Component-Based Software Engineering

- **Main concerns:**
  - Development of software from pre-produced parts
  - Reuse of those parts in other applications
  - Easily maintaining and customizing those parts to produce new features

- **Goals (users’ demands)**
  - Build more reliable software
  - Less time between versions
Component Everywhere

- Concerns and goals are similar in many other engineering disciplines
  - Precise assembly of reusable, well-documented, quality and trusted parts

- For them components are well established
  - Civil engineering -> house prefab, …
  - Chemical engineering -> proteins, …
  - Electronic engineering -> circuit, …
  - Industrial engineering -> cars, …

But why not in Software engineering?

- If the goals of CBSE are no different from industrial and civil engineering why is CBS an hype only now?

- Industrial and civil engineering develop final products

- Software is a generic meta-product that can be used to create families of instances
  - using different parameters
Standards

- Industrial and civil engineering successfully develop components because of:
  - Standards
  - Regulations
  - Laws

- Before the software crisis (1968) software had no standard.

Standard needed

- **Interaction standard** for specifying the explicit context dependency on other software elements

- **Composition standard** for defining how components can be composed to create a larger structure and how a producer can replace one existing component with another one.
What is a Component?

- We can find several definitions of a component in literature, however everyone agrees that a component is a piece of software.

- These definitions begin from the consideration of CBSE from different viewpoints and focus on different aspects of software engineering such as:
  - Different phases (design, implementation and run-time phases)
  - Business aspects
  - Architectural issues

The Component: Szyperski

- Szyperski defines a component precisely by enumerating its characteristic properties as follows:

  - 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 party.
Implications of Szyperski’s Definition

- The following implications arise as a result of Szyperski’s definition:
  - For a component to be deployed independently, a clear distinction from its environment and other components is required.
  - A component must have clearly specified interfaces.
  - The implementation must be encapsulated in the component and is not directly reachable from the environment.

The Component: D'Souza and Wills

- D'Souza and Wills define a component as:
  - A reusable part of software, which is independently developed, and can be brought together with other components to build larger units. It may be adapted but may not be modified.
  - A component can be, for example, a compiled code without a program source, or a part of a model and/or design.
Objects and Components

- Szyperski
  - View a component as a collection of objects, in which the objects co-operate with each other, and are intertwined tightly.

- D’Souza and Wills
  - Assert that if a class were packaged together with the explicitly defined interfaces which it requires and implements, then this class would be a component.

Describing a Component

- To be able to describe a component completely the component should consist of the following elements:
  
  - A set of interfaces provided to, or required from the environment.

  - An executable code, which can be coupled to the code of other components via interfaces.
Improving a Component

- To improve the component quality, the following elements can be included in the specification of a component:
  - The specification of non-functional characteristics
  - The validation code
  - Additional information

Interfaces

- An interface of a component can be defined as a specification of its access point, offering no implementation for any of its operations.

- This separation makes it possible to:
  - Replace the implementation part without changing the interface
  - Add new interfaces (and implementations) without changing the existing implementation
Describing an Interface

- Interfaces defined in standard component technologies using techniques such as Interface Definition Language (IDL) are:
  - Sufficient in describing functional properties
  - Insufficient in describing extra-functional properties such as quality attributes like:
    - Accuracy
    - Availability
    - Latency
    - Security
    - etc

Export and Import Interfaces

- We can distinguish two kinds of interfaces. Components can export/import interfaces to/from environments which may include other components.
  - An exported interface describes the services provided by a component to the environment.
  - An imported interface specifies the services required by a component from the environment.
Contracts

- A more accurate specification of a component's behavior can be achieved through contracts.

- A contract is comprised of:
  - The Invariant, the global constraints which the component will maintain
  - The Pre-condition, the constraints which need to be met by the client
  - The Post-condition, the constraints which the component promises to establish in return

Component Interaction

- A contract specifies the interactions among components, in terms of:
  - The set of participating components
  - The role of each component through its contractual obligations, such as type and casual obligations
  - The invariant to be maintained by the components
  - The specification of the methods which instantiate the contract.
Patterns

- Patterns define recurring solutions to recurring problems capturing non-obvious solutions, not just abstract principles or strategies.

  - The solutions should be proven to solve the problem rather than being theories or speculations.

  - Patterns describe relationships between deeper system structures and mechanism.

  - A component, as a reusable entity, can be seen as a realization of some design pattern.

Categories of Patterns

- Patterns can be classified into three major categories:
  - *Architectural Patterns*, capture the overall structure and organization of a software system.
  - *Design Patterns*, refine the structure and the behavior of the subsystems as well as the components of a software system, and the relationships which exist between them.
  - *Idioms*, are low-level patterns which are dependent on the chosen paradigm and the programming language used.
Design Patterns

- Design patterns are *widely used in the process of designing component-based systems* in which the reusable units must be identified.

- By using design patterns, it is easier to *recognize those reusable parts* and either find them in the form of pre-existing components, or develop them as reusable units.

Software Frameworks

- CBSE means that we build software by "*putting pieces together*".

- *Frameworks provide the context in which the pieces can be used.*

- A framework may be seen as:
  - A *reusable design* of a system
  - A *skeleton of an application* which can be customized by an application developer
Component Frameworks

- Frameworks in general describe a typical and reusable situation at a model level.

- A component framework describes a “circuit-board” with empty slots into which components can be inserted to create a working instance.

![Component Framework Diagram]

Component Models

- The two concepts Component Models and Component Frameworks are sometimes intermixed.

- A component model defines a set of standards and conventions used by the component developer.

- A component framework is a support infrastructure for the component model.
Relationships Between Concepts

Interface that satisfies contracts

Frameworks and Contracts

- Frameworks focus on the *overall properties* of component compositions.

- Contracts give *specifications for relationships between concrete components*.

  - These specifications may be different for components within one composition.
Frameworks and Patterns

- It is important to realize that design patterns and frameworks are distinct concepts of different natures.
  
  - *Design patterns* are of a *logical nature*, representing knowledge of and experience gained with software.
  
  - *Frameworks* are of a *physical nature*, and are executable software used in either the design or the runtime phase.

The major differences between design patterns and frameworks are as follows:

- The *level of abstraction* of frameworks and design patterns

- Design patterns are *smaller architectural elements* than frameworks

- The *level of specialization* of frameworks and design patterns.
Concepts: Summary (1)

- Component specifications are essential for component users who are focused on the component features, functional and non-functional.

- The main purpose of frameworks is to support the process of component composition.

- Component developers must obey the rules and formats specified by the framework to develop and to specify the component.

- Component users will use frameworks to compose systems from components in a more efficient and accurate way.

Concepts: Summary (2)

- Patterns give an abstract and more general view of a function, procedure or similar.

- Patterns can be implemented in the form of systems or components.

- Component designers will use patterns in the design process to design components more efficiently.
A Component Comprised of?

- Some code...
  - The code represents the operations that the component will perform when invoked

- An interface...
  - The interface tells the component-user everything he needs to know in order to deploy the component
  - The interface of a component should provide all the information needed by its users

*The specification of a component is therefore the specification of its interface*

Specification of an Interface

- This must consist solely of:
  - A precise definition of the component's operations
  - All context dependencies
Component Specification

- **For users**
  - The specification provides a definition of its interface, viz. its operations and context dependencies.
  - Since it is only the interface that is visible to users, its specification must be precise and complete.

- **For developers**
  - The specification of a component also provides an abstract definition of its internal structure.

Components and Interfaces

- A component provides:
  - The implementation of a set of:
    - named interfaces
    - types
  - Each interface being a set of named operations

- The following diagram is a UML metamodel
  - This model allows an interface to be implemented by several different components, and an operation to be part of several different interfaces
Components and Interfaces

IDL Example

```idl
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;
    }
};
```
Specification Usage

- The primary usage of such specifications are:
  - Type checking of client code
  - A base for interoperability between independently developed components and applications

- An important aspect of interface specifications is how they relate to substitution and evolution of components

Substitution

- *Substituting a component Y for a component X* is said to be safe if:
  - All systems that work with X will also work with Y

- From a syntactic viewpoint, a component can safely be replaced if:
  - The new component implements *at least the same interfaces* as the older components, or
  - The interface of the new component is a *subtype of the interface* of the old component
Interfaces

- A component:
  - Implements a set of interfaces that each consists of a set of operations
  - In addition, a set of pre-conditions and post-conditions is associated with each operation

Interface Specification

This model allows the same state to be associated with several interfaces
Semantic Specification in a UML Meta-Model

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->excluding(word)
Component Specification Diagram

Similarly to interface specification diagrams, components specification diagrams are used to specify which interfaces components provide and require.

```
context SpellChecker
ISpellCheck::words = ICustomSpellCheck::words
```

Specifying a component that provides interfaces

Inter-interface Constraints

- The component specification is completed by the specification of its inter-interface constraints, an example constraint is formulated in OCL below.
Credentials

- A Credential is a triple <Attribute, Value, Credibility>
  - Attribute: is a description of a property of a component
  - Value: is a measure of that property
  - Credibility: is a description of how the measure has been obtained

Attributes in .NET
- A component developer can associate attribute values with a component and define new attributes by sub-classing an existing attribute class.

Architectural Component

- **Structural Properties**
  - Governing how a component can be composed with other components.

- **Extra-Functional Properties**
  - Such as performance, capacity, and environmental assumptions.

- **Family Properties**
  - Specifying relations among similar or related components.
Ensemble

- In Ensemble, a set of credentials may be associated with a single technology, product, or component, or with a group of such elements.

- A UML metamodel with the concepts of syntactic specification augmented with credentials is shown in the following slide.

Extra-Functional Properties
Conclusion

- A component has two parts
  - an interface
  - some code

- In current practice, component specification techniques specify components only syntactically

- The use of UML and OCL to specify components represents a step towards semantic specifications

- Specification of extra-functional properties of components is still an open area of research, and it is uncertain what impact it will have on the future of software component specification