Systems Integration:

Component-based software engineering
Objectives

• To explain that CBSE is concerned with developing standardised components and composing these into applications
• To describe components and component models
• To show the principal activities in the CBSE process
• To discuss approaches to component composition and problems that may arise
Topics covered

• Components and component models
• The CBSE process
• Component composition
Component-based development

• Component-based software engineering (CBSE) is an approach to software development that relies on software reuse.

• It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific.

• Components are more abstract than object classes and can be considered to be stand-alone service providers.
CBSE essentials

- Independent components specified by their interfaces.
- Component standards to facilitate component integration.
- Middleware that provides support for component interoperability.
- 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 platforms are shared and reduce development costs.
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?
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 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.
Component characteristics 1

| Standardised | Component standardisation means that a component that is used in a CBSE process has to conform to some standardised component model. This model may define component interfaces, component meta-data, documentation, composition and deployment. |
| Independent | A component should be independent so 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 2

<table>
<thead>
<tr>
<th>Deployable</th>
<th>To be deployable, a component has to be self-contained and must be able to operate as a stand-alone entity on some component platform that implements the component model. This usually means that the component is a binary component that does not have to be compiled before it is deployed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documented</td>
<td>Components have to be fully documented so that potential users of the component can decide whether or not they meet their needs. The syntax and, ideally, the semantics of all component interfaces have to be specified.</td>
</tr>
</tbody>
</table>
Component interfaces

• Provides interface
  - Defines the services that are provided by the component to other components.

• Requires interface
  - Defines the services that must be made available for the component to execute as specified.
Component Integration Problems

- Example: Mars Lander

What is necessary for “correct” component interaction?
Component interfaces

- **Requires interface**: Defines the services from the component’s environment that it uses.
- **Provides interface**: Defines the services that are provided by the component to other components.
Provides/requires interfaces

- These interfaces are NOT the same as input/output interfaces.
- Both of these interfaces define both the inputs needed and the outputs produced by the provided and required services.
- Requires
  - You can think of this as defining the methods that are called by a component or before a component executes.
- Provides
  - You can think of this as defining how the component is called.
A data collector component

Requires interface

sensorManagement

sensorData

Provides interface

addSensor
removeSensor
startSensor
stopSensor
testSensor
initialise
report
listAll
Components and objects

- Components are deployable entities.
- Components do not define types.
- Component implementations are opaque.
- Components are language-independent.
- Components are standardised.
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.
Elements of a component model

- Composition
  - Interface definition
  - Specific interfaces
- Customisation
  - Naming convention
  - Meta-data access
- Documentation
  - Packaging
  - Evolution support
- Interfaces
- Usage information
- Deployment and use

Component model
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;
  - Horizontal services that are application-independent services used by different components.

• 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.
Component model services

Horizontal services
- Component management
- Concurrency
- Transaction management
- Persistence
- Resource management
- Security

Platform services
- Addressing
- Interface definition
- Exception management
- Component communications
Component development for reuse

- Components are rarely created by simply using part of the code of another application system.
- Rather, components for reuse have to be specifically developed so that they are reusable across a range of applications.
Component development for reuse

- Components developed for a specific application usually have to be generalised 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.
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

- 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.
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.
- We will return to issues of legacy system reuse in the discussion of service-oriented systems.
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.
CBSE Processes

- As discussed, conventional software engineering processes have to be adapted for reuse.
The CBSE process

- 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.
The CBSE process

Outline system requirements → Identify candidate components

Architectural design → Identify candidate components

Modify requirements according to discovered components → Compose components to create system
The component identification process

Component search \leadsto \text{Component selection} \leadsto Component validation
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?
Component composition

- Designing a system by integrating a number of components.
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.
Component Integration

- Integrating components can be illustrated as a mechanical process of “wiring” components together to form assemblies.
- Standardization in form of component models like EJB, CORBA and COM.
- Still Difficult to make components play well together.
Component Integration (2)

Architectural mismatch stems from mismatched assumptions a reusable part makes about the structure of the system it is to be part of. These assumptions often conflict with the assumptions of other parts and are almost always implicit, making them extremely difficult to analyze before building the system.”

Component Integration (3)

- 2 Cases
    - AESOP
  - P. Inverardi, A.L. Wolf, and D. Yankelevich, Static Checking of System Behaviors Using Derived Component Assumptions
    - Compressing proxy
Component Integration (4)

- four classes of structural assumptions
  - The nature of components (infrastructure, control model, and data model)
  - The nature of connectors (protocols and data models)
  - The architecture of the assemblies (constraints on interactions)
  - The run-time construction process (order of instantiations).
Component Integration (5)

- Process
- Component
- Channel

Function call interface
UNIX pipe interface

Compressing Proxy

Filter
Pseudo Filter (Adaptor)
gzip
Filter

1 2 3 4
Component Integration (6)

- Formal architectural description and analysis to uncover what they call “behavioral mismatch”
- Not component mismatch
- Components must express assumptions made about it’s environment such as data formats or buffer sizes
- Careful analysis of the assumptions components make about the context in which they are to be used can help prevent design errors and result in building the right product faster
From Integration to Composition

- All assemblies are potential subsystems
- Predicting the emergent behavior of assemblies
- The result of component composition is a component assembly which can be used as a part of a larger composition
- Composition goes beyond integration by allowing prediction of the emergent behavior of assemblies
Predictable Assembly from Certifiable Components

• What types of system quality attributes are developers interested in predicting?
• What types of analysis techniques support reasoning about these quality attributes, and what component property values do they require as input parameters?
• How are these component properties specified, measured, and certified?
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.
Types of composition

(a) 

(b) 

(c)
Sequential composition

- In this case, the components are executed in sequence to provide some required effect.
- The outputs from the ‘provides’ interface from the first component executed become the inputs for the provides interface for the 2nd unit called (perhaps with some modification through an adapter component).
- Each component is executed independently and does not have to be aware of the other components in the sequence.
Hierarchical composition

- In this case, one component (defined in the requires interface) is called directly from within the body of the other component.
- The calling component must know the name and the interface signature of the called component.
Additive composition

- In this case, we put two components together so that the provides interface includes operations that come from both of the composed components.
- Essentially, this is normally implemented by defining a new small component that offers the combined interface and which than calls one of the composed components, depending on the call to the composed component.
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.
Incompatible components

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)
Incompatibility

- The component addressFinder through its location method produces a string which is the address of the property, including street number and name and town.
- The component mapper through its displayMap method, expects a string which is a postcode only (not a complete address).
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 (called postCodeStripper) 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.

```plaintext
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000)
```
Adaptor for data collector
Adaptor functionality

- The data collector component requires a sensorManagement component that provides facilities to stop and start sensors and to query sensors for data.
- The adapter component in this case turns a string of sensor management commands e.g. sensor.stop into the commands required for a specific sensor device.
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:

```java
public void addItem (Identifier pid ; Photograph p; CatalogEntry photodesc) ;
public Photograph retrieve (Identifier pid) ;
public CatalogEntry catEntry (Identifier pid) ;
```
Photo library composition

- Photo Library
  - addItem
  - retrieve
  - catEntry

- adaptor
  - getCatalogEntry

- User Interface
  - getImage

- Image Manager
“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.
- It is based around the notion of pre and post condition specification - similar to the approach used in System Z.
Formal description of 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
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
Composition trade-offs

- For composition (a), reporting and data management are separate so there is more flexibility if changes in one of these functions but not the other have to be made.
- For composition (b), there are fewer discrete components so faster communications within the component offering both data management and report generation.
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.
- During the CBSE process, the processes of requirements engineering and system design are interleaved.
Key points

• 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.