301AA - Advanced Programming

Lecturer: **Andrea Corradini**
andrea@di.unipi.it
http://pages.di.unipi.it/corradini/

***AP-06**:* *Software Components*

Overview

- Needs of components: Motivations
- Component-Based Software Development
- Component Models
- Successful component models
- Successful component-based systems
- Component-based software frameworks

➔ Chapters 1 and 4 of *Component Software: Beyond Object-Oriented Programming. C. Szyperski, D. Gruntz, S. Murer, Addison-Wesley, 2002.*

Some historical remarks

- Need of software built from *prefabricated components* first stated by **MALCOLM DOUGLAS MCILROY** in a SE conference in 1968.
  - He included *pipes* and *filters* in Unix, and developed several Unix tools, such as *spell*, *diff*, *sort*, *join*, *graph*, *speak*, and *tr*.
- Brad Cox's Integrated Circuit analogy:
  - Software components should be like integrated circuits (ICs) (IEEE Software, 1990)
- Other analogies:
  - Components of stereo equipments
  - Lego blocks, …
- Full maturity of the field in 1990-2000

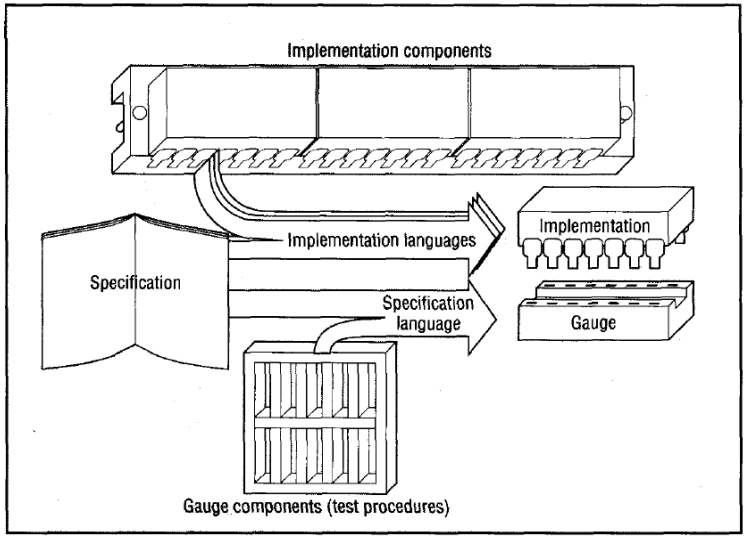


Figure 7. A development process in which specification is given the same emphasis as implementation.

Why Component-Based Software Development (CBSD)?

CBSD provides significant advantages:

- reducing costs through reuse
- facilitating faster development cycles
- leveraging off-the-shelf solutions for complex functionalities
- making system construction more modular and manageable

These benefits are consistent with the trends in modern software engineering where scalability, reusability, and time-to-market are critical factors.

Advantages of Component-Based Software Development (CBSD)

From Software Products to Product Families

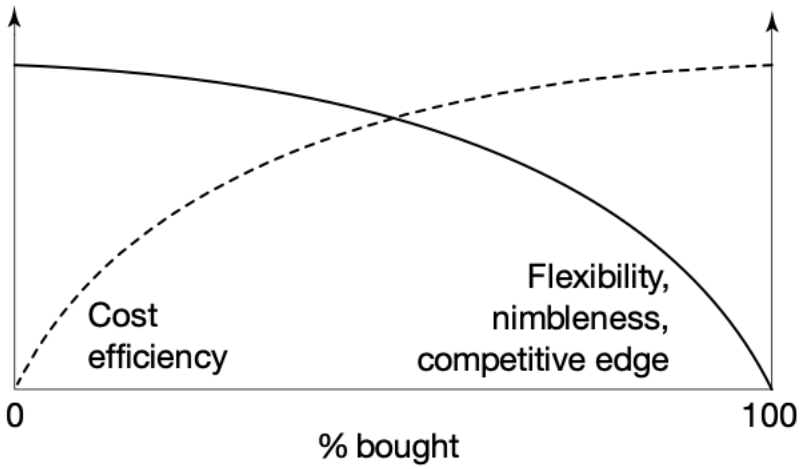
- Shared Infrastructure Across Products
- Consistency and Standardization
- Faster Time-to-Market

Need to Reuse Software to Reduce Costs

- Lower Development Costs
- Reduction in Testing and Debugging
- Focus on New Features

Preference for Off-the-Shelf Components over Re-Implementing

- Cost-Efficiency
- Access to Proven Solutions
- Faster Deployment



Advantages of Component-Based Software Development (CBSD)

Easier System Construction by Composing Components

- Modularity and Flexibility
- Separation of Concerns
- Interoperability and Customization

Additional Benefits

- Improved Maintainability
- Parallel Development
- Better Scalability

Desiderata for software components

Bertrand Meyer, in *Object Oriented Software Construction (1997)*:

1. **modular** (IC chips, disk drivers, are self-contained: packaged code)
    1. **compatible** (chips or boards that plug in easily, simple interfaces)
    2. **reusable** (same processor IC can serve various purposes)
    3. **extendible** (IC technology can be improved: inheritance)
2. **reliable** (an IC works most of the time!)
    1. **correct** (it does what it's supposed to, according to **specification**)
    2. **robust** (it functions in abnormal conditions)
3. **efficient** (ICs are getting faster and faster!)
4. **portable** (ease of transferring to different platforms)
5. **timely** (released when or before users want it)

What is a Software Component?

A **software component** can be defined as a ***modular***, ***reusable***, and ***encapsulated*** unit of software that provides a specific functionality or set of functionalities. It is designed to ***interact with other components*** via ***well-defined interfaces*** and can be ***independently developed***, ***deployed***, and ***maintained***.

Key Characteristics of a Software Component:

1. **Encapsulation**: A component hides its internal implementation details from the outside world and exposes only the necessary interfaces. This abstraction allows developers to interact with the component without needing to understand its inner workings.
2. **Reusability**: Components are designed to be reused across different applications, systems, or contexts. They are generic enough to be applied in multiple scenarios without needing major modifications.
3. **Modularity**: Each component represenats a distinct part of the system, which can be developed and deployed independently. This modularity ensures that components can be composed together to form more complex systems.

What is a Software Component? (cont)

4. **Composability**: Components are designed to be combined with other components to build larger systems. They interact via defined interfaces, making it easy to integrate them into existing systems or applications.
5. **Well-Defined Interfaces**: Components interact with other components through clearly defined interfaces, such as APIs (Application Programming Interfaces). These interfaces specify what services or functionalities the component offers and how other components can use them.
6. **Independent Deployment**: A component can be deployed independently of other components. This characteristic allows for better scalability and maintainability since individual components can be updated or replaced without affecting the entire system.
7. **Interchangeability**: A component can often be replaced by another component that implements the same interface, allowing flexibility in system evolution and adaptation to changing requirements.

Basic concepts of a Component Model

- **Component interface**: describes the operations (method calls, messages, . . .) that a component implements and that other components may use
- **Composition mechanism**: the manner in which different components can be composed to work together to accomplish some task.
  For example, using message passing.
- **Component platform**: A platform for the development and execution of components
- Concepts are **language/paradigm agnostic**
- Lays the ground for **language interoperability**

Successful Component Models (1)

Each of these models offers distinct characteristics for designing, deploying, and managing software components in a modular and reusable way:

1. **Enterprise JavaBeans (EJB)**: is a ***server-side component model*** used in **Java EE (Jakarta EE)** for building modular, transactional, and scalable business components. It supports transaction management, security, and concurrency control. Easy integration with other Java EE technologies. Ideal for: large-scale, distributed enterprise systems.

   Heavy. Spring (see later) more successful because lighter.

2. **COM (Component Object Model)**: Developed by **Microsoft**, it is a platform-independent, distributed, and object-oriented component model. It allows objects to interact within the same process or across network boundaries.
   Components communicate via language-independent interfaces.
   Supports **binary-level** software reusability and integration. Foundation for **OLE** and **ActiveX** controls. *Legacy. Windows only.*

Successful Component Models (2)

3. **CORBA (Common Object Request Broker Architecture)** is an open, vendor-neutral standard developed by the **Object Management Group (OMG)** to enable distributed systems and components to communicate in a language-agnostic manner.
   Components communicate through an **Object Request Broker (ORB).**
   Support for language-independent interoperable and distributed applications.
   Not successful as hoped, despite the standardization efforts
4. **OSGi (Open Services Gateway initiative)** is a dynamic module system for Java that allows applications to be divided into reusable components called ***bundles***. These bundles can be dynamically installed, updated, or removed without requiring a system restart. Used in ***Eclipse IDE*** and enterprise software, especially in environments requiring dynamic updates and runtime flexibility.

Successful Component Models (3)

5. **Spring Beans (Spring Framework)** offers a component-based development model where business logic is encapsulated in ***Spring Beans***. The ***Inversion of Control (IoC) container*** manages the lifecycle and dependencies of String Beans.
   Supports ***dependency injection*** for wiring components together. Highly modular and lightweight. Allows configuration using annotations, XML, or Java configuration. Application: Ideal for building Java applications across different architectures, including microservices and web applications.
6. **JavaBeans** is a reusable component model for Java that allows developers to create objects (**beans**) that can be manipulated in a visual development environment, such as **NetBeans** or **Eclipse**.

Beans follow specific conventions for property access (getter/setter methods). Supports visual component assembly in IDEs.. Commonly used in graphical user interface (GUI) applications and systems that require visual programming and component reuse.

This is the model we will play with…

Successful Component Models (4)

7. **Microsoft .NET Component Model:** it enables developers to create reusable components, often packaged as assemblies (.dll files), which can be shared across .NET applications. Supports languages like C#, VB.NET, and F#. Integrates with COM and ActiveX for legacy systems. Common in Windows-based systems, including desktop, web, and cloud applications.
8. **Model-Driven Architecture (MDA)** is a framework that promotes the generation of software components through the use of high-level models. Components are designed based on Platform-Independent Models (PIMs), which are later transformed into Platform-Specific Models (PSMs). Used in environments where model-driven development is key, such as in telecommunications and embedded systems.
9. **SCA (Service Component Architecture)**: Developed by OASIS, is a specification for creating service-oriented architecture (SOA) systems. Language-neutral; Facilitates the development of loosely-coupled services.

Successful Component Models (5)

10. **Web Components** is a modern browser technology that allows developers to create custom HTML elements with encapsulated behavior. These components are reusable across different web applications. Used for building modern web applications, providing reusable, framework-agnostic UI components.
11. **Fractal Component Model:** Provides introspection and reflection mechanisms for runtime reconfiguration. Primarily used in research, embedded systems, and large-scale systems that need runtime flexibility and dynamic adaptation.
12. **Koala Component Model** is a component model primarily designed for consumer electronics and embedded systems.

Successful Component Models (6)

13. **SOFA (Software Appliances) Component Model** focuses on building reliable distributed systems. It is designed to support dynamic reconfiguration and runtime adaptation of software components.

    Strong emphasis on dynamic reconfiguration. Components are designed to be adaptable at runtime without downtime. Primarily used in systems requiring high availability, such as telecommunication systems and enterprise infrastructures.

14. **Pi Calculus-Based Models (e.g., ArchJava)** These models use process algebras, like Pi Calculus, to model the interaction between components. ArchJava, for instance, incorporates architectural constraints directly into the programming language.
    Useful in research and high-assurance systems, where formal verification of component interaction is necessary.

Successful systems exploiting components (1)

Component-based software architecture has been successfully applied across a wide range of industries. By using modular, reusable components, organizations can build scalable, maintainable, and flexible systems that can evolve over time.

1. **Enterprise Resource Planning (ERP) Systems – SAP**
   - **Overview:** SAP's ERP system is a classic example of a component-based system. It consists of numerous modules (components) that handle specific business functions like finance, human resources, supply chain management, and more.
   - **Component Usage**: Each module is a self-contained component that can be deployed independently and interacts with other components via APIs and services.
2. **Microsoft Office Suite**
   - **Overview**: Microsoft Office (Word, Excel, PowerPoint, etc.) is a software suite with different applications (components) that share common services, like spell checkers, file formats, and user interface elements.
   - **Component Usage**: The suite is a collection of components that integrate seamlessly but can also be used independently. The components share a common architecture, and through APIs like **Visual Basic for Applications (VBA)**, they enable developers to build custom solutions that integrate Office applications.

Successful systems exploiting components (2)

3. **Web Browsers – Google Chrome & Mozilla Firefox**
   - **Overview**: Modern web browsers like **Google Chrome** and **Mozilla Firefox** are built from numerous software components that handle different tasks, such as rendering HTML/CSS, handling JavaScript execution, network communication, and providing security features.
   - **Component Usage**: For instance, **Chrome** uses the **Blink rendering engine** (which is a component itself) and separates tabs into independent processes (components) for better stability and security. Each tab operates as an isolated component, meaning if one crashes, the others can continue to function.
4. **Cloud Computing Platforms – Amazon Web Services (AWS)**
   - **Overview**: AWS provides a vast array of cloud services (components) like storage (S3), computing (EC2), databases (RDS), and messaging services (SQS).
   - **Component Usage**: These services are modular components that can be used independently or combined to create custom cloud-based architectures. Each service is self-contained, with defined APIs for interaction, enabling customers to build highly scalable and flexible systems using different AWS components.
5. **Operating Systems – Linux**
   - **Overview**: Linux is built on a modular architecture where the kernel is at the core, and various components (modules) provide additional functionality such as device drivers, file system support, and networking capabilities.
   - **Component Usage**: The Linux kernel is highly modular, and components (known as kernel modules) can be dynamically loaded or unloaded to provide additional functionality as needed, without having to recompile the entire kernel.

Successful systems exploiting components (3)

6. **Microservices-Based Applications – Netflix**
   - **Overview**: Netflix is a pioneer in building large-scale, cloud-based systems using microservices, which are essentially small, independently deployable software components.
   - **Component Usage**: Each microservice (component) in the Netflix architecture handles a specific part of the system (e.g., user recommendations, streaming management, authentication). These microservices communicate via APIs, enabling Netflix to scale and evolve its system continuously while maintaining high availability.
7. **Integrated Development Environments (IDEs) – Eclipse**
   - **Overview**: Eclipse is an open-source IDE for software development. It's built on a component-based architecture called the **Eclipse Rich Client Platform (RCP**), based on the OSGi component framework.
   - **Component Usage**: Eclipse is highly modular, consisting of a core platform and numerous plugins (components) that provide additional functionality such as debugging, version control, and language support. Developers can create their own plugins or customize the IDE by integrating third-party components.
8. **Content Management Systems (CMS) – WordPress**
9. **E-Commerce Platforms – Shopify**
10. **Automotive Software Platforms – AUTOSAR (Automotive Open System Architecture)**
11. **Banking Systems – Murex**
12. **Healthcare Systems – Epic Systems**

What do all the above examples have in common?

- In all cases there is an **infrastructure** providing rich foundational functionality for the addressed domain.
- Components can be purchased from **independent providers** and deployed by clients.
- The components provide services that are substantial enough to make **duplication of their development** too difficult or **not cost- effective**.
- Multiple components from different sources **can coexist** in the same installation.

- Components exist on a **level of abstraction** where they directly mean something to the deploying client
- With Visual Basic, this is obvious – a **control** has a direct visual representation, displayable and editable properties, and has meaning that is closely attached to its appearance.
- With **plugins**, the client gains some explicable, high-level feature and the plugin itself is a user-installed and configured component

Software Frameworks for Component-Based Application

These frameworks enable developers to build modular, scalable, and maintainable systems by encouraging the reuse and composition of components.

- **Spring Framework**
- **Angular (Web Components), by Google**
- **React (Component-Based UI), by Facebook**
- **Vue.js: progressive JavaScript framework for building web interfaces**
- **Apache Struts: open-source web application framework based on (MVC)**
- **Apache Wicket: Java-based web application framework**
- **Vaadin: Java-based web application framework that provides component-based model**
- **Microsoft .NET Core**
- **OSGi (Open Services Gateway initiative)**
- **JBoss Seam (Red Hat)**
- **MEAN Stack (MongoDB, Express, Angular, Node.js)**

OOP vs CBSD

Object-Oriented Programming promotes reuse primarily through:

- Inheritance
- Polymorphism: Allowing methods to operate differently based on the object type
- Encapsulation

OOP has been successful in promoting modularity and reuse ***within individual applications or projects***, but it has ***several limitations*** when it comes to large-scale or cross-project software reuse:

1. **Tight Coupling and Inflexibility**: Inheritance often leads to tight coupling between classes. If one class is changed, all its subclasses may need to be modified as well. This can reduce flexibility and make reuse across different applications or contexts harder.

   OOP assumes that you have access to the source code or are working within the same project or system. In many cases, external or third-party components cannot easily be adapted using traditional OOP techniques.

OOP vs CBSD (2)

2. **Granularity of Reuse**: OOP is primarily focused on ***reuse at the class level***. However, software reuse at a ***higher level***, like ***services*** or ***components***, requires a more coarse-grained approach. Reusing entire modules, components, or services is often more beneficial than reusing individual classes.
3. **Difficulty in Cross-Project Reuse**: OOP relies on a shared type hierarchy and a common environment (like a shared codebase). Reusing components across different projects, with different dependencies, frameworks, or runtime environments, can be challenging.
   Reuse in OOP can be limited to projects that use the same language, libraries, and framework versions, making it less effective in heterogeneous or multi-language environments.

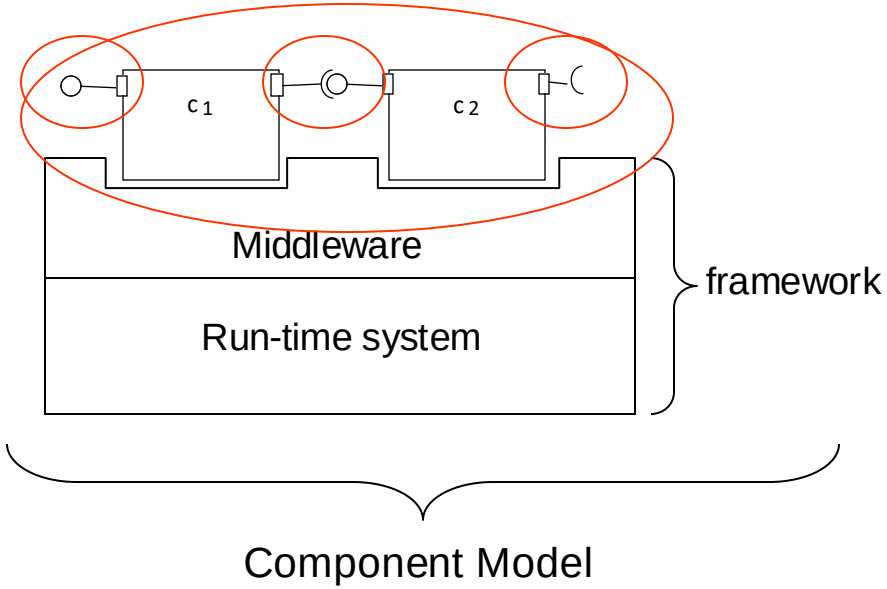
OOP vs CBSD (3)

4. **Evolving Requirements**: Modern software often needs to evolve rapidly in response to changing user needs and market conditions. The rigidity of OOP's inheritance hierarchies can make it difficult to adapt existing classes to new requirements without extensive modification, which is contrary to the principles of agile development.
5. **Interoperability Issues**: OOP paradigms can lead to challenges in integrating with components or services that were designed using different paradigms or technologies (e.g., functional programming, procedural programming, or microservices-based architectures).

Component Based Software Engineering

basic definitions

- The basis is the Component
- Components can be assembled according to the rules specified by the component model
- Components are assembled through their interfaces
- A Component Composition is the process of assembling components to form an assembly, a larger component or an application
- Component are performing in the context of a component framework
- All parts conform to the component model
- A component technology is a concrete implementation of a component model



Summary

- Component technology in Software Development reached maturity in 1990/2000
- Main motivation: cost-efficiency thanks to reuse
- Various Component Models
- Component based software development supported by several frameworks
- We shall play with JavaBeans as Component Model, with NetBeans as framework