Software Reuse and Component-Based SE

ITSE422

Lecture #4: CBSE Processes & Component Composition & Component Specification I

Main References

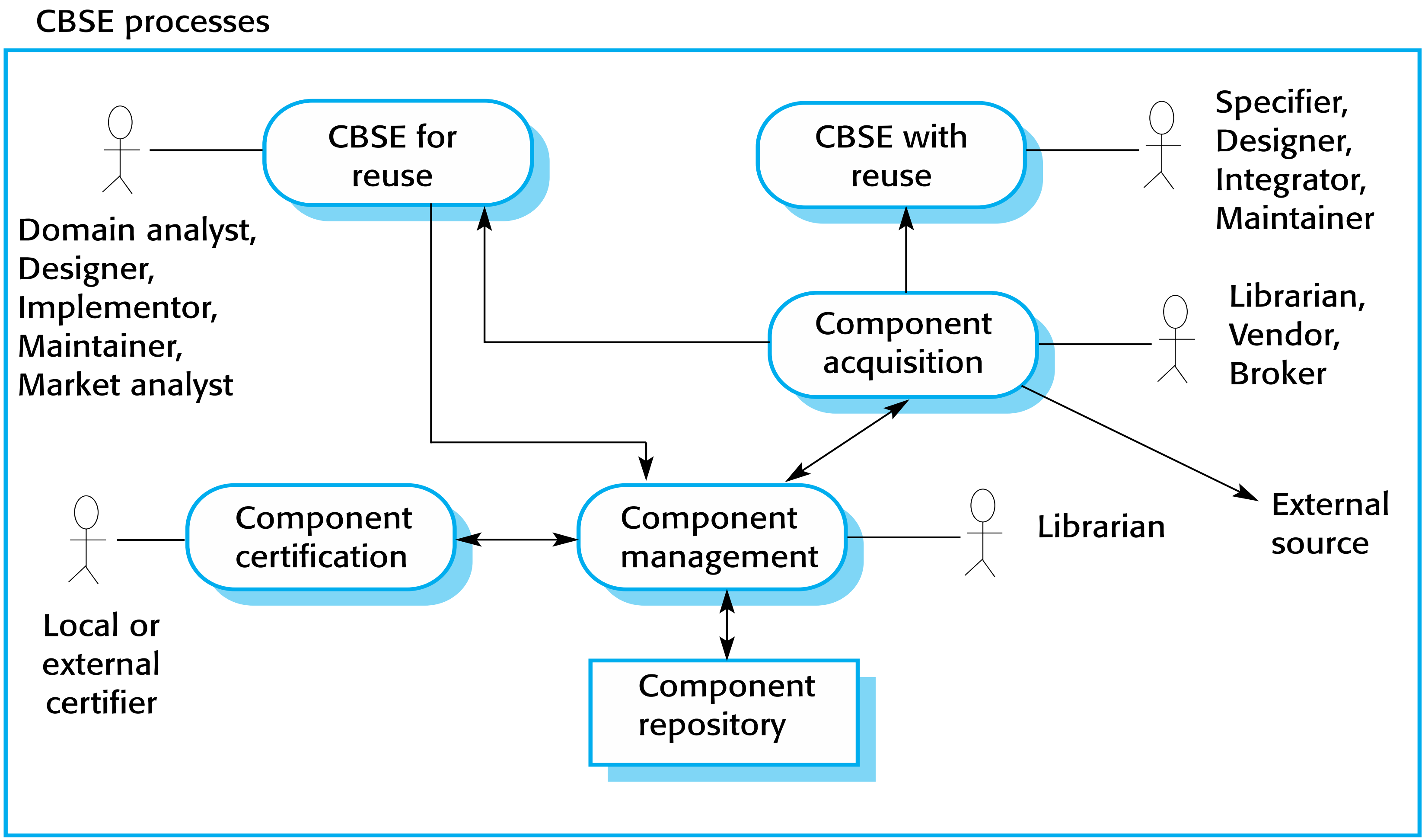
- Ian Sommerville, *Software Engineering*, 8th edition, chapter 19.1 (*Components and component models*)
- Ivica Crnkovic, Magnus Larsson. *Building reliable component based software systems*, Artech House, 2002.
- Roger S. Pressman, Software Engineering: A Practitioner's Approach, Eighth Edition, McGraw-Hill Higher Education, 2015

CBSE processes

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.

CBSE processes



Supporting processes

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

CBSE for reuse

- CBSE for reuse focuses on component development.
- 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

- 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.

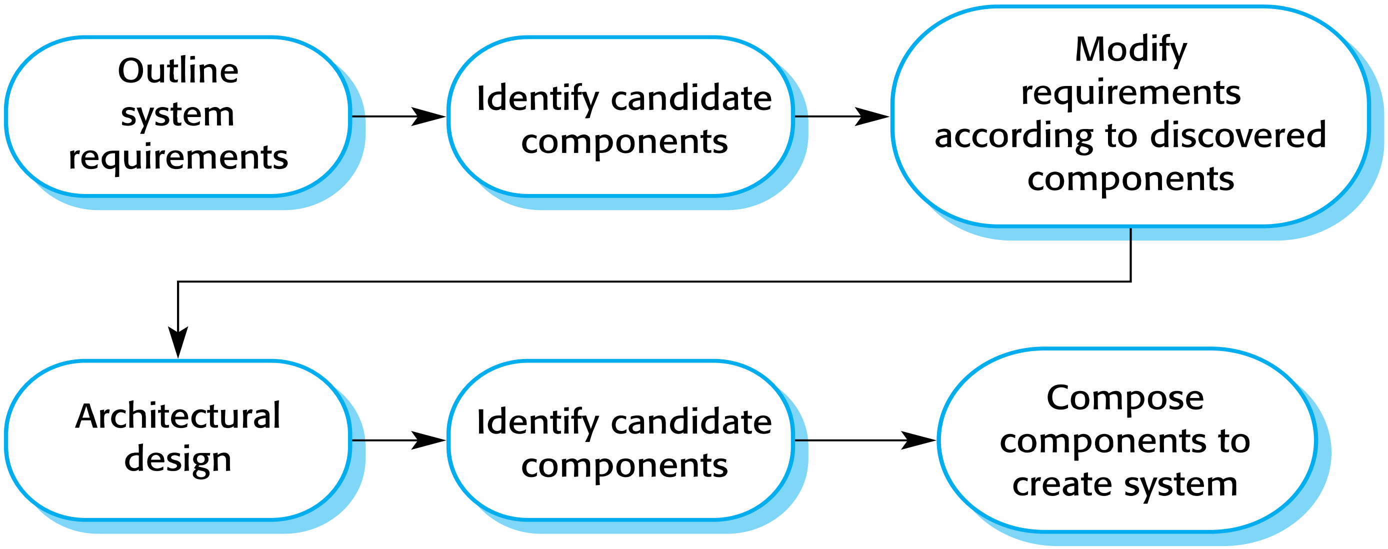
Component management

- 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.

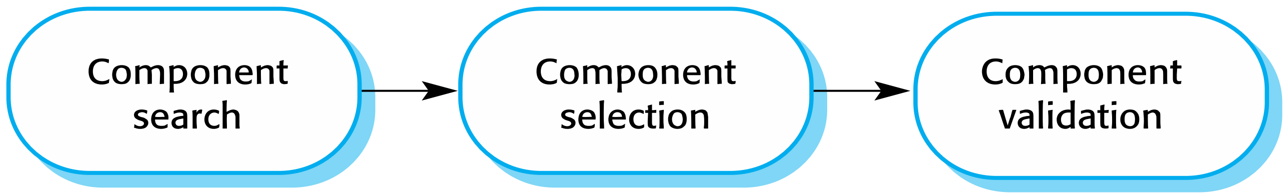
CBSE with reuse

- 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.

CBSE with reuse



The component identification process



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 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.

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.

Component composition

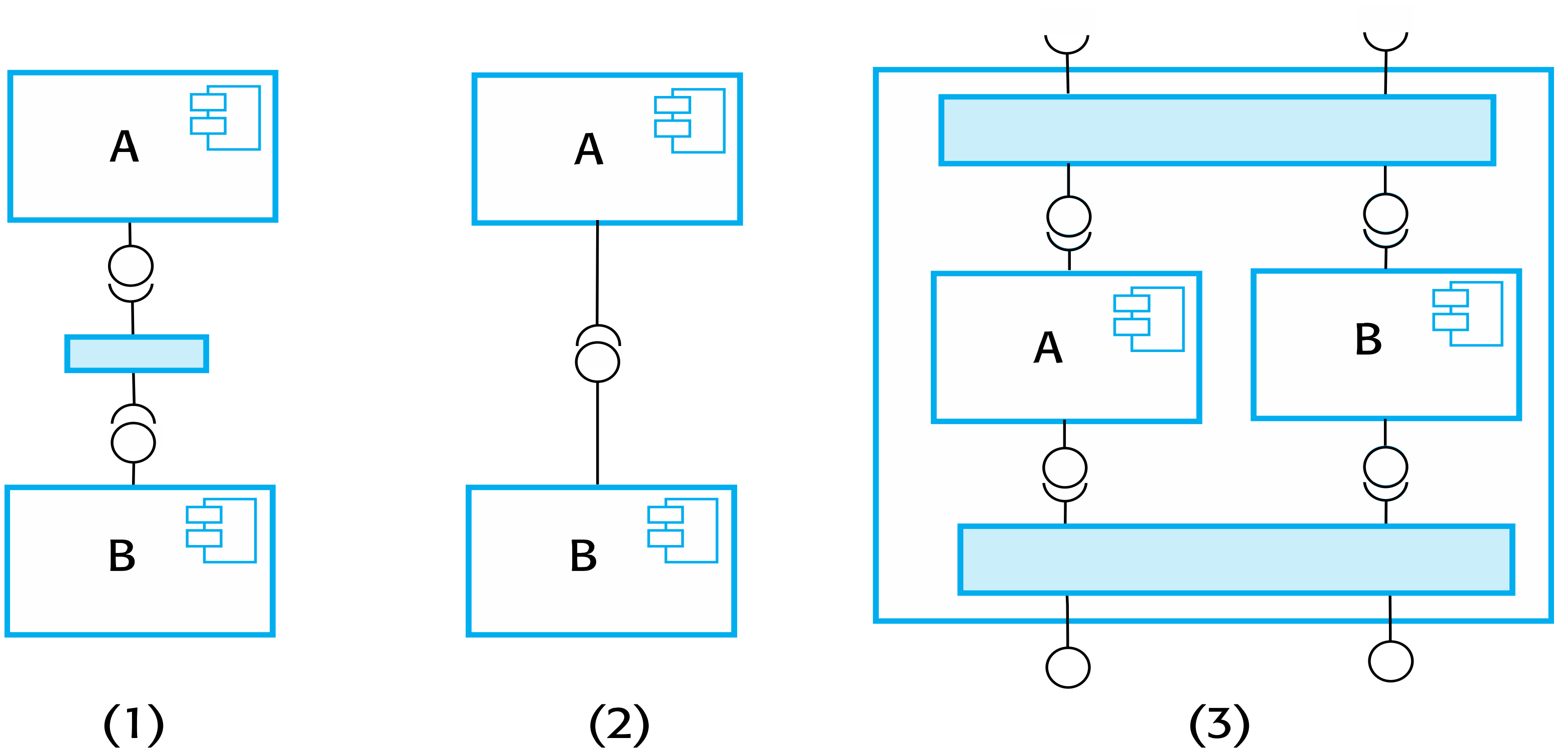
Component composition

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

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.

Types of component composition



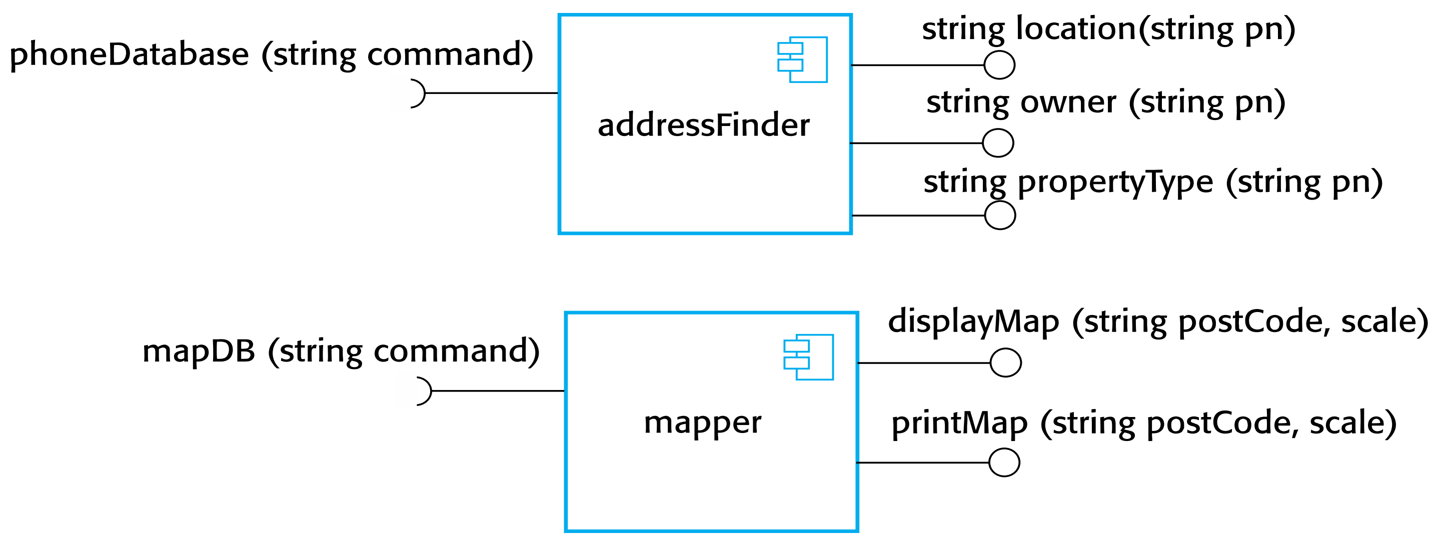
Glue code

- 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.

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.

Components with incompatible interfaces



Adaptor components

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

Composition through an adaptor

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

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

An adaptor linking a data collector and a sensor

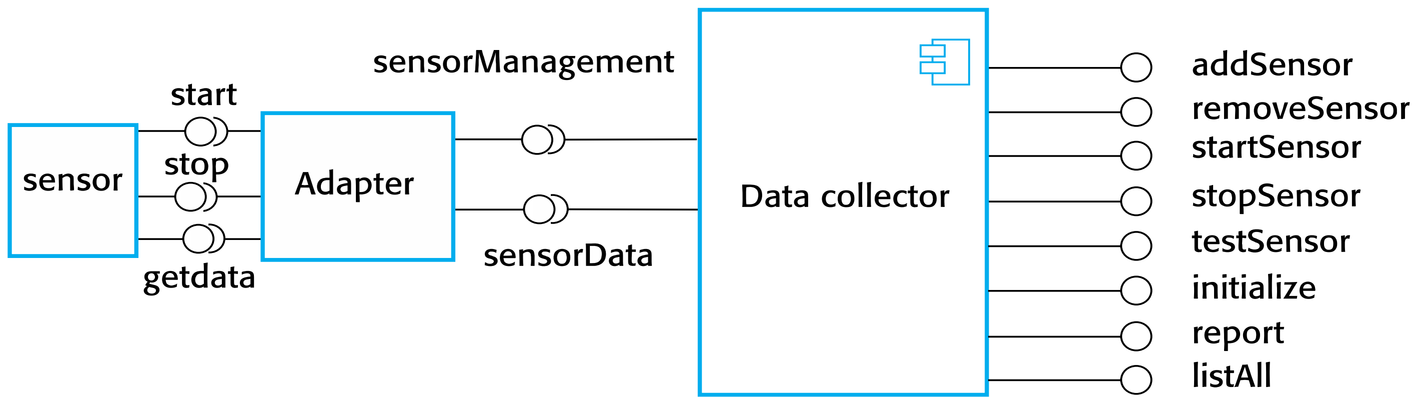
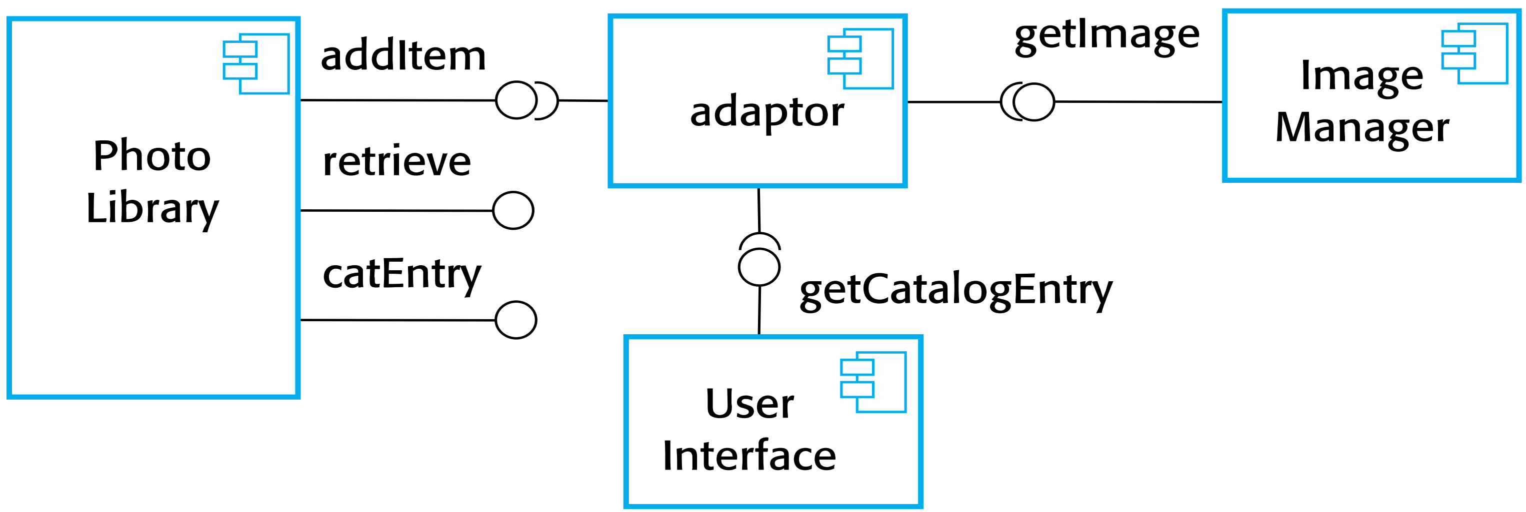


Photo library composition



Interface semantics

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

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

Photo Library documentation

"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.

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[em]1

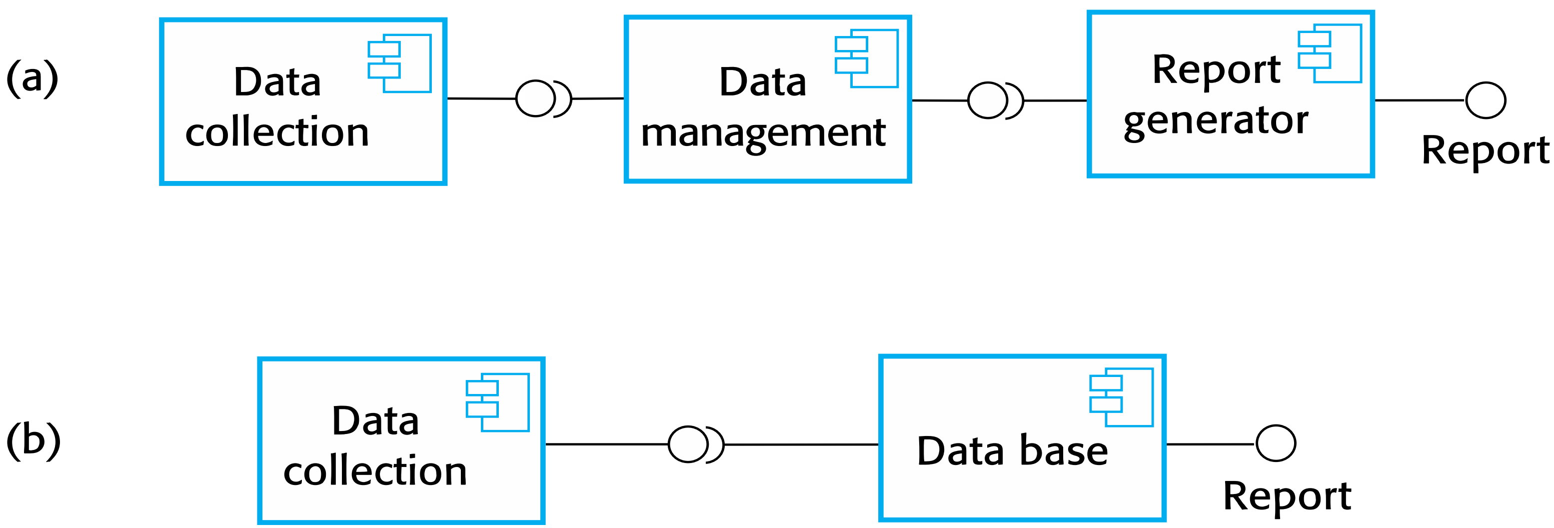
Photo library conditions

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

Composition trade-offs

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

Data collection and report generation components



- Here, there is a potential conflict between adaptability and performance.
- Composition (a) is more adaptable but composition (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.
- In composition (b), a database component with built-in reporting facilities (e.g., Microsoft Access) is used. The key advantage of composition (b) is that there are fewer components.
- Furthermore, data integrity rules that apply to the database will also apply to reports
- In general, a good composition principle to follow is the principle of separation of concerns

Component Specification I

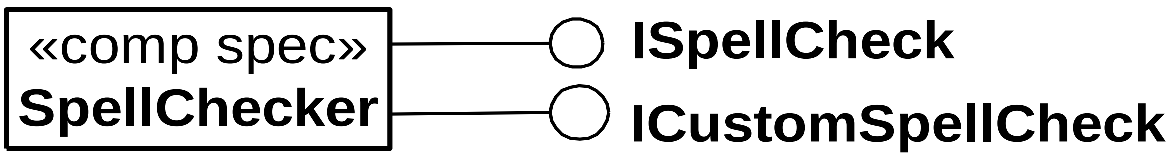
Component specification

- There should be no difference between:
  - What a component does
  - What we *know* it does
- The only way we get to know what a component does is from its *component specification*
- Levels of a component specification:
  - Syntax: includes specifications on the programming language level.
  - Semantic: functional contracts

syntactic specification

- All component models use syntactic specification of interfaces:
  - Programming language
  - IDL
- Examples
  - Microsoft's Component Object Model (COM)
  - Common Object Request Broker Architecture (CORBA)
  - JavaBeans

Example: component SpellChecker



Implementation as a COM (Component Object Model) component:

- Uses an IDL (Interface Description Language)

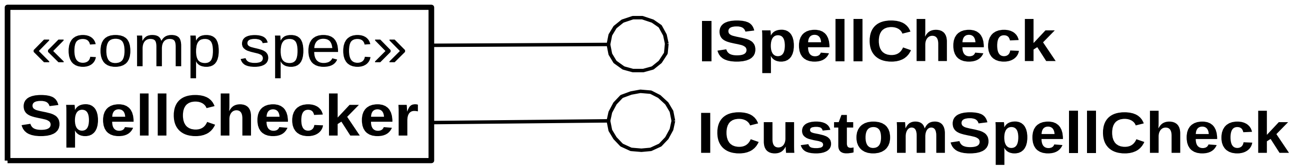
IDL (Interface Description Language) Example

```
interface ISpellCheck : IUnknown
{
    HRESULT check([in] BSTR *word, [out] bool *correct);
};

interface ICustomSpellCheck : IUnknown
{
    HRESULT add([in] BSTR *word);
    HRESULT remove([in] BSTR *word);
};

library SpellCheckerLib
{
    coclass SpellChecker
    {
            [default] interface ISpellCheck;
            interface ICustomSpellCheck;
    };
};
```

Semantic Specification

- Tool support for component developers
- Tool support for developers of component-based applications

Example: SpellChecker component



Example: OCL Interface Specification

```
context ISpellCheck::check(in word : String, out correct :       Boolean):
HRESULT
pre:
word <> “”
post:
SUCCEEDED(result) implies correct = words->includes(word)

context ICustomSpellCheck::add(in word : String) : HRESULT
pre:
word <> “”
post:
SUCCEEDED(result) implies  words = words@pre->including (word)

context ICustomSpellCheck::remove(in word : String) : HRESULT
pre:
word <> “”
post:
SUCCEEDED(result) implies words = words@pre->exluding(word)
```

- Questions?