3.4
Design Evaluation,Software Reuse, Component-Based Software Engineering

Steps in Interface Design

- Interface design, like all software engineering design, is an iterative process.
- Each user interface design step:
  - occurs a number of times,
  - elaborating and
  - refining information developed in the preceding step.

Steps in Interface Design

1. Using information developed during interface analysis,
   - ❑ Define interface objects and actions (operations).

2. Define events (user actions)
   - ❑ that will cause the state of the user interface to change.
   - ❑ Model this behavior.

3. Depict each interface state
   - ❑ as it will actually look to the end user.

4. Indicate how the user interprets the state of the system
   - ❑ from information provided through the interface.

Design Issues

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

Response time

- System response time has two important characteristics:
  - **length**
  - **variability.**
- If system response is too long, user frustration and stress are inevitable.
- *Variability* refers to the deviation from average response time, and in many ways.
  - Low variability enables the user to establish an interaction rhythm, even if response time is relatively long.
  - For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds.
  - When variability is significant, the user is always off balance, always wondering whether something “different” has occurred behind the scenes.

Help facilities

Design issues with a help facility:

• Will help be available for all system functions and at all times during system interaction?

- Options include help for:
  - only a subset of all functions and actions or
  - help for all functions.

• How will the user request help? Options include:

- a help menu,
- a special function key, or
- a HELP command.

• How will help be represented? Options include:

- a separate window,
- A reference to a printed document (less than ideal), or
- a one- or two-line suggestion produced in a fixed screen location.

• How will the user return to normal interaction? Options include:

- A return button displayed on the screen,
- a function key, or
- Control sequence.

Error handling

Error messages and warnings are “bad news” delivered to users of interactive systems when something has gone awry. Every error message or warning should have the following characteristics:

- The message should describe the problem in jargon that the user can understand.
- The message should provide constructive advice for recovering from the error.
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).
- The message should be accompanied by an audible or visual cue
- The message should be “non-judgmental.” That is, the wording should never place blame on the user.

Menu and command labelling

Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands,

- but some power-users continue to prefer a command-oriented mode of interaction

A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?
- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

Application accessibility

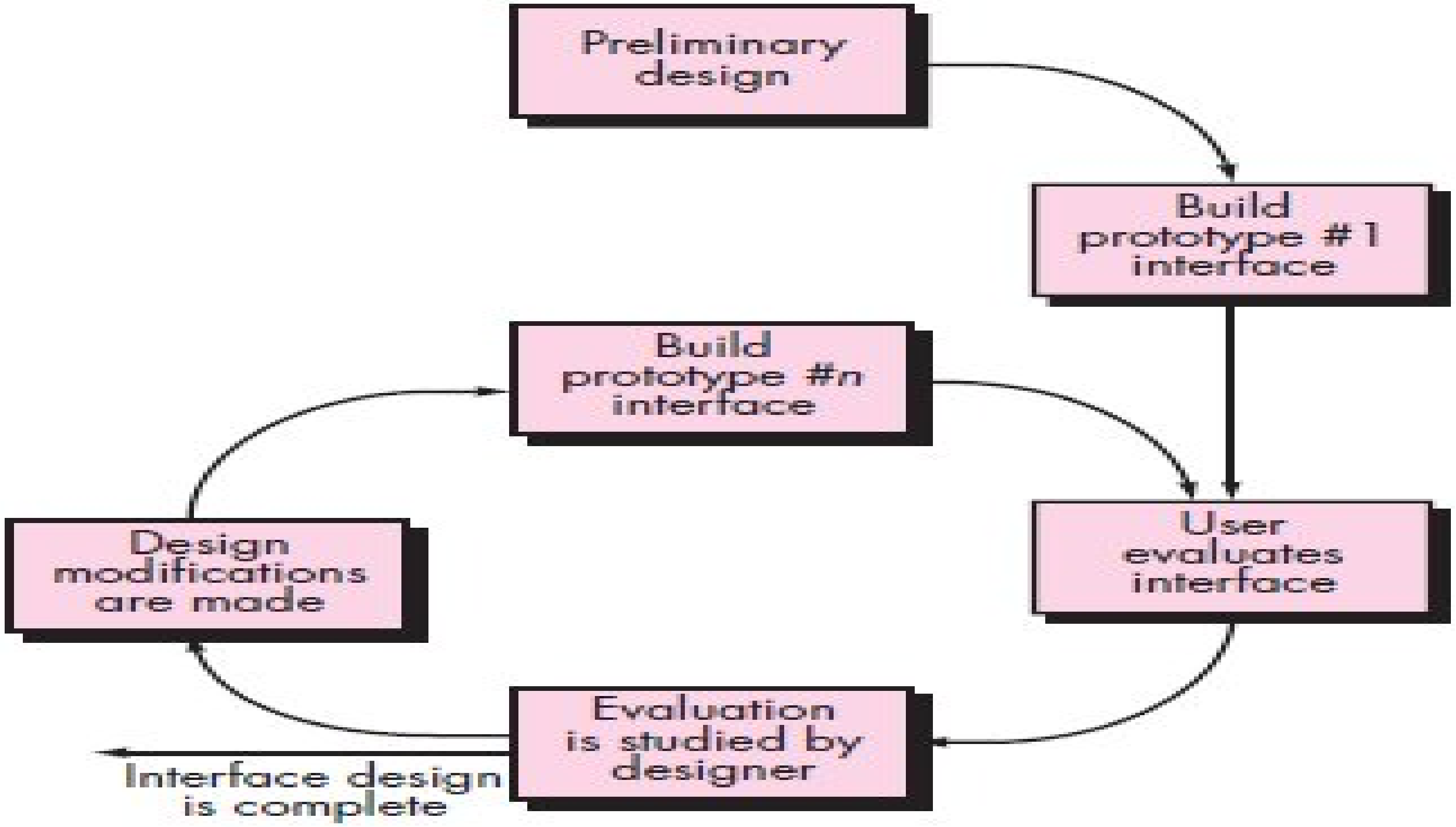
- Interface design must encompass mechanisms that enable easy access for those with special needs.
- *Accessibility* for users (and software engineers) who may be physically challenged is an imperative for ethical, legal, and business reasons.
- The interface should be designed to achieve varying levels of accessibility.
- Should address the needs of those with :
  - visual,
  - hearing,
  - mobility,
  - speech, and
  - learning impairments

Internationalization

- Software engineers and their managers invariably underestimate the effort and skills required to:
  - create user interfaces that accommodate the needs of different locales and languages.
  - Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries.

- The challenge for interface designers is to create “globalized” software.

Design Evaluation


Preliminary design
Build prototype #1 interface
Build prototype #n interface
User evaluates interface
Design modifications are made
Evaluation is studied by designer
Interface design is complete

Design Evaluation

- After the design model has been completed
  - a first-level prototype is created.
- The prototype is evaluated by the user,
  - who provides you with direct comments about the efficacy of the interface.
  - If formal evaluation techniques are used (e.g., questionnaires, rating sheets), you can extract information from these data.
- Design modifications are made based on user input.
- The next level prototype is created.
- The evaluation cycle continues:
  - until no further modifications to the interface design are necessary.

Design with Reuse

- Building software from reusable components.

Software Reuse

- In most engineering disciplines, systems are designed by composing existing components that have been used in other systems

- Software engineering has been more focused on original development but it is now recognised that to achieve better software, more quickly and at lower cost, we need to adopt a design process that is based on *systematic reuse*

Reuse-based software engineering

- **Application system reuse**
  - The whole of an application system may be reused either by incorporating it without change into other systems or by configuring the application for different customers.
  - application families that have a common architecture, but which are tailored for specific customers, may be developed.

Reuse-based software engineering

- **Component reuse**
  - Components of an application from sub-systems to single objects may be reused
- **Object and Function reuse**
  - Software components that implement a single function, such as a mathematical function, or an object class may be reused

Benefits of reuse

- Increased reliability
  - Components exercised in working systems
- Reduced process risk
  - Less uncertainty in development costs
- Effective use of specialists
  - Reuse components instead of people
- Standards compliance
  - Embed standards in reusable components
- Accelerated development
  - Avoid original development and hence speed-up production

Component-based software engineering

- *Component-based software engineering* (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software "components."
- [CBSE] embodies the "buy, don't build" philosophy.
- CBSE emphasizes reusability—that is, the creation and reuse of software building blocks.
- Such building blocks, often called *components*, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Component-based software engineering

Four software resource categories that should be considered for planning Reuse:

- *Off-the-shelf components.*
  - <u>Existing software that can be acquired from a third party or from a past project.</u>
  - COTS (commercial off-the-shelf) components are purchased from a third party, are <u>ready for use on the current project, and have been fully validated.</u>
- *Full-experience components.*
  - <u>Existing specifications, designs, code, or test data developed for past projects that are similar to the software</u> to be built for the current project.
  - <u>Members of the current software team have had full experience in the application area</u> represented by these components. Therefore, modifications required for full-experience components will be relatively low risk.

Component-based software engineering

- *Partial-experience components*.
  - Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification.
  - Members of the current software team have only limited experience in the application area represented by these components.
  - Therefore, modifications required for partial-experience components have a fair degree of risk.
- *New components*.
  - Software components must be built by the software team specifically for the needs of the current project.

Ironically, reusable software components are often neglected during planning, only to become a paramount concern later in the software process.

Component-based software engineering

“Never forget that integrating a variety of reusable components can be a significant challenge. Worse, the integration problem resurfaces as various components are upgraded.”

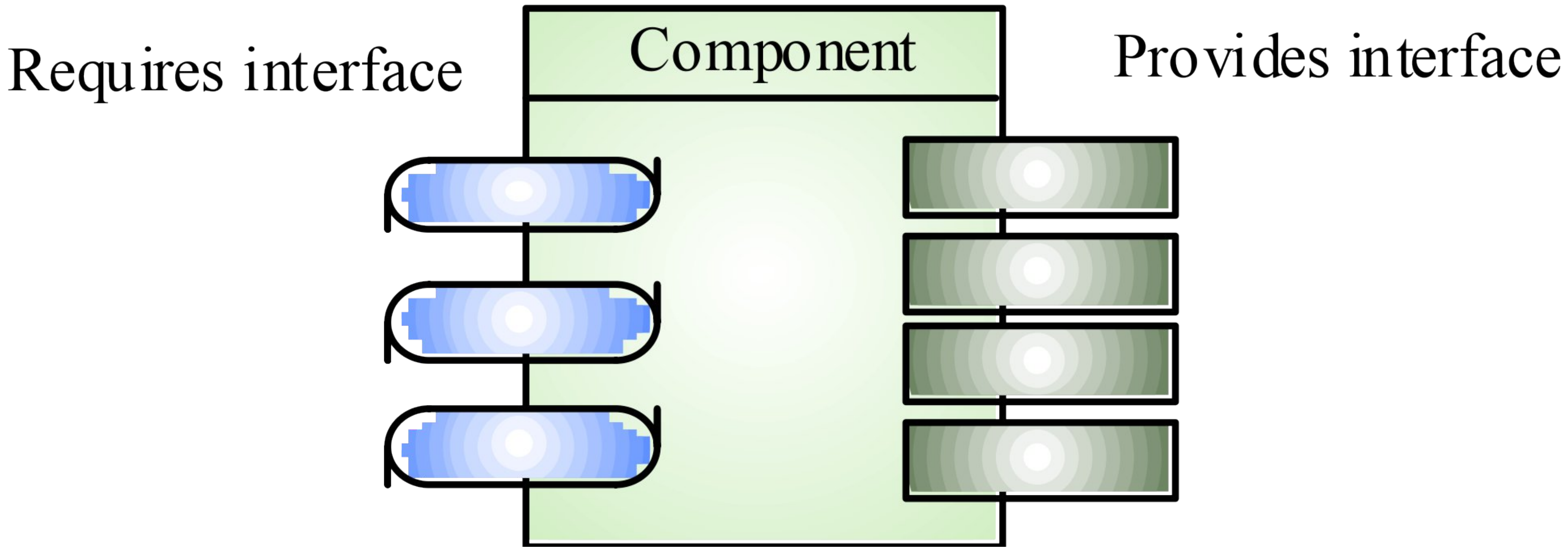
Component-based software engineering

- Component-based software engineering (CBSE) is an **approach to software development that relies on reuse**
- It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific
- **Components are more abstract than object classes** and can be considered to be stand-alone service providers

Component-based software engineering

- Components can range in size from simple functions to entire application systems

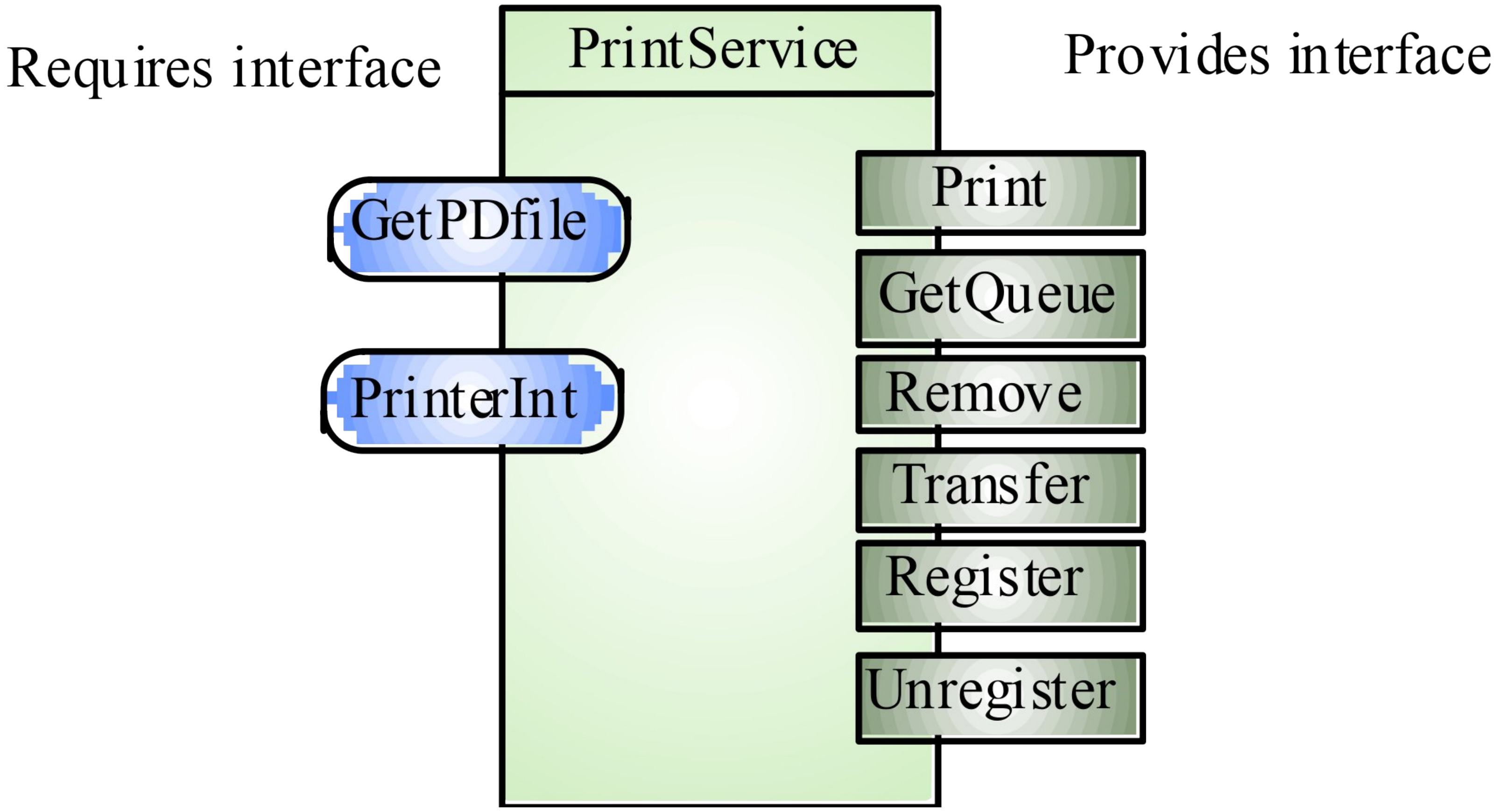
Component interfaces



Component Interfaces

- These interfaces reflect the services that the component provides and the services that the component requires to operate correctly
- **Provides interface**
  - Defines the services that are provided by the component to other components
- **Requires interface**
  - Defines the services that specifies what services must be made available for the component to execute as specified

Example: Printing services component



Component abstractions

- *Functional abstraction*
  - The component implements a single function such as a mathematical function
- *Casual groupings*
  - The component is a collection of loosely related entities that might be data declarations, functions, etc.
- *Data abstractions*
  - The component represents a data abstraction or class in an object-oriented language
- *Cluster abstractions*
  - The component is a group of related classes that work together
- *System abstraction*
  - The component is an entire self-contained system

CBSE processes

- CBSE processes are software processes that support component-based software engineering.

- They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components.

CBSE processes

- There are two types:

1. **Development for reuse**

- This process is concerned with **developing components or services that will be reused** in other applications.

- It usually involves generalizing existing components.

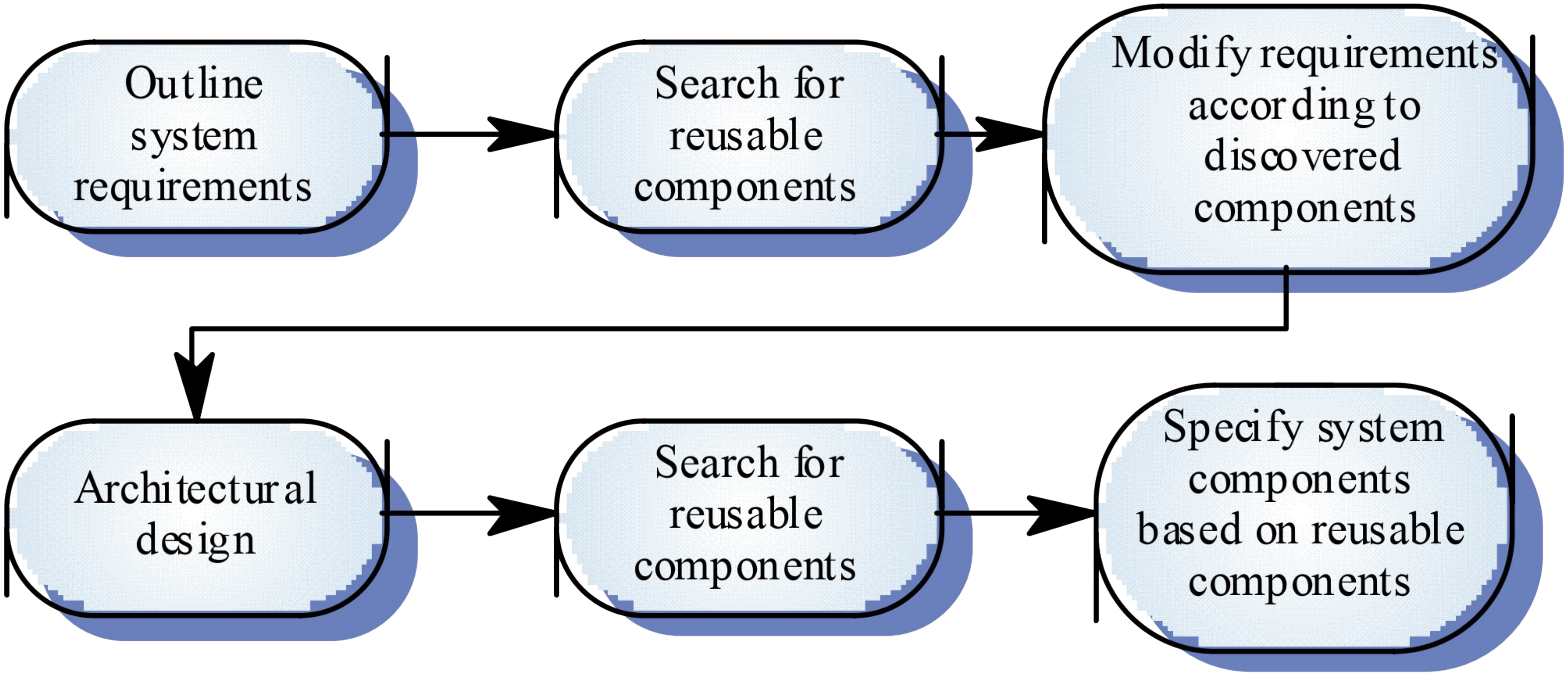
2. Development with reuse

- This is the process of **developing new applications using existing components and services**.

CBSE processes - **Development for reuse**

- CBSE for reuse is the <u>process of developing reusable components and making them available for reuse</u> through a component management system.

CBSE processes - Development with reuse


Outline system requirements
Search for reusable components
Modify requirements according to discovered components
Architectural design
Search for reusable components
Specify system components based on reusable components