Software Components: from Business to Metacomputing

Denis Caromel
OASIS Team
INRIA -- CNRS - I3S -- Univ. of Nice Sophia-Antipolis -- IUF

Invited Talk at ORAP Forum / SpeedUp, October 25-26th 2001, Lyon

Business Components:
1. Component Basics
2. Examples: JavaBeans, EJB, Corba 3, .Net

Metacomputing Components:
3. ProActive components, Demo: C3D and IC2D
4. CCA architecture and tools: CCAT

INRIA
Basic Ideas and Definition - What it is?

A Component = a unit of Composition and Deployment

From Objects (Classes) to Components:

• Objects:
  ➡ Programming in the small
  ➡ Composition by programming (Inheritance, Instantiation/Aggregation)

• Components:
  ➡ Building software in the large
  ➡ Tools for assembling and configuring the execution

Component = a module (80s!) but subject to:
  ✦ Configuration (variation on Non-Functional Properties)
  ✦ Instantiation

To be deployed on various platforms (some portability)
Characteristics -- How?

How it works --- Common characteristics

• A **standardized** way to describe a component:
  ➔ a specification of what a component does:
    ✤ **Provide** (Interfaces, Properties to be configured)
    ✤ **Require** (services, etc.)
    ✤ **Accept** as parameterization

• Usually dynamic discovery and use of components:
  ➔ **auto-description** (Explicit information: text or XML, reflection, etc.)

• Usually components come to life through several classes, objects

• **Legacy code**: OO code wrapper to build components from C, Fortran, etc.
My Definition of Software Components

A component in a given infrastructure is:

• a software module,
• with a standardized description of what it needs and provides,
• to be manipulated by tools for Composition and Deployment
A first example: JavaBeans
Graphical components in Java

Quite simple:

- a Java class (or several)
- a naming convention to identify properties:
  - method: `public T getX()`
  - method: `public void setX()`
  - an attribute: `private T X = <default value>`;
- a communication pattern: Events, Source, Listeners

and … a class is turned into a graphical component!

The Java introspection allows to discover dynamically the properties, and to configure them, assemble JB interactively.
JavaBeans (2)

The BeanBox

So for JavaBeans:

Software module = Java Class
Standardized description = getX, setX, X, listeners
Tools:
Composition = BeanBox
Deployment = JVM

Nothing very new (cf. NeXTStep Interface Builder), but life made a bit easier with byte code and introspection
Deploying and Executing Components

Components have to be configured on their Non-Functional Properties:

- **Functional Properties (Def.):**
  - Application level services a component provides (e.g. Balance, Saxpy)

- **Non-Functional Properties (Def.):**
  - The rest, mainly infrastructure services:
    - Transaction, Security, Persistence, Remote/Asynchronous Com., Migration, etc.

**so, Typical Infrastructure : Container for Isolation**

Allows to manage and implement:

- the non-functional properties
- Life cycle of components

![Diagram of Client-Server-Container Architecture](attachment:image.png)
Example: Enterprise Java Beans

A 3 tiers architecture (Interface, Treatment, DB), in Java

➤ Objectives: ease development + deployment + execution
➤ Java portability

A few concepts and definitions:

• EJB Home object:
  ➤ management of life cycle: creation, lookup, destruction, etc.

• EJB Remote object:
  ➤ Client view and remote reference

• EJB object (or Bean):
  ➤ the component itself (Session Stateless or Statefull, Entity Bean)

• Functional Properties = Business Methods
Example: Enterprise Java Beans (2)

Home and Remote are generated:

- to implement basic services (remote access)
- and Non-Fct. Prop.,

from a (text, XML)

Deployment Descriptor:

```xml
<ejbml>
<entity-bean name="Books"
home="Books.BooksHome"
remote="Books.Books"
bean="Books.BooksBean"
primary-key="Books.BooksPK"
<container managed-persistence
<field name="price" />
```

From www.tripod.com, G. S. Raj article
Example: Enterprise Java Beans (3)

So for EJB:

Software module =

- Java Classes and Interfaces (Home, Remote, Beans, …)

Standardized description =

- a file with a standard format

Tools:

- Composition = ? EJBrew ?
- Deployment = JVM + RMI, JTS, + Generators + EJB Servers

From www.tripod.com, G. S. Raj article
Components in Windows .Net

.Net basics:

• A VM designed for several languages (C, C++, VB, + others)
  ➤ CLR (Common Language Runtime)
  ➤ CIL (Common Intermediate Language, MSIL) wider than ByteCode
    ✦ Boxing/Unboxing (value type <--> object), etc.

• A new language: C#

• An interactive tool (Visual Studio) to manipulate the “components”

A key choice: Extraction of description from program code

➤ C# introduces language constructions for component information:
  ✦ Properties
  ✦ Attributes
  ✦ XML tags in source code (in Attributes)
Components in Windows .Net (2)

Example of Attributes, and Properties in C#:

```csharp
[HelpUrl ("http://someUrl/Docs/SomeClass")]
class SomeClass {
    private string caption;
    public string Caption {
        get { return caption; }
        set { caption = value; Repaint (); }
    }
}
```

- **An attribute**: HelpUrl
  - It is actually a user define class (derive from System.Attribute)
  - Attribute exists at RT.

- **A Property**: Caption
  - JavaBeans in a language
  - Also: Indexed properties

Components for Web program: WSDL (Web Services Description Lang.)

- WSDL (Def. of Web callable methods) + Directories +
- SOAP as wire format + Classes with Attributes and properties,
Components in Windows .Net (3)

Components characteristics:

Software module =
- Classes and Interfaces in various languages, but focus on C#

Standardized description =
- Still the COM, DCOM interfaces
- Extraction of Attributes, Properties from source code!
- WSDL

Tools:
- Composition = Visual Studio, etc.
- Deployment = Windows, .NET CLR
Corba 3 and CCM

CCM: Corba Components Model =

• EJB + a few things:
  ➤ More types of Beans defined: Service, Process, ...
  ➤ Not bound to Java (Corba IDL, but … usually in Java ...)
  ➤ Assembly Descriptor: towards interactive composition tools
Current Limitations

Easy to describe what a component provides:

- Interfaces, Classes

- not so common to describe what it needs:
  - Also Interface and Classes, but several object references in code:
    - which one to bind to?
    - which one are bound to the same object? Need dedicated factories

The specification of a component is given by Method Profiles

- But no Behavior
  - What about correct order on method calls,
  - Resource consumption (CPU, ...), etc.

Hard to add new “Non-Functional Properties”

- in existing platforms

(RNTL project: ARCAD: Architecture Répartie extensible pour Composants ADaptables)
Metacomputing Components

Specificity:

- **Communication**:
  - High performance
  - Latency Hiding
- **Deployment complexity**
- **Debugging and Monitoring**
  - Across the world ??
- **Some load balancing**:
  - at least dealing with Load Sharing tools (LSF, etc.)
  - Impact on “Container’’?
- **Security**
  - Crossing administrative boundaries

High-Performance a specificity ?
Not sure: an EJB component handling 1 000 000 of requests already needs High-Performance!
3. Components in ProActive PDC

- Library for Parallelism, Distribution, CSCW
- 100 % Java
- Active object = an object + a thread = a server + an agent = Actor

Seamless

Sequential  Multithreaded  Distributed
**ProActive Non-Functional Properties**

- Remotely Accessible (Interface, Classes, existing objects)
- Asynchronous Communications + Synchronization (Futures)
- Migration
- Group Communications
- Graphical Visualization, Monitoring, and Control (IC2D)
- XML Deployment descriptors (November release)
- Security (no release plan yet)
DEMO: C3D with the IC2D monitor

C3D: Collaborative 3D renderer in //
a standard ProActive application

IC2D: Interactive Control & Debug for Distribution
work with any ProActive application

Features:
Graphical and Textual visualization
Monitoring and Control
C3D: distributed-collaborative

Denis Caromel
IC2D on several machines (1)
IC2D on several machines (2)
ProActive Components: Characteristics

Components characteristics:

Software module =
  · Java Classes and Interfaces + threads: forming Active Objects

Standardized description =
  · In source: Active interface marker, newActive, turnActive
  · .proActiveDescriptor : an XML file per component

Tools:
  · Composition = none
  · Deployment = Java VM, IC2D Gui tool
4. CCAT: Common Component Architecture
D. Gannon et al.

CCA:
- DOE funded project,
- Defining standard features for HP Components

CCAT: Common Component Architecture Toolkit
- D. Gannon’s team implementation of CCA
- Based on:
  - HPC++
  - Globus, SSH
  - Java for GUI, JPython, Matlab interface

A focus on:
- Composition
CCAT Components

A few original characteristics:

- Dependencies: Interfaces
  - Provides-Port
  - Uses-Port
- Core Services are components
  - Flexibility, Higher-level services from core
- GUI for composition is a component
  - connected to Provides-Port of core service components

Others:

- Standard services:
  - Directory, Registry, Creation, Connection, Events
- XML description of components
CCAT session with Java GUI: Composition

Composition Tool:

- Select
- Connect
- But also: Test and Execute

From D. Gannon et al. article
CCAT Components Characteristics

So for CCAT:

Software module =
- Any Code + wrappers

Standardized description =
- XML
- Interfaces (Provide+Use)

Tools:
- Composition = Java GUI
- Deployment = some

From D. Gannon et al. article

SC are mainly wrappers for external services
**XCAT:**  
D. Gannon et al. current project

- A Java-based web server (Tomcat) on the client:  
  - Java Servlets

- A Browser on the client as well
  - Script Editing

- Scripts in JPython

- WSDL services, to generate SOAP communications
  - XML fire format

---

**Notebook Browsing Controls**

**Notebook Scripts can be “parameterized” by web forms**

**Script Editing**

**Courtesy of D. Gannon**
Conclusion

• The concept and technology of Business Components is attracting, and useful for Metacomputing Components

• Of course, performance is an issue (not even talking about QoS...):
  • What balance between Commodity and Dedicated Services?

• Will Components make parallelism and clusters much more wide spread?
  • … some hope here ...

• About platforms:
  ➤ **CCAT,XCAT**: focus on Composition, Assembly/Mon. of simply connected code: WFI.
  ➤ **ProActive**: focus on Deployment and Monitoring

• Would be good to have both, but … IMHO:
  ➤ **Well Known Interfaces** needed before Composition is useful
  ➤ Deploying // applications in Metacomputing framework still a challenge!

• **Security** is the next key issue, with performance complications!
ProActive PDC

The ProActive Java Library for Parallel, Distributed, and Concurrent Computing.

ProActive PDC is a Java library for Parallel, Distributed, and Concurrent computing and metacomputing. In the framework of a MIMD model (active object model), from a reduced set of rather simple primitives, a comprehensive and versatile library is defined for parallel, distributed, and concurrent programming (middleware).

In the absence of any syntactical extension, ProActive PDC programmers write standard code. The library is itself extensible by the programmers, making the system open for adaptations and optimizations.

ProActive PDC is only made of standard Java classes, and requires no changes to the Java Virtual Machine, no preprocessing or compiler modification.

ProActive PDC is based on the RMI Java standard library.

ProActive features the following:

- Active objects, Asynchronous calls, Messages (Request and Reply)
- Migration, Mobile Agents
- Seamless sequential, multi-threaded, and distributed programming
- Reuse: polymorphism between standard object and Active objects
- Automatic future-based synchronizations (wait-by-necessity)
- Libraries for sophisticated synchronizations, collaborative applications
- Interfaced with RMI registry, Jini, and Globus
- Compatible with Swing and AWT

New:
- ProActive support: proactive-support@inria.fr
- Job opportunities: PostDoes
- Interactive Control and Debugging of Distribution: IC2D

Formerly

Denis Caromel

08591
DIVA: Distributed Int. Virtual World in Java
**ProActive** : Creating active objects

Single inheritance: using interface

An object created with

\[
A a = \text{new} \ A \ (\text{obj}, \ 7);
\]

can be turned into an active object:

**Class-based**

```java
class \textit{pA} \ extends \ A \ \text{implements} \ \text{Active} \ \{
\}
A a = (A)\text{ProActive}.\text{newActive}(\langle\textit{pA}\rangle,\text{params}, \ \text{node});
```

The most general case. Allows to provide a specific behavior.

**Instanciation-based**

```java
A a = (A)\text{ProActive}.\text{newActive}(\langle\textit{A}\rangle,\text{params}, \ \text{node});
```

**Object-based**

```java
A a = \text{new} \ A \ (\text{obj}, \ 7);
a = (A)\text{ProActive}.\text{turnActive} \ (a, \ \text{node});
```
ProActive: Reuse and seamless

Two key features:

- Polymorphism between standard and active objects
  - Type compatibility for classes (and not only interfaces)
  - Needed and done for the future objects also
  - Dynamic mechanism (dynamically achieved if needed)

- Wait-by-necessity: inter-object synchronization
  - Systematic, implicit and transparent futures
  
  Ease the programming of synchronizations, and the reuse of routines

```c
void foo (A a)
{
    a.g (...);
    v = a.f (...);
    ...
    v.bar (...);
}
```
ProActive: Intra-object synchronization

Explicit control:

Library of service routines:

- Non-blocking services,...
  - serveOldest ();
  - serveOldest (f);

- Blocking services, timed, etc.
  - serveOldestBl ();
  - serveOldestTm (ms);

- Waiting primitives
  - waitARequest();
  - etc.

Implicit (declarative) control: library classes

e.g.: myBody.forbid ("put", "isFull");
**ProActive**: Migration of active objects

Migration is initiated by the active object itself through a primitive: `migrateTo`

Can be initiated from outside through any public method

The active object migrates with:
- all pending requests
- all its passive objects
- all its future objects

Automatic and transparent forwarding of:
- requests (remote references remain valid)
- replies (its previous queries will be fulfilled)
**ProActive**: API for Mobile Agents

- Mobile agents (active objects) that communicate
- Basic primitive: `migrateTo`
  - `public static void migrateTo (String u)`
    // string to specify the node (VM)
  - `public static void migrateTo (Object o)`
    // joining another active object
  - `public static void migrateTo (Node n)`
    // ProActive node (VM)
  - `public static void migrateTo (JiniNode n)`
    // ProActive node (VM)
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)
Characteristics and optimizations

Same semantics guaranteed (RDV, FIFO order point to point, asynchronous)
Safe migration (no agent in the air!)
Local references if possible when arriving within a VM
Tensionning (removal of forwarder)