Component-based Architecture

"Buy, don't build"
Fred Broks

1. Why use components?

- Problem with OOP:
  - ∗ Objects are too complicated and provide too limited functionality to be useful to many clients, while components such as plug-ins provide a high-level feature that can be installed and configured by users (such as web-browser plug-ins).
  - ∗ Objects do not allow for plug-and-play, integrating an object into a particular system may not be possible and therefore objects cannot be provided independently.
- Composition and assembly of components can be done by a larger group of people who do not have to have the specialist skills required for component development.
- Component-based development is a critical part of the maturing process of developing and managing distributed applications.
- Where are we in the software life cycle?

Requirements	**Design**	**Implementation**	**….**
	Software Architectures		
		Components	
	Software Component Architecture		
DSSA: Domain-Specific Software Architectures			
	Frameworks		
	Design Patterns		
		Which Programming language?	

2. What are software components?

- “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.” (Workshop on Component-Oriented Programming, ECOOP, 1996.)

- A **component** is a software object, meant to **interact** with other components, encapsulating certain functionality or a set of functionalities. A component has a clearly defined interface and conforms to a prescribed behavior common to all components within an architecture. Multiple components may be composed to build other components.

- Components are expected to exhibit certain behaviors and characteristics that let them participate in the component structure and interact with its environment and other components.

3. Component-based Systems: A Reality!! [SEI reference]

- Component-based systems encompass both commercial-off-the-shelf (COTS) products and components acquired through other means, such as existing applications.

- Developing component-based systems is becoming feasible due to the following:

    - the increase in the quality and variety of COTS products economic pressures to reduce system development and maintenance costs
    - the emergence of component integration technology
    - the increasing amount of existing software in organizations that can be reused in new systems.

- CBSD shifts the development emphasis from **programming software** to **composing software** systems [Clements 95].

4. Major elements of a component:

- **Specification:**

  It is more than just list of available operations. It describes the expected behavior of the component for specific situations, constraints the allowable states of the component, and guides the clients in appropriate interactions with the component. In some cases these descriptions may be in some formal notation. Most often they are informally defined.

- **One or more implementations:**

  The component must be supported by one or more implementations. These must conform to the specification. The implementer can choose any programming language.

- **Component Model:**
  - Software components exist within a defined environment, or component model.
  - Established component models include MS's COM+, Sun's Java J2EE or JEE 5, and the Object Management Group OMG's CORBA component standard.
  - A component model is a set of services that support the software, plus a set of rules that must be obeyed by the component in order for it to take advantage of the services.
  - Each of these component models addresses the following issues:
    - ✓ How a component makes its services available to others?
    - ✓ How component are named?
    - ✓ How new components and their services are discovered at runtime.
    - ✓ Component Types: [Felix Bachman et al. 2000]

        - A component's type may be defined in terms of the interfaces it implements.
        - If a component implements three different interfaces X, Y and Z, then it is of type X, Y and Z. We say that this component is polymorphic with respect to these types (it can play the role of an X, Y, or Z at different times.
        - Component types are found in both Microsoft/COM and Sun/Java technologies.
        - A component model requires that components implement one or more interfaces, and in this way a component model can be seen to define one or more component types. Different component types can play different roles in systems, and participate in different types of interaction schemes.

    - Each model also provides other capabilities such as:
        - ✓ Transaction management,
        - ✓ persistence, and
        - ✓ Security.

- **A packaging approach:**
    - Components must be grouped to provide a set of services. It is these packages that are bought and sold when acquiring from a third-party sources. They represent units of functionality that must be installed on the system.

    - A J2EE application is packaged as an **E**nterprise **AR**chive (**EAR**) file, a standard Java JAR file with an .ear extension.

The goal of this file format is to provide an application deployment unit that is assured of being portable.

  - Different components (modules) of an application may be packaged separated to achieve maximum reusability.

- **A deployment approach:**

  - Once the packaged components are installed in an operational environment, they will be deployed. This occurs by creating an executable instance of a component and allowing interactions with it to occur. Note that we might have different instances of a component running on the same machine.

  - J2EE uses deployment descriptors that are defined as in XML files named **ejb-jar.xml**. Example:

```
<?xml version="1.0" encoding="UTF-8"?>

<application xmlns="http://java.sun.com/xml/ns/j2ee"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
          http://java.sun.com/xml/ns/j2ee/application_1_4.xsd"
          version="1.4">

  <display-name>Simple example of application</display-name>
  <description>Simple example</description>

  <module>
    <ejb>ejb1.jar</ejb>
  </module>
  <module>
    <ejb>ejb2.jar</ejb>
  </module>

  <module>
```

```
    <web>
      <web-uri>web.war</web-uri>
      <context-root>web</context-root>
    </web>
  </module>
</application>
```

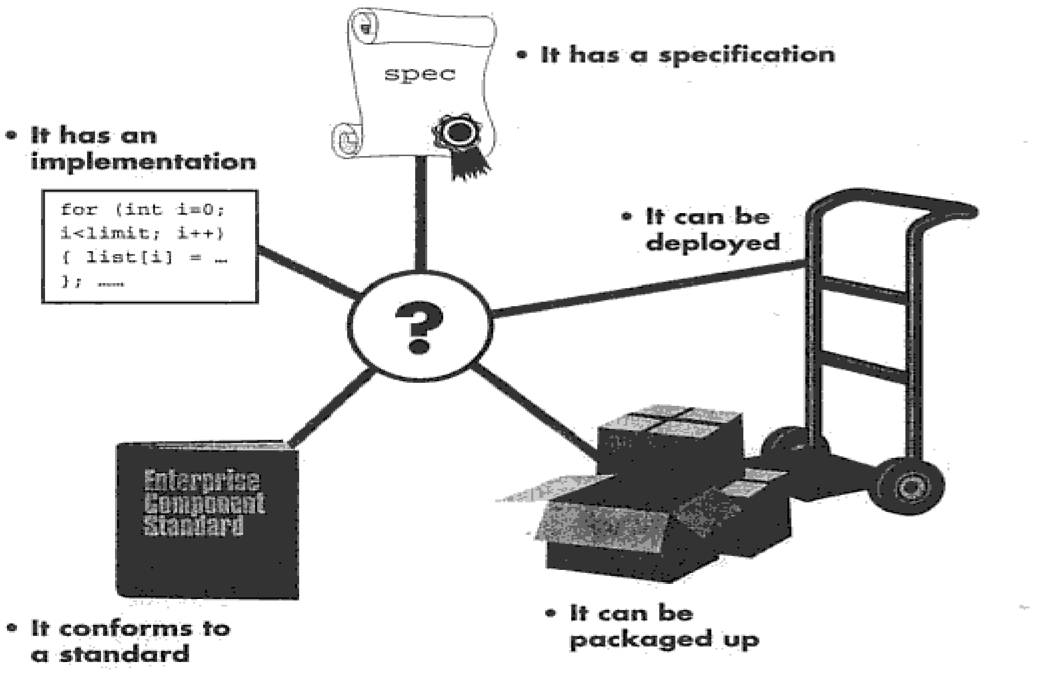


FIGURE 4.4. What is a component?

5.Component Architecture

- A **component architecture** is a system defining the rules of linking components together.

- A standard component model includes definitions for the following (WebSphere Advisor 2000):
  - ✓ How each component must make its services available to others?
  - ✓ How to connect one component to another.
  - ✓ Which common utility services can be assumed to be provided by the infrastructure supporting the component model?
  - ✓ How new components announce their availability to others?

- Component Architecture Principles (Rijsenbrij)

  - ✓ Component architecture is a set of **principles** and **rules** according to which a software system is designed and built with the use of components.

  - ✓ It must be independent from the business domain or the technology of the application.

  - ✓ The component architecture covers three aspects of a software system. These are:
    - ▪ Building blocks:
      The architecture specifies the type of building blocks systems are composed of.
    - ▪ Construction of the software system:
      The architecture specifies how the building blocks are joined together when developing an

application. The architecture describes the ***role*** that the building blocks play in the system.

        - Organisation:
          Components are divided in categories based on their functionality.

- The **component interface** is a set of methods supported by a component, and type definitions for the data used for arguments to those methods. An interface itself is a type and can be an argument for a component method.

- The Common Component Architecture Forum (http://www.cca-forum.org/glossary/index.html)

    - An **Interface Definition Language** understandable to all components. Interface definitions expressed in a language allow components to find out about each other either through introspection or through consulting a repository, and give component architecture the potential to dynamically add and delete components in multi-component applications (whether this potential is actually realized or not depends on a specific implementation of the architecture).
    - **Introspection**: Inspection is the process of exposing the properties, methods, and Events that a component supports. Example: Java provides Java provide an interface java.beans.BeanInfo to accomplish it.
    - A **Reusable Combining Infrastructure** provides the implementation necessary to **link components**. It contains mechanisms enabling the components to reference each other, understands the interface definition syntax and is capable of transferring data types and component references between components.

- A **Binding** between the interface definition syntax and a language or framework of actual component implementation.
- A **Composition API** allows the programmer to link components into multi-component applications and save those compositions. Such a mechanism could be provided for example by a GUI or a scripting language. Examples:
  - BML (Bean Markup language) from IBM
  - CoML: (http://www.springerlink.com/content/k4hy95n563m8agv8/)

6. Blackbox vs. Whitebox

- Abstractions and Reuse Blackbox vs. whitebox abstraction refers to the ***visibility of an implementation*** behind its interface.

- Ideally, a blackbox's clients don't know any details beyond the interface and its specification.

- For a whitebox, the interface may still enforce encapsulation and limit what clients can do (although implementation inheritance allows for substantial interference). **However, the whitebox implementation is available and you can study it to better understand what the box does.**

- **Blackbox reuse** refers to reusing an implementation without relying on anything but its interface and specification. For example, typical application programming interfaces (APIs) reveal no implementation details. Building on such an API is thus blackbox reuse of the API's implementation.

- In contrast, **whitebox reuse** refers to using a software fragment, through its interfaces, while relying on the understanding you gained from studying the actual implementation. Most class libraries and application frameworks are delivered in source form and application developers study a class implementation to understand what a subclass can or must do.

- There are serious problems with **whitebox** reuse across components, since whitebox reuse renders it unlikely that the reused software can be replaced by a new release. Such a replacement will likely break some of the reusing clients, as these **depend on implementation** details that may have changed in the new release.

- Some authors further distinguish between **whiteboxes** and **glassboxes** where a whitebox lets you manipulate the implementation, and a glass merely lets you study the implementation.

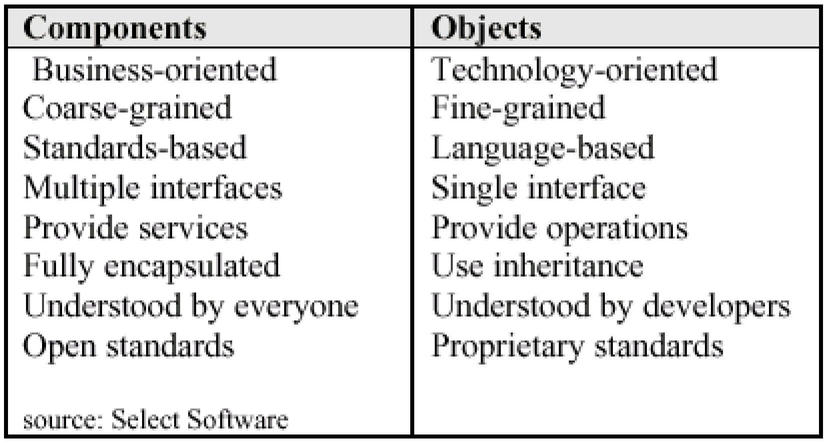
7. Components vs. Objects

- How does component architecture differ from object architecture?

- An object is built around the following ideas:
  - Inheritance
  - Needs other objects to be (re)used properly
  - The interface defines only methods
  - Has only properties (state) and behavior.

- A component differs in the following ways:
  - No inheritance (although the object that make up the component may inherit behavior from other objects, possibly in other components.
  - The component always appears as one of multiple interfaces.
  - The interface formalizes properties, events and behavior.
  - Easily reused due to its well-defined interface.
  - Flat hierarchy: no direct dependencies on other external objects.
  - Guaranteed to function in any configuration.
  - Has the ability to describe its own interface at runtime.

- List of properties contrasting Components and Objects:



Components	Objects
Business-oriented	Technology-oriented
Coarse-grained	Fine-grained
Standards-based	Language-based
Multiple interfaces	Single interface
Provide services	Provide operations
Fully encapsulated	Use inheritance
Understood by everyone	Understood by developers
Open standards	Proprietary standards

source: Select Software

Components and Objects

8. Components in industry verses in-house solutions

- Fixed-price contracts can be agreed on, limiting financial risks.
- Existing software can be customized to business needs.
- Interoperability problems are left to vendor
- In-house developers may not have the required skill. In this case component vendors may provide better solutions.

9. Component disadvantages

- Must upgrade configuration for next release
- Business processes may have to be changed to suit software (rather than developing software to suit business processes)
- Fully testing components for integration testing will be infeasible, customers may have to proceed on a most-likely will work basis (compare with applets and browsers).
- Components must handle downloading and dynamic (late) integration with other components.
- Reliance on vendors may make adjustments to software slower.

10. Summary

- Why use components?
- Major elements of a component:
  - **Specification**
  - **One or more implementations**
  - **Component Model:**
    - Each of these component models addresses the following issues:
      - ✓ How a component makes its services available to others?
      - ✓ How component are named?
      - ✓ How new components and their services are discovered at runtime.
  - **A packaging approach:**
    - Example: 2EE application is packaged as an **E**nterprise **AR**chive (**EAR**) file, a standard Java JAR file with an .ear extension.
  - **A deployment approach:**
    - J2EE uses deployment descriptors that are defined as in XML files named **ejb-jar.xml**.
- Component Architecture
- Blackbox vs. Whitebox
- Components vs. Objects
- Components in industry verses in-house solutions
- Component disadvantages