. Thus, for example, they are not intended to capture specifications about how things must be computed (for example, equations relating inputs to outputs), how a system must be implemented (for example, what sort of database system it must use), and most non-functional requirements (like reliability, safety, security, portability, and performance requirements).

Another problem with use case models for documenting requirements is that they are organized to trace interactions and not to specify behaviors in relation to system features. This can sometimes make it difficult to see the overall organization of a user interface, and to check whether an interaction specification is complete.

In summary, use case models are good for interaction design and they can help in expressing interaction requirements, but they cannot capture all requirements, and they may not always be the best vehicle for expressing interaction requirements.

Product backlog items (PBIs) in Scrum are the standard means for specifying features to be included in a product. As explained in the chapter about requirements, PBIs are high-level (mainly) functional requirements that must be elaborated before they can be implemented. A PBI may specify some aspect of a product's interaction with its environment, but as high-level specifications they are really not suited to capture the details of interactions. Use case descriptions do capture such details, and so they are a good tool for elaborating interaction PBIs during sprints. They can also provide supplementary information if a PBI is elaborated to a fairly low-level of detail during backlog grooming.

Interaction Design Processes

Like other design processes, interaction design is mainly a top-down activity that begins with the most abstract models and gradually refines them until a detailed specification is complete. After eliciting needs, desires, ideas, and preferences from stakeholders, a use case diagram can be formulated to pin down product features and capabilities. Use case descriptions should follow as a means to determine the best overall flow of interaction. Screen layout diagrams and storyboards can

be made to aid in design, to obtain feedback from stakeholders, and as specifications for implementation.

In both traditional and agile approaches, it is often advisable to conduct usability testing at various points during interaction design. **Usability testing** is empirical evaluation of (parts of) an interaction design to determine whether it meets interaction design goals [4]. Usability testing can take many forms, but it typically requires specifying the interaction design goals to be tested and the response that will be considered adequate. Several test subjects are chosen to be representative of the user population and a testing scenario and environment are selected and set up. The scenario can be "executed" on anything from a rough paper prototype to a full-featured product. Users are exposed to the scenario and their behavior measured. Measurements might include pre- and post-test questionnaires or surveys, observation of user actions, timing how long it takes users to achieve goals, counting user mistakes, and so forth. The data is analyzed and conclusions drawn about the effectiveness of the design and how it might be improved. Several rounds of usability testing may be done to refine important parts of an interaction design.

Interaction Design Principles

A **design principle** is statement that certain characteristics make a design better, used as a quality criterion in generating and evaluating designs. There are many interaction design principles; here we mention nine of them in three categories that seem particularly helpful to novice interaction designers.

The first group of three principles applies to interaction design in general. The principles can be remembered using the acronym SAC, which stand for Simplicity, Accessibility, and Consistency.

> **Simplicity**—Simpler designs are better. Simplicity is a principle that is fundamental in every design discipline. An example in interaction design is proliferation of interface features. Many commercial products include (apparently) every feature that anyone has ever thought of, making them very complex. The user is confronted with a plethora of controls, and long, multi-level menus. Just finding basic features becomes difficult, not to mention the difficulty of choosing between a wealth of similar options. A better alternative is a product that offers all basic functions in a simple, elegant design. Rarely done tasks can still be completed, though perhaps with a bit more work. The principle of simplicity says it is better to opt for streamlined, simplified access to basic functions in exchange for slightly more effort to accomplish rarely done tasks.
>
> **Accessibility**—Designs that can be used by more people are better. In the US, about eight percent of the population has red-green color blindness; older people generally have difficulty reading text in small fonts; many people have tremors or fine-motor control problems that make it difficult or impossible for them to use small controls with a mouse. Designs that rely on red or green to make important distinctions, use small fonts, or have small controls will be unusable by these populations. Better design will accomplish their ends while still being accessible to a wide range of users.
>
> **Consistency**—Designs that present similar data in similar ways, and provide similar ways of accomplishing similar tasks, are better. For example, all the pages of web site should have the same navigation mechanisms in the same places on the page, and invoking them should take the user to the same location. Fonts, colors, and layout should be used in a consistent fashion throughout a site.

The SAC principles apply to both product appearance and behavior. The CAP principles apply in particular to appearance. CAP stands for Contrast, Alignment, and Proximity.

Contrast—Designs that make things that are different appear different are better. This principle has a long history in typography and has lead to conventions about the use of serif versus sans-serif fonts (use one for headings, the other for text), italics and boldface (for emphasis), and font-size (for distinguishing different levels of headers, for example) [5]. It applies as well to colors, images, and controls.

Alignment—Designs that line-up elements in a grid are better. Alignment makes designs look neater and may also convey information about relationships between elements. For example, text that is aligned on both the left and right looks neater and more formal than text that is only aligned on the left (so-called ragged-right text). As another example, it is conventional to use indentation (and alignment on the left) for different levels of an outline or hierarchical list, and of course we use left alignment in source code to show the bodies of control structures.

Proximity—Designs that group related items together spatially are better. For example, when arranging a group of related radio buttons, it is better to place them neatly in a tight group separated from other elements by vertical or horizontal space. This signals to users that these buttons are connected and also makes it easier for users to discern the range of choices available to them.

The final group of three design principles applies especially to behavior. These are the FeVER principles, which stands for Feedback, Visibility, and Error prevention and Recovery.

Feedback—Designs that acknowledge user actions are better. Without feedback users often cannot know whether their actions are having any effect. For example, imagine a mail program that did not indicate in any way whether a message had actually been sent when the send button was pressed. Most users would probably press it several times to make sure their mail was sent, resulting in involuntary spam. Feedback can be very subtle, such as changing the color of an icon or emitting a small noise. Nevertheless, some sort of feedback is essential for efficient and effective use of most programs.

Visibility—Designs that prominently display their state and available operations are better. The state of a program often has consequences for what a user can do at a given time. For example, the popular UNIX text editor vi has two states: insertion mode and edit mode. In insertion mode, whatever the user types is inserted into the text, but in edit mode, whatever the user types is interpreted as editing commands. Unfortunately, vi makes its state visible only through display of "insert" or nothing at the bottom of the screen, so it is hard for users to keep track of the program state. Veteran users learn to press the escape key whenever they lose track of the program state because it forces the program into edit mode. Wouldn't it be better for the program to display its state in a more overt fashion? Programs that do not make available operations visible are similarly hard to use, and again vi provides a bad example. The only way to discover editor commands is to type ":h" in edit mode (for help)—and there is no way to discover this incantation in the program. Then the user has to remember the commands when returning to editing. This makes vi very difficult to use until the user has mastered many of its commands.

Error Prevention and Recovery—Designs that prevent user errors and provide error recovery mechanisms are better. There are many ways that a design may prevent errors. For example, buttons that should not be pressed or menu items that should not be selected in some program state should be disabled. If the program needs a value from a small set of values (like months or a range of numbers), don't ask the user to type in the value: have the user select from a list or menu. Not all errors can be prevented, so programs should also provide ways to recover from them. A common example is the Undo feature in many programs.

References

1. Preece, Jenny, Helen Sharp, and Yvonne Rogers. *Interaction Design: Beyond Human-Computer Interaction 4th Edition*. Wiley, 2015.
2. Shneiderman, Ben, Catherine Plaisant, Maxine Cohen, and Steven Jacobs. *Designing the User Interface: Strategies for Effective Human-Computer Interaction 5th Edition*. Pearson, 2009.
3. Cockburn, Alistair. *Writing Effective Use Cases*. Addison-Wesley, 2000.
4. Leventhal, Laura, and Julie Barnes. *Usability Engineering: Process, Products, and Examples*. Prentice Hall, 2008.
5. Williams, Robin. *The Non-Designers Design Book*. Peachpit Press, 1994.