

MODULE 4

DESIGN TOOLS AND TECHNOLOGY

11. VISUAL DESIGN BASICS:

11.1 THE GOLDEN RULES:

In his book on interface design, Theo Mandel [Man97] coins three golden rules:

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

11.1.1 Place the User in Control

Understanding the User's Perspective

- Users desire systems that anticipate their needs and simplify their tasks.
- An ideal interface should **respond to the user's actions naturally** rather than imposing unnecessary restrictions.
- A **well-designed user experience** puts the user in charge rather than letting the system dictate how tasks should be performed.

Real-World Example

- A user, when asked about interface preferences, humorously requested a system that could "read their mind."
- While unrealistic, this request highlights a key expectation: users want intuitive and effortless interaction.
- The best user interfaces feel **natural and frictionless**, ensuring the user controls the system—not the other way around.

Design Pitfalls: Constraints That Benefit Developers, Not Users

- Many **design limitations** exist to simplify **interface development** rather than enhancing usability.
 - A system that's **easy to build** may **frustrate users** if it forces them into rigid workflows.
 - UI designers should prioritize **user flexibility and efficiency** over developer convenience.
-

Key Principles for User Control (Mandel, 1997)

1. Avoid Forcing Unnecessary Actions

- **Definition:** The interface should allow users to move freely between tasks without unnecessary restrictions.
- **Example:** A word processor's spell-check mode should allow users to edit text **without exiting** spell check mode.

2. Provide Flexible Interaction

- Different users prefer **different input methods** (e.g., keyboard, mouse, touch, voice commands).
- **Example:**
 - A user drawing a complex shape will likely prefer a **stylus or mouse** over voice input.
 - Not all actions should be **locked to a single interaction mode**.

3. Allow Interruptions and Undoing Actions

- Users should be able to **pause** or **switch tasks** without losing progress.
- Actions should always be **reversible** (Undo function).
- **Example:**
 - A user in an image editing app should be able to undo changes **step-by-step** rather than restarting from scratch.

4. Streamline Interaction for Advanced Users

- Users **repeat common actions**—interfaces should provide **shortcuts** or **automation** for efficiency.

- **Example:**
 - Software should offer **macros** or **customized shortcuts** for frequent tasks (e.g., batch editing).

Additional Design Considerations

5. Hide Technical Complexity from Users

- Users should not be required to interact with **technical system components** (OS, file management, system commands).
- **Example:**
 - A **text editing application** should not require users to manually enter OS commands to save a file.

6. Enable Direct Interaction with On-Screen Objects

- Users feel more **in control** when they can **manipulate objects intuitively**.
- **Example:**
 - A user resizing an image should be able to **drag and stretch** it rather than inputting precise numerical dimensions.

Key Takeaways

- ✓ **Empower users** by minimizing restrictions and allowing freedom of interaction.
- ✓ **Avoid forcing** users into rigid workflows—provide flexibility and undo options.
- ✓ **Enhance efficiency** by enabling shortcuts and automation for advanced users.
- ✓ **Simplify interactions** by hiding unnecessary technical complexities.
- ✓ **Encourage direct interaction** with screen objects for an intuitive experience.

11.1.2 Reduce the User's Memory Load

Key Concept

- The more a user has to **remember**, the higher the chances of making **errors** during system interaction.

- A **well-designed UI** minimizes memory load by allowing the system to “remember” relevant details and assist the user in recalling past actions.
-

Key Principles for Reducing Memory Load (Mandel, 1997)

1. Reduce Demand on Short-Term Memory

- Users working on complex tasks often struggle with remembering previous actions, inputs, and results.
 - The UI should provide **visual cues** to help **recognize past actions** instead of requiring recall.
 - **Example:**
 - A search bar should display **recent search history** to prevent users from retyping previous queries.
 - A form-filling system should **autofill fields** based on past user inputs.
-

2. Establish Meaningful Defaults

- **Defaults** should be pre-set to suit the **average user’s needs** while still allowing customization.
 - Users should be able to **modify preferences**, but a **reset option** should be available to restore original defaults.
 - **Example:**
 - In a word processor, the default **font size and style** should be readable and commonly used (e.g., Arial 12pt).
 - A software installation process should **preselect** commonly used features while allowing advanced users to customize settings.
-

3. Define Intuitive Shortcuts

- Shortcuts should be **easy to learn and remember**.
- Mnemonics (shortcut keys) should be logically tied to their functions.

- **Example:**
 - **Ctrl + P** for printing (P stands for Print).
 - **Ctrl + Z** for undo (Z represents “going back”).
 - **Cmd + S** for saving (S stands for Save).
-

4. Use Real-World Metaphors

- A UI should mimic **real-world objects** and workflows to make interaction intuitive.
 - Users can **rely on familiar visual cues** instead of memorizing abstract sequences.
 - **Example:**
 - A **bill payment system** should resemble a **checkbook** to help users intuitively understand the process.
 - A **shopping cart system** in e-commerce mirrors a **physical shopping cart** experience.
-

5. Disclose Information Progressively

- The UI should be **structured hierarchically**, presenting **high-level information first** and revealing more details **only when needed**.
 - This prevents cognitive overload by not overwhelming the user with too much information at once.
 - **Example:**
 - A word processor groups text formatting options under a **"Text Style" menu**. When a user selects "Underline," further options appear (e.g., **single, double, dashed underline**).
-

Key Takeaways

- ✓ **Minimize memory demands** by providing visual aids instead of requiring users to remember past actions.
- ✓ **Use meaningful defaults** that suit most users but allow customization.

- ✓ **Make shortcuts intuitive** and easy to remember.
 - ✓ **Leverage real-world metaphors** to make UI interactions natural.
 - ✓ **Present information progressively**, revealing details only when necessary.
-

11.1.3 Make the Interface Consistent

Core Idea

- **Consistency in user interface (UI) design** is crucial for usability and efficiency.
 - A well-designed interface should present and acquire information in a **predictable and uniform manner**.
 - Consistency reduces the user's learning curve and **ensures seamless interaction** across different screens, tasks, and applications.
-

Key Principles for Creating a Consistent Interface (Mandel, 1997)

1. Organize Visual Information According to Design Rules

- **All screen elements should follow a common design pattern** throughout the application.
 - **Color schemes, fonts, icons, and layouts** should be standardized across all screens.
 - **Example:**
 - If a **submit button** is blue on one screen, it should not be **green on another**.
 - **Navigation menus** should appear in the same location across all pages.
-

2. Maintain Consistency in Input Mechanisms

- **A limited set of input methods** should be used consistently across the system.
- Users should not have to **learn different ways to interact with different screens**.
- **Example:**
 - If pressing **Enter** submits a form in one section, it should not require clicking a button in another.

- If a **drag-and-drop** feature exists in one part of the system, users should expect it elsewhere if applicable.
-

3. Define and Standardize Navigation Mechanisms

- **Task navigation should be uniform** across the application.
 - Users should know **where they are, how they got there, and how to move forward or back**.
 - **Example:**
 - **Breadcrumb navigation** helps users understand their location in a multi-step process.
 - **Consistent icons** (e.g., a house icon always leading to the homepage) improve predictability.
-

4. Allow Users to Understand the Current Task Context

- In complex applications with **multiple layers of interaction**, users must always be aware of their current task.
 - The system should provide **context indicators** such as:
 - **Window titles**
 - **Icons and graphical elements**
 - **Consistent color coding**
 - **Example:**
 - A **highlighted menu option** can indicate the current section of the app.
 - A **progress bar** in a multi-step form shows how far along the user is.
-

5. Maintain Consistency Across a Family of Applications

- If multiple applications (or products) are part of the same ecosystem, they should **share the same UI principles**.

- This ensures users can **transition between different apps** without having to relearn interactions.
 - **Example:**
 - Microsoft Office apps (Word, Excel, PowerPoint) share **similar toolbars and menu structures**.
 - Google services (Gmail, Drive, Docs) use a **consistent material design approach**.
-

6. Avoid Changing Established Interactive Models Without a Strong Reason

- Users develop expectations based on **common UI patterns**.
 - Once an interactive sequence becomes a **de facto standard**, changing it can **frustrate users**.
 - **Example:**
 - **Ctrl + S** is universally used to **save files**. Changing it to **Ctrl + Z** for saving would confuse users.
 - A **red 'X'** in the corner of a window always means **close**; changing it to minimize would break user expectations.
-

Key Takeaways

- ✓ Consistency in design, input methods, and navigation enhances usability.
 - ✓ Users should always know where they are and what actions are available.
 - ✓ Applications within the same family should follow a shared UI design.
 - ✓ Avoid unnecessary changes to well-established UI conventions.
-

11.4 INTERFACE DESIGN STEPS

Overview

Once the **interface analysis** is completed, the next step is **interface design**. At this stage, all **tasks, objects, and actions** required by the user are clearly defined, and the interface is structured accordingly. Like other **software engineering** processes, **interface design** is **iterative**, meaning it goes through multiple refinements to enhance usability and effectiveness.

Various **user interface design models** (e.g., Norman's model [Nor86] and Nielsen's model [Nie00]) have been proposed, but they generally follow a set of core **design steps** that help create an effective user interface.

Interface Design Steps

1. Define Interface Objects and Actions

Using information obtained from the **interface analysis phase**, the first step is to **identify interface objects and actions**.

- **Objects** are elements that users interact with (e.g., buttons, forms, icons).
- **Actions** are operations that users perform on these objects (e.g., click, drag, enter text).

The objects and actions are structured in a way that makes user interaction smooth and intuitive.

2. Define Events That Change the Interface State

User interactions (**events**) can cause the **state of the interface to change**.

- These events can include **mouse clicks, keyboard inputs, or touch gestures**.
- The behavior of the interface in response to these events must be clearly defined.

For example, clicking a **"Submit"** button should transition the system from an input screen to a confirmation screen. Modeling this behavior ensures that users understand how the system reacts to their actions.

3. Depict Each Interface State as Seen by the User

The next step is to **visually represent each interface state**.

- This includes designing the **actual layout, colors, buttons, icons, and menus** as they will appear to users.
- Creating **wireframes or mockups** of different interface states helps refine usability.

For example, a **shopping cart** may have different states:

- **Empty cart state** → Displays a message: *"Your cart is empty."*

- **Cart with items** → Shows products, total price, and checkout button.
- **Checkout state** → Displays payment options and confirmation details.


By clearly defining how each interface state appears, **users can understand and navigate the system easily**.

4. Indicate How Users Interpret the Interface

A well-designed UI should help users **understand what is happening** at all times. This involves:

- **Providing meaningful feedback** (e.g., error messages, success confirmations).
- **Using clear labels and instructions** to guide users.
- **Ensuring consistency** in design elements, so users do not get confused.

For instance, when entering a **wrong password**, the system should clearly indicate:

 *"Incorrect password. Please try again."* instead of just failing silently.

By carefully considering how users **interpret and interact** with the interface, designers can create a system that is easy to use and minimizes confusion.

11.4.1 Applying Interface Design Steps

Defining Interface Objects and Actions

To achieve an effective design, **user scenarios** are analyzed to extract **nouns (objects)** and **verbs (actions)**. This helps determine:

- **What the user interacts with** (e.g., buttons, icons, menus).
- **What actions the user performs** (e.g., drag, drop, click).

Objects are further categorized into three types:

1. **Source objects** → Items that initiate actions (e.g., a report icon).
2. **Target objects** → Items that receive an action (e.g., a printer icon).
3. **Application objects** → Data that the system manages (e.g., mailing lists).

For example, in a **reporting application**:

- A **source object** (report icon) is **dragged and dropped** onto a **target object** (printer icon)
→ This triggers a **hard-copy printout**.

Once objects and actions are finalized, the **screen layout** is designed, focusing on:

- **Graphical design & icon placement.**
- **Menu structures & screen text.**
- **Title bars & window arrangements.**

If applicable, a **real-world metaphor** (e.g., a desktop with files and folders) is chosen to improve user familiarity and ease of use.

SafeHome System Example

To illustrate these steps, consider a **SafeHome security system**. A homeowner wants to access their **security system remotely via a web browser**.

User Scenario:

- The homeowner logs in using an **ID and password**.
- They check the **system's current security status**.
- They can **arm or disarm the system**.
- They can view **floor plans, security zones, and sensor locations**.
- They can **view live video feeds from installed cameras**.
- They can **pan and zoom** the cameras for better viewing.

Extracted Objects and Actions:

Object	Action
SafeHome System	Access remotely
Login Form	Enter username & password
Floor Plan	View security zones
Video Camera	View live feed, Zoom, Pan

Object	Action
--------	--------

Arm/Disarm Button	Activate or deactivate system
-------------------	-------------------------------

A **preliminary screen layout sketch** is created to visualize the **video monitoring interface**:

- The **floor plan** is displayed with **camera icons**.
- Users **drag and drop** a camera icon onto the **video monitor window** to view live footage.
- The **zoom and pan controls** adjust the camera view.

Each **menu item** and **screen state** is expanded into detailed sketches to refine functionality and improve usability.

11.4.2 User Interface Design Patterns

What Are UI Design Patterns?

User interfaces have evolved into **common patterns** that help users navigate applications more easily.

A **UI design pattern** is a **reusable design solution** to a specific, well-defined problem.

For example, one common **UI problem** is selecting a **date in the future**.

◆ Pattern: CalendarStrip

A **CalendarStrip** is a **scrollable calendar** where:

- The **current date is highlighted**.
- The user can **select a future date** by scrolling.

This design **follows a well-known metaphor (a physical calendar)**, making it intuitive for users.

Why Use UI Design Patterns?

- ✓ **Familiarity** → Users already recognize standard patterns.
- ✓ **Efficiency** → Reduces learning curve and increases usability.
- ✓ **Consistency** → Ensures the same design approach across applications.

Example of UI Patterns:

Problem	UI Pattern
Navigating pages	Breadcrumb navigation
Filling forms	Multi-step form wizard
Searching for items	Autocomplete search bar

These patterns help maintain a **smooth and user-friendly interface**, making the system easier to use.

Conclusion

The **interface design process** follows a structured set of **steps** that guide designers in creating an intuitive, efficient, and user-friendly UI.

- ✓ **Defining objects and actions** ensures clarity.
- ✓ **Modeling user events** helps anticipate system responses.
- ✓ **Depicting interface states** allows users to understand system behavior.
- ✓ **Providing clear feedback** improves usability.
- ✓ **Using design patterns** ensures consistency and familiarity.

By following these principles, an interface can be designed to enhance **user experience** while minimizing confusion and errors. 🚀

Here's a detailed description of the provided content on WebApp interface design:

11.5 WebApp Interface Design

A user interface is a critical component of any software system, whether a **WebApp**, a traditional application, or an industrial device. The usability of an interface significantly influences user experience.

Dix [Dix99] suggests that an effective WebApp interface should help users answer three key questions:

1. **Where am I?** – The interface should clearly indicate the WebApp being accessed and provide a sense of location within its content hierarchy.

2. **What can I do now?** – It should help users understand their available options, including active functions and links.
3. **Where have I been, and where am I going?** – The interface must offer intuitive navigation aids, such as breadcrumbs or maps, to track past and future navigation paths.

11.5.1 Interface Design Principles and Guidelines

A well-designed interface plays a crucial role in **user retention**. Regardless of the WebApp's functionality, a **poor interface design** can drive users away.

Fundamental Characteristics of Effective WebApp Interfaces

Bruce Tognozzi [Tog01] identifies essential traits that every WebApp interface should embody:

- **Visually apparent and forgiving** – Users should feel in control, clearly see options, and be able to recover from errors.
- **Hides complexity** – The interface should not expose users to the system's inner workings; work should be saved automatically, with an undo option.
- **Minimizes user input** – The system should do most of the work while requiring minimal input from the user.

Here's a detailed description of **Tognozzi's Design Principles for WebApp Interfaces** from the provided data:

Tognozzi's Design Principles for WebApp Interfaces

Bruce Tognozzi [Tog01] outlines several **core design principles** that WebApp interfaces should follow to ensure they are effective, user-friendly, and efficient. These principles aim to **optimize user experience** by guiding WebApp designers to anticipate needs, streamline interactions, and minimize complexity. Below are the detailed descriptions of each principle:

1. Anticipation

- **Principle:** A WebApp should be designed to **predict** the user's next actions, anticipating what the user may need or want to do.
- **Example:** Consider a WebApp for a printer company. If a user searches for a specific printer driver for a newly released operating system, the WebApp should predict that the user will want to download this driver and provide an **easy way to do so**, perhaps by

offering a download link alongside the relevant information, without requiring the user to search for it.

- **Goal: Reduce effort** for the user by proactively offering next steps and relevant options, enhancing user convenience and satisfaction.

2. Communication

- **Principle:** The WebApp interface should clearly **communicate the status** of any activities initiated by the user, as well as the user's current location and status within the system.
- **Example:** If a user submits a print command, the interface could show an animation of paper moving through a printer or provide a text notification saying, "Printing in progress." Additionally, the system should notify the user of **who they are** (e.g., logged in as a customer) and **where they are** in the WebApp (e.g., within the "support section").
- **Goal:** Keep the user informed about ongoing processes and their interaction status to reduce confusion and enhance confidence in the system.

3. Consistency

- **Principle:** The interface should be **consistent** in its use of design elements such as navigation controls, icons, and layout. Users should expect uniform behavior across the entire WebApp.
- **Example:** If underlined blue text indicates a hyperlink in one part of the WebApp, it should **always** function as a link elsewhere. If a yellow triangle is used to represent a caution message in one part of the system, it should not be repurposed for something unrelated elsewhere in the interface.
- **Goal:** Users should have **predictable experiences** based on the interface design, improving usability and minimizing errors.

4. Controlled Autonomy

- **Principle:** While users should be allowed to **navigate freely** within the WebApp, the interface must enforce certain **navigation conventions** and security protocols.
- **Example:** Navigation to secure areas of the WebApp should be **restricted by login credentials**, such as a username and password. There should be **no bypassing** of these security measures, and the WebApp should prevent unauthorized actions or routes.

- **Goal:** Empower users to explore the WebApp freely, but within **defined boundaries** that ensure **security and consistency**.

5. Efficiency

- **Principle:** The interface should be designed to **optimize the user's work efficiency**, not necessarily the efficiency of the developers or the system environment. The design should aim to make the user's workflow smooth and fast.
- **Example:** A well-designed WebApp may automatically fill in details such as shipping address or payment method if the user has used the service before, **minimizing repetitive data entry**.
- **Goal: Maximize user productivity** by designing an interface that supports efficient task completion without unnecessary steps or delays.

6. Flexibility

- **Principle:** The WebApp interface should offer enough **flexibility** to accommodate both structured tasks and free-form exploration, depending on user preferences or skill levels.
- **Example:** Some users may prefer a straightforward, **step-by-step** approach to filling out a form, while others might want to explore different sections or features without following a linear flow. The WebApp should cater to both types of users.
- **Goal: Adapt to different user preferences**, allowing users to choose how they interact with the WebApp, while still ensuring they can achieve their goals effectively.

7. Focus

- **Principle:** The interface should maintain **focus** on the user's **current task**, avoiding unnecessary distractions.
- **Example:** A WebApp might contain multiple content areas, but the interface should prioritize the main task at hand. For example, if the user is in a **checkout process**, the interface should avoid leading the user to unrelated information like promotions or social media feeds.
- **Goal:** Keep the user on track by ensuring that they do not get **sidetracked** by irrelevant information or features.

8. Fitt's Law

- **Principle:** According to Fitt's Law, the time it takes for a user to acquire a target (e.g., clicking a button) depends on the **distance to the target** and the **size of the target**. To optimize interface design, elements requiring frequent interaction should be **closer** to the user's current position and **larger** in size for easy selection.
- **Example:** On an e-commerce site, the **"Buy Now" button** should be **large enough** and placed **close to the product details** to minimize the effort needed for users to click it.
- **Goal: Reduce interaction time** and effort by making interactive elements easy to access and use.

9. Human Interface Objects

- **Principle:** A large library of **pre-built human interface objects** (such as buttons, forms, and menus) is available for use. These elements should be reused wherever possible to create a consistent, familiar user experience.
- **Example:** Rather than designing custom forms or buttons from scratch, use standardized, widely recognized interface components that users are already familiar with.
- **Goal: Leverage existing design elements** to create a more intuitive, recognizable, and standardized interface that reduces the learning curve for users.

10. Latency Reduction

- **Principle:** The WebApp should minimize **waiting times** and reduce user frustration due to latency. While certain operations might take time, the user should be able to continue working, and any delays should be acknowledged visually or audibly.
- **Example:** If a user is waiting for a large image to download, the WebApp could **continue loading other parts of the page** in the meantime, or show a progress bar to indicate that the operation is in progress.
- **Goal: Improve responsiveness** and maintain user engagement even during operations that require time.

11. Learnability

- **Principle:** The interface should be easy to **learn** and **quick to remember**. It should minimize the time required for a new user to become comfortable and effective in using the WebApp.

- **Example:** If a user revisits the WebApp after a long break, they should not need to relearn the interface. The design should remain simple, with well-organized content that doesn't change drastically from one visit to the next.
- **Goal:** Make the WebApp **easy to use from the start** and ensure minimal effort is needed to learn or re-learn the system.

12. Metaphors

- **Principle:** Use real-world **metaphors** to make interactions easier for users to understand. A metaphor is a representation that draws upon the user's existing knowledge of real-world experiences.
- **Example:** An **online checkbook metaphor** for bill payment allows users to "write checks" without having to manually enter all the details. Instead, they can choose from a list of payees or have the system auto-complete fields.
- **Goal:** Use **familiar analogies** that simplify complex tasks and make the WebApp intuitive and easier to understand.

13. Maintain Work Product Integrity

- **Principle:** Any work a user is doing (such as filling out a form or creating a list) should be **saved automatically** to prevent data loss if an error occurs.
- **Example:** If a user is filling out a lengthy form and the WebApp crashes or the user accidentally navigates away, the entered data should be preserved and available when the user returns to the page.
- **Goal:** **Protect the user's work** and prevent frustration caused by losing data.

14. **Readability** – Text should be legible for all users, with appropriate font choices and contrast.

15. **Track State** – Users should be able to log out and resume later without losing their progress, achieved through cookies or other tracking methods.

16. **Visible Navigation** – Navigation should be effortless, allowing users to retrieve content without confusion.

Tognozzi's principles aim to make WebApp interfaces **intuitive, efficient, and user-centered**, enhancing the user experience by anticipating needs, ensuring clarity in communication, and maintaining consistency across the design. These principles provide a clear framework to create

interfaces that cater to a broad range of user preferences while keeping the focus on usability, efficiency, and minimal frustration.

Additional Guidelines by Nielsen and Wagner [Nie96]

- **Avoid long blocks of text** – Since reading speed on screens is slower, keep instructions concise.
- **Do not use “Under Construction” signs** – Users expect all links to be functional.
- **Minimize scrolling** – Important content should fit within a typical browser window.
- **Ensure consistent navigation menus** – Every page should include familiar navigation elements.
- **Prioritize functionality over aesthetics** – A simple, clear button is better than an obscure but visually appealing icon.
- **Make navigation intuitive** – Users should instantly recognize how to move within the WebApp.

Conclusion

A well-crafted **WebApp interface** significantly enhances usability, ensuring users can quickly locate information, perform tasks efficiently, and navigate seamlessly. **Clear, consistent, and intuitive design** is the key to success in WebApp development.

11.5.2 Interface Design Workflow for WebApps

The **interface design workflow** for Web applications (WebApps) follows a structured process that ensures a **user-friendly and efficient** interface. This workflow begins with **analyzing user needs** and progresses through **design iterations, prototyping, and refinement**.

Key Steps in WebApp Interface Design Workflow

1. Review Requirements Model

- Examine user, task, and environmental requirements.
- Ensure **user objectives align with interface functions**.
- **Example:** A **news portal** should prioritize easy navigation between categories like politics, sports, and entertainment.

2. Develop a Rough Sketch of the Interface

- Create **wireframes or prototypes** to visualize layout and navigation.
- If a **preliminary prototype** exists from requirements modeling, refine it.
- **Example:** A **banking WebApp** might sketch out login pages, account summary screens, and transaction forms.

3. Map User Objectives into Interface Actions

- Identify **primary user objectives** and match them to specific UI actions.
- **Example:** In an **e-commerce WebApp**, objectives might include:
 - Searching for products → Search bar
 - Adding to cart → "Add to Cart" button
 - Completing purchase → "Checkout" process

4. Define User Tasks for Each Action

- Break down actions into step-by-step user tasks.
- **Example:** For a **product purchase**, steps might include:
 1. Select product
 2. Choose specifications (size, color)
 3. Add to cart
 4. Enter shipping details
 5. Confirm and pay

5. Storyboard Screen Images for Each Interaction

- Create a **visual sequence** to represent user interactions.
- Define **navigation links, content objects, and WebApp functions**.
- **Example:** A **music streaming app** might storyboard steps for searching, playing, and adding songs to a playlist.

6. Refine Layout Based on Aesthetic Design Principles

- Adjust **spacing, alignment, colors, and typography** for usability.

- Balance **visual appeal with functionality** (e.g., avoiding cluttered designs).
- **Example:** A **news WebApp** should have a clear separation between headlines, images, and article text.

7. Identify Required Interface Objects

- Specify **UI components** like buttons, menus, sliders, and forms.
- Search for **existing reusable components** to optimize development.
- **Example:** A **video-sharing platform** might use:
 - A **play button** for videos
 - A **like/dislike** icon
 - A **comment section** for user interaction

8. Develop a Procedural Flow Representation (Optional)

- Use **UML sequence diagrams or activity diagrams** to depict interaction flow.
- Helps in defining **navigation sequences and error handling**.
- **Example:** A **food delivery app** flow diagram might show:
 1. User selects a restaurant
 2. Browses menu
 3. Places an order
 4. Payment processing
 5. Order confirmation and tracking

9. Develop a Behavioral Model for Interface (Optional)

- Use **UML state diagrams** to define interface state transitions.
- Identify **how external events (e.g., clicks, form submissions) trigger changes**.
- **Example:** A **social media app** might have states for:
 - **Logged out** → User must enter credentials
 - **Logged in** → Feed, messages, and profile options available
 - **Posting state** → User can type, upload, and publish a post

10. Describe the Interface Layout for Each State

- **Finalize layout** based on insights from previous steps.
 - Document **all UI states** to ensure clarity for developers.
 - **Example:** A travel booking WebApp should provide:
 - A **home screen** with search filters
 - A **results page** displaying available flights
 - A **booking confirmation page** with itinerary details
-

Best Practices for WebApp Interface Design

- ✓ **Ensure mobile compatibility** – The UI should be **responsive** across devices.
 - ✓ **Optimize navigation** – Use **clear menus, breadcrumbs, and intuitive icons**.
 - ✓ **Prioritize speed** – Avoid **heavy visuals** that slow down page load times.
 - ✓ **Improve accessibility** – Support **keyboard navigation, high-contrast themes, and screen readers**.
 - ✓ **Avoid unnecessary scrolling** – Keep **important content within the visible screen area**.
-

Conclusion

A well-defined **WebApp interface workflow** ensures a structured approach to **designing user-friendly, visually appealing, and functional interfaces**. By following these steps, developers can create **efficient, intuitive WebApps that meet user needs**.

12.1 Design Patterns

A **design pattern** provides a **reusable solution** to a common software design problem. It describes a standard way of solving issues related to software architecture, components, or interaction flow. Design patterns improve **modularity, flexibility, and maintainability** of software by **reusing proven solutions**.

12.1.1 Kinds of Patterns

Design patterns are a fundamental concept in software engineering, as they enable the reuse of solutions to common problems in software design. The importance of design patterns stems from **human beings' inherent ability to recognize patterns**. This ability, deeply ingrained over a

lifetime of experiences, allows people to quickly identify recurring phenomena and use these insights for problem-solving. In the context of software design, this ability translates to recognizing patterns in system structures, algorithms, and processes, which leads to efficient, repeatable, and adaptable solutions.

Generative vs. Nongenerative Patterns

A significant distinction in the world of design patterns is between **generative** and **nongenerative** patterns:

- **Nongenerative Patterns:** These patterns **describe phenomena** or situations without offering a clear solution or a way to implement them. A good example of a nongenerative pattern is the **RubberNecking** pattern in traffic. When an accident occurs on an interstate, traffic in the opposite direction often slows down as drivers attempt to view the accident, causing a **traffic jam**. While this pattern is predictable, it does not suggest a solution to the problem.
- **Generative Patterns:** In contrast, **generative patterns** are much more useful in design contexts. These patterns not only describe recurring problems and contexts but also provide **pragmatic solutions** to those problems. In software engineering, generative design patterns are used to **build systems** that can adapt to change. By applying a set of generative patterns, developers can **construct applications** and systems that are flexible, scalable, and maintainable over time. This process, known as **generativity**, involves using patterns that encapsulate both problems and solutions, unfolding into larger, more comprehensive solutions.

Categories of Design Patterns

Design patterns can be categorized based on their **level of abstraction** and the types of problems they address. Some of the most important categories of design patterns include:

1. **Architectural Patterns:** These patterns address broad, **high-level design issues** related to the overall structure of a system. They provide solutions for structuring the entire system in a way that promotes efficiency, scalability, and maintainability.
2. **Data Patterns:** Data patterns focus on **recurrent data-related problems** and their solutions. These patterns often guide how data should be modeled, stored, and manipulated to achieve the best performance and organization in a system.
3. **Component Patterns (Design Patterns):** Component patterns, also known simply as **design patterns**, deal with problems related to the **development of subsystems and components** within a software system. They address issues such as how these

components communicate with one another and how they should be structured within the larger system architecture.

4. **Interface Design Patterns:** These patterns focus on **user interface issues**, specifically how to design interfaces that provide effective, intuitive, and user-friendly interactions with the system. These patterns take into account the specific **characteristics of end-users** and the needs of the system they are interacting with.
5. **WebApp Patterns:** WebApp patterns refer to the specific design challenges encountered in the development of **web applications**. They often overlap with other pattern categories but are unique in addressing the challenges specific to web-based systems, including user interaction, responsiveness, and data management in a web context.
6. **Idioms:** At the **lowest level of abstraction**, **idioms** describe how to implement specific algorithms or data structures in a particular programming language. These patterns are highly specific to the language in question and provide detailed implementation strategies.

Gamma's Three Main Types of Patterns (GoF)

In their influential book, **Gamma and colleagues (the "Gang of Four" or GoF)** focus on three key types of patterns that are especially relevant to **object-oriented design**:

1. **Creational Patterns:**

- **Focus:** These patterns deal with the **creation** of objects in a system. They abstract the process of object instantiation, allowing the system to **hide the specifics** of how objects are created and composed. This promotes flexibility and allows the system to adapt to changes without altering the way objects are instantiated.
- **Purpose:** They help define the **rules and constraints** around the creation of objects, such as how many objects of a particular class can exist at one time and how these objects should be created. This approach provides mechanisms for object creation while maintaining **flexibility**.

2. **Structural Patterns:**

- **Focus:** Structural patterns address problems related to how **objects and classes** are organized and integrated into a larger system. They help establish and optimize **relationships** between different entities within the system.

- **Purpose:** These patterns focus on creating efficient relationships between entities, such as **inheritance mechanisms** in class-based designs or **object composition** techniques in object-oriented systems. They aim to create flexible and easily maintainable systems by optimizing the structure of the system.

3. Behavioral Patterns:

- **Focus:** These patterns deal with how objects interact and communicate with one another within a system. They focus on the **distribution of responsibility** between objects and how they collaborate to achieve specific goals.
- **Purpose:** Behavioral patterns emphasize the **coordination of behaviors** between objects, helping to determine the most effective way to assign tasks and handle object interactions. They ensure that objects can communicate in a flexible and adaptable way, depending on the situation at hand.

Conclusion

Design patterns are crucial in software engineering because they encapsulate reusable solutions to common design problems. By recognizing and applying these patterns, software engineers can build systems that are **efficient, scalable, and easy to maintain**. The distinction between **generative** and **nongenerative patterns** highlights the value of providing not only an understanding of the problem but also practical, implementable solutions. The categories of patterns—such as creational, structural, and behavioral patterns—help developers to approach different aspects of software design with **tried and tested solutions**.

12.1.2 Frameworks:

- **Definition of Framework:** A **framework** provides a **skeletal infrastructure** that serves as a reusable **mini-architecture**. It offers a generic structure and behavior for a family of software abstractions. It is typically tailored to a specific problem domain and defines how these abstractions interact within the context of the application. A framework is a collection of cooperating classes, with customizable plug points to integrate domain-specific functionality.
- **Skeletal Structure with Plug Points:** A framework is a skeleton structure that includes pre-defined areas or **plug points** (also referred to as **hooks** and **slots**) where developers can insert domain-specific classes or functionality. These plug points allow the framework to be adapted to the unique needs of a given problem domain while maintaining the overall structure.

- **Difference Between Design Patterns and Frameworks:** The **Gang of Four** (Gamma et al.) highlight the differences between design patterns and frameworks in the following ways:
 1. **Abstraction Level:**
 - Design patterns are **more abstract** than frameworks. They offer general solutions without prescribing specific implementation details.
 - Frameworks are **concrete** solutions that can be directly implemented in code. Frameworks provide a tangible structure that can be executed and reused.
 2. **Size and Scope:**
 - Design patterns are **smaller architectural elements** that address specific problems. They focus on individual aspects of a system.
 - Frameworks are **larger and more comprehensive**. They typically combine multiple design patterns into a cohesive system that forms the backbone of an application.
 3. **Specialization:**
 - Design patterns are **general-purpose** solutions applicable across a broad range of software applications.
 - Frameworks are **domain-specific** and tailored to a specific application domain. They provide infrastructure for building software within a particular context and are often not reusable outside that domain.
- **Effectiveness of Frameworks:** Frameworks are most effective when applied **as-is**, without changes to their core structure. While developers can add additional design elements, these should be integrated through the plug points, ensuring that the framework remains intact while adapting to specific needs.

12.2 PATTERN-BASED SOFTWARE DESIGN:

- **Pattern Recognition in Design:** Successful designers, regardless of the field, excel in recognizing patterns that define problems and identify corresponding patterns that can be used to create effective solutions. In software design, this ability helps in identifying reusable patterns that can streamline the development process.
- **Pattern-Based Design in Software vs. Industrial Technology:** While pattern-based design is relatively new in software development, it has been widely used in industrial

technology for many years. Fields like automotive engineering, aerospace, machine tool manufacturing, and robotics have long relied on **pattern catalogs**—standardized mechanisms and configurations that serve as building blocks for designing complex systems.

- **Benefits of Pattern-Based Design:** Just as pattern-based design in industrial sectors brings predictability, risk mitigation, and increased productivity, applying similar principles to software development offers the same advantages. These benefits come from using proven solutions to solve recurring design problems, reducing the risk of failure and enhancing the efficiency of the development process.
- **Use of Existing Patterns:** During the design process, developers are encouraged to **leverage existing design patterns** whenever possible instead of reinventing new ones. This not only saves time but also ensures the design is grounded in tried-and-tested solutions that are more likely to succeed.

12.2.1 Pattern-Based Design in Context:

- **Not Used in Isolation:** Pattern-based design is not an isolated approach. It works in conjunction with other software design techniques and concepts such as architectural design, component-level design, and user interface design (discussed in Chapters 9 through 11). These approaches are interconnected and support the overall pattern-based design process.
- **Quality Guidelines and Fundamental Design Concepts:** In software design, quality guidelines and attributes are critical, forming the foundation for all design decisions. These guidelines are influenced by fundamental design concepts such as **separation of concerns**, **stepwise refinement**, and **functional independence**. Over time, heuristics and best practices have evolved to help designers create more effective and high-quality systems.
- **The Role of Pattern-Based Design:** Pattern-based design plays a crucial role in transitioning from abstract requirements models to concrete software designs. The **requirements model** represents an abstract version of the system, describing the problem, context, and forces that influence the design. This model helps to set the stage for the design but doesn't explicitly define it.
- **Quality Attributes in Design:** When starting the design process, keeping quality attributes in mind is essential. These attributes ensure that the design aligns with the requirements outlined in the requirements model. However, they don't directly guide

how to achieve the design but offer a way to assess the quality of the design after it has been created.

- **Using Proven Techniques:** When transforming abstract requirements into a concrete software design, it's important to use proven methods and tools for architectural, component-level, and interface design. These techniques have been developed to make the design process more efficient and effective. However, the key takeaway is that if a solution to a particular problem already exists in the form of a design pattern, use it. This is where pattern-based design proves most valuable: if a solution exists, applying it saves time and ensures reliability.

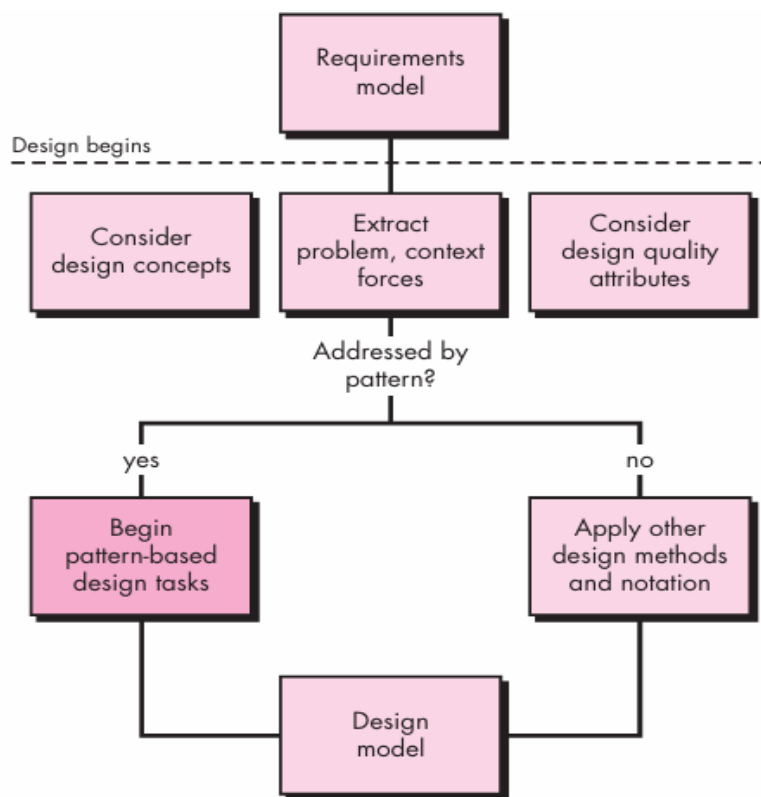


FIGURE 12.1 Pattern-based design in context

12.2.2 Thinking in Patterns

In their book on pattern-based design, Shalloway and Trott [Sha05] discuss a "new way of thinking" that comes with using patterns in the design process. They highlight a critical point

shared by Christopher Alexander: “good software design cannot be achieved simply by adding together performing parts.”

Good design starts with considering the **context**—the big picture. As you evaluate the context, you extract a hierarchy of problems that need solving. Some of these problems will be **global**, while others will address specific features or functions of the software. All will be influenced by a system of forces that shape the nature of the solution.

Shalloway and Trott propose the following approach to help designers think in patterns:

1. **Understand the Big Picture:** Be sure you grasp the context in which the software will be developed. The **requirements model** should communicate this to you.
2. **Extract Patterns at the High Level:** From the big picture, identify the patterns that exist at this level of abstraction.
3. **Start with Big Picture Patterns:** Begin your design with the broad, high-level patterns that form the foundation or skeleton of the software.
4. **Work Inward:** Look for patterns at lower levels of abstraction that contribute to the overall design solution.
5. **Repeat the Process:** Continue the steps until the entire design is fully developed.
6. **Adapt Patterns to the Software:** Refine the design by adjusting each pattern to suit the specifics of the software you are building.

Key Insights about Patterns in Design

- **Patterns Are Not Independent:** Patterns at higher levels of abstraction influence those at lower levels.
- **Collaboration of Patterns:** Patterns often work together. For example, selecting an **architectural pattern** can influence the **component-level design patterns** you choose. Similarly, an **interface design pattern** might require the use of other patterns that work alongside it.

Example: SafeHomeAssured.com WebApp

To illustrate how patterns work together, consider the **SafeHomeAssured.com WebApp**, which must solve fundamental problems such as:

- Providing information about SafeHome products and services
- Selling SafeHome products and services to customers

- Setting up Internet-based monitoring and control of an installed security system

Each of these problems can be broken down into smaller subproblems. For example, the problem of "selling via the Internet" suggests the need for an **E-commerce pattern**, which will lead to several patterns at lower levels of abstraction. The **E-commerce pattern** may require:

- Setting up a **customer account** (a potential architectural pattern)
- Displaying products for sale
- Selecting products for purchase

As you consider the design, it's important to check if patterns like **SetUpAccount** exist. If they do, they may collaborate with other patterns like **BuildInputForm**, **ManageFormsInput**, and **ValidateFormsEntry**, which each solve specific problems in the design.

12.5 User Interface Design Patterns

Hundreds of user interface (UI) patterns have been proposed in recent years. Most fall within one of the following 10 categories, each with representative examples as described by Tidwell [Tid02] and van Welie [Wel01]:

1. Whole UI

Provides design guidance for top-level structure and navigation throughout the entire interface.

Pattern: TopLevelNavigation

- **Brief Description:** Used when a site or application implements a number of major functions. It provides a top-level menu, often coupled with a logo or identifying graphic, that enables direct navigation to any of the system's major functions.
- **Details:** Major functions are listed across the top of the display, typically between four to seven functions. Each name provides a link to the relevant function or information source. Often used with the **BreadCrumbs** pattern.
- **Navigation Elements:** Each function/content name represents a link to the appropriate function or content.

2. Page Layout

Addresses the general organization of pages (for websites) or distinct screen displays (for interactive applications).

Pattern: CardStack

- **Brief Description:** Used when multiple specific subfunctions or content categories related to a feature must be selected in random order. It provides the appearance of a stack of tabbed cards.
 - **Details:** Tabbed cards are easy to manipulate. Each tab represents a specific subfunction or content category. Some tabs require input; others may be informational. This can be combined with other patterns like **DropDownList**, **Fill-in-the-Blanks**, etc.
 - **Navigation Elements:** A mouse click on a tab causes the appropriate card to appear.
-

3. Forms and Input

Focuses on design techniques for completing form-level input.

Pattern: Fill-in-the-Blanks

- **Brief Description:** Allows alphanumeric data to be entered in a “text box.”
 - **Details:** Data is entered into a text box, validated, and processed after selecting a text or graphic indicator (e.g., "go," "submit," or "next"). Often combined with other patterns like **DropDownList**.
 - **Navigation Elements:** A text or graphic indicator that initiates validation and processing.
-

4. Tables

Provides design guidance for creating and manipulating tabular data.

Pattern: SortableTable

- **Brief Description:** Displays a long list of records that can be sorted by selecting a toggle mechanism for any column label.
- **Details:** Each row represents a record, and each column represents a field. Column headers serve as toggle buttons to sort records in ascending or descending order. The table is resizable and may have a scrolling mechanism.

- **Navigation Elements:** Column headers initiate sorting, with potential navigation within records to access other content or functions.
-

5. Direct Data Manipulation

Addresses data editing, modification, and transformation.

Pattern: BreadCrumbs

- **Brief Description:** Provides a full navigation path when the user is working with a complex hierarchy of pages.
 - **Details:** The path to the current location is displayed in a predefined location and takes the form: home > major topic page > subtopic page > specific page > current page.
 - **Navigation Elements:** Entries in the breadcrumb path can be clicked to link back to a higher level of the hierarchy.
-

6. Navigation

Helps users navigate through hierarchical menus, Web pages, and interactive display screens.

Pattern: EditInPlace

- **Brief Description:** Provides simple text editing capabilities directly within the display location.
 - **Details:** A mouse double-click on content signals the start of editing. The content is highlighted, and the user can make changes.
 - **Navigation Elements:** None.
-

7. Searching

Enables content-specific searches through information in a website or persistent data stores.

Pattern: SimpleSearch

- **Brief Description:** Provides the ability to search a website or data source for a simple item described by an alphanumeric string.

- **Details:** Search can be local (one page) or global (entire site or database). A list of “hits” is generated based on their relevance.
 - **Navigation Elements:** Each entry in the list links to the data it references.
-

8. Page Elements

Implements specific elements of a Web page or display screen.

Pattern: Wizard

- **Brief Description:** Guides the user through a complex task step-by-step with a series of simple window displays.
 - **Details:** A classic example is a multi-step registration process, where each step requests specific information.
 - **Navigation Elements:** Forward and back buttons allow navigation through the wizard process.
-

9. E-commerce

Specific to websites, these patterns implement recurring elements of e-commerce applications.

Pattern: ShoppingCart

- **Brief Description:** Provides a list of items selected for purchase.
 - **Details:** Lists items, quantities, product codes, availability, prices, shipping costs, and other purchase details. It also allows the user to edit (remove, change quantity).
 - **Navigation Elements:** Includes options to proceed with shopping or go to checkout.
-

10. Miscellaneous

Patterns that do not easily fit into the above categories. They may be domain-specific or used by certain user groups.

Pattern: ProgressIndicator

- **Brief Description:** Shows an indication of progress when an operation takes longer than a few seconds.
 - **Details:** Represented by an animated icon (e.g., rotating "barber pole," progress bar, or percentage slider) that indicates processing is ongoing.
 - **Navigation Elements:** May include a button to pause or cancel processing.
-

Each of the above UI patterns would also have a complete **component-level design**, including design classes, attributes, operations, and interfaces. For a more comprehensive discussion of user interface patterns, refer to [Duy02], [Bor01], [Tid02], and [Wel01].

12.6 WebApp Design Patterns

Throughout this chapter, you've learned that there are different types of patterns and ways to categorize them. When considering the design problems to solve when building a WebApp, it's valuable to focus on two dimensions: **design focus** and **granularity**.

- **Design focus** identifies which aspect of the design model is relevant (e.g., information architecture, navigation, interaction).
 - **Granularity** identifies the level of abstraction being considered (e.g., does the pattern apply to the entire WebApp, a single Web page, a subsystem, or an individual WebApp component?).
-

12.6.1 Design Focus

The focus becomes "narrower" as you move further into design. In earlier chapters, a design progression was emphasized—starting from architecture, component-level issues, and user interface representations. At each step, the problems and solutions become more detailed and specific. Design focus becomes narrower as you delve deeper into the design.

The problems (and solutions) encountered when designing an information architecture for a WebApp are different from those encountered during interface design. Patterns for WebApp design are developed for different levels of focus to address unique problems at each level.

WebApp patterns can be categorized using the following levels of **design focus**:

- **Information Architecture Patterns:** Relate to the overall structure of the information space and how users interact with the information.
- **Navigation Patterns:** Define navigation link structures, such as hierarchies, rings, tours, etc.
- **Interaction Patterns:** Contribute to the design of the user interface, addressing how the interface informs users of actions, expands content based on context, and communicates ongoing interaction status.
- **Presentation Patterns:** Assist in presenting content via the interface. These patterns address the organization of UI control functions, showing relationships between actions and content, and establishing content hierarchies.
- **Functional Patterns:** Define workflows, behaviors, processing, communication, and other algorithmic elements within a WebApp.

In most cases, it's unnecessary to explore information architecture patterns when working on interaction design. Instead, interaction patterns should be explored, as they are more relevant to the task at hand.

12.6.2 Design Granularity

When a problem involves "big picture" issues, solutions (and patterns) should focus on the broader scope. On the other hand, when the focus is narrow (e.g., selecting one item from a small set), the solution (and the corresponding pattern) is targeted more specifically. Patterns can be described at the following levels of **granularity**:

- **Architectural Patterns:** These relate to the overall structure of the WebApp, defining relationships among different components, increments, and rules for specifying relationships among architecture elements (pages, components, subsystems).
- **Design Patterns:** Address specific elements of the design, such as aggregating components to solve a design problem, relationships among elements on a page, or mechanisms for component-to-component communication. For example, the **Broadsheet** pattern for the layout of a WebApp home page.
- **Component Patterns:** Relate to individual small-scale elements of the WebApp, such as individual interaction elements (e.g., radio buttons), navigation items (e.g., formatting links), or functional elements (e.g., specific algorithms).

It's also possible to define the relevance of different patterns to various application domains. For instance, a collection of patterns at different levels of design focus and granularity might be especially relevant to **e-business** applications.