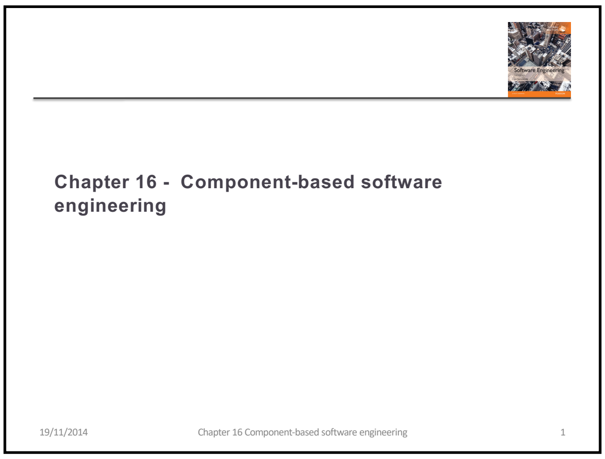
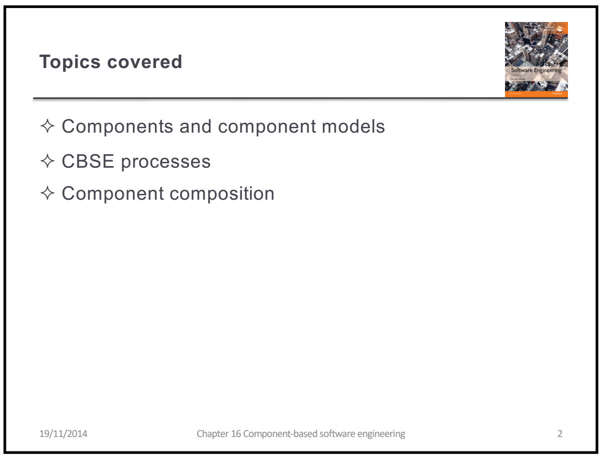


Chapter 16 - Component-based software engineering

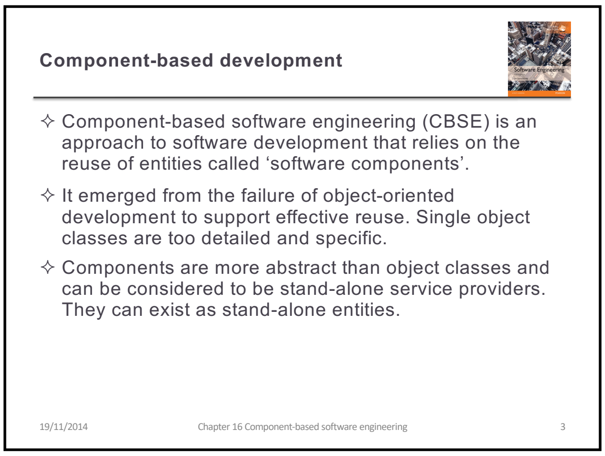
1



Topics covered

- ✧ Components and component models
- ✧ CBSE processes
- ✧ Component composition

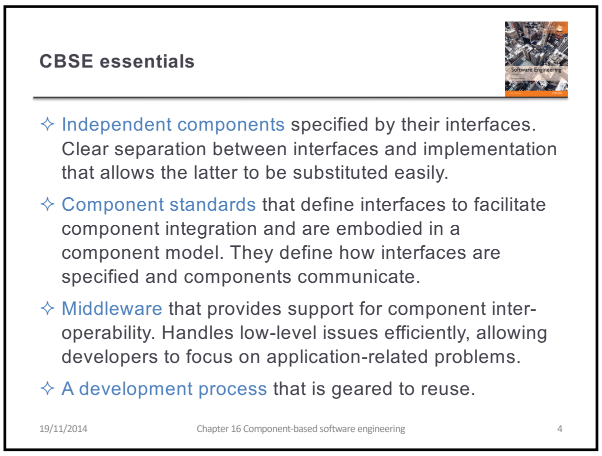
2



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.

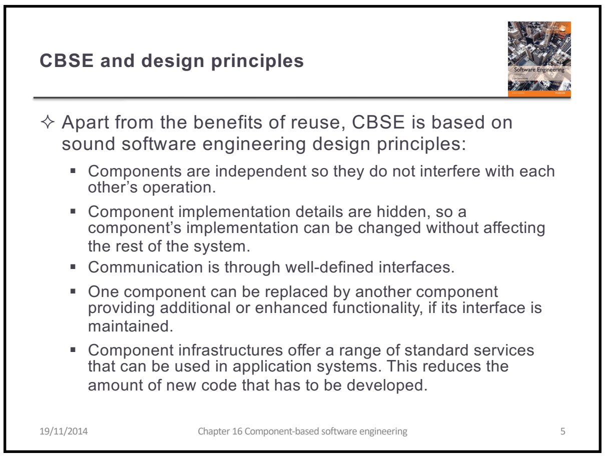
3



CBSE essentials

- ✧ Independent components specified by their interfaces. Clear separation between interfaces and implementation that allows the latter to be substituted easily.
- ✧ Component standards that define interfaces to facilitate component integration and are embodied in a component model. They define how interfaces are specified and components communicate.
- ✧ Middleware that provides support for component inter-operability. Handles low-level issues efficiently, allowing developers to focus on application-related problems.
- ✧ A development process that is geared to reuse.

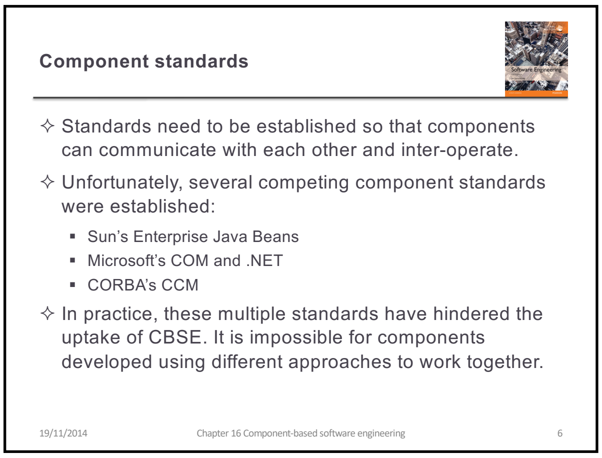
4



CBSE and design principles

- ✧ Apart from the benefits of reuse, CBSE is based on sound software engineering design principles:
  - Components are independent so they do not interfere with each other’s operation.
  - Component implementation details are hidden, so a component’s implementation can be changed without affecting the rest of the system.
  - Communication is through well-defined interfaces.
  - One component can be replaced by another component providing additional or enhanced functionality, if its interface is maintained.
  - Component infrastructures offer a range of standard services that can be used in application systems. This reduces the amount of new code that has to be developed.

5



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.

6

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.
- ✧ Covered in Chapter 18.

7

Components and component models

8

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.

9

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.*

10

Component characteristics

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.

11

Component characteristics

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.

12

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.

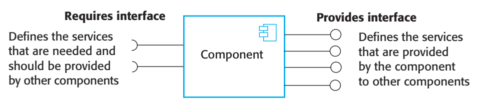
13

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.

14

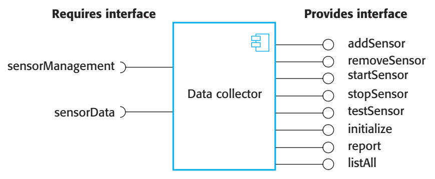
Component interfaces


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.

15

A model of a data collector component



16

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.
- ✧ Therefore it can be called in a similar way as a procedure or method running on a local computer.

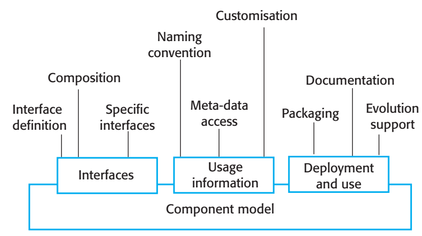
17

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.

18

Basic elements of a component model



19

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.

✧ For web services, interface specification uses XML-based languages.

✧ EJB uses Java as the interface definition language.

✧ .NET uses Microsoft Common Intermediate Language (CIL).

✧ 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.

20

Elements of a component model: 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.

✧ For example, in EJB a hierarchical name is generated with the root based on an Internet domain name. Services have a unique Uniform Resource Identifier (URI).

✧ Component meta-data is data about the component itself, such as information about its interfaces and attributes. The meta-data is important because it allows users of the component to find out what services are provided and required.

✧ Components are generic entities and when deployed they have to be configured to fit into an application system. For example, you could configure the `Data collector` component by defining the maximum number of sensors in a sensor array. The component model may therefore specify how the binary components can be customized for a particular deployment environment.

21

Elements of a component model: Deployment

✧ The component model includes a specification of how components should be packaged for deployment as independent, executable entities.

✧ Because components are independent entities, they have to be packaged with all supporting software that is not provided by the component infrastructure or is not defined in the "requires" interface.

✧ Deployment information includes information about the contents of a package and its binary organization.

✧ Inevitably, as new requirements emerge, components will have to be changed or replaced. The component model may therefore include rules governing when and how component replacement is allowed.

✧ Finally, the component model may define the component documentation that should be produced. This is used to find the component and to decide whether it is appropriate.

22

Middleware support

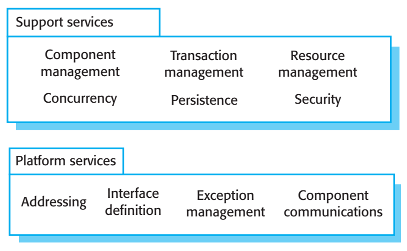
✧ Component models are the basis for middleware that provides support for executing components.

✧ Component model implementations provide:
- Platform services that allow components written according to the model to communicate in a distributed environment and which are fundamental and must be available in all component-based systems;
- Support services that are standard type of application-independent services (e.g. authentication) used by different components, thus reducing component development costs and potential component incompatibility can be avoided.

✧ 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.

23

Middleware services defined in a component model



24

CBSE processes

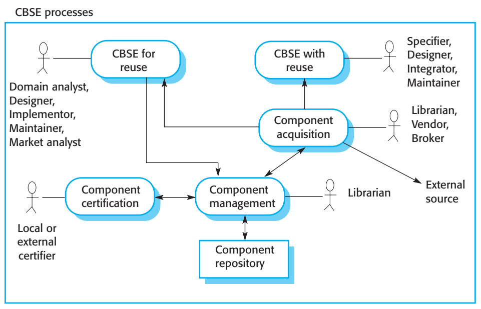
25

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

26

CBSE processes


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

27

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.

28

CBSE for reuse

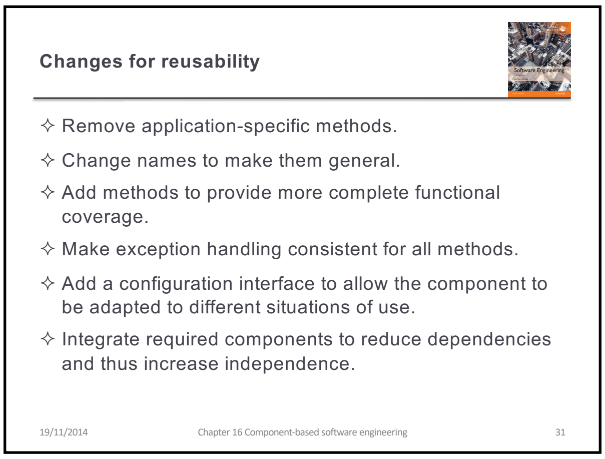
- ✧ CBSE for reuse focuses on component development.
- ✧ Components developed for a specific application usually have to be generalized to make them reusable.
- ✧ A component is most likely to be reusable if it is associated with a stable domain abstraction (business object).
- ✧ For example, in a hospital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

29

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.

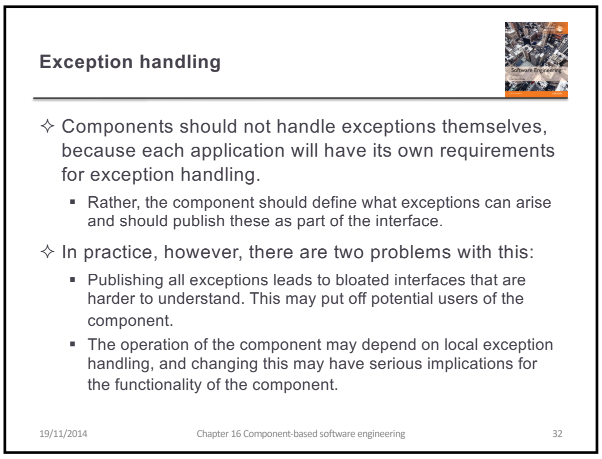
30



Changes for reusability

- ✧ Remove application-specific methods.
- ✧ Change names to make them general.
- ✧ Add methods to provide more complete functional coverage.
- ✧ Make exception handling consistent for all methods.
- ✧ Add a configuration interface to allow the component to be adapted to different situations of use.
- ✧ Integrate required components to reduce dependencies and thus increase independence.

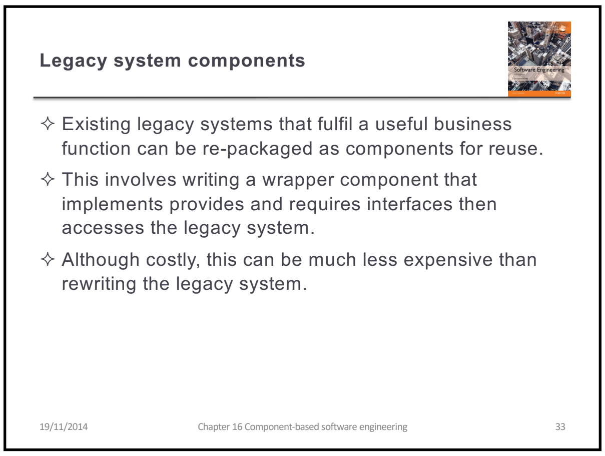
31



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.

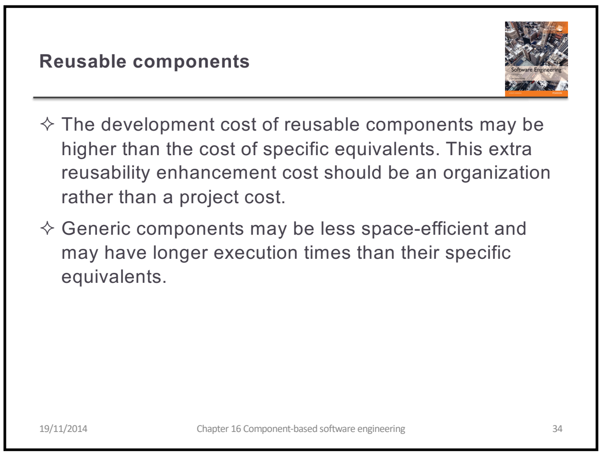
32



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.

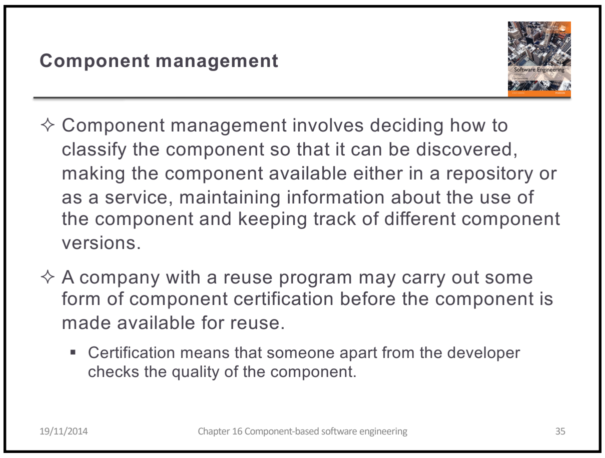
33



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.

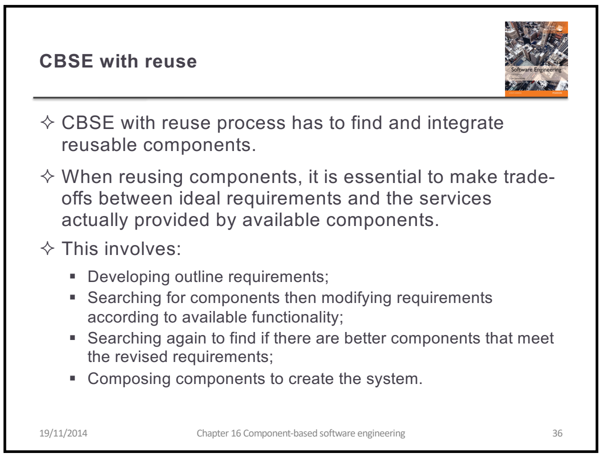
34



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.

35

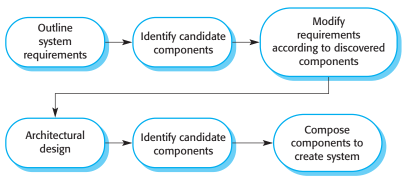


CBSE with reuse

- ✧ 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.
- ✧ 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.

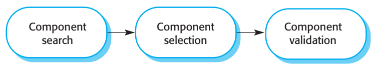
36

CBSE with reuse


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

37

The component identification process



38

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?

39

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.

40

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.

41

Component composition

42

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.

43

Types of composition

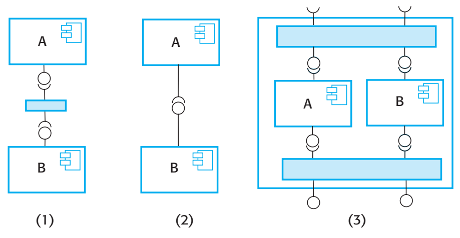
✧ Sequential composition (1) where the composed components are executed in sequence. This involves composing the provides interfaces of each component.

✧ Hierarchical composition (2) 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 (3) 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.

44

Types of component composition



45

Glue code

✧ Code that allows components to work together.

✧ If A and B are composed sequentially, then glue code has to call A, collect its results then call B using these results, transforming them into the format required by B.

✧ Glue code may be used to resolve interface incompatibilities.

46

Interface incompatibility

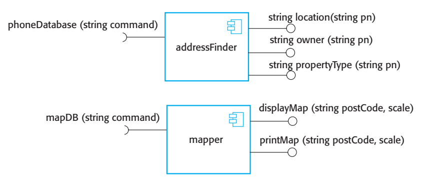
✧ Parameter incompatibility where operations have the same name but are of different types.

✧ 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 or vice versa.

47

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)

48

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.

49

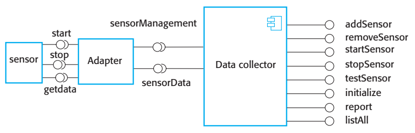
Composition through an adaptor

- ✧ 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)
```

50

An adaptor linking a data collector and a sensor



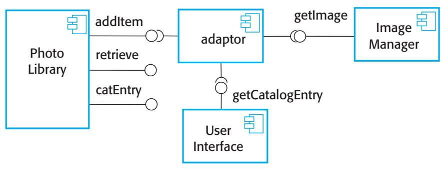
51

Reconciling `sensor` and `Data collector` components

- ✧ Here the adaptor reconciles the “requires” interface of the data collector component with the “provides” interface of the sensor component.
- ✧ To issue a collect command, `Data collector` sends the message `sensorData(“collect”)`.
- ✧ The adaptor parses the input string, identifies the command (`collect`), and then calls `Sensor.getdata` to collect the sensor value.
- ✧ It then returns the result (as character string) to the data collector component.
- ✧ A separate adaptor can be used for each sensor device.

52

Photo library composition



53

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) ;
```

54

Photo Library documentation

"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?"

55

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.

56

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
```

57

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.
- To delete an item, it must exist and after deletion the photo can no longer be retrieved and the size of the library is reduced by 1.

58

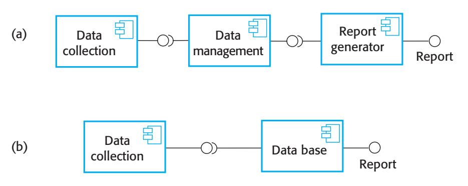
Composition trade-offs

✧ When composing components, you may find conflicts between functional and non-functional requirements, and conflicts between the need for rapid delivery and system evolution.

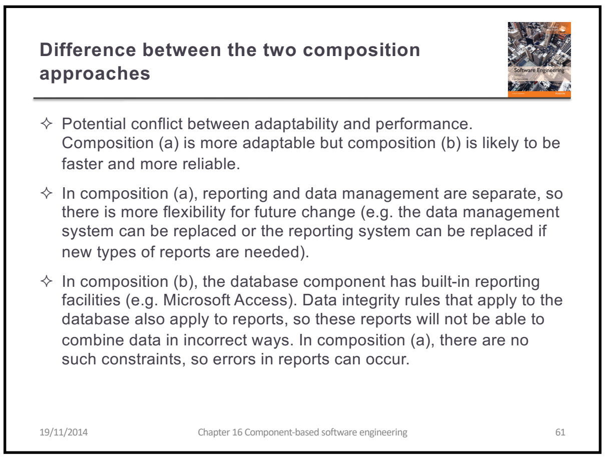
✧ You need to make decisions such as:
- What composition of components is more effective for delivering the functional requirements for the system?
- What composition of components will make it easier to adapt the composite component when its requirements change?
- What will be the emergent properties of the composed system? These properties include performance and dependability. You can only assess these properties once the complete system is implemented.

59

Data collection and report generation components



60

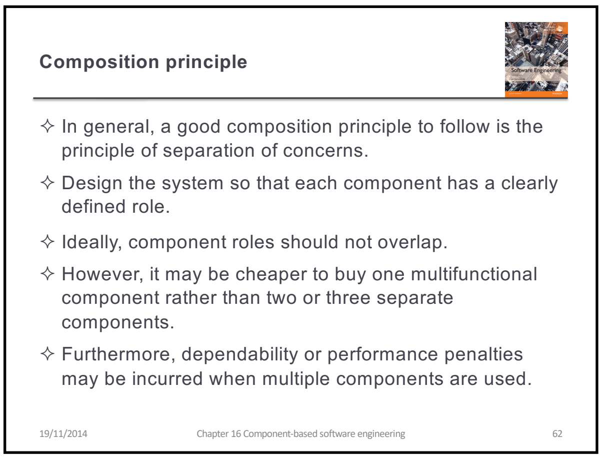


Difference between the two composition approaches

- ✧ Potential conflict between adaptability and performance. Composition (a) is more adaptable but composition (b) is likely to be faster and more reliable.
- ✧ In composition (a), reporting and data management are separate, so there is more flexibility for future change (e.g. the data management system can be replaced or the reporting system can be replaced if new types of reports are needed).
- ✧ In composition (b), the database component has built-in reporting facilities (e.g. Microsoft Access). Data integrity rules that apply to the database also apply to reports, so these reports will not be able to combine data in incorrect ways. In composition (a), there are no such constraints, so errors in reports can occur.

19/11/2014 Chapter 16 Component-based software engineering 61

61

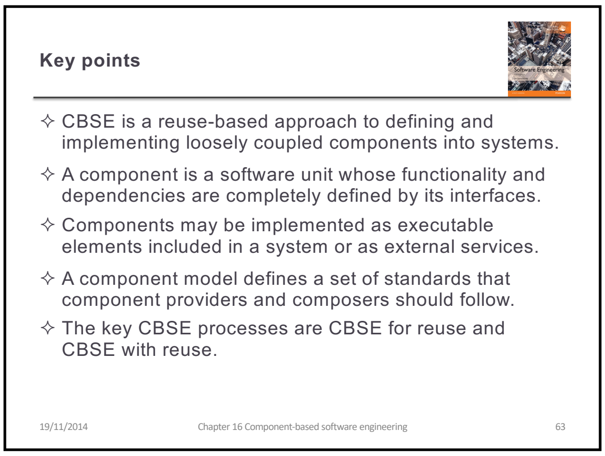


Composition principle

- ✧ In general, a good composition principle to follow is the principle of separation of concerns.
- ✧ Design the system so that each component has a clearly defined role.
- ✧ Ideally, component roles should not overlap.
- ✧ However, it may be cheaper to buy one multifunctional component rather than two or three separate components.
- ✧ Furthermore, dependability or performance penalties may be incurred when multiple components are used.

19/11/2014 Chapter 16 Component-based software engineering 62

62

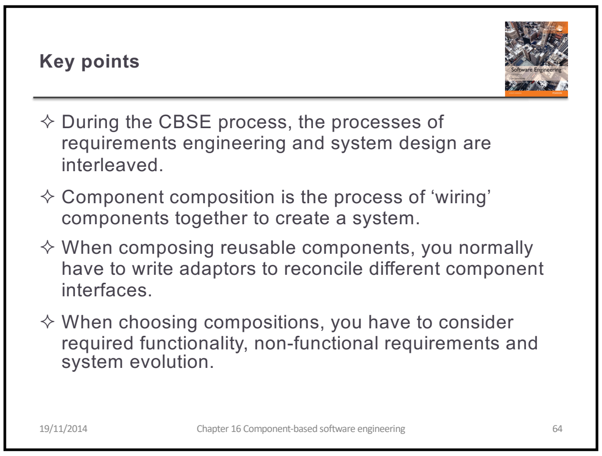


Key points

- ✧ 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.
- ✧ 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.

19/11/2014 Chapter 16 Component-based software engineering 63

63



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.

19/11/2014 Chapter 16 Component-based software engineering 64

64