

## 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 focused on original development but it is now recognized 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

3

- **Application system reuse**
  - The whole of an application system may be reused either by incorporating it without change into other systems. COTS (Commercial Off The Shelf)
- **Component reuse**
  - Components of an application from sub-systems to single objects may be reused
- **Function reuse**
  - Software components that implement a single well-defined function may be reused

## Reuse practice

4

- **Application system reuse**
  - Widely practiced as software systems are implemented as application families. COTS reuse is becoming increasingly common
- **Component reuse**
  - Now seen as the key to effective and widespread reuse through component-based software engineering. However, it is still relatively immature
- **Function reuse**
  - Common in some application domains (e.g. engineering) where domain-specific libraries of reusable functions have been established

## 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

## Requirements for design with reuse

- It must be possible to find appropriate reusable components
- The reuser of the component must be confident that the components will be reliable and will behave as specified
- The components must be documented so that they can be understood and, where appropriate, modified

## Reuse problems

- Lack of tool support
- Not-invented-here syndrome
- Maintaining a component library
- Finding and adapting reusable components

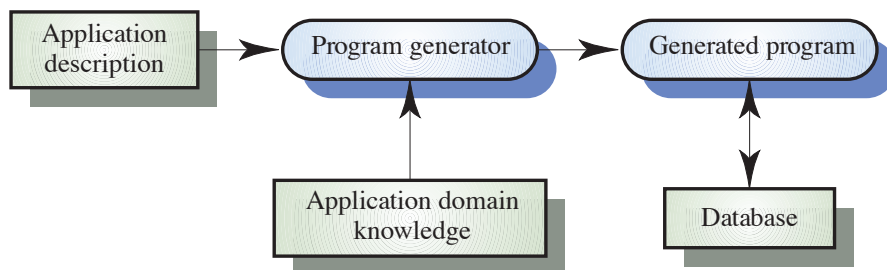
## Generator-based reuse

- Program generators involve the reuse of standard patterns and algorithms
- These are embedded in the generator and parameterized by user commands. A program is then automatically generated
- Generator-based reuse is possible when domain abstractions and their mapping to executable code can be identified
- A domain specific language is used to compose and control these abstractions

## Types of program generator

- **Types of program generator**
  - Application generators for business data processing
  - Parser and lexical analyser generators for language processing
  - Code generators in CASE tools
- **Generator-based reuse is very cost-effective but its applicability is limited to a relatively small number of application domains**
- **It is easier for end-users to develop programs using generators compared to other component-based approaches to reuse**

## Reuse through program generation



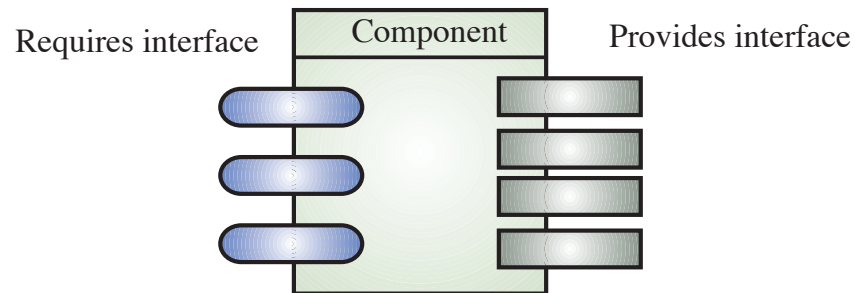
## Component-based development

- 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

## Components

- Components provide a service without regard to where the component is executing or its programming language
  - A component is an independent executable entity that can be made up of one or more executable objects
  - The component interface is published and all interactions are through the published interface
- Components can range in size from simple functions to entire application systems

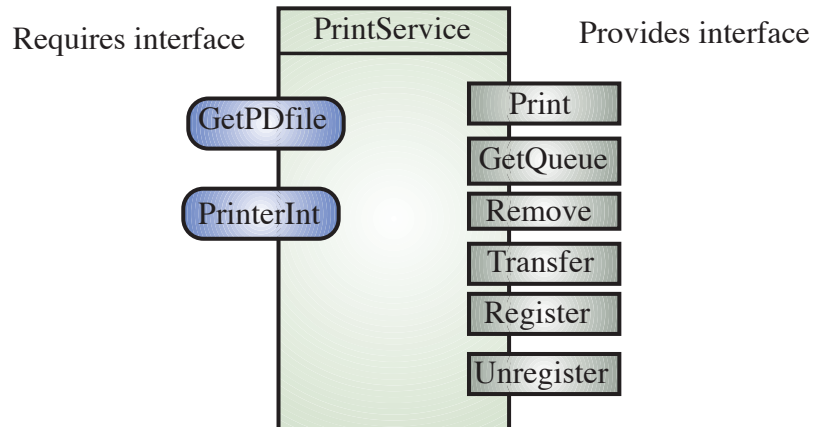
## Component interfaces



## Component interfaces

- **Provides interface**
  - Defines the services that are provided by the component to other components
- **Requires interface**
  - Specifies what services must be made available for the component to execute

## Printing services component



## Component abstractions

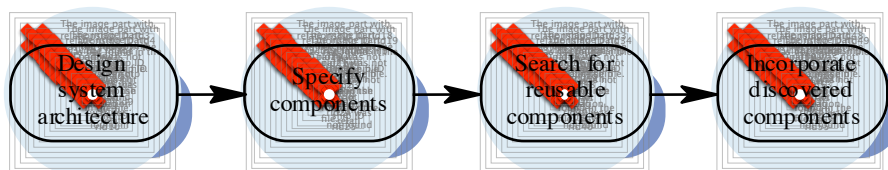
- **Functional abstraction**
  - The component implements a single function such as a mathematical function
- **Casual groupings**
  - The component is a collection of loosely related entities that might be data declarations, functions, etc.
- **Data abstractions**
  - The component represents a data 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



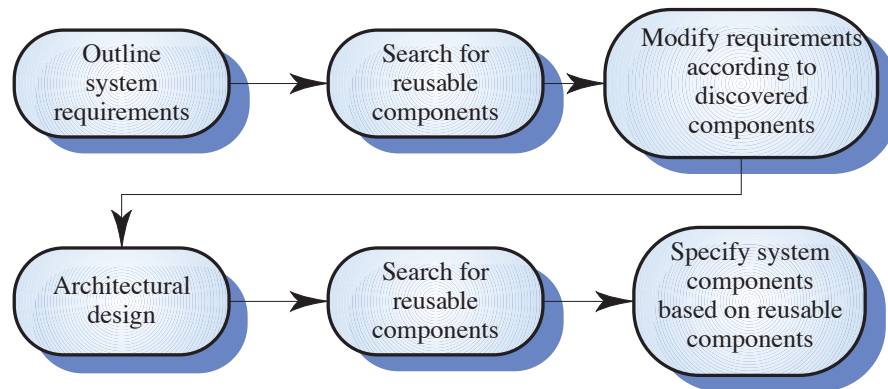
## CBSE processes

- Component-based development can be integrated into a standard software process by incorporating a reuse activity in the process
- However, in reuse-driven development, the system requirements are modified to reflect the components that are available
- CBSE usually involves a prototyping or an incremental development process with components being 'glued together' using a scripting language

## An opportunistic reuse process



## Development with reuse



## CBSE problems

- Component incompatibilities may mean that cost and schedule savings are less than expected
- Finding and understanding components
- Managing evolution as requirements change in situations where it may be impossible to change the system components

## COTS product reuse

- COTS - Commercial Off-The-Shelf systems
- COTS systems are usually complete application systems that offer an API (Application Programming Interface)
- Building large systems by integrating COTS systems is now a viable development strategy for some types of system such as E-commerce systems

## COTS system integration problems

- Lack of control over functionality and performance
  - COTS systems may be less effective than they appear
- Problems with COTS system inter-operability
  - Different COTS systems may make different assumptions that means integration is difficult
- No control over system evolution
  - COTS vendors not system users control evolution
- Support from COTS vendors
  - COTS vendors may not offer support over the lifetime of the product

## Component development for reuse

- Components for reuse may be specially constructed by generalizing existing components
- Component reusability
  - Should reflect stable domain abstractions
  - Should hide state representation
  - Should be as independent as possible
  - Should publish exceptions through the component interface
- There is a trade-off between reusability and usability.
  - The more general the interface, the greater the reusability but it is then more complex and hence less usable

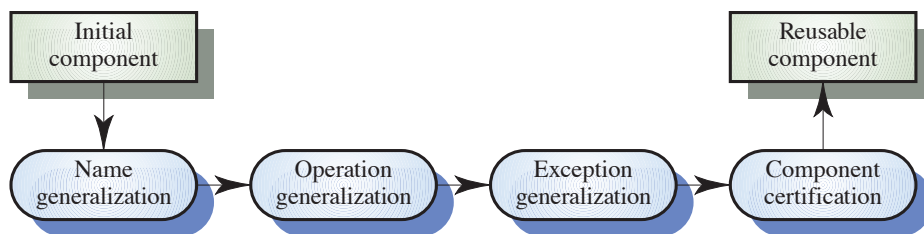
## Reusable components

- The development cost of reusable components is higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents

## Reusability enhancement

- **Name generalization**
  - Names in a component may be modified so that they are not a direct reflection of a specific application entity
- **Operation generalization**
  - Operations may be added to provide extra functionality and application specific operations may be removed
- **Exception generalization**
  - Application specific exceptions are removed and exception management added to increase the robustness of the component
- **Component certification**
  - Component is certified as reusable

## Reusability enhancement process



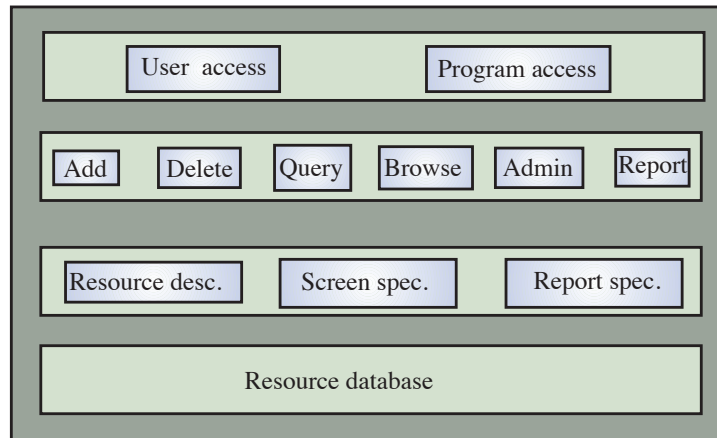
## Application families

- An application family or product line is a related set of applications that has a common, domain-specific architecture
- The common core of the application family is reused each time a new application is required
- Each specific application is specialized in some way

## Application family specialization

- **Platform specialization**
  - Different versions of the application are developed for different platforms
- **Configuration specialization**
  - Different versions of the application are created to handle different peripheral devices
- **Functional specialization**
  - Different versions of the application are created for customers with different requirements

## A resource management system



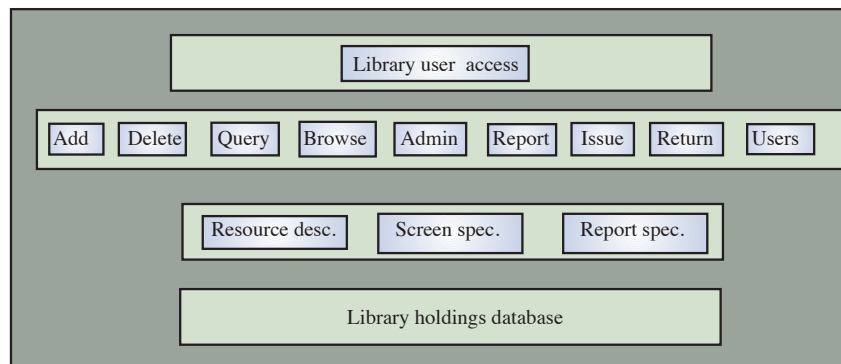
## Inventory management systems

- **Resource database**
  - Maintains details of the things that are being managed
- **I/O descriptions**
  - Describes the structures in the resource database and input and output formats that are used
- **Query level**
  - Provides functions implementing queries over the resources
- **Access interfaces**
  - A user interface and an application programming interface

## Application family architectures

- Architectures must be structured in such a way to separate different sub-systems and to allow them to be modified
- The architecture should also separate entities and their descriptions and the higher levels in the system access entities through descriptions rather than directly

## A library system





## Testing Issues

---

- **Components**
  - Code may not be available
- **Unit test the component**
  - What does it mean to test a component
- **Integration testing**
  - In the context