Software components and distributed systems

INF 5040/9040 autumn 2012

lecturer: Frank Eliassen

Literature

  - copies available at http://heim.ifi.uio.no/~frank/inf5040/CBSE/
- Coulouris chap 8.4, 8.5 and 8.6
- Recommended
A history of middleware

First generation middleware
- Exclusively based on *client-server model*
- Examples include Open Group’s DCE

Second generation middleware
- Based on *distributed object technology*
- Examples include CORBA and Java RMI

Third generation middleware
- Based on *component technology*

Issues with object-oriented middleware

- Implicit dependencies
  - It is not clear what dependencies an object have on other objects

- Interaction with the middleware
  - Many low-level details

- Lack of separation of distributed concerns
  - Security, transactions, coordination, etc.

- No support for deployment
Background for Java and CORBA platforms

- Known problems with CORBA and Java-RMI
  - How to deploy the components of my application?
  - Which services will be available on a given host?
  - Who activates my objects?
  - Who manages the life-cycle of my objects?

=> We need a standard development, deployment and runtime environment for distributed objects (CORBA, Java)

Explicit middleware: lack of “separation of concerns”

- Programs directly towards a middleware API
- Application logic entangled with logic for life cycle management, transactions, security, persistence, etc.
**Implicit middleware: better support for “separation of concerns”**

- Logic for life cycle management, transactions, security, persistence, etc. managed by the middleware
- Requirements for middleware services declared separately and can later be changed without changing the application code
- Middleware can be changed without changing the application code

![Diagram of component technologies]

**Component technologies**

- What is a component [Szyperski]?  

  “a unit of composition with contractually specified interfaces and explicit context dependencies only”

  “in this context, a component can be deployed independently and is subject to third-party composition”
Rationale for components

- Time to marked
  - Improved productivity/ reduced complexity
  - Focus on reuse
- Programming by assembly rather than by engineering
  - Reduced requirements to knowledge
- Most important advantage: development of server side?
  - (cf. EJB/JEE or CORBA Component Model - later)

Component platform

- A standard development, deployment and runtime environment can be designed as a set of contractually specified interfaces
- Contracts agreed between components and a component platform
- Component platform defines the rules for deployment (installation), composition and activation of components.
- For delivering and deploying a component is required a standardized archive format that packages component code and meta-data
Contracts

- What is in a contract?
  - Set of provided interfaces.
    - Some of these may be required by the component platforms
  - Set of required interfaces.
    - These must be offered by other components available in the container
  - Pre and post conditions/invariants
  - Extra-functional requirements: transactions, security, performance, ...

- Functions defined both syntactically and semantically
  - int add(int a, int b)
  - pre: a + b <= Integer.MAXINT
  - post: result' = a + b

- Extra-functional requirements
  - Guarantees: Response within 10 ms
  - Conditions: Needs 1000 CPU-cycles
  - Transaction requirements: e.g., create new transaction when component is invoked, serializable, ...

Composition

- Components and composition
  - Composition is the fundamental method for construction, extension and reuse of component-based software development
  - In contrast to (implementation) inheritance in object-oriented approaches

“Components are made for composition”
Connection-oriented programming

- Composition of pre-manufactured components
- Binding of incoming and outgoing interfaces
  - provided/required interfaces
  - Reflects direction of method calls
    - Not the direction of data flow
  - Outgoing interface
    - The method calls a component potentially may issue
- Support for distribution?
  - When the binding can be made across address spaces and computers

Third party composition

- The composition can be done by a third party external to the components themselves (loading and binding)
- Example
  - Connections (bindings), outgoing and incoming interfaces
  - Connects (binds) “matching” interfaces
  - Can be done during runtime by a third party
    - Can typically be realized by setting an appropriate attribute of the component with the outgoing interface (for C1, methods: setB, setV)
Composition: Reuse and assembly of components

An implementation of a component platform is often called a *container*.

Responsibilities of the container:
- life cycle management, system services, security
- dynamic deployment and activation of new components
  - e.g., resolve dependencies dynamically or activate components requested in method calls
  - Front-end for remote communication including interception of incoming invocations (cf implicit middleware)

Middleware that supports the container pattern: Application Server
Application Servers

➤ Advantage
  - Comprehensive support for one style of distributed programming
    - Three-tier approach, nodes complexities

➤ Disadvantages
  - Mandates a particular architectural style
    - E.g. three-tier architecture
  - Large and complex systems that works best on high-end servers
    - Performance and resources overhead

Key players

➤ OMG and components
  - CORBA v3 standard with CORBA Component Model (CCM)

➤ Microsoft and components
  - Development of COM/DCOM, COM+ and .NET

➤ SUN and components
  - Development of Java Beans and EJB
Enterprise Java Beans (EJB)

- Component architecture for deployable server side components in Java.
- EJB 3.0: based on Metadata facility in Java 5
  - annotations in source code
- EJB is managed
  - Container handles: transactions, security and lifecycle;
- Component Model
  - A bean is a component offering one or more business interfaces (provided interfaces)
    - Session Beans and Message-driven beans
  - Plain Old Java Objects (POJOs)
  - Annotations for Dependency injection (required interfaces)
  - Interception
    - Method Invocations
    - Lifecycle events (Creation and deletion of components)

Lightweight Component Model

- Component Models as EJB are heavyweight and prescriptive
  - Cannot be used for different classes of DS, such as peer-to-peer
  - Not suitable for constrained and embedded devices
- Need for a more stripped-down, domain-independent and minimal component model
  - Fractal
  - OpenCOM
  - OSGi
Fractal

- Programming with interfaces
  - Uniform model for *provided* and *required* interfaces
  - Explicit representation of the architecture
- No support for deployment, full container patterns, etc.
- Configurable and reconfigurable at runtime

---

Fractal

- Component Model
  - *Server* (provided) and *Client* (required) interfaces
  - Composition: *bindings* between interfaces
    - *Primitive Binding*: client and server interfaces within the same address space
    - *Composite Binding*: arbitrarily complex architectures (consisting of components and bindings) implementing communication between two or more interfaces potentially on different machines
  - Component model is *hierarchical*
  - System is fully configurable and reconfigurable: including components and their interconnections
Fractal

Fractal Ø

Ø Architecture Description Language (ADL)

<definition name="HelloWorld">
  <interface name="r" role="server" signature="Runnable"/>
  <component name="client">
    <interface name="l" role="server" signature="Runnable"/>
    <interface name="s" role="client" signature="Service"/>
    <content class="ClientImpl"/>
  </component>
  <component name="server">
    <interface name="s" role="server" signature="Service"/>
    <content class="ServerImpl"/>
  </component>
  <binding client="this.r" server="client.r"/>
  <binding client="client.s" server="server.s"/>
</definition>

Frank Eliassen, SRL & Ifi/UiO

Fractal

Fractal Ø

Ø Resulting architecture

Frank Eliassen, SRL & Ifi/UiO
Composing adaptive software using components

- Importance and interest in adaptive software is increasing dramatically
  - mobile, ubiquitous and autonomic computing
- Components play a major part
- Compositional adaptation (component reconfiguration)
  - dynamic adaptation of architecture of component-based application
    - change component impl
    - redeploy component
    - parameter adaptation
    - change overall architectural framework
    - combinations of the above

MUSIC middleware: Adaptability through component frameworks

Resources: ist-music.berlios.de
Summary

- Components
  - Programming according to LEGO-principle
  - Contractually specified interfaces and composition
  - Support for connection oriented programming

- Component architecture
  - Contractually specified interfaces between components and application servers
  - Realizes “implicit middleware”
  - Application servers: Java: EJB, CORBA: CCM ...
  - Light-weight component models: Fractal, OSGi ...