Design with Reuse

- Building software from reusable components.

Objectives

- To explain the benefits of software reuse and some reuse problems
- To describe different types of reusable component and processes for reuse
- To introduce application families as a route to reuse
- To describe design patterns as high-level abstractions that promote reuse

Topics covered

- Component-based development
- Application families
- Design patterns

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 (COTS reuse) or by developing application families
- 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

- Application system reuse
  - Widely practised 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

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

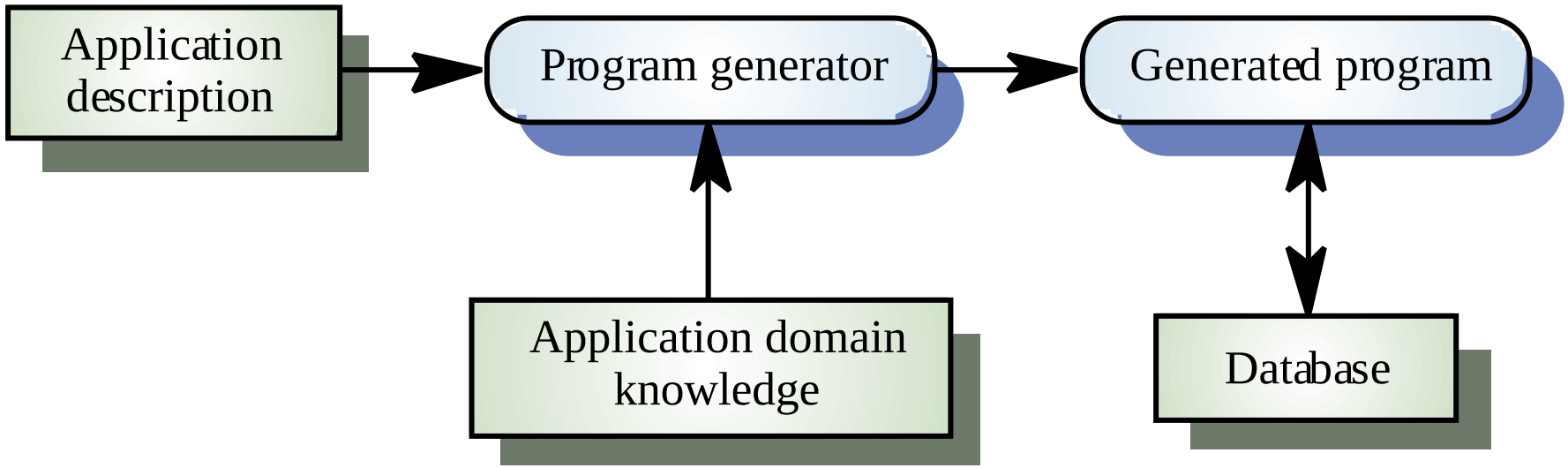
Generator-based reuse

- Program generators involve the reuse of standard patterns and algorithms
- These are embedded in the generator and parameterised 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



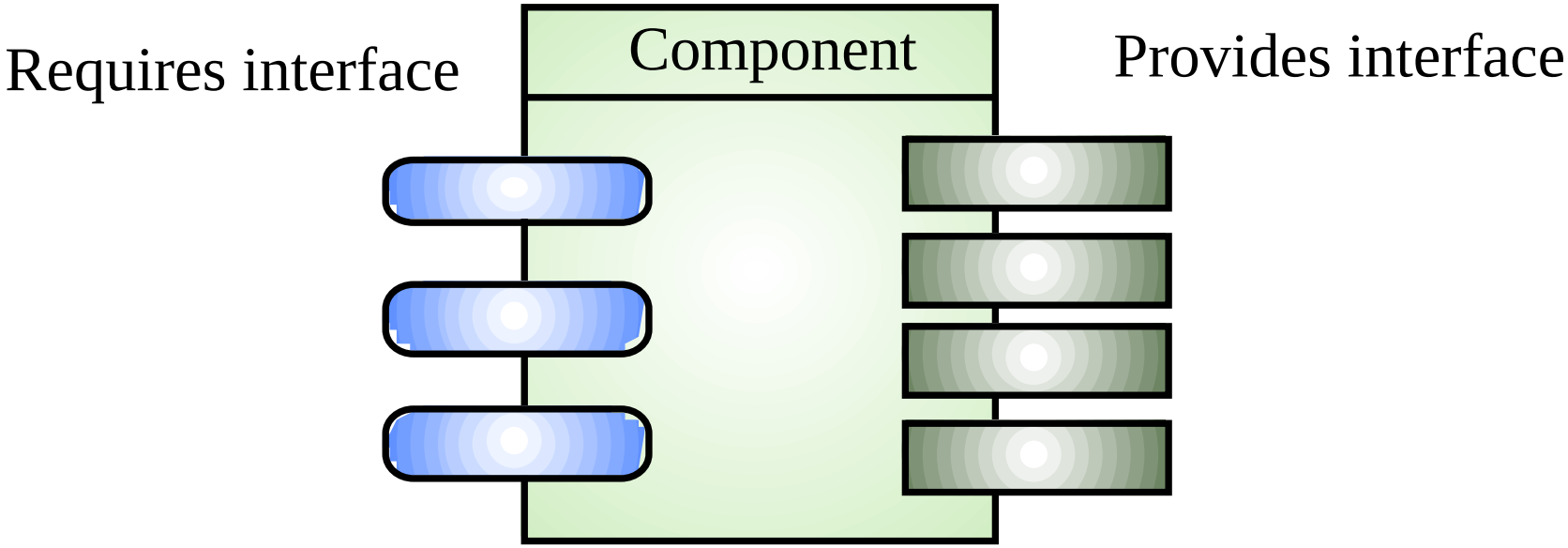
Component-based development

- 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

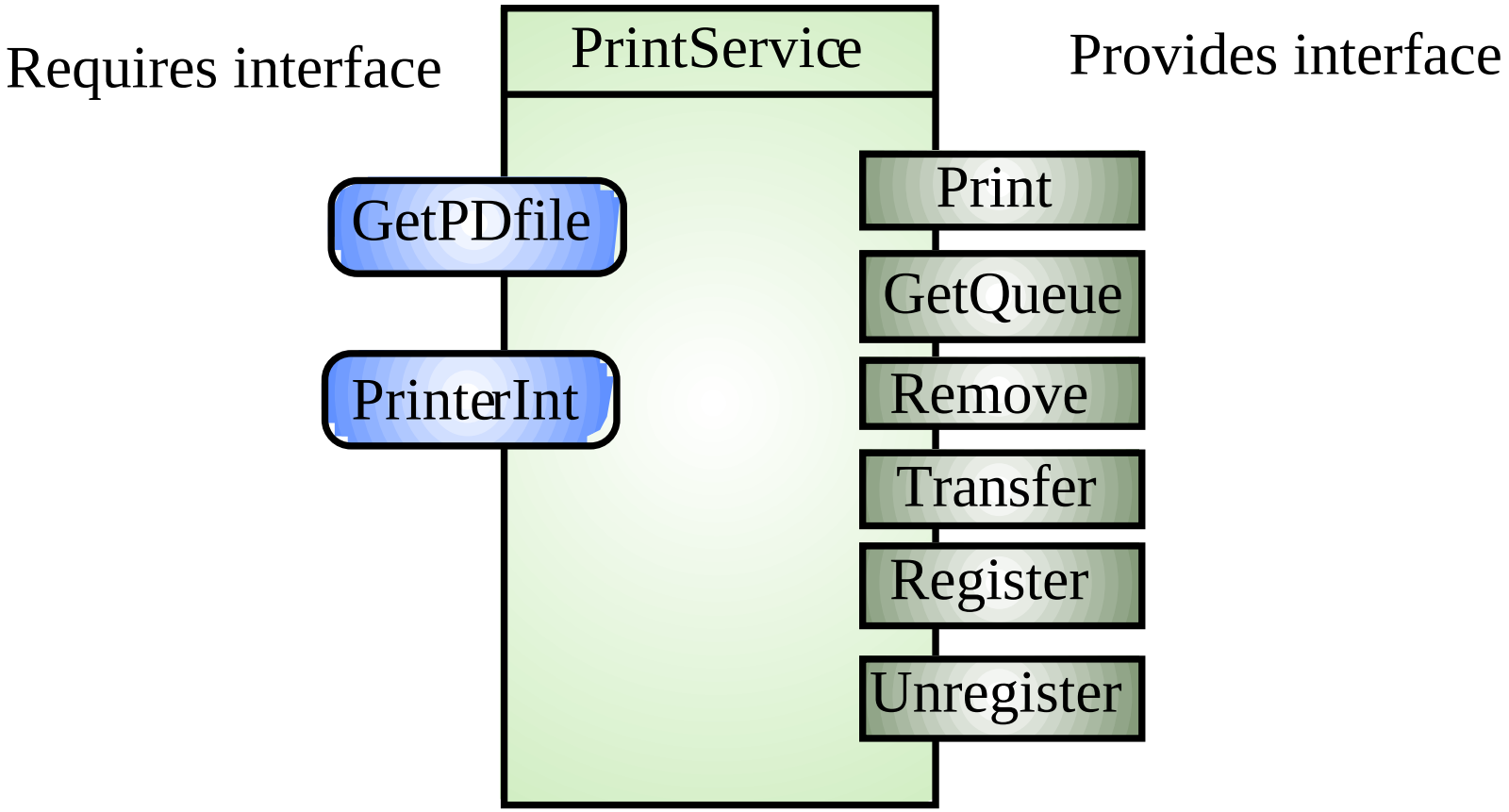
Component interfaces



Component interfaces

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

Printing services component



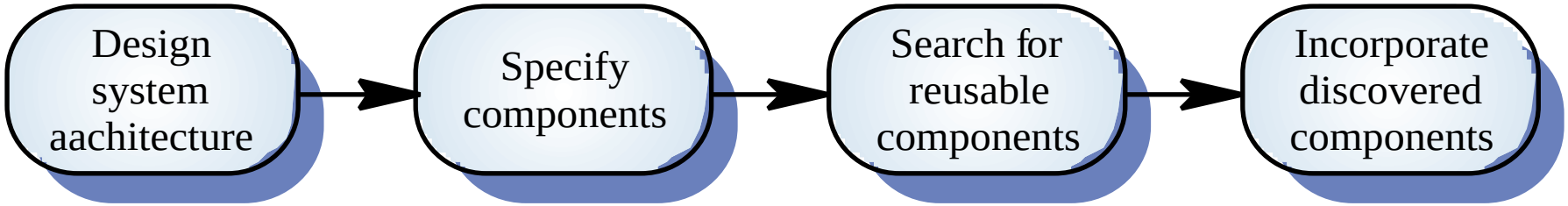
Component abstractions

- *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

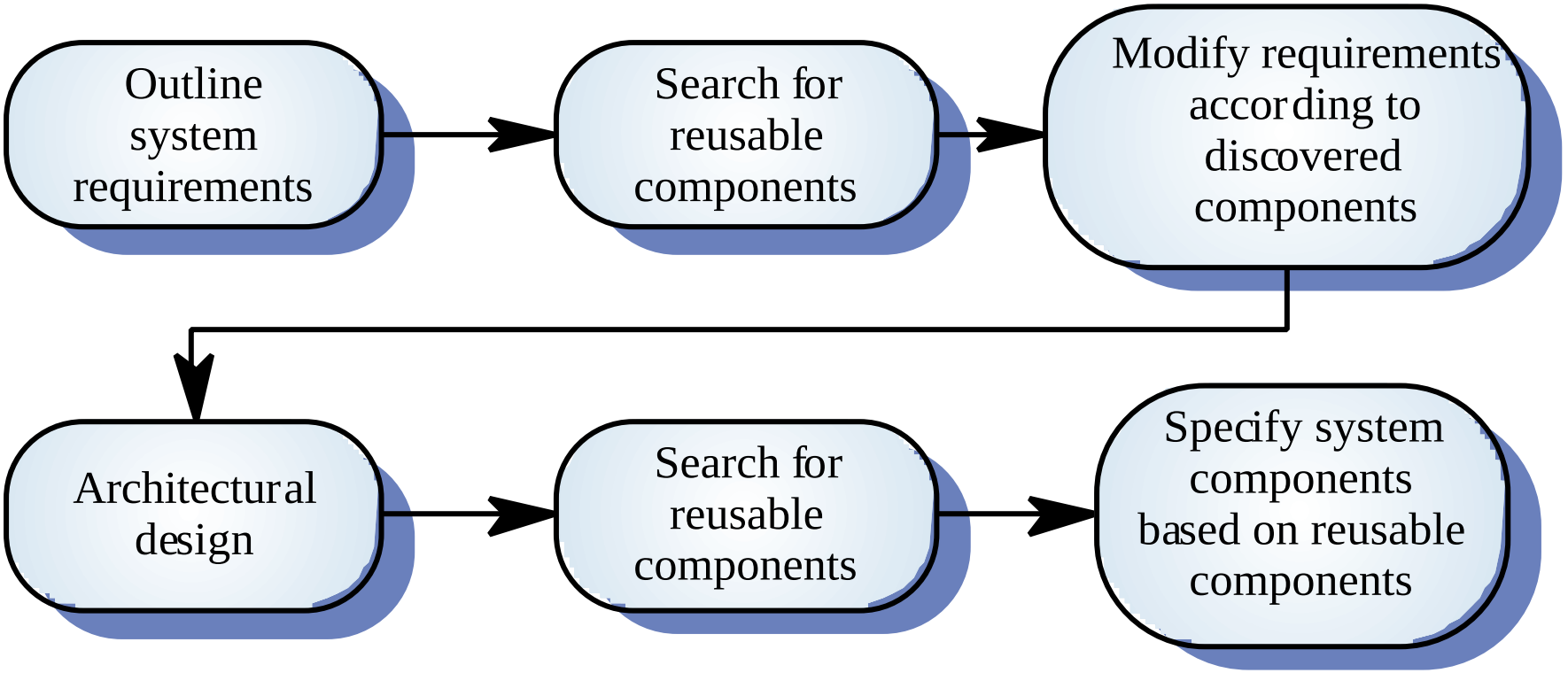
CBSE processes

- 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 then expected
- Finding and understanding components
- Managing evolution as requirements change in situations where it may be impossible to change the system components

Application frameworks

- Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them
- The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework
- Frameworks are moderately large entities that can be reused

Framework classes

- System infrastructure frameworks
  - Support the development of system infrastructures such as communications, user interfaces and compilers
- Middleware integration frameworks
  - Standards and classes that support component communication and information exchange
- Enterprise application frameworks
  - Support the development of specific types of application such as telecommunications or financial systems

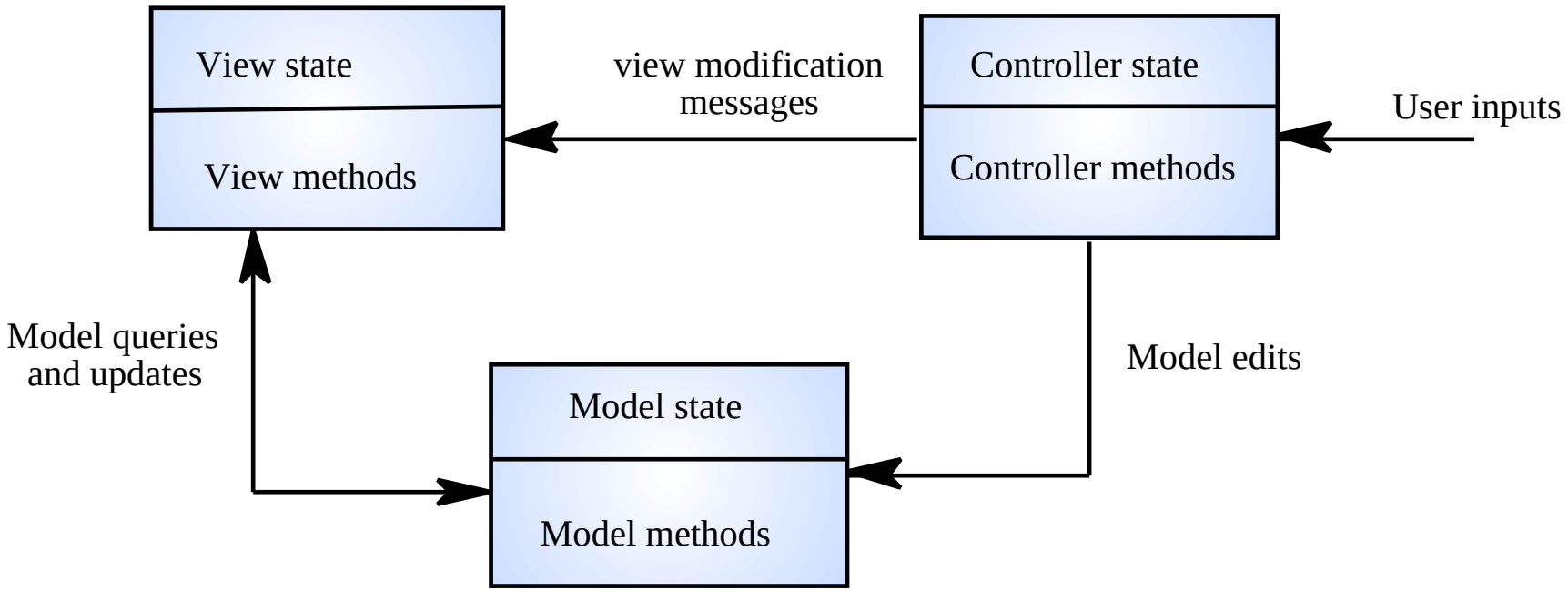
Extending frameworks

- Frameworks are generic and are extended to create a more specific application or sub-system
- Extending the framework involves
  - Adding concrete classes that inherit operations from abstract classes in the framework
  - Adding methods that are called in response to events that are recognised by the framework
- Problem with frameworks is their complexity and the time it takes to use them effectively

Model-view controller

- System infrastructure framework for GUI design
- Allows for multiple presentations of an object and separate interactions with these presentations
- MVC framework involves the instantiation of a number of patterns (discussed later)

Model-view controller



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 generalising 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

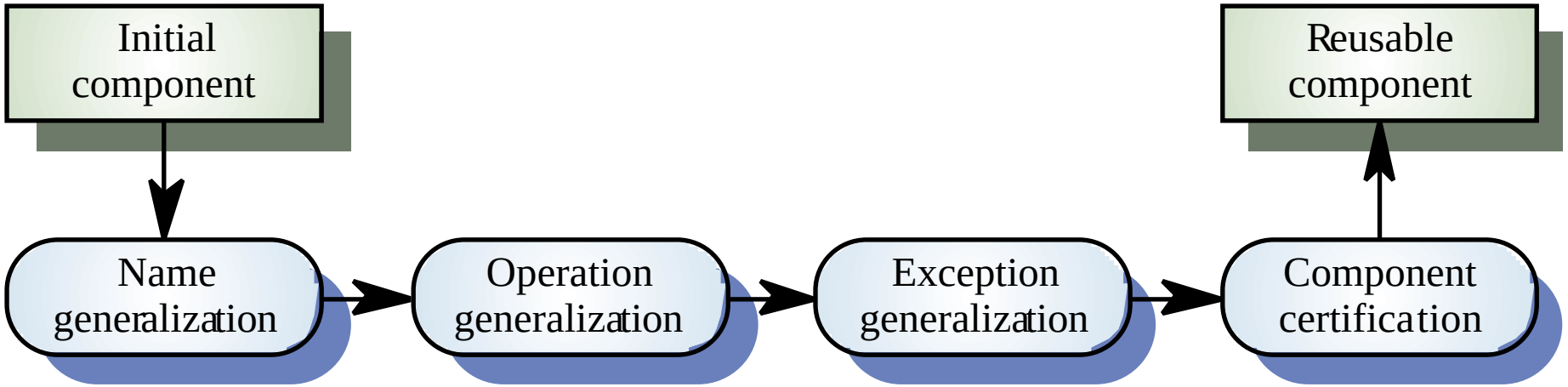
Reusable components

- 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 generalisation
  - Names in a component may be modified so that they are not a direct reflection of a specific application entity
- Operation generalisation
  - Operations may be added to provide extra functionality and application specific operations may be removed
- Exception generalisation
  - 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



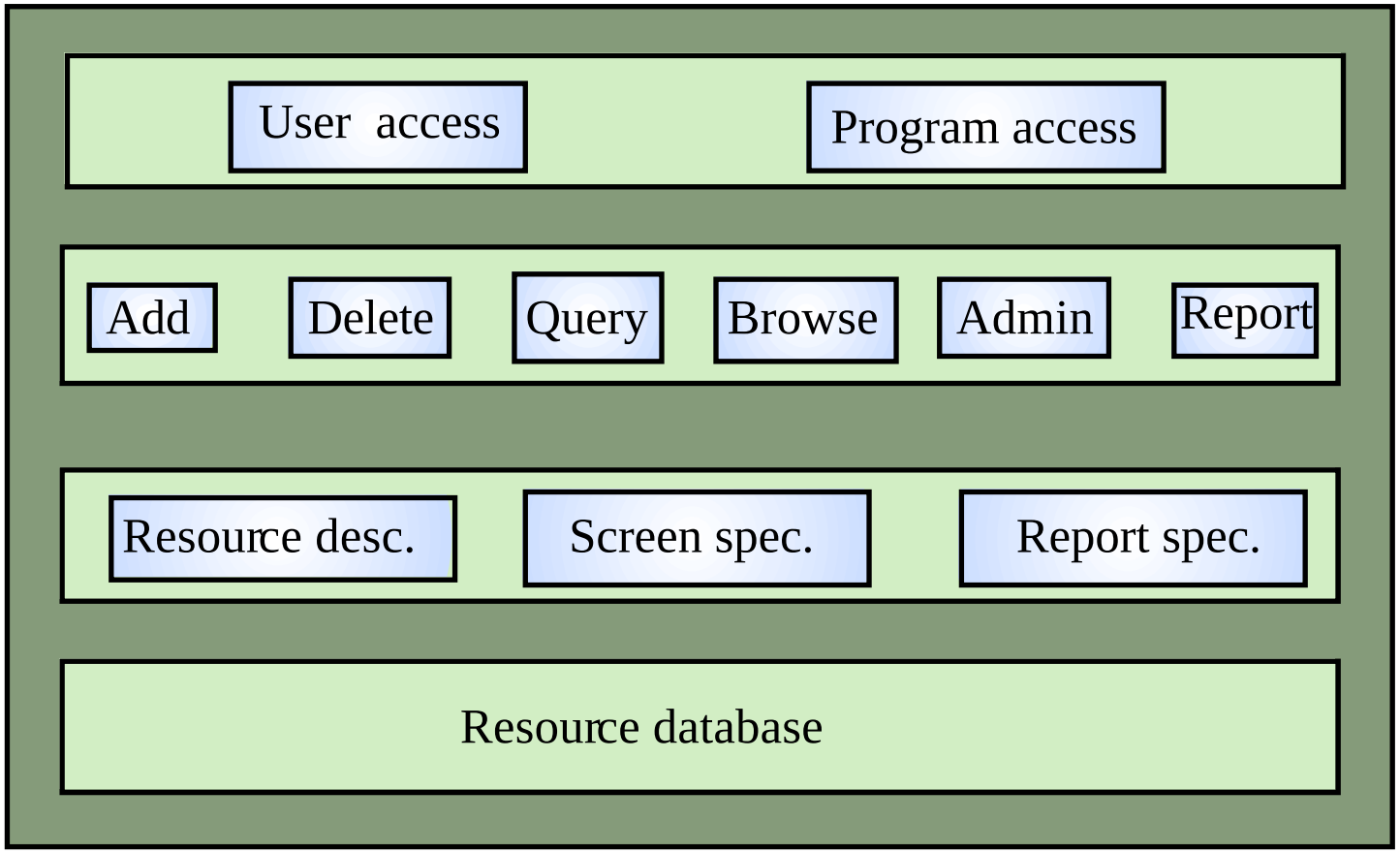
Application families

- 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 specialised in some way

Application family specialisation

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

A resource management system



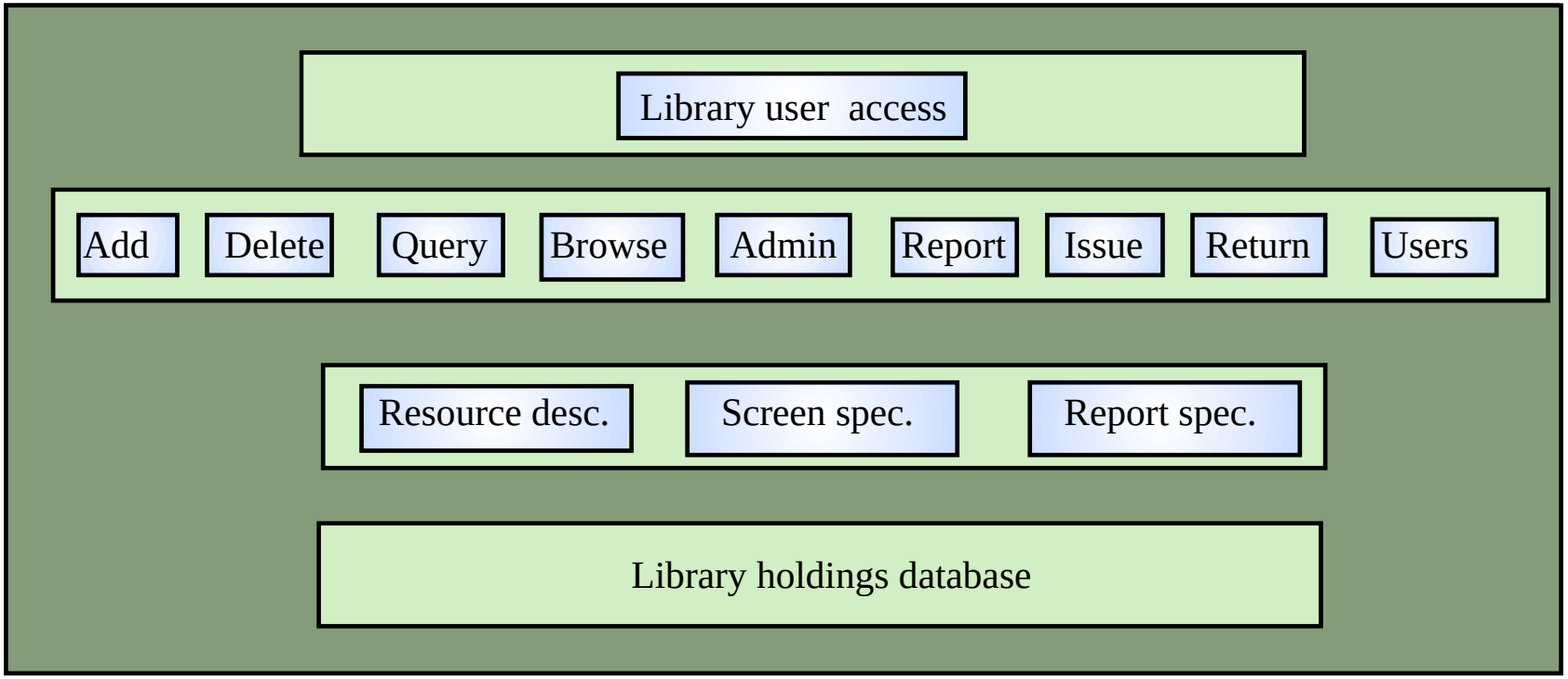
Inventory management systems

- 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

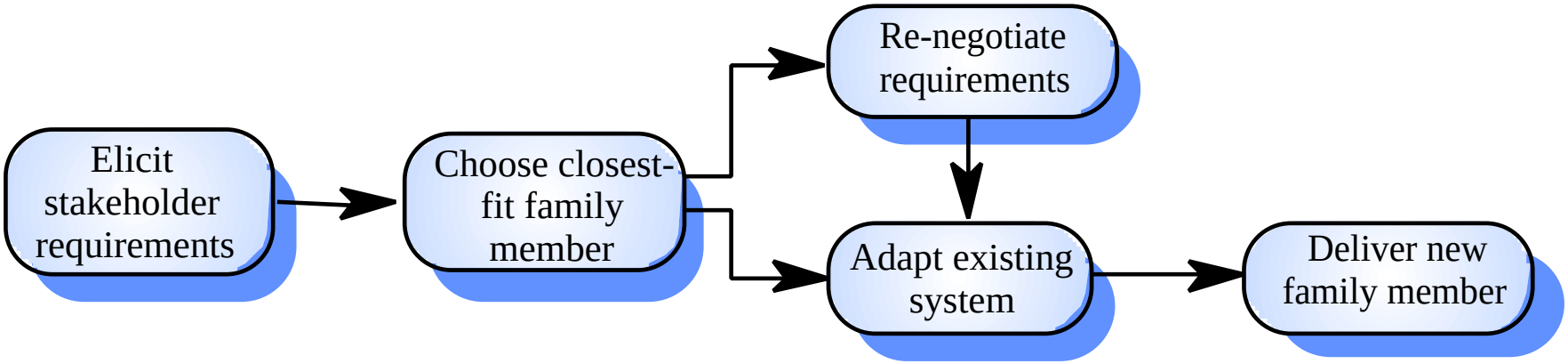
A library system



Library system

- The resources being managed are the books in the library
- Additional domain-specific functionality (issue, borrow, etc.) must be added for this application

Family member development



Family member development

- Elicit stakeholder requirements
  - Use existing family member as a prototype
- Choose closest-fit family member
  - Find the family member that best meets the requirements
- Re-negotiate requirements
  - Adapt requirements as necessary to capabilities of the software
- Adapt existing system
  - Develop new modules and make changes for family member
- Deliver new family member
  - Document key features for further member development

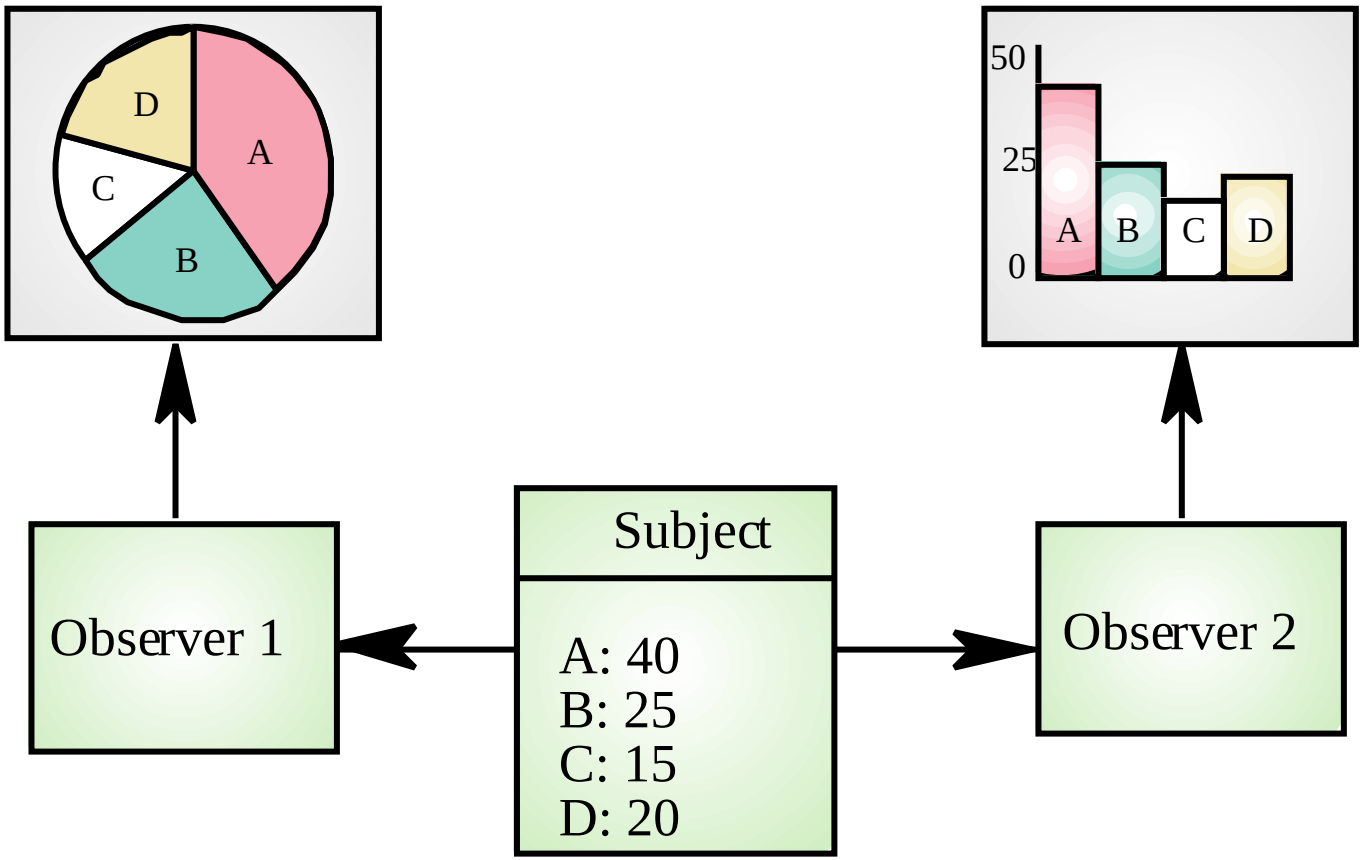
Design patterns

- A design pattern is a way of reusing abstract knowledge about a problem and its solution
- A pattern is a description of the problem and the essence of its solution
- It should be sufficiently abstract to be reused in different settings
- Patterns often rely on object characteristics such as inheritance and polymorphism

Pattern elements

- Name
  - A meaningful pattern identifier
- Problem description
- Solution description
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- Consequences
  - The results and trade-offs of applying the pattern

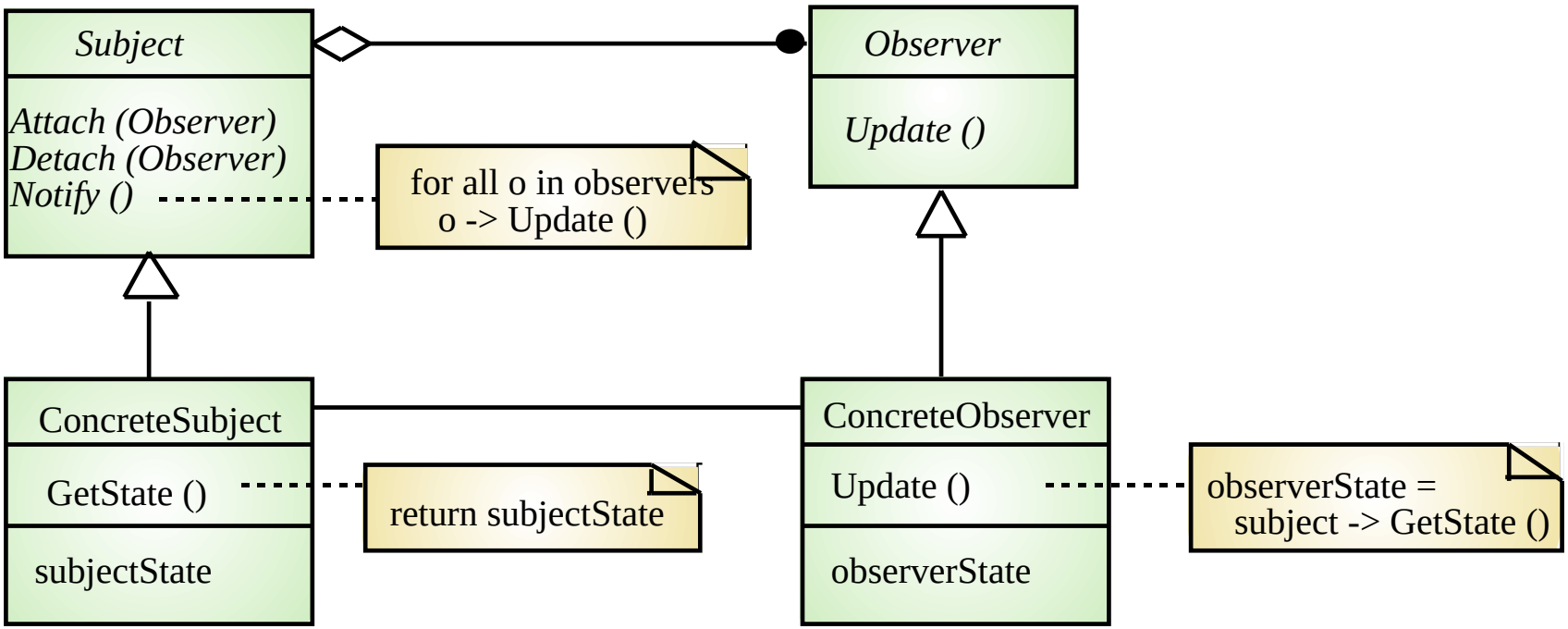
Multiple displays



The Observer pattern

- Name
  - Observer
- Description
  - Separates the display of object state from the object itself
- Problem description
  - Used when multiple displays of state are needed
- Solution description
  - See slide with UML description
- Consequences
  - Optimisations to enhance display performance are impractical

The Observer pattern



Key points

- Design with reuse involves designing software around good design and existing components
- Advantages are lower costs, faster software development and lower risks
- Component-based software engineering relies on black-box components with defined requires and provides interfaces
- COTS product reuse is concerned with the reuse of large, off-the-shelf systems

Key points

- Software components for reuse should be independent, should reflect stable domain abstractions and should provide access to state through interface operations
- Application families are related applications developed around a common core
- Design patterns are high-level abstractions that document successful design solutions