301AA - Advanced Programming

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

AP-07: Software Components
Overview

• Needs of components
• Definition of Component Software
• Components and other programming concepts
• Example of components: short history

Why component-based software?

• Cost of software development
  – from software products to product families
  – need to re-use software to reduce costs
  – better to buy off-the-shelf than re-implementing
  – constructing systems by *composing* components is easier

*Figure 1.1* Spectrum between make-all and buy-all.
Why component-based software?

- **Component software**: composite systems made of software components
- More reliable software
  - more reliable to reuse software than to create
  - system requirements can force use of certified components (car industry, aviation, . . . )
- Emergence of a component marketplace
  - Apple’s App Store, Android Market, . . .
- Emergence of distributed and concurrent systems
  - we need to build systems composed of independent parts, by necessity
Components as in Engineering...

• 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, ...
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)
Software Components: a definition

“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.” Clemens Szyperski

Workshop on Component-Oriented Programming, 1996 European Conference on Object-Oriented Programming

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.

- Binary units – black boxes, not source code
- Partial deployment not possible
- System can be built by combining components
- No (externally) observable state
- Indistinguishable from copies
What is a contract?

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.

- **Interface** – component specification

- **Contract** - A specification attached to an interface that mutually binds the clients and providers of the components.
  - Functional Aspects (API)
  - Pre- and post-conditions for the operations specified by API.
  - Non functional aspects (different constrains, environment requirements, etc.)
"Contractually specified interfaces"

• Require mechanism for interface definition, such as Interface Definition Language (IDL)
• Contracts specify more than dependencies and interfaces
  – how the component can be deployed
  – how can be instantiated
  – how the instances behave through the advertised interfaces
• Note: this is more than a set of per-interface specifications
• **Example**: a *queuing component* has a *stable storage requires* interface and *enqueue* and *dequeue provides* interfaces. The contract states that:
  – what is enqueued via one interface can be dequeued via the other
  – instances can only be used by connecting them to a provider implementing the stable storage interface
What is an explicit context dependency?

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.

- Provided and Required Interface

- Context dependencies - Specification of the deployment environment and run-time environment
  - Example: Which tools, platforms, resources or other components are required?
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.

- Late binding - dependencies are resolved at load or run-time.
What does it mean deployed independently?

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.

• The component can be plugged into a system or composed with other components by third parties, not aware of the internals of the component.
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**.
Before Components: Modules

- Support for **modules** in several languages since the 1970’s
- Modules as main feature of programming languages for supporting development of large applications
  - Support *information hiding* through *encapsulation*: explicit import and export lists
  - Reduce risks of *name conflicts*; support *integrity of data abstraction*
- Teams of programmers can work on separate modules in a project
  - No language support for modules in C and Pascal
  - Modula-2 **modules**, Ada **packages**
  - Java **packages** (?), new notion of module in Java 9
Scoping Rules for Modules

- Scoping: modules encapsulate variables, data types, and subroutines in a package
  - Objects inside are visible to each other
  - Objects inside are not visible outside unless exported
  - Objects outside are visible [open scopes], or are not visible inside unless imported [closed scopes], or are visible with “qualified name” [selectively open scopes] (eg: B.x)

- A module interface specifies exported variables, data types and subroutines

- The module implementation is compiled separately and implementation details are hidden from the user of the module
Module Types, towards Classes

• Modules as abstraction mechanism: collection of data with operations defined on them (sort of abstract data type)

• Various mechanism to get module instances:
  – Modules as manager: instance as additional arguments to subroutines (Modula-2)
  – Modules as types (Simula, ML)

• Object-Oriented: Modules (classes) + inheritance

• Many OO languages support a notion of Module (packages) independent from classes
Modules vs. Components

- Several component-related concepts already present in modules
- Modules as part of a program, component as part of a system
- Components can include static resources
- Modules may expose observable state
- Modules encompassed by classes in OO languages in the 1990’s
- Now present in most modern languages
Components and Programming Concepts

• Component can be anything and can contain anything
  – (Collections of) classes, objects, functions/algorithms, data structures
• Typically granularity is coarser than classes
• Components support:
  – Unification of data and function
  – Encapsulation: no visible state
  – Identity: each software entity has a unique identity
  – Use of interfaces to represent specification dependencies
OOP vs COP

• Object orientation is not primarily concerned with reuse, but with appropriate domain/problem representation using concepts like:
  – Objects, classes, inheritance, polymorphism
• Experience has shown that the use of OO does not necessarily produce reusable software
CBSE – Component-Based Software Engineering

• Provides methods and tools for
  – Building systems from components
  – Building components as reusable units
  – Performing maintenance by replacement of components and introducing new components into the system
  – System architecture detailed in terms of components
Component Forms

1. Component specification
2. Component interface
3. Component implementation
4. Installed component
5. Component object
Component Specification

• The specification of a unit of software that describes the behavior of a set of Component Objects and defines a unit of implementation.

• Behavior is defined as a set of Interfaces. A Component Specification is realized as a Component Implementation.
Component Interface

• A definition of a set of behaviors that can be offered by a *Component Object*. 
Component Implementation

• A realization of *Component Specification*, which is independently deployable.

• This means it can be installed and replaced independently of other components.
  – It does not mean that it is independent of other components – it may have many dependencies.
  – It does not necessarily mean that it is a single physical item, such as a single file.
Installed Component

• An installed (or deployed) copy of a *Component Implementation*.

• A Component Implementation is deployed by registering it with the runtime environment.
  – This enables the runtime environment to identify the *Installed Component* to use when creating an instance of the component, or when running one of its operations.
Component Object

• An instance of an *Installed Component*.
• A runtime concept.
• An object with its own data and a unique identity.
• The thing that performs the implemented behavior. An Installed Component may have multiple Component Objects (which require explicit identification) or a single one (which may be implicit).
Summary CBSE – 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
Some successful components: In the past...

• Mathematical libraries
  – NAGLIB - Fortran Library
  – Mathematical and physical functions

• Characteristics
  ✌ Well defined theory behind the functions - very well standardized
  ✌ Simple Interface - **procedural type** of communication between client (application) and server (component)
  ✌ Well defined input and output
  ✌ Relative good error handling
  ❌ Difficult for adaptation (not flexible)
Some successful components: The big ones...

Client - server type

- **Database Servers**
  - Relational databases, (Object-oriented databases, hierarchical databases)
  - Standard API - SQL
  - Different dialects of the standard

- **X-windows**
  - Standard API, *callback type* of communication
  - High level of adaptation
  - Too general - difficult to use it
Even bigger components: Operating systems

• Example - Unix
  – A general purpose OS, used as a platform for dedicated purposes
  – Standard API - POSIX
    ✤ Commands used as components in a shell-process
    ✤ Low-level but well-defined interfaces (file sharing, pipes and filter)
    ✖ Different variants, POSIX is not sufficient
    ✖ Not a real component behavior (difficult to replace or update)

• MS Windows ...
More recent components...

• **Plugin architectures** (finer-grained components)
  – Netscape’s Navigator web browsers
  – Active Server Pages (ASP) and Java Server Pages (JSP) architectures for web servers
• Microsoft’s **Visual Basic**
• Java **Beans**, Enterprise JavaBeans (EJB)
• Microsoft’s **COM+**
• Android’s component based apps
• Modern application and integration servers around J2EE and COM+ / .NET
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.