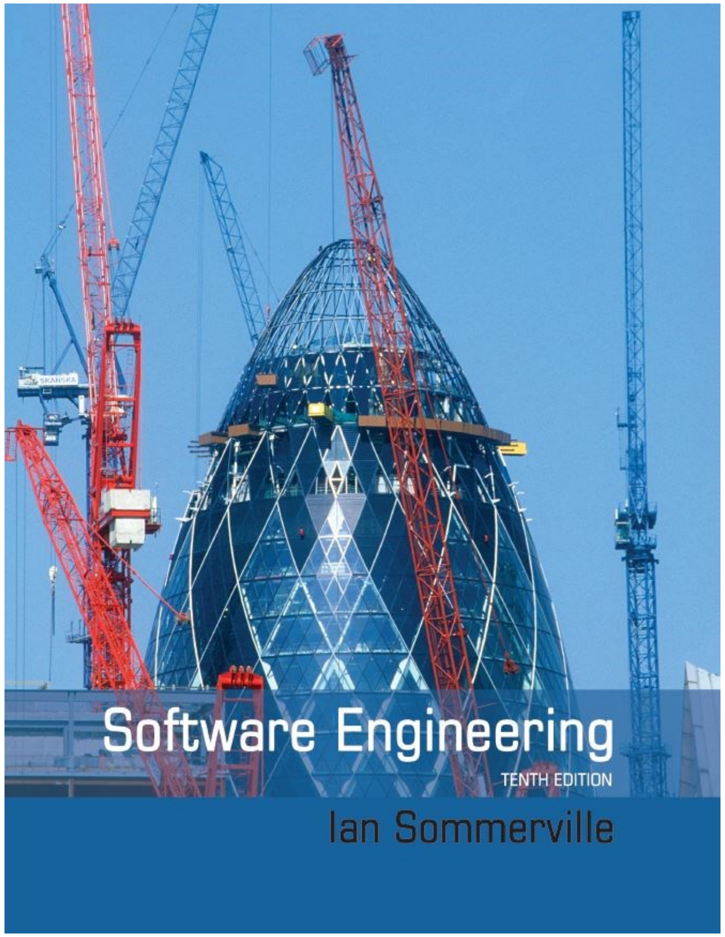
Software Engineering

Tenth Edition



Chapter 16

Component-based Software Engineering

Learning Objectives

16.1 Components and component models

16.2 CBSE processes

16.3 Component composition

Component-based development

- Component-based software engineering (CBSE) is an approach to software development that relies on the reuse of entities called ‘software components’.
- 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. They can exist as stand-alone entities.

Component-based development

- Component-based software engineering is the process of defining, implementing, and integrating or composing these loosely coupled, independent components into systems.
- CBSE has become as an important software development approach for large-scale enterprise systems, with demanding performance and security requirements.
- The only way that these demands can be met is to build software by reusing existing components.

CBSE Essentials

- Independent components specified by their interfaces.
  - There should be a clear separation between the component interface and its implementation.
- Component standards to facilitate component integration.
  - These standards are embodied in a component model.
  - If components conform to standards, then their operation is independent of their programming language.
- Middleware that provides support for component inter-operability.
  - In addition, middleware may provide support for resource allocation, transaction management, security, and concurrency.
- A development process that is geared to reuse.

CBSE and Design Principles

- Apart from the benefits of reuse, CBSE is based on sound software engineering design principles:
  - Components are independent so do not interfere with each other;
  - Component implementations are hidden;
  - Communication is through well-defined interfaces;
  - One component can be replaced by another if its interface is maintained;
  - Component infrastructures offer a range of standard services.

Component Standards

- Standards need to be established so that components can communicate with each other and inter-operate.
- Unfortunately, several competing component standards were established:
  - Sun’s Enterprise Java Beans
  - Microsoft’s COM and .NET
  - CORBA’s CCM
- In practice, these multiple standards have hindered the uptake of CBSE. It is impossible for components developed using different approaches to work together.

Service-oriented Software Engineering

- An executable service is a type of independent component. It has a ‘provides’ interface but not a ‘requires’ interface.
- From the outset, services have been based around standards so there are no problems in communicating between services offered by different vendors.
- System performance may be slower with services but this approach is replacing CBSE in many systems.

Components and Component Models

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;

Component Definitions

- Councill and Heinmann:
  - A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.
- Szyperski:
  - A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.

Component Characteristics (1 of 2)

Component characteristic	**Description**
Composable	For a component to be composable, all external interactions must take place through publicly defined interfaces. In addition, it must provide external access to information about itself, such as its methods and attributes.
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have to be compiled before it is deployed. If a component is implemented as a service, it does not have to be deployed by a user of a component. Rather, it is deployed by the service provider.

Component Characteristics (2 of 2)

Component characteristic	**Description**
Documented	Components have to be fully documented so that potential users can decide whether or not the components meet their needs. The syntax and, ideally, the semantics of all component interfaces should be specified.
Independent	A component should be independent—it should be possible to compose and deploy it without having to use other specific components. In situations where the component needs externally provided services, these should be explicitly set out in a ‘requires’ interface specification.
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.

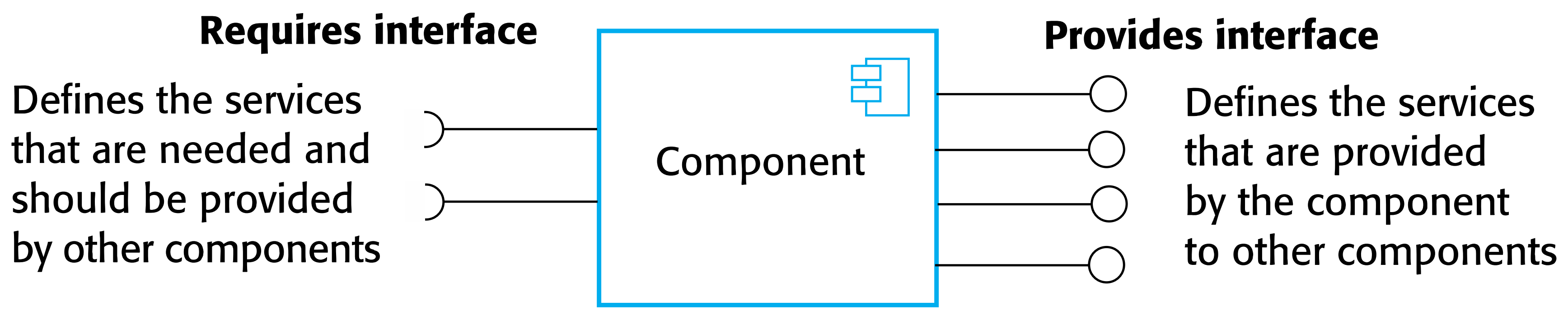
Component as a Service Provider

- The component is an independent executable entity that is defined by its interfaces.
  - You don't need any knowledge of its source code to use it. It can either be referenced as an external service or included directly in a program.
- The services offered by a component are made available through an interface, and all interactions are through that interface.
  - The component interface is expressed in terms of parameterized operations, and its internal state is never exposed.

Component Interfaces

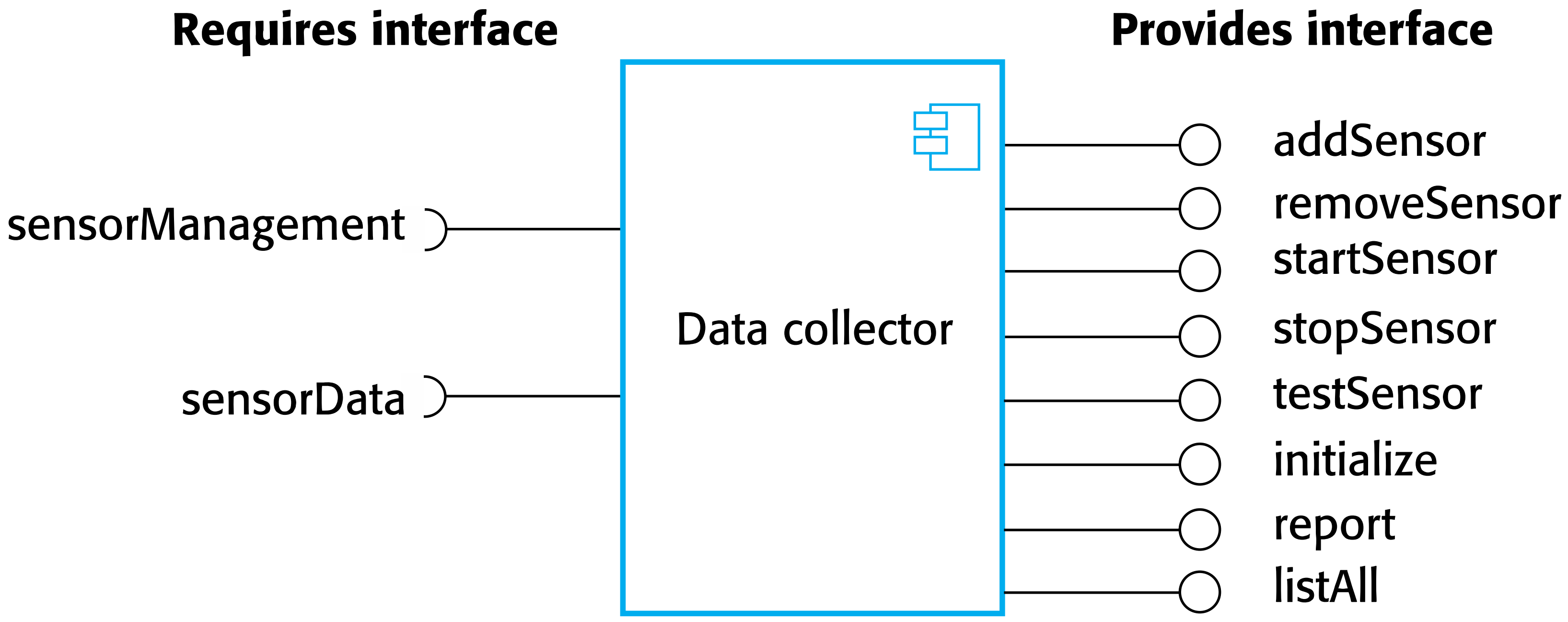
- Provides interface
  - Defines the services that are provided by the component to other components.
  - This interface, essentially, is the component API. It defines the methods that can be called by a user of the component.
- Requires interface
  - Defines the services that specifies what services must be made available for the component to execute as specified.
  - This does not compromise the independence or deployability of a component because the ‘requires’ interface does not define how these services should be provided.

Component Interfaces


Requires interface
Defines the services that are needed and should be provided by other components
Component
Provides interface
Defines the services that are provided by the component to other components

Note UML notation. Ball and sockets can fit together.

A Model of a Data Collector Component



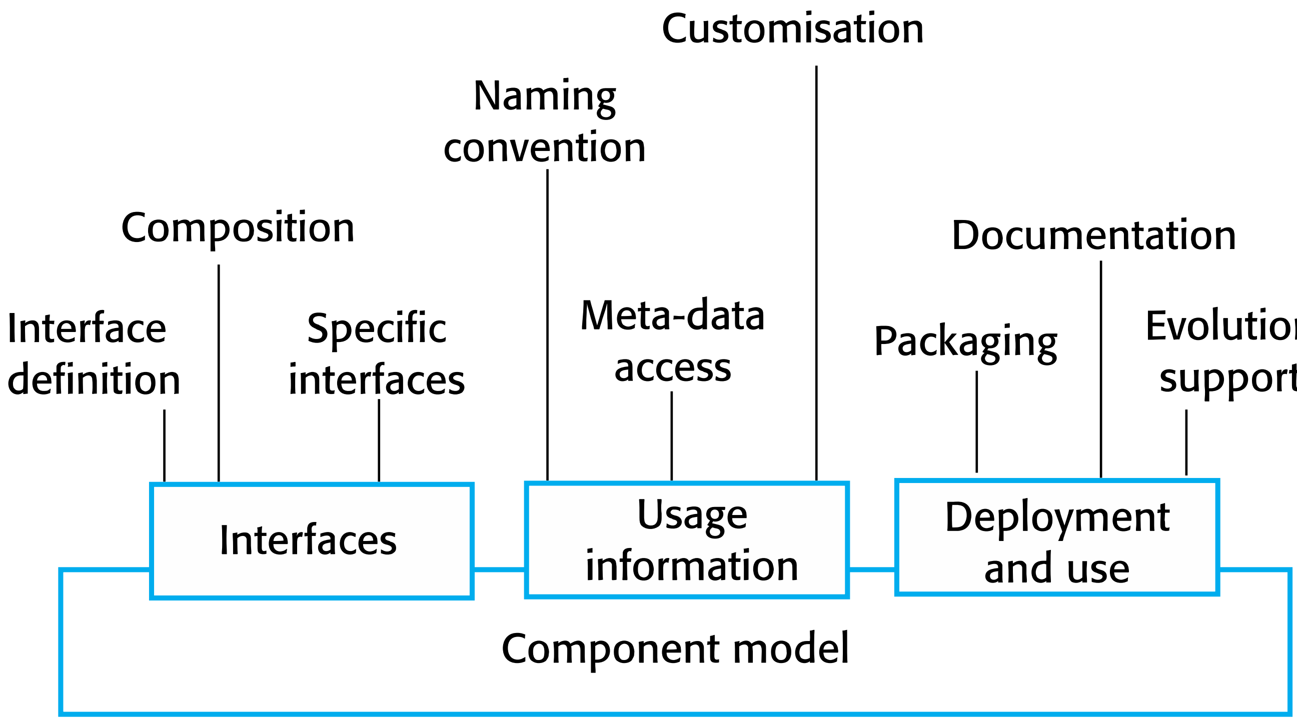
Component Access

- Components are accessed using remote procedure calls (RPCs).
- Each component has a unique identifier (usually a URL) and can be referenced from any networked computer.
  - The called component uses the same mechanism to access the “required” components that are defined in its interface.
- An important difference between a component as an external service and a component as a program element accessed using a remote procedure call is that services are completely independent entities. They do not have an explicit “requires” interface.

Component Models

- A component model is a definition of standards for component implementation, documentation and deployment.
- Examples of component models
  - EJB model (Enterprise Java Beans)
  - COM+ model (.NET model)
  - Corba Component Model
- The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.

Basic Elements of a Component Model



Interfaces

- Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters, and exceptions, which should be included in the interface definition. The model should also specify the language used to define the component interfaces.
- Some component models require specific interfaces that must be defined by a component. These are used to compose the component with the component model infrastructure, which provides standardized services such as security and transaction management.

Usage

- In order for components to be distributed and accessed remotely via RPCs, they need to have a unique name or handle associated with them. This has to be globally unique.
- Component meta-data is data about the component itself, such as information about its interfaces and attributes.
- Components are generic entities, and, when deployed, they have to be configured to fit into an application system.

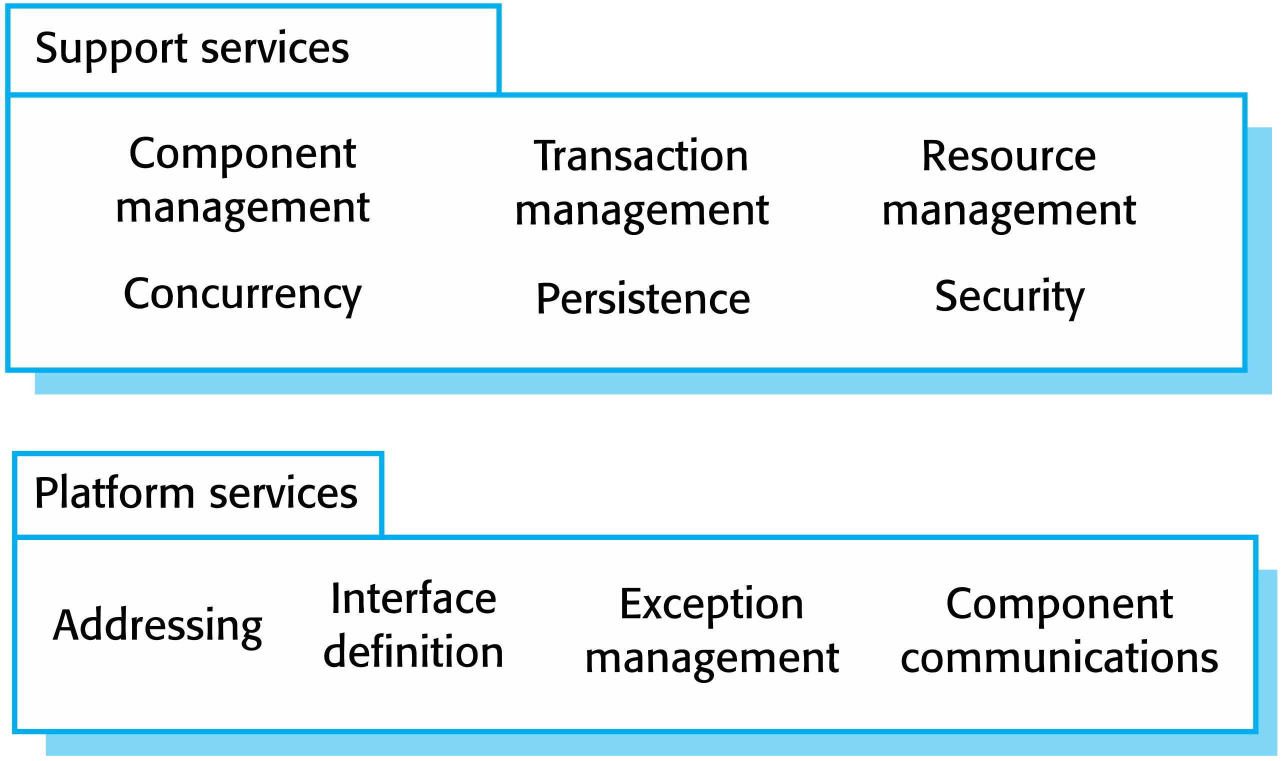
Deployment

- The component model includes a specification of how components should be packaged for deployment as independent, executable routines.
- Deployment information includes information about the contents of a package and its binary organization.
- Finally, the component model may define the component documentation that should be produced.

Middleware Support

- Component models are the basis for middleware that provides support for executing components.
- Component model implementations provide:
  - Platform services that enable components to communicate and interoperate in a distributed environment;
  - Support services that are application-independent services used by different components.
- Middleware implements the common component services and provides interfaces to them. To use services provided by a model, components are deployed in a container. This is a set of interfaces used to access the service implementations.

Middleware Services Defined in a Component Model

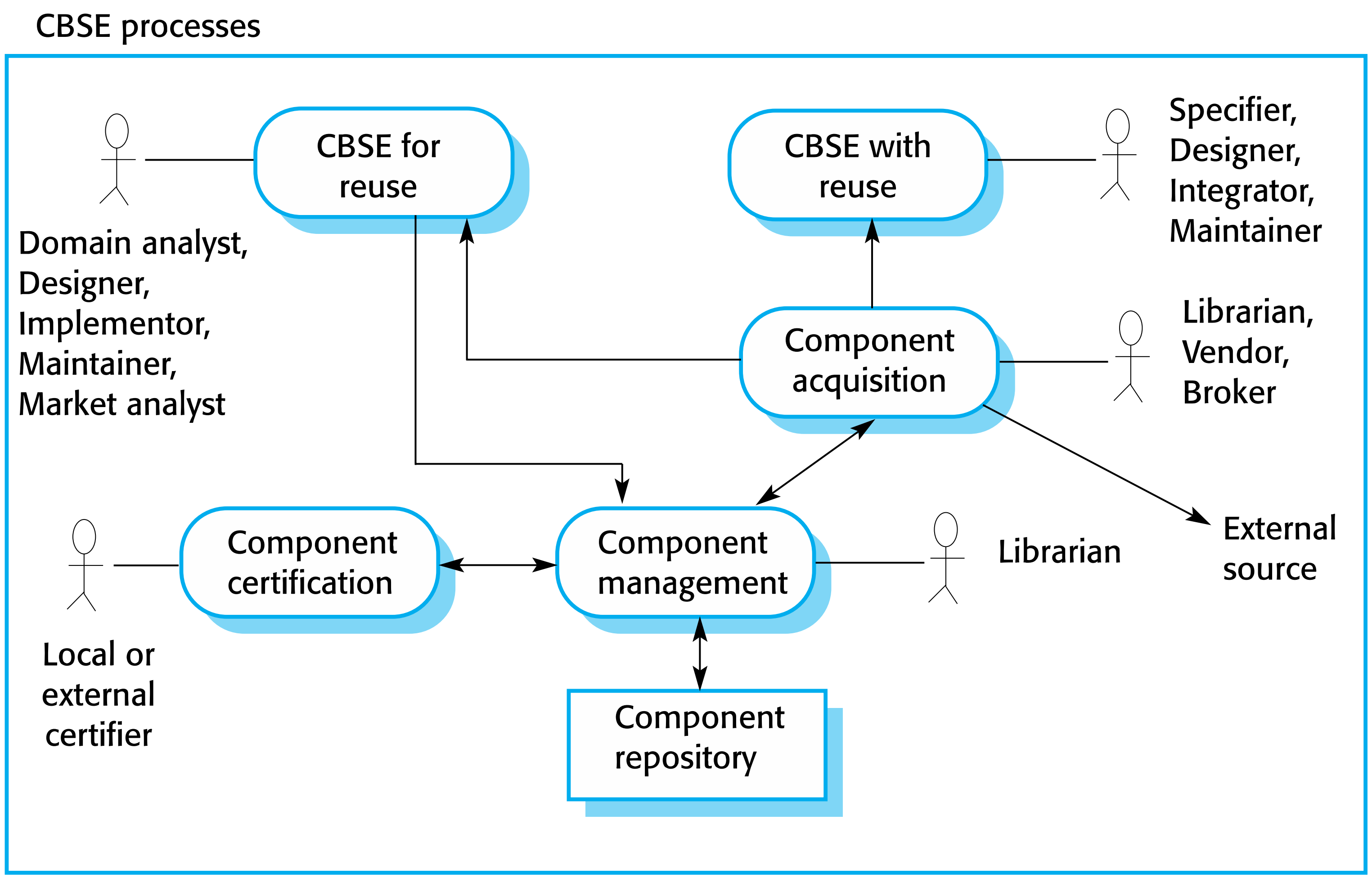


CBSE Processes

CBSE Processes (1 of 2)

- 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.
- 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.
- Development with reuse
  - This process is the process of developing new applications using existing components and services.

CBSE Processes (2 of 2)


CBSE processes
CBSE for reuse
CBSE with reuse
Specifier, Designer, Integrator, Maintainer
Domain analyst, Designer, Implementor, Maintainer, Market analyst
Component acquisition
Librarian, Vendor, Broker
External source
Component certification
Component management
Librarian
Local or external certifier
Component repository

Supporting Processes

- Component acquisition is the process of acquiring components for reuse or development into a reusable component.
  - It may involve accessing locally developed components or services or finding these components from an external source.
- Component management is concerned with managing a company's reusable components, ensuring that they are properly catalogued, stored and made available for reuse.
- Component certification is the process of checking a component and certifying that it meets its specification.

CBSE for Reuse

- CBSE for reuse focuses on component development.
- To make components reusable, you have to adapt and extend the application-specific components to create more generic and therefore more reusable versions. This adaptation has an associated cost. Decide:
  - whether a component is likely to be reused
  - whether the cost savings from future reuse justify the costs of making the component reusable

Component Development for Reuse

- For reusability, you have to decide whether or not the component implements one or more stable domain abstractions.
- Stable domain abstractions (business objects) are fundamental elements of the application domain that change slowly.
- If the component is an implementation of a commonly used domain abstraction or group of related business objects, it can probably be reused.

Component Development for Reuse

- To answer the question about cost-effectiveness, you have to assess the costs of changes that are required to make the component reusable.
- These costs are the costs of component documentation and component validation, and of making the component more generic. They include:
  - Remove application-specific methods.
  - Change names to make them general.
  - Add methods to broaden coverage.
  - Make exception handling consistent.
  - Add a configuration interface for component adaptation.
  - Integrate required components to reduce dependencies.

Exception Handling

- Components should not handle exceptions themselves, because each application will have its own requirements for exception handling.
  - Rather, the component should define what exceptions can arise and should publish these as part of the interface.
- In practice, however, there are two problems with this:
  - Publishing all exceptions leads to bloated interfaces that are harder to understand. This may put off potential users of the component.
  - The operation of the component may depend on local exception handling and changing this may have serious implications for the functionality of the component.

Reusable Components

- The development cost of reusable components may be 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.

Legacy System Components

- Existing legacy systems that fulfil a useful business function can be re-packaged as components for reuse.
- This involves writing a wrapper component that implements provides and requires interfaces then accesses the legacy system.
- Although costly, this can be much less expensive than rewriting the legacy system.

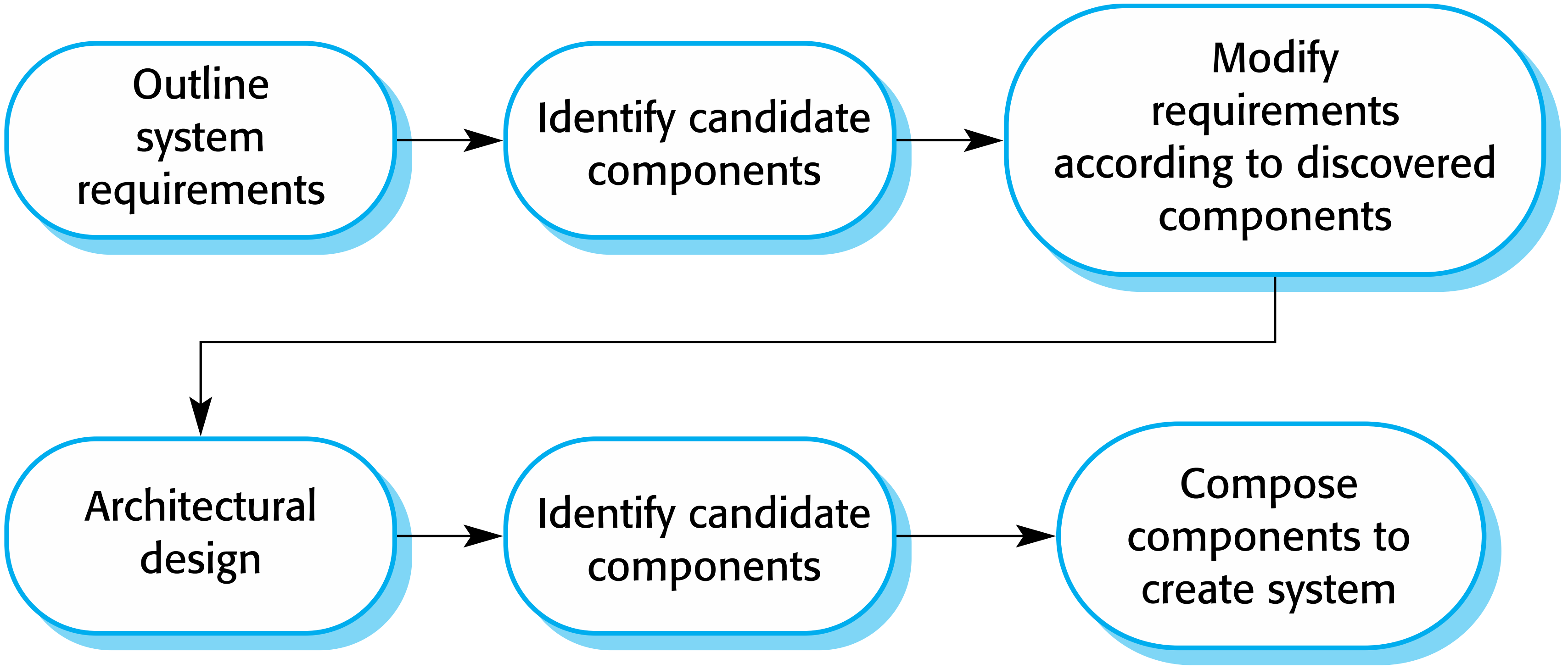
Component Management

- Component management involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component and keeping track of different component versions.
- A company with a reuse program may carry out some form of component certification before the component is made available for reuse.
  - Certification means that someone apart from the developer checks the quality of the component.

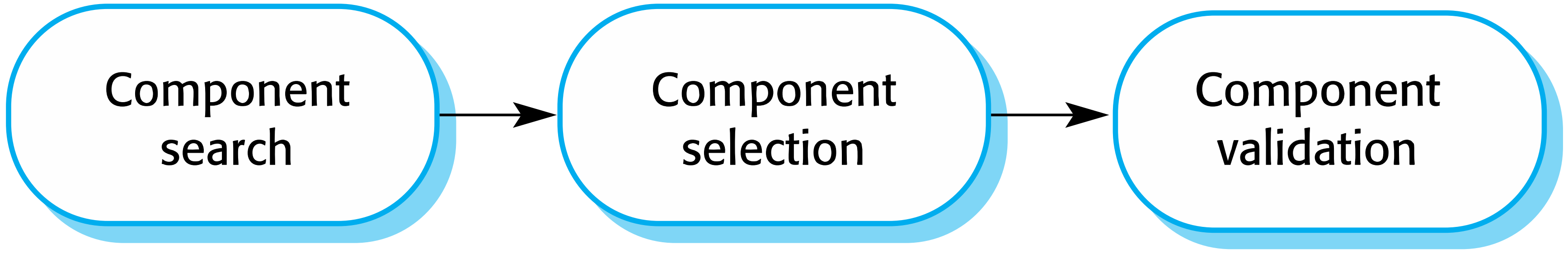
CBSE with Reuse (1 of 2)

- CBSE with reuse process has to find and integrate reusable components.
- When reusing components, it is essential to make trade-offs between ideal requirements and the services actually provided by available components.
- This involves:
  - Developing outline requirements;
  - Searching for components then modifying requirements according to available functionality.
  - Searching again to find if there are better components that meet the revised requirements.
  - Composing components to create the system.

CBSE with Reuse (2 of 2)


Outline system requirements
Identify candidate components
Modify requirements according to discovered components
Architectural design
Identify candidate components
Compose components to create system

The Component Identification Process



Component Search

- The first step in identifying components is to look for components that are available within your company or from trusted suppliers.
- Software development companies can build their own database of reusable components without the risks inherent in using components from external suppliers.
- Alternatively, you may decide to search code libraries available on the web, such as Sourceforge, GitHub, or Google Code, to see if source code for the component that you need is available.

Component Selection

- In some cases, this will be a straightforward task. Components on the list will directly implement the user requirements, and there will not be competing components that match these requirements.
- In other cases, however, the selection process is more complex. There will not be a clear mapping of requirements onto components. You may find that several components have to be integrated to meet a specific requirement or group of requirements. You therefore have to decide which of these component compositions provide the best coverage of the requirements.

Component Validation

- Component validation involves developing a set of test cases for a component (or, possibly, extending test cases supplied with that component) and developing a test harness to run component tests.
  - The major problem with component validation is that the component specification may not be sufficiently detailed to allow you to develop a complete set of component tests.
- As well as testing that a component for reuse does what you require, you may also have to check that the component does not include any malicious code or functionality that you don’t need.

Component composition

Component Composition

- The process of assembling components to create a system.
- Composition involves integrating components with each other and with the component infrastructure.
- Normally you have to write ‘glue code’ to integrate components.
- You can compose components in different ways: Sequential composition, Hierarchical composition, Additive composition

Sequential composition

- In a sequential composition, you create a new component from two existing components by calling the existing components in sequence.
- You can think of the composition as a composition of the "provides interfaces."
- The components do not call each other in sequential composition but are called by the external application. This type of composition may be used with embedded or service components.

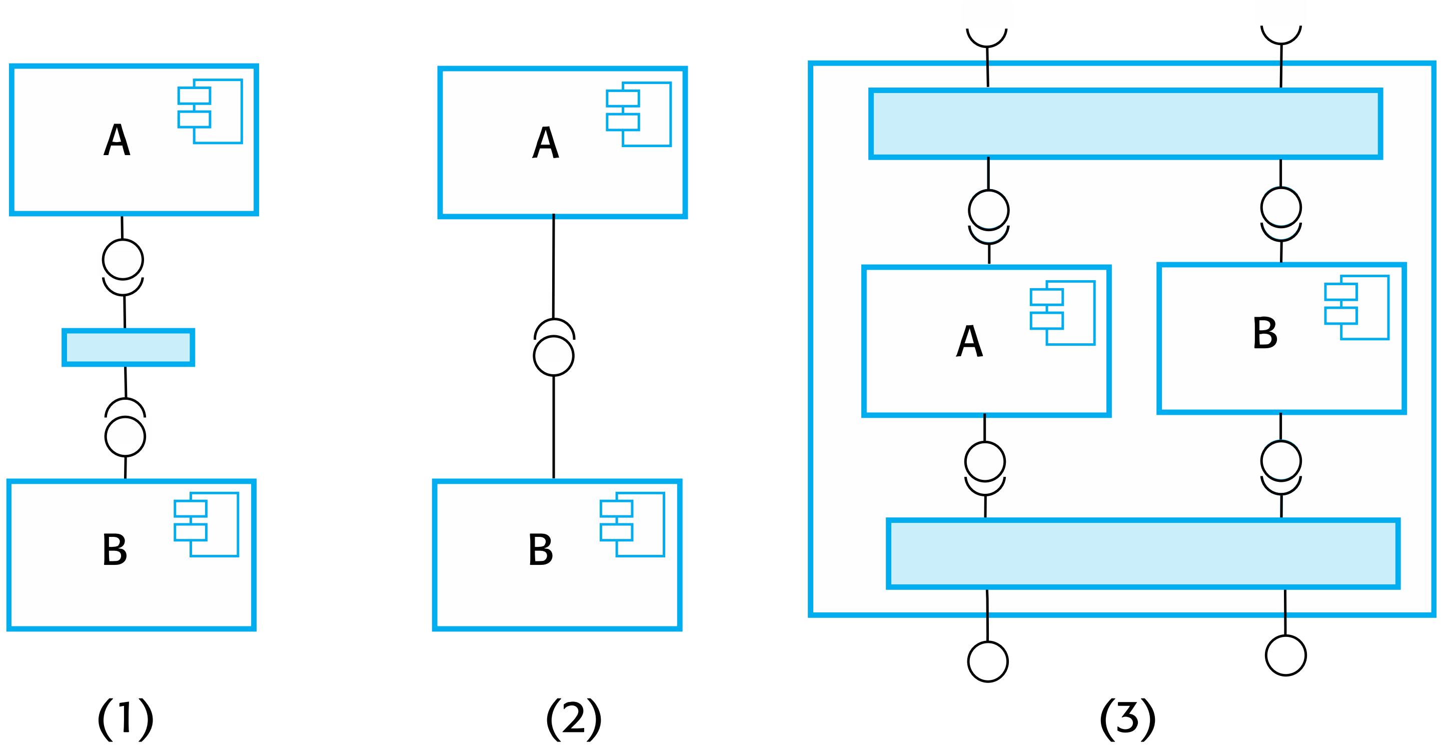
Hierarchical composition

- Occurs when one component calls directly on the services provided by another component.
- The called component provides the services that are required by the calling component. Therefore, the “provides” interface of the called component must be compatible with the “requires” interface of the calling component.
- As services do not have a “requires” interface, this mode of composition is not used when components are implemented as services accessed over the web.

Additive composition

- Occurs when two or more components are put together (added) to create a new component, which combines their functionality.
- The “provides” interface and “requires” interface of the new component are a combination of the corresponding interfaces in components
- The components are called separately through the external interface of the composed component and may be called in any order.
- May be used with embedded or service components.

Types of Component Composition



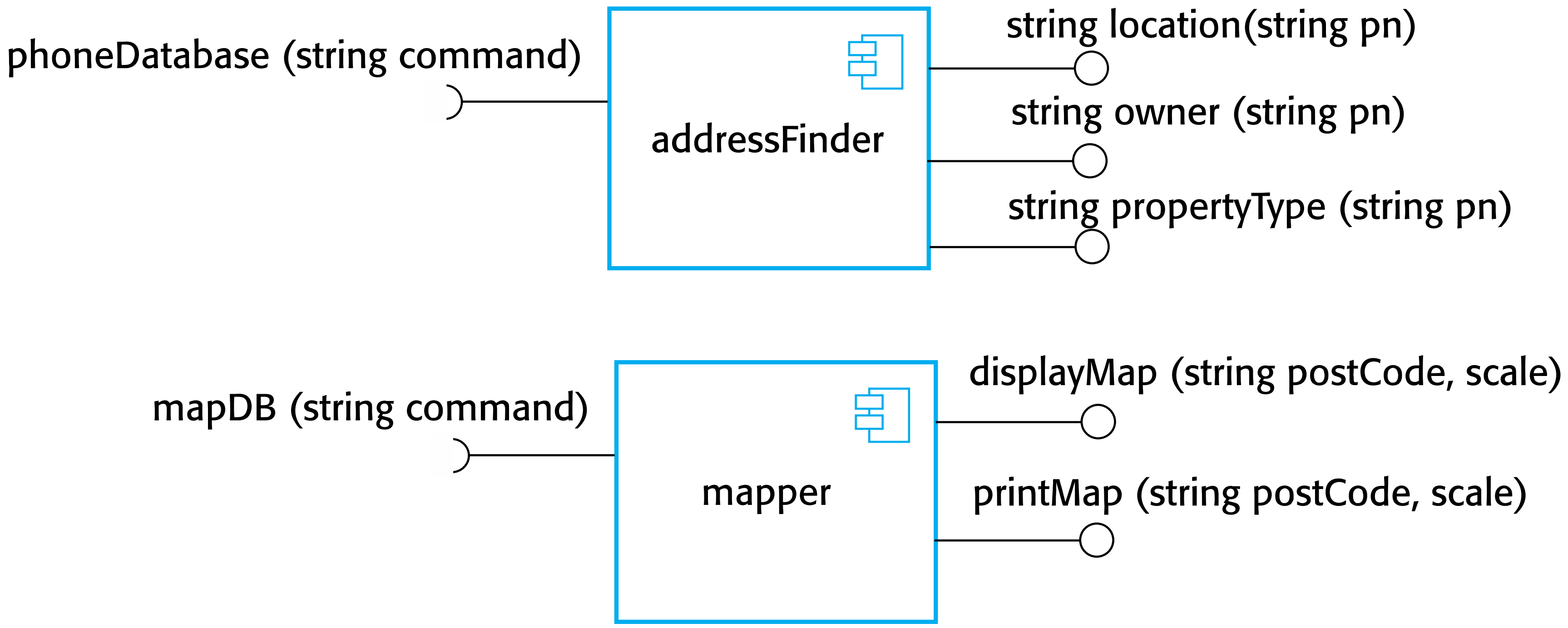
Glue Code

- Code that allows components to work together
- You might use all the forms of component composition when creating a system. In all cases, you may have to write “glue code” that links the components.
- When one component calls another, you may need to introduce an intermediate component that ensures that the “provides” interface and the “requires” interface are compatible.

Interface Incompatibility

- *Parameter incompatibility* where operations have the same name but parameter types or the number of parameters are different.
- *Operation incompatibility* where the names of the operations in the provides and requires interfaces are different.
- *Operation incompleteness* where the provides interface of one component is a subset of the requires interface of another, or vice versa.

Components with Incompatible Interfaces


phoneDatabase (string command)
addressFinder
string location(string pn)
string owner (string pn)
string propertyType (string pn)
mapDB (string command)
mapper
displayMap (string postCode, scale)
printMap (string postCode, scale)

Adaptor Components

- Address the problem of component incompatibility by reconciling the interfaces of the components that are composed.
- Different types of adaptor are required depending on the type of composition.
- An addressFinder and a mapper component may be composed through an adaptor that strips the postal code from an address and passes this to the mapper component.

Composition Through an Adaptor

- The component postCodeStripper is the adaptor that facilitates the sequential composition of addressFinder and mapper components.

```
address addressFinder.location (phonenumber) ;
postCode postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000)
```

An Adaptor Linking a Data Collector and a Sensor

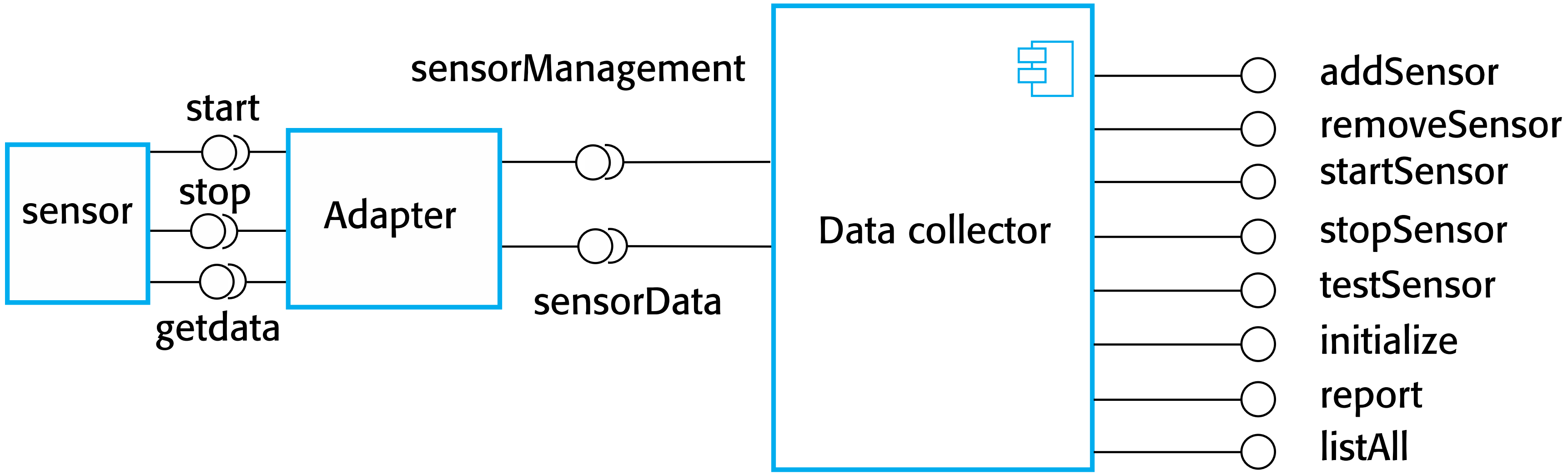
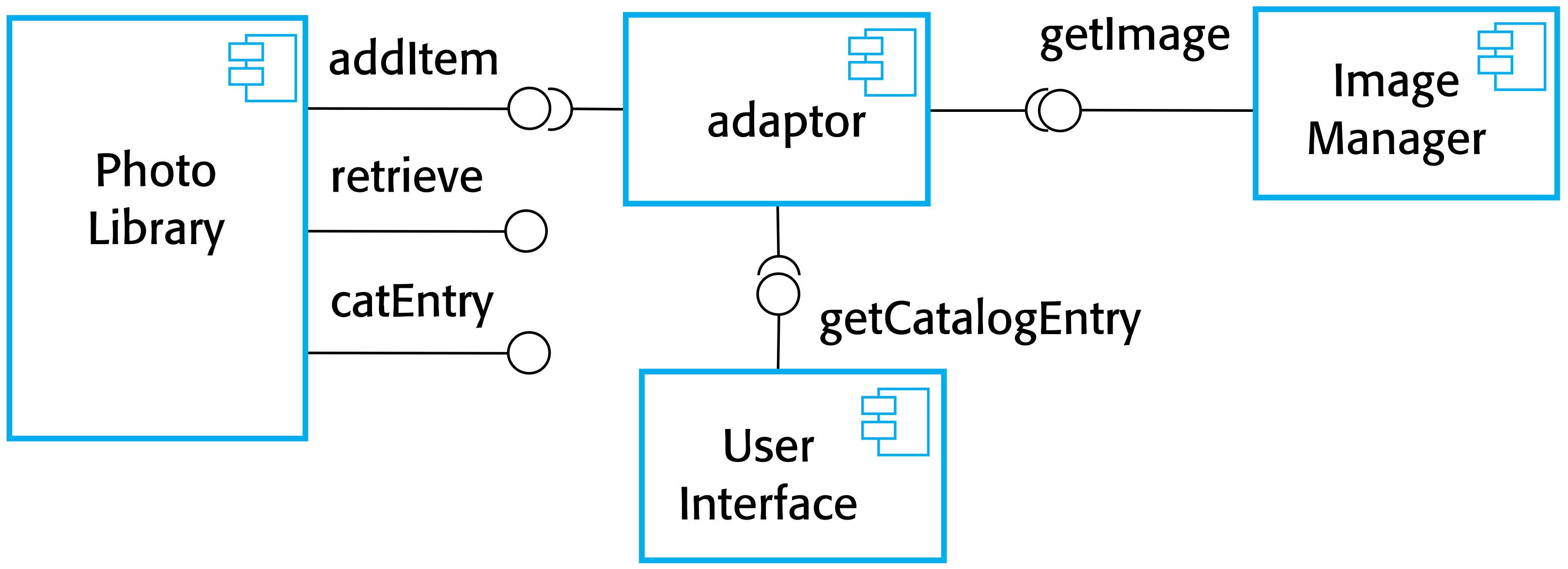


Photo Library Composition



Interface Semantics

- You have to rely on component documentation to decide if interfaces that are syntactically compatible are actually compatible.
- Consider an interface for a PhotoLibrary component:

```
public void addItem (Identifier pid ; Photograph p; CatalogEntry photodesc) ;
public Photograph retrieve (Identifier pid) ;
public CatalogEntry catEntry (Identifier pid) ;
```

Photo Library Documentation

addItem: “This method adds a photograph to the library and associates the photograph identifier and catalogue descriptor with the photograph.”

“what happens if the photograph identifier is already associated with a photograph in the library?”

“is the photograph descriptor associated with the catalogue entry as well as the photograph i.e. if I delete the photograph, do I also delete the catalogue information?”

The Object Constraint Language

- The Object Constraint Language (OCL) has been designed to define constraints that are associated with UML models.
- Allows you to express predicates that must always be true, that must be true before a method has executed; and that must be true after a method has executed. These are invariants, preconditions, and postconditions.
  - age = age@pre + 1
- OCL-based approaches are primarily used in model-based software engineering to add semantic information to UML models. The OCL descriptions may be used to drive code generators in model-driven engineering.

The OCL Description of the Photo Library Interface

```
-- The context keyword names the component to which the conditions apply

context addItem

-- The preconditions specify what must be true before execution of addItem
pre: PhotoLibrary.libSize() > 0
PhotoLibrary.retrieve(pid) = null

-- The postconditions specify what is true after execution
post:libSize () = libSize()@pre + 1
PhotoLibrary.retrieve(pid) = p
PhotoLibrary.catEntry(pid) = photodesc

context delete

pre: PhotoLibrary.retrieve(pid) <> null ;

post: PhotoLibrary.retrieve(pid) = null
PhotoLibrary.catEntry(pid) = PhotoLibrary.catEntry(pid)@pre
PhotoLibrary.libSize() = libSize()@pre—1
```

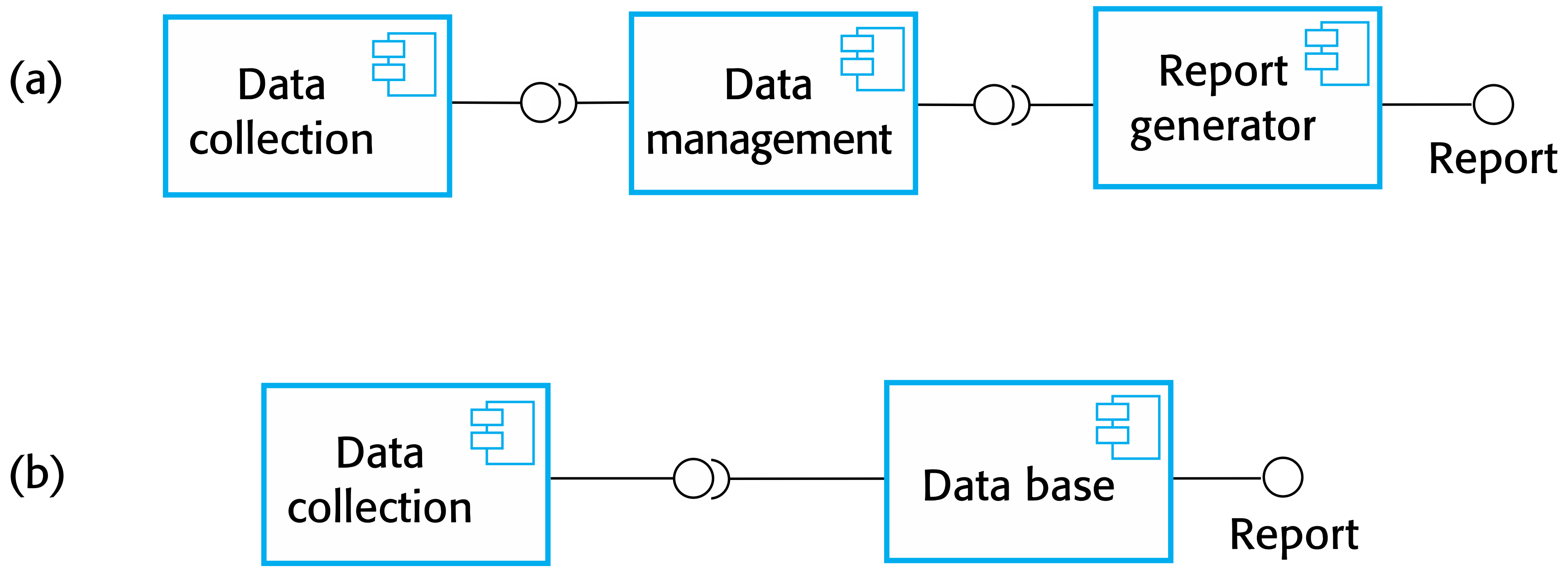
Photo Library Conditions

- As specified, the OCL associated with the Photo Library component states that:
  - There must not be a photograph in the library with the same identifier as the photograph to be entered;
  - The library must exist - assume that creating a library adds a single item to it;
  - Each new entry increases the size of the library by 1;
  - If you retrieve using the same identifier then you get back the photo that you added;
  - If you look up the catalogue using that identifier, then you get back the catalogue entry that you made.

Composition Trade-offs

- When composing components, you may find conflicts between functional and non-functional requirements, and conflicts between the need for rapid delivery and system evolution.
- You need to make decisions such as:
  - What composition of components is effective for delivering the functional requirements?
  - What composition of components allows for future change?
  - What will be the emergent properties of the composed system?

Data Collection and Report Generation Components



Key Points (1 of 2)

- CBSE is a reuse-based approach to defining and implementing loosely coupled components into systems.
- A component is a software unit whose functionality and dependencies are completely defined by its interfaces.
- Components may be implemented as executable elements included in a system or as external services.
- A component model defines a set of standards that component providers and composers should follow.
- The key CBSE processes are CBSE for reuse and CBSE with reuse.

Key Points (2 of 2)

- During the CBSE process, the processes of requirements engineering and system design are interleaved.
- Component composition is the process of ‘wiring’ components together to create a system.
- When composing reusable components, you normally have to write adaptors to reconcile different component interfaces.
- When choosing compositions, you have to consider required functionality, non-functional requirements and system evolution.

Copyright