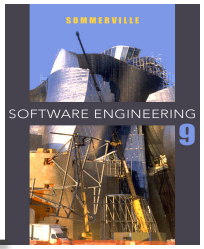
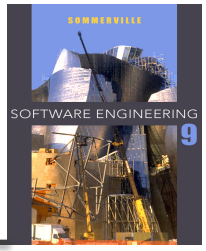


Chapter 17 Component-based software engineering

Lecture 1

Topics covered

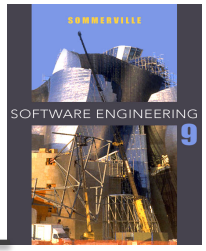


17.1 Components and component models

17.2 CBSE processes

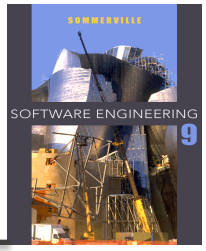
17.3 Component composition

Component-based development



- Component-based software engineering (CBSE) is an approach to software development that relies on the reuse of entities called 'software components'. (CBSE is the process of defining, implementing, and integrating or composing loosely coupled, independent components into systems)
- It emerged from the failure of object-oriented development to support effective reuse in the late 1990s. Single object classes are too detailed and specific.
- Components are more abstract (defined by their interfaces) than object classes and can be considered to be stand-alone service providers. They can exist as stand-alone entities.

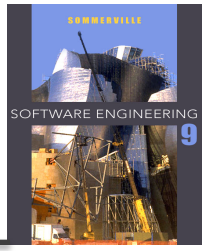
CBSE essentials



The essentials of CBSE are:

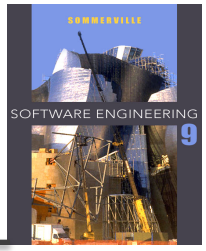
- ✧ Independent components specified by their interfaces.
- ✧ Component standards to facilitate component integration.
- ✧ Middleware that provides support for component inter-operability.
- ✧ A development process that is geared to reuse.

CBSE and design principles



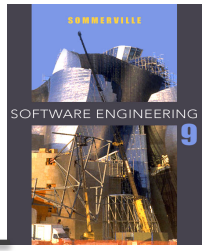
- Apart from the benefits of reuse, CBSE is based on sound software engineering design principles:
  - Components are independent so do not interfere with each other;
  - Component implementations are hidden;
  - Communication is through well-defined interfaces;
  - Component infrastructures/platforms offer a range of standard services that can be used in application systems. These services are shared and reduce development costs.

Component standards



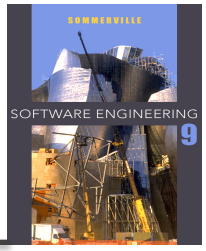
- Standards need to be established so that components can communicate with each other and inter-operate.
- Unfortunately, several competing component standards were established:
  - Sun's Enterprise Java Beans(EJB)
  - Microsoft's COM and .NET
  - CORBA's CCM(OMG)(CCM-CORBA Component Model, CORBA-Common Object Request Broker Architecture, OMG-Object Management Group)
- In practice, these multiple standards have hindered the uptake of CBSE. It is impossible for components developed using different approaches to work together.

CBSE problems



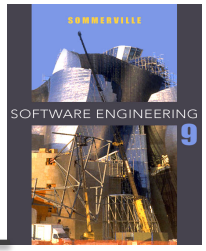
- Component trustworthiness - how can a component with no available source code be trusted?
- Component certification - who will certify the quality of components?
- Emergent property prediction - how can the emergent properties of component compositions be predicted?
- Requirements trade-offs - how do we do trade-off analysis between the features of one component and another?

17.1 Components and component models



- 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



People have proposed varying definitions of a component:

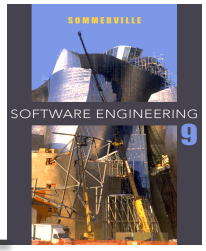
✧ Councill and Heinmann:

- *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:

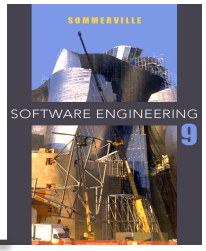
- *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third-parties.*

Component characteristics



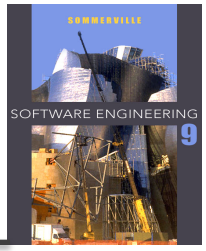
Component characteristic	Description
Standardized	Component standardization means that a component used in a CBSE process has to conform to a standard component model. This model may define component interfaces, component metadata, documentation, composition, and deployment.
Independent	A component should 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.
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.

Component characteristics



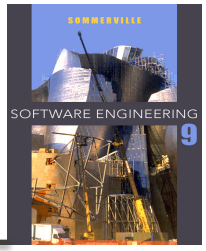
Component characteristic	Description
Deployable	To be deployable, a component has to be self-contained. It must be able to operate as a stand-alone entity on a component platform that provides an implementation of the component model. This usually means that the component is binary and does not have 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.
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.

Component as a service provider



- The component is an independent, executable entity. It does not have to be compiled before it is used with other components.
- The services offered by a component are made available through an interface and all component interactions take place through that interface.
- The component interface is expressed in terms of parameterized operations and its internal state is never exposed.

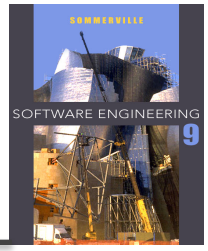
Component interfaces



Components have two related 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
  - Specifies what services must be provided by other components in the system if a component is to operate correctly. If these are not available, then the component will not work.
  - This does not compromise the independence or deployability of a component because the ‘requires’ interface does not define how these services should be provided.

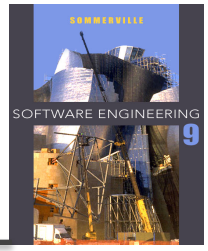
Component interfaces

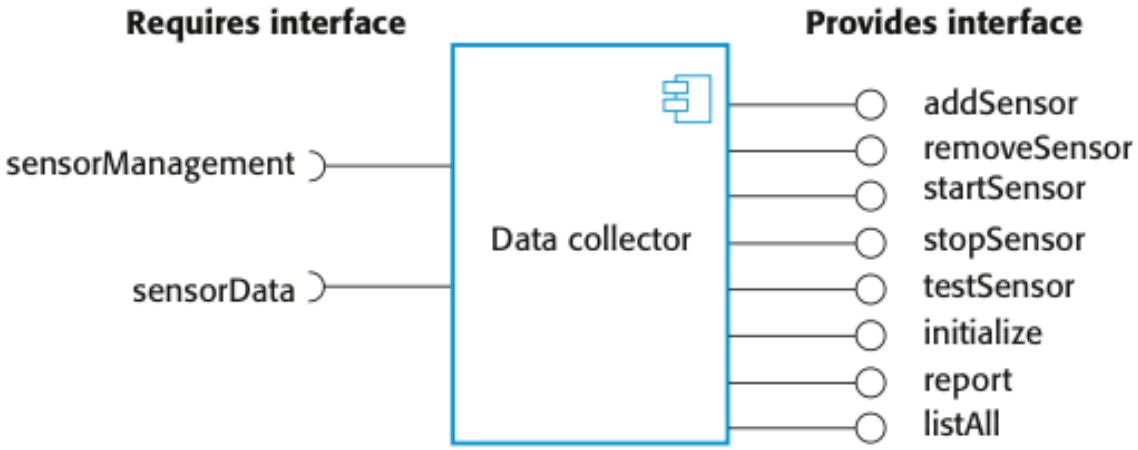


In the UML component diagram, the 'provides' interface is a circle and the 'requires' interface is a semicircle, at the end of a line from the component icon;

Note UML notation. Ball (Provides interface) and sockets (Requires interface) can fit together.

Example: A model of a data collector component





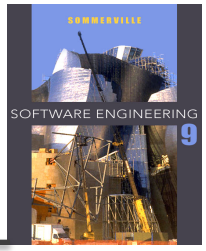
• This figure shows a model of a component that has been designed to collect information from an array of sensors. It runs autonomously to collect data over a period of time and, on request, provides collected data to a calling component.

• The 'provides' interface includes methods, such as addSensor(), removeSensor(),…;

• These methods have associated parameters specifying the sensor identifiers, locations and so on.

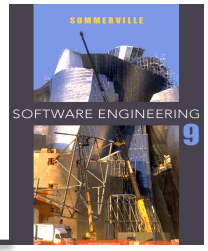
• The 'requires' interface is used to connect the component to the sensors.

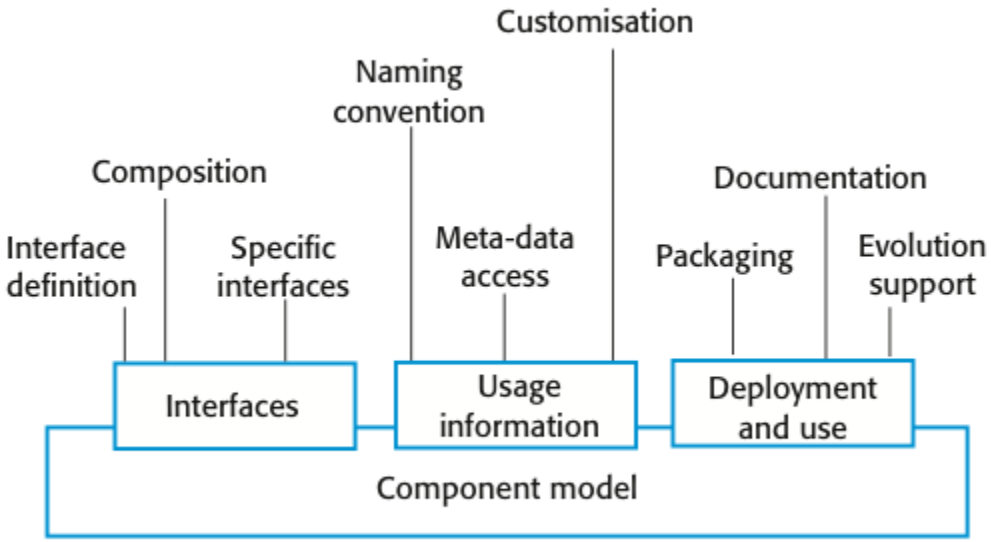
17.1.1 Component models



- 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(CCM) (Corba - Common Object Request Broker Architecture)
- The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.

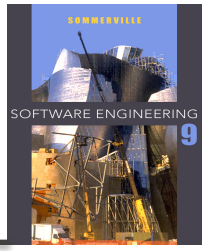
Basic elements of a component model





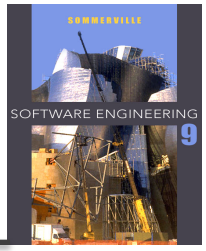
- The diagram summarizes the model elements.
- It shows that the elements of a component model define the component interfaces, the information that you need to use the component in a program, and how a component should be deployed. More details are shown in next slide.

Elements of a component model



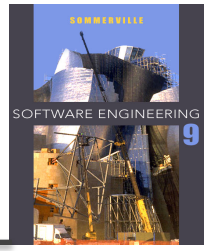
- Interfaces
  - Components are defined by specifying their interfaces. The component model specifies how the interfaces should be defined and the elements, such as operation names, parameters and exceptions, which should be included in the interface definition.
- Usage
  - In order for components to be distributed and accessed remotely, they need to have a unique name or handle associated with them. This has to be globally unique.
- Deployment
  - The component model includes a specification of how components should be packaged for deployment as independent, executable entities.

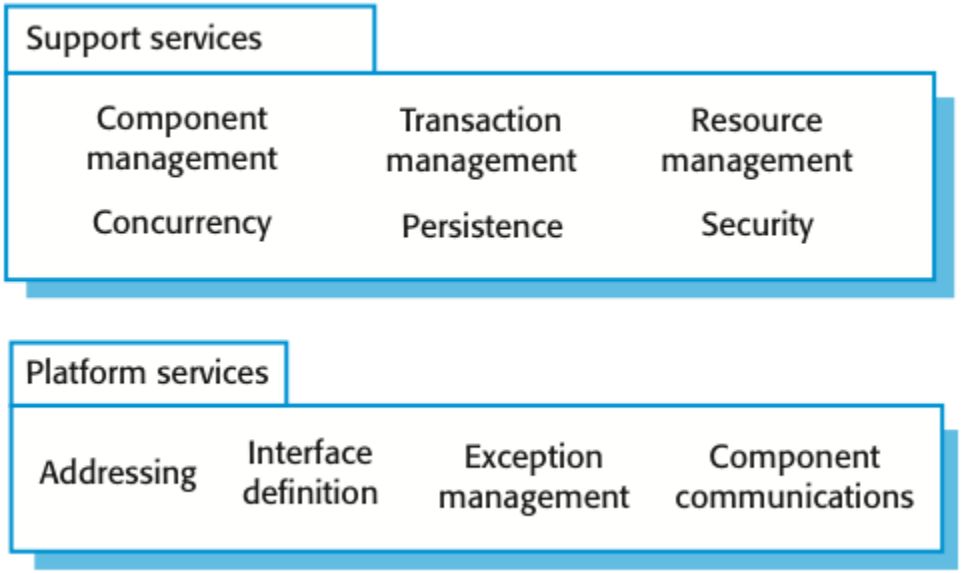
Middleware support



- Component models are the basis for middleware that provides support for executing components.
- Component model implementations provide:
  - Platform services that allow components written according to the model to communicate;
  - Support services that are application-independent services used by different components.
  - More details in next slide.
- 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.

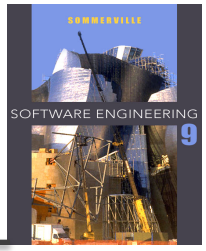
Middleware services defined in a component model





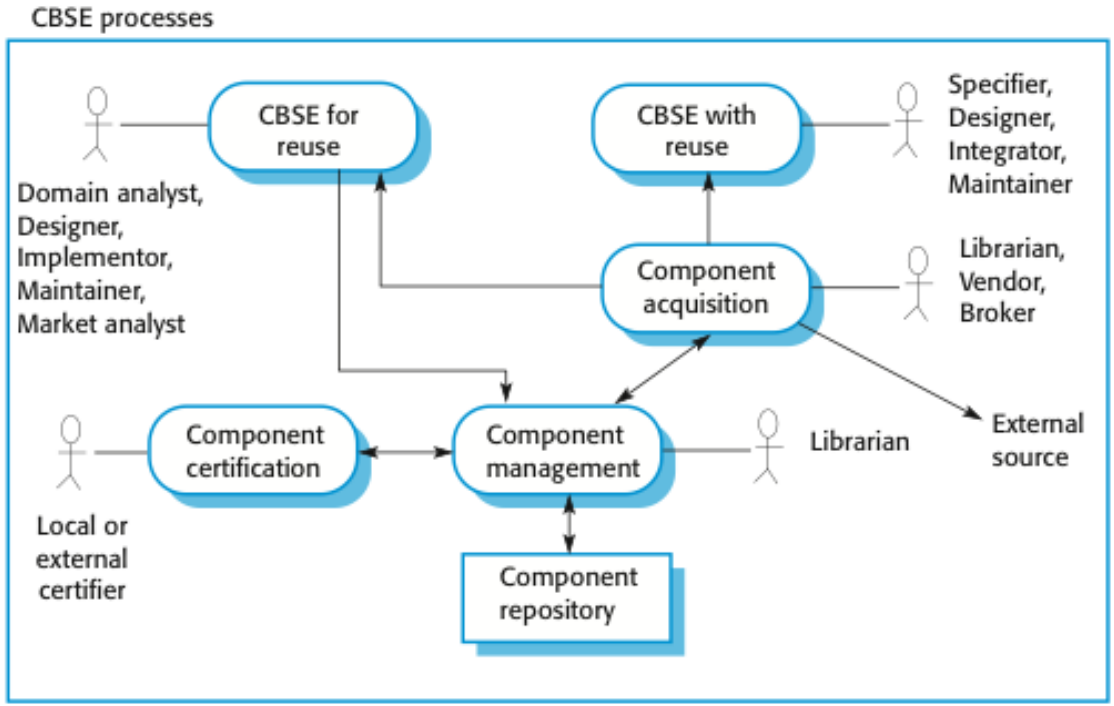
- Platform services, which enable components to communicate and interoperate in a distributed environment. These are the fundamental services that must be available in all component-based systems;
- Support services, which are common services that are likely to be required by many different components. For example, many components require authentication to ensure that the user of component services is authorized. It makes sense to provide a standard set of middleware services for use by all components. This reduces the costs of component development and potential component incompatibilities can be avoided.

17.2 CBSE processes



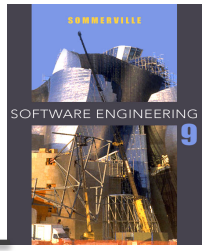
- CBSE processes are software processes that support component-based software engineering.
  - They take into account the possibilities of reuse and the different process activities involved in developing and using reusable components.
- Development for reuse
  - This process is concerned with developing components or services that will be reused in other applications. It usually involves generalizing existing components.(reuse-oriented)
- Development with reuse
  - This process is the process of developing new applications using existing components and services.(reuse-based)

CBSE processes



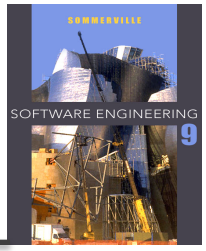
- The basic processes of CBSE with and for reuse have supporting processes that are concerned with component acquisition, component management, and component certification. More details in next slide.

Supporting processes

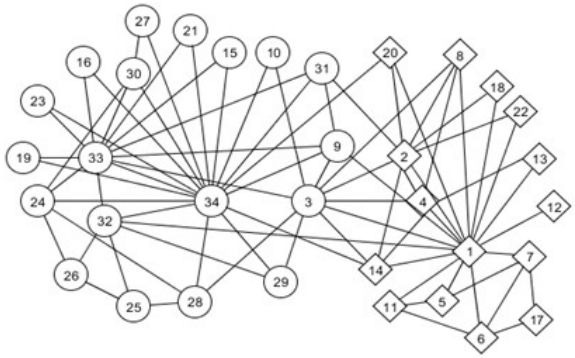


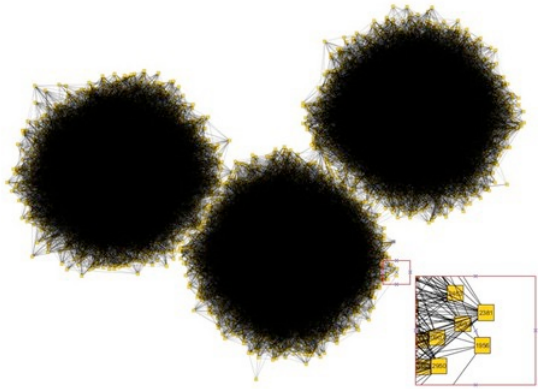
- Component acquisition is the process of acquiring components for reuse or development into a reusable component.
  - It may involve accessing locally-developed components or services or finding these components from an external source.
- Component management is concerned with managing a company's reusable 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.

Key points

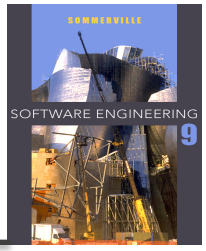


- CBSE is a reuse-based approach to defining and implementing loosely coupled components into systems.





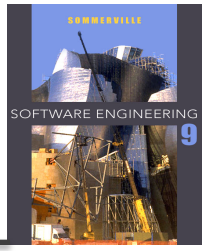
- A component is a software unit whose functionality and dependencies are completely defined by its interfaces.
- A component model defines a set of standards that component providers and composers should follow.
- The key CBSE processes are CBSE for reuse and CBSE with reuse.



Chapter 17 Component-based software engineering

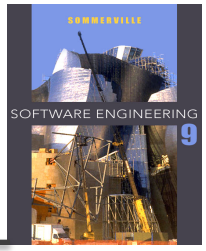
Lecture 2

17.2.1 CBSE for reuse



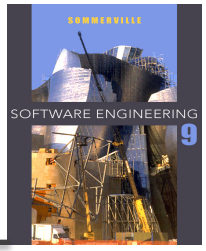
- CBSE for reuse focuses on component development.
- Components developed for a specific application usually have to be generalized to make them reusable.
- A component is most likely to be reusable if it associated with a stable domain abstraction (business object).
- For example, in a hospital stable domain abstractions are associated with the fundamental purpose - nurses, patients, treatments, etc.

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.

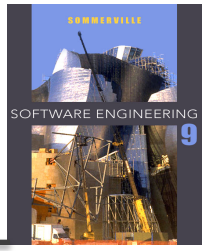
Changes for reusability



Changes that you may make to a component to make it more reusable include:

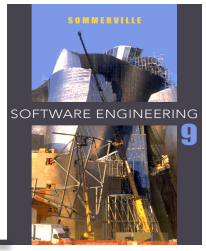
- ✧ Remove application-specific methods.
- ✧ Change names to make them general.
- ✧ Add methods to broaden coverage.
- ✧ Make exception handling consistent.
- ✧ Add a configuration interface for component adaptation.
- ✧ Integrate required components to reduce dependencies.

Exception handling



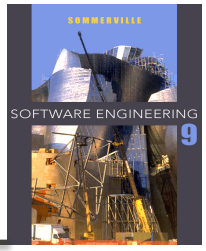
- Components should not handle exceptions themselves, because each application will have its own requirements for exception handling.
  - Rather, the component should define what exceptions can arise and should publish these as part of the interface.
- In practice, however, there are two problems with 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.

Legacy system components



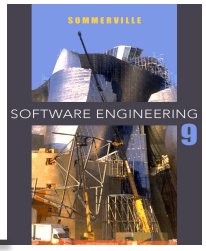
- Existing legacy systems that fulfil a useful business function can be re-packaged as components for reuse.
- This involves writing a wrapper component that implements provides and requires interfaces then accesses the legacy system.
- Although costly, this can be much less expensive than rewriting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

Reusable components



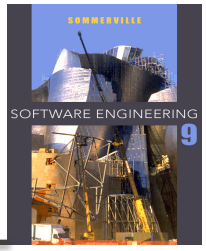
- The development cost of reusable components may be higher than the cost of specific equivalents. This extra reusability enhancement cost should be an organization rather than a project cost.
- Generic components may be less space-efficient and may have longer execution times than their specific equivalents.

Component management



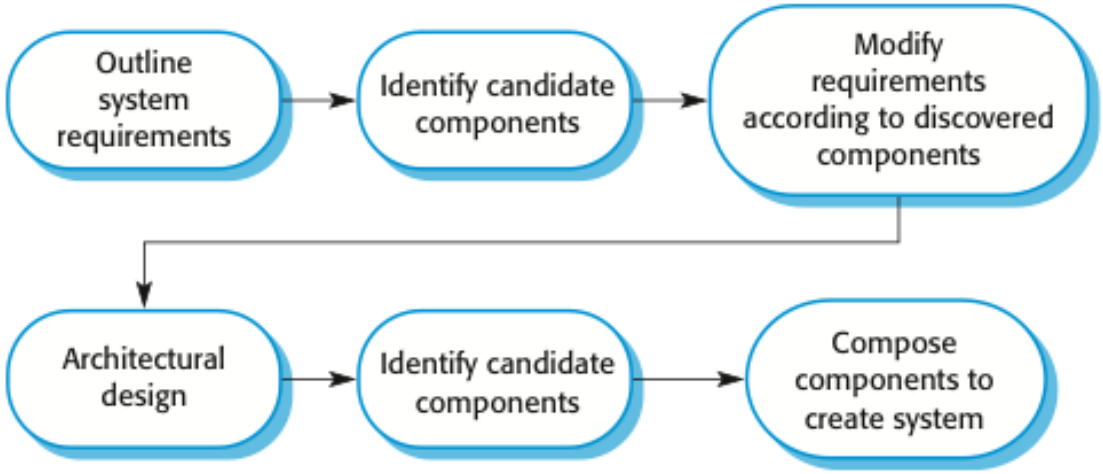
- Component management involves deciding how to classify the component so that it can be discovered, making the component available either in a repository or as a service, maintaining information about the use of the component and keeping track of different component versions.
- A company with a reuse 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.

17.2.2 CBSE with reuse



- 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.
- The process 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.
  - Figured in next slide.

CBSE with reuse

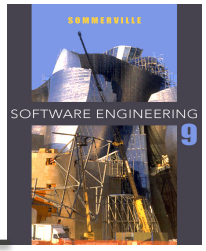


The principal activities of CBSE with reuse process

The essential differences between CBSE with reuse and software processes for original software development are:

1. The user requirements are initially developed in outline rather than in detail, and stakeholders are encouraged to be as flexible as possible in defining their requirements.
2. Requirements are refined and modified early in the process depending on the components available.
3. There is a further component search and design refinement activity after the system architecture has been designed.
4. Development is a composition process where the discovered components are integrated.

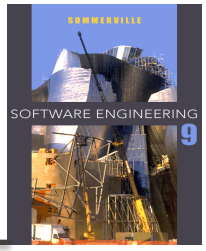
The component identification process



An activity that is unique to the CBSE process is identifying candidate components or services for reuse. This involves a number of subactivities, as shown in the following figure:

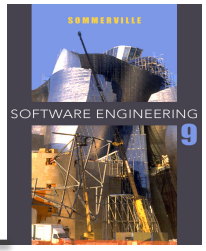
1. Initially, your focus should be on search and selection.
2. You need to convince yourself that there are components available to meet your requirements.
3. Obviously, you should do some initial checking that the component is suitable but detailed testing may not be required.
4. In the later stage, after the system architecture has been designed, you should spend more time on component validation.
5. You need to be confident that the identified components are really suited to your application; if not, then you have to repeat the search and selection processes.

Component identification issues



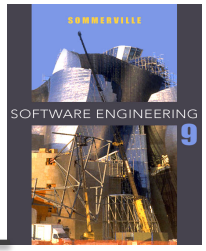
- Trust. You need to be able to trust the supplier of a component. At best, an untrusted component may not operate as advertised; at worst, it can breach your security.
- Requirements. Different groups of components will satisfy different requirements.
- Validation.
  - The component specification may not be detailed enough to allow comprehensive tests to be developed.
  - Components may have unwanted functionality. How can you test this will not interfere with your application?
  - More details in next slide.

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. An example is shown in next slide.

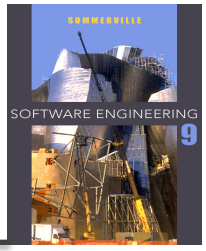
Ariane launcher failure – validation failure?



- In 1996, the 1st test flight of the Ariane 5 rocket ended in disaster when the launcher went out of control 37 seconds after take off.
- The problem was due to a reused component from a previous version of the launcher (the Inertial Navigation System) that failed because assumptions made when that component was developed did not hold for Ariane 5.
- The functionality that failed in this component was not required in Ariane 5, the validation tests for the reused software were based on Ariane 5 requirements.(Because there were no requirements for the function that failed, no tests were developed. Consequently, the problem with the software was never discovered during launch simulation tests.)

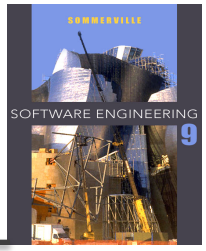
More description: The cause of the problem was an unhandled exception when a conversion of a fixed-point number to an integer resulted in a numeric overflow. This caused the run-time system to shut down the inertial reference system and launcher stability could not be maintained. The fault had never occurred in Ariane 4 because it had less powerful engines and the value that was converted could not be large enough for the conversion to overflow.

17.3 Component composition



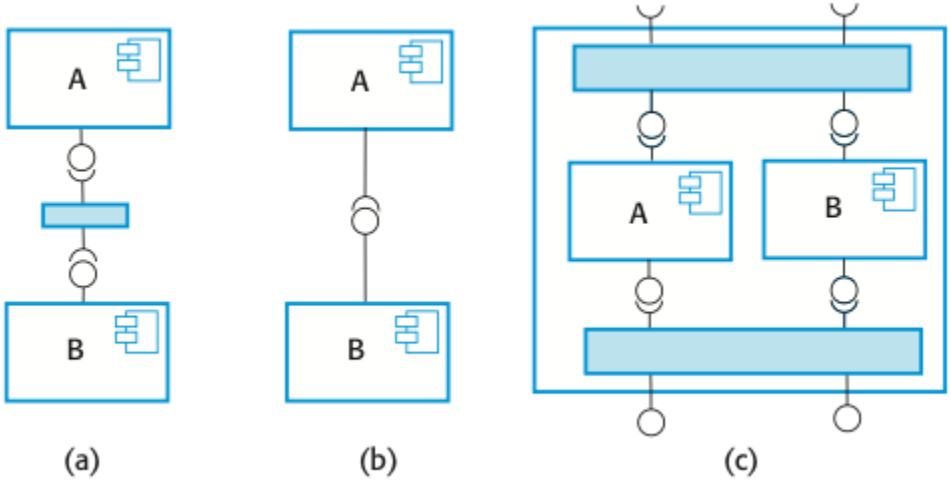
- The process of assembling components to create a system(or another component).
- Composition involves integrating components with each other and with the component infrastructure.
- Normally you have to write ‘glue code’ to integrate components.

Types of composition



- Sequential composition where the composed components are executed in sequence. This involves composing the ***provides interfaces*** of each component.
- Hierarchical composition where one component calls on the services of another. The ***provides interface*** of one component is composed with the ***requires interface*** of another.
- Additive composition where the interfaces of two components are put together to create a new component. Provides and requires interfaces of integrated component is a combination of interfaces of constituent components.

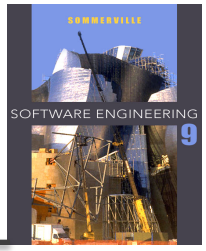
Types of component composition



Types of component composition

(a) Sequential composition(composing the ***provides interfaces***, the services offered by component A are called, and the results returned by A are then used in the call to the services offered by component B.)
(b) Hierarchical composition(***provides interface*** of one component is composed with the ***requires interface*** of other, these two interfaces must be **compatible.** ***Additional code*** may need if there is a ***mismatch*** between ***requires interface*** of A and ***provides interface*** of B.)
(c) Additive composition(Provides and requires interfaces is a ***combination*** of the corresponding interfaces in components A and B. The components are ***called separately*** through the external interface of the composed component. A and B are not dependent and do not call each other.)

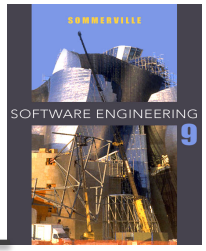
Interface incompatibility



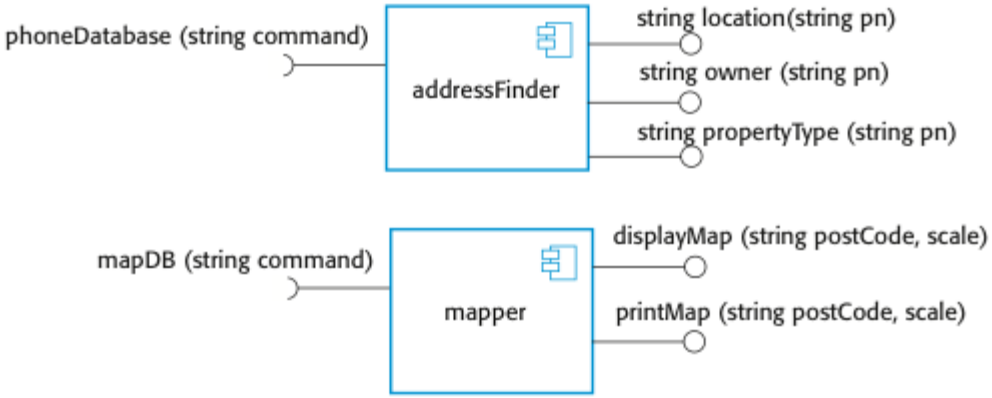
- Parameter incompatibility where operations have the same name but are of different types.
- Operation incompatibility where the names of operations in the composed interfaces are different.
- Operation incompleteness where the '*provides' interface* of one component is a subset of the '*requires' interface* of another.

In all cases, you tackle the problem of incompatibility by writing an adapter that reconciles the interfaces of the two components being reused. An adaptor component converts one interface to another.

Components with incompatible interfaces



To illustrate adaptors, consider the two components shown below, whose interfaces are incompatible:



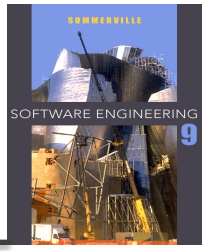
These might be part of a system used by the emergency services(simplified).

When the emergency operator takes a call, the phone number is input to the **addressFinder** component to locate the address.

Then using the **mapper** component, the operator prints a map to be sent to the vehicle dispatched to the emergency.

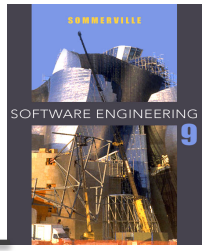
The component **addressFinder** finds the address that matches a phone number. The **mapper** component takes a post code and displays or prints a street map of the area around that code at a specified scale.

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

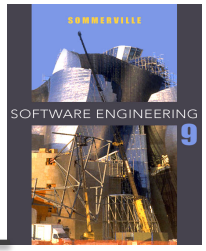


- The component 'postCodeStripper' is the adaptor that facilitates the sequential composition of 'addressFinder' and 'mapper' components.
- 'postCodeStripper' takes the location data from '**addressFinder**' and strips out the post code. This post code is then used as an input to '**mapper**' and the street map is displayed at a scale of 1:10,000. The following code, which is an example of sequential composition, illustrates the sequence of calls that is required to implement this:

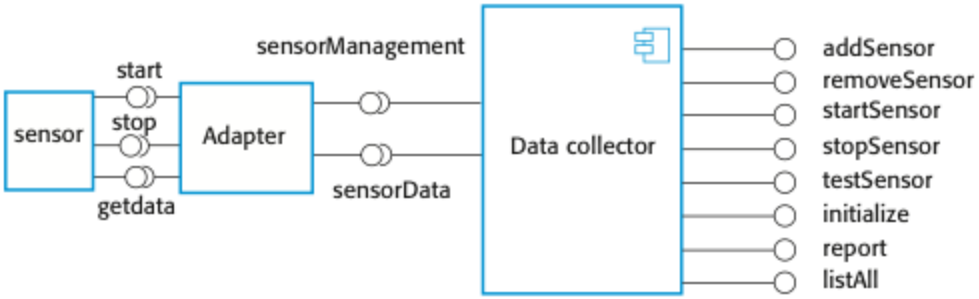
The picture can't be displayed.

```
Address=addressFinder.location(phonenumber);
Postcode=postCodeStripper.getPostCode(address);
Mapper.displayMap(postCode,10000);
```

An adaptor linking a data collector and a sensor



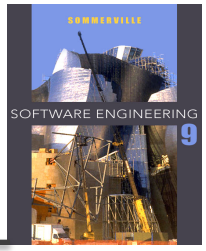
The use of an adaptor in the following Figure where an adaptor is used to link a '**Data Collector**' and a '**sensor**' component. These could be used in the implementation of a wilderness weather station system.



A separate 'adaptor', which converts the sensor commands from 'Data collector' to the actual sensor interface, is implemented for each type of sensor.

The 'Data Collector' component has been designed with a generic 'requires' interface that supports sensor data collection and sensor management.
For each of these operations, the parameter is a text string representing the specific sensor commands. For example, to issue a 'collect' command, you would say sensorData("collect").
The adaptor parses the input string, identifies the command(e.g., collect) and then calls Sensor.getdata() to collect the sensor value. It then returns the result(as a character string) to the 'Data Collector' component. This interface style means that the data collector can interact with different types of sensor.
The 'sensor' itself has separate operations such as 'start', 'stop', and 'getdata'.

Photolibrary composition



The above discussion of component composition assumes you can tell from the component documentation whether or not interfaces are compatible.
Of course, the interface definition includes the operation name and parameter types, so you can make some assessment of the compatibility from this.
To illustrate this problem, consider the composition shown in the figure:

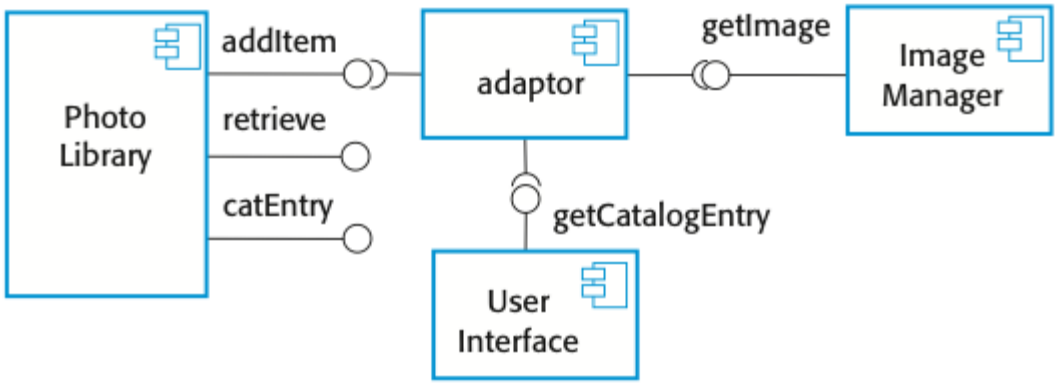
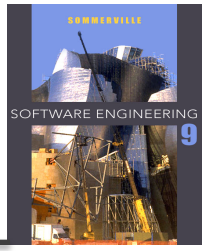


Photo Library Composition

These components are used to implement a system that downloads images from a digital camera and stores them in a photograph library.

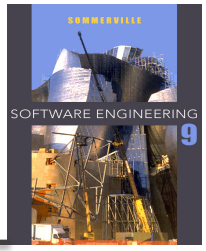
Interface semantics



- You have to rely on component documentation to decide if interfaces that are syntactically compatible are actually compatible.
- Consider an interface(methods in the inteface) 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



Assume that the documentation for the 'addItem' method is:

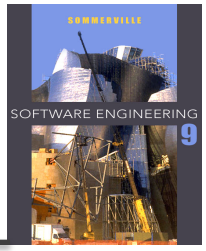
"This method adds a photograph to the library and associates the photograph identifier and catalogue descriptor with the photograph."

This description appears to explain what the component does, but consider the following questions:

- *"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?"*

There is not enough information in the informal description of 'addItem' to answer these questions. So some information based on Object Constraint Language(OCL) is added to the informal description. See next slide.

The Object Constraint Language

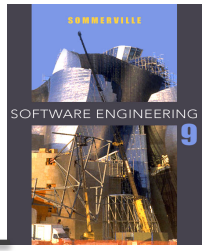


- The Object Constraint Language (OCL) has been designed to define constraints that are associated with UML models.
- It is based around the notion of pre and post condition specification – common to many formal methods.
- OCL allows you to express predicates that must always be true, that must be true before a method has executed; and that must be true after a method has executed. These are invariants, pre-conditions, and post-conditions.
- To access the value of a variable before an operation, you add @pre after its name. Therefore, using ‘age’ as an example:

  age=age@pre+1

  This statement means that the value of ‘age’ after an operation is one more than it was before that operation.

The OCL description of the Photo Library interface



The following includes a specification for the *addItem* and *delete* methods in Photo Library:

-- 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

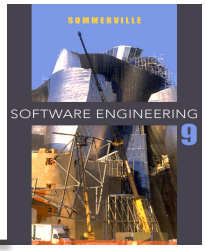
context delete

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

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

The next slide shows more details about the conditions.

Photo library conditions



- As specified, the OCL associated with the Photo Library component states that(conditions for 'addItem' state):

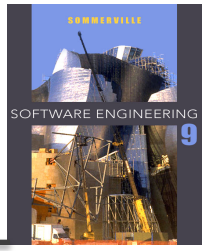
Pre-conditions:

- 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;

Post-conditions:

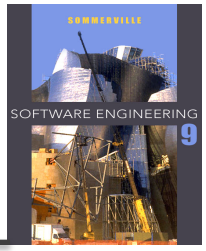
- Each new entry increases the size of the library by 1;
- If you retrieve using the same identifier then you get back the photo that you added;
- If you look up the catalogue using that identifier, then you get back the catalogue entry that you made.

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



Consider a situation such as that illustrated in the figure, where a system can be created through two alternative compositions:

In general, a good composition principle to follow is the principle of separation of concerns. That is, you should try to design your system in such a way that each component has a clearly defined role and that, ideally, these roles should not overlap.

The
perf
Con

(b) is perhaps faster and more reliable.

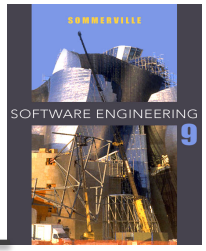
The ***advantages*** of composition (a) are that reporting and data management are separate, so there is more flexibility for future change. The data management system could be replaced and, if reports are required that the current reporting component cannot produce, that component can also be replaced without having to change the data management component.

In composition(b), a database component with built-in reporting
ss) is

osition(b)
nents,
entation
nent

les that apply to the database will also apply to reports. These reports will not be able to combine data in incorrect ways. *In composition(a), there are no such constraints so errors in reports could occur.*

Key points



- 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.