8. Component Architectures

1. Motivation and Fundamental Concepts
2. Revisiting Object-Oriented Analysis, Design, and Implementation
3. Design Patterns
4. Pipes & Filter Architectures
5. Event-based Architectures
7. Framework Architectures

8. Component Architectures

8.1 Software Components
8.2 Client-Side Component Architecture
8.3 Foundation for Server-Side Component Architecture
8.4 Server-Side Component Architecture
Software Components

**Software components** are (binary) modular units which are subject for assembly and integration (by third parties) with other software components. They export and import functionality using a standardized interface mechanism. [Szy97,Broy98]

The driving force behind component architecture [Griffel98]:
- Programming as an engineering discipline: analysis and design have already been understood as important tasks for building successful (one-time, one-product) systems.

The idea behind software components is to
- **reuse** (pre-fabricated, matured) software (many-times, many-products)
- stop the NIH (not-invented-here) paradigm (buy instead of make)
- add structure to design and implementation (design for reuse)

This requires planning ahead of products and product families.

NOTE: Compared to hardware, the reuse of software nowadays is marginal (No „Software-IC“ [Cox87]).

---

**Differences between OO and Component Software**

<table>
<thead>
<tr>
<th>Components:</th>
<th>Object-oriented software:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Interactivity, connectivity, exchangeability of components</td>
<td>- Encapsulation, Inheritance, Polymorphism</td>
</tr>
<tr>
<td>- Building blocks (components) usually consist of several classes</td>
<td>- Building blocks are classes</td>
</tr>
<tr>
<td>- (Externally) visible functionality is defined by interfaces</td>
<td>- (Externally) visible functionality is defined by class declaration</td>
</tr>
<tr>
<td>- Metadata to describe the purpose and interfaces of components (esp. for third-party use):</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Event notification</td>
</tr>
<tr>
<td></td>
<td>- Persistence</td>
</tr>
<tr>
<td></td>
<td>- Visual representation (desktop components only)</td>
</tr>
<tr>
<td></td>
<td>- Distribution</td>
</tr>
</tbody>
</table>

[Griffel98, Stal97]
Large, Distributed Component Architectures

- Components allow for distributed architectures
  (distributed, cooperating information systems can be regarded as distributed components of a superordinated, dynamic system)
- Components can be realized in different programming languages
- Coordination languages are needed to make components interact
  (in JavaBeans and Enterprise JavaBeans, the coordination language is again Java, but applied at another level)

Black-Box vs. White-Box Components

**Black-Box Components:**
- Internal structure is invisible to component user
- Re-use via interfaces (delegation)
- User cannot verify the component’s code (security, bugs)

Examples:
- Java Collections: Stack, List, Map, etc.

**White-Box Components:**
- Internal structure is (partly) visible to component user
- Re-use via inheritance (subclassing)

Examples:
- Java Swing Classes (JPanel, JFrame, JButton, ...)

metadata
Component Evolution

GUI-related components:

- Buttons, Textfields, Panels, Canvasses
- GUI trash bin

First, obvious “components”: Elements that can be plugged into larger elements. Visible, client-side components.

Business Objects:

A business object is a software module / component that is associated with a concept and therefore can be understood by a software user. A business object can be an

- Application-/domain-specific or
- Application-independent (generic) object.

Identified as important for building enterprise application systems. Invisible, server-side components. [Griffel98]

Technical View on Components

A component is a physical replaceable part of a system. Its implementation conforms to a set of interfaces.

Components have two different aspects:

- Code: A component contains Code (Source code, libraries, executable programs, ...) (e.g. JavaBeans, Windows DLL, Windows EXE-File, CORBA ORB, ...)
  Components may contain (sub-)components or use (delegate to) other components.

- Identity and State: A Component may have an identity and a state, represented by objects (see State Pattern). A client object trying to access and use a component’s services usually has to provide the components instance (e.g. via a reference to a JavaBean, a DLL-Handle, a process, a CORBA-IOR, etc.)

Example:

- Dictionary for different languages / with different word bases
UML Component Diagrams (1)

Notation

Component

Dictionary

Spellchecker

Thesaurus

“implements”

Interfaces

Component with identity and state

myDictionary : Dictionary

o : Options

wb : WordBase

Component's Objects

UML Component Diagrams (2)

A **Component Diagram** describes organization and dependencies between components.

Client-side Applet

Update

Server-Side Servlet

Reservation

Interface usage ⇒ Dependency

Interface
Problem with Semantics

A component consists of interface(s), semantics, and metadata.

- Component = Interfaces + Semantics + Metadata [OMG97].

How to specify the semantics of a component?

- What is the meaning of the interface methods?
  - Usually only human-understandable
  - Formal calculi exist to describe semantics (operational, axiomatic, denotational)

- What are valid method invocation sequences?
  - Interface + semantics can be specified by a service contract.

Service Contracts

Method invocations are valid only in certain states (of the component). The states can be described by

- automata, state charts, UML state diagrams,
- methods’ pre-, postconditions and invariants (Programming by Contract [Meyer97]).

```c
/** @pre a>0, b>0  
* @post a==p*x, b==q*x,  
* \forall x': a==p*x', b==q*x'  => x'<=x */
int gcd (int a, int b) ;

/** @pre getCustomer ().getCreditRating () >= CreditRating.GOOD */
void orderItem (Item i) ;
```

Sidestep: Specification languages (VDM, Z) use pre-, postcondition and invariants to ensure that implementation follows specification.
Are Components Successful?

Are there successful component architectures?

- The Java server-side EJB (Enterprise JavaBeans) architecture is evolving as a (economically) successful component architecture, while the emphasis here is to reuse components is different EJB server/containers, not connect EJBs.
- The Java client-side desktop component architecture (JavaBeans) is only successful for visual components (Java Swing).
- MS ActiveX, COM, DCOM components are successful within MS-based systems.
- Business Object Frameworks (SanFrancisco Framework, CORBA BOCA)

Are there successful components (without a component architecture)?

- Web-Server components: Plug-ins, “Counters”, Bulletin-Boards, ...
- XML technology: XML Parsers, XML Transformers (XSLT), ...

Components – The Silver Bullet?

Do we need components and component architectures? Generally yes, to

- manage increasing software system complexity
- reduce the risks of vendor lock-in (and, to a certain extent, technology lock-in)
- reduce the time and cost to build and test software
- achieve better time-to-market by building systems mostly out of existing components
- adding components to extend a system by functionality that competitor systems have just integrated

But note: Component-oriented software development is still in its infancy compared to

- object-oriented software engineering (replacing structured programming)
- structured programming (replacing, basically, hacking)

Therefore, the perceptible differences between OO and components are faint.
References & Further Reading


[Broy98] Manfred Broy et al., What characterizes a software component?


