Challenges in component based programming

Lena Buffoni
Challenge: Size & complexity

• Software is everywhere and increasingly complex (embedded systems, internet of things...)

• Single products have become product families
  • different versions of the same product
  • compatible products from the same set
Challenge: Software life-cycle

- 40-80% development time is dedicated to modifying existing software*
- up to 70% of software cost goes to maintenance*

*approximate statistics

Needs change!
Challenge: Software life-cycle

• Some software is meant to operate 30+ years
• Applications need to integrate components they were never intended to use
• Software flexibility is very important
  → systems should be built to facilitate change
  → easy removal and addition of functionality
Challenge : Time & Cost

• Time-to-market needs to be short in a competitive environment

• Incorporate successful innovations quickly

• Pressure to reduce development costs

• In a short development cycle software still needs to be reliable and stable between updates
Are component models the solution?

*They are a big step in the right direction*

“It is not the strongest of the species that survive, nor the most intelligent, but the one most responsive to change”

- Charles Darwin
What is included in a component model?

A definition of standards for
- component implementation
- component documentation
- component deployment

- The component model specifies how interfaces should be defined and the elements that should be included in an interface definition.

Relate to meta-modeling from previous lecture
Component-based Software Engineering is concerned with the rapid assembly and maintenance of component-based systems, where components and platforms have certified properties these certified properties provide the basis for predicting properties of systems built from components.

Volume II:
Technical Concepts of Component-Based Software Engineering, 2nd Edition
SEI in CMU/SEI-2000-TR-008
So what are the goals of a good component model?

- Substitutability
- Extensibility
- Decomposability
- Reliance on standards
Horizontal vs. Vertical composition

Vertical standards

- User GUI
- Hardware drivers

Horizontal standards

Standardization to avoid market fragmentation
What is independent deployment?

- never partially deployed
- no dependencies on peer-components
- some ‘meaningful’ functionality by itself components tend to be ‘large grained’
- deployment-friendly
Architecture vs. Generic Components

A specific place for every component

Comparable to top-down vs. bottom-up design strategy?
Reality?

Modern software development
Building systems from existing components

• Assuming that a collection of high-quality components is available, a system built from these components will be high-quality
  – The cost of high quality components is amortized multiple times
  – Reuse of the same components will reduce errors and improve testing
Interoperability problems

• How will one component find another component at run-time?
• Which instance should be used if multiple versions of a component are present at run-time?
• What happens if one component needs to be shut down temporarily?
• How does development and evolution of one component of a family of systems impact other components?
Interoperability problems (continued)

• Are components bloated by code that is unrelated to its specific task in a system?
• A component is almost a perfect fit for a new system. Can the existing component be extended in unanticipated ways without touching the source code?
• How can components that run on different platforms interoperate? What if the platforms run different operating systems?
How do we tackle these problems?

- Communication standards and protocols for:
  - Calling concept (messages, method-calls...)
  - Data types (standard types, user defined types...)
  - Argument passing (by value, by reference...)
  - Object persistence (data-base, garbage-collection...)
  - Object identification (unique identifiers, paths...)
  - ...
Example of underspecified standards

Provider encodes in little endian

This kind of error might be very hard to detect

Client interprets result as big endian
So a component model specifies:

- Life-cycle management:
  - instantiation,
  - (de)activation,
  - removal
- Binding mechanisms
- Interaction style
- Data exchange format
- Process model
- Packaging Model
Break here?
Component Platform

- A component platform is the run-time infrastructure of the component model.
- For example

  *Java components*  
  *JVM*  
  *Windows*  
  *Unix*
Middleware

• Software that connects two or more software applications so that they can exchange data.

"the software layer that lies between the operating system and the applications on each side" of a distributed computing system
The stub & skeleton proxy pattern

Client-side proxy of the component

Call

Language adaptation

local version

Client 1 in Java
Stub

Client 2 In Python
Stub

Application In C++
Skeleton
Proxy pattern

Client

calls

<<interface>>

Application

+ call(Data)

Proxy

+ call(Data)

represents

implements

ApplicationImpl

+ call(Data)
Distributed component systems

- Client 1 in Java
- Client 2 in Python
- Application in C++
- Stub
- Stub
- Skeleton

On another machine

Local Call

Remote Call

Now need to: handle data transportation, remote procedure calls...
Reference problem

• How to keep references across multiple platforms?
• References in composite parameters and results
• Scope of references?
• Target of calls
Unique naming conventions

• World-wide unique addresses
  E.g. computer address + local address URL, URI
  (uniform resource identifiers)

• Mapping tables for local references
  – Logical-to-physical
    Consistent change of local references possible
  – (In principle) one adapter per computer manages references
Stubs, Skeletons, and Adapters

Groups everything need to handle communication protocols, identification, referencing...
Who implements Stubs and Skeletons?

- Stubs and Skeletons are
  - Component interface dependent
  - Implementation language dependent
- Manually – error prone, time consuming
- Automatically?
Interface Definition Language (IDL)

- A language to define the interfaces of components
- IDL interfaces can be used to automatically generate stubs and skeletons (OMG IDL, MIDL...)

Define component interface → Automatically generate Stub and Skeleton → Implement the component
Persistence of objects

• The ability of an object to survive the lifetime of the OS process in which it resides.
• The object state needs to be retained
• Store and load the object

Persistent storage can be

– File system
– Relational Database
– Object-Database
– Flash-RAM
The OMG way: CORBA, CCM, OMA, and MDA

OMG never settled on “binary” standards (standards at the level of deployable executables) – everything is carefully standardized to allow for many different implementations and for individual vendors of CORBA-compliant products to add value. The downside of this very open approach is that individual CORBA-compliant products cannot interoperate on an efficient binary level, but must engage instead in costly high-level protocols. The most prominent, although only moderately efficient, interoperability protocol is OMG’s inter-net inter-ORB protocol (IIOP), standardized with CORBA 2.0 in July 1995. Any ORB claiming interoperability compliance has to support IIOP. In the July 1996 update of the CORBA 2.0 standard, an interworking standard was added, which specifies the interworking of CORBA-based systems with systems based on Microsoft’s COM (see Chapter 15).

CORBA essentially has three parts: a set of invocation interfaces, the object request broker (ORB), and a set of object adapters. Invocations of object-oriented operations – also called method invocations – require late binding of the implementation. The method implementing the invoked operation is selected based on the object implementation to which the receiving object’s reference refers. Invocation interfaces enable various degrees of late binding. They also marshal an invocation’s arguments such that the ORB core can locate the receiver object and the invoked method and transport the arguments. At the receiving end, an object adapter unmarshals the arguments and invokes the requested method on the receiver object. Figure 13.1 illustrates the basic CORBA structure in simplified form.

For invocation interfaces and object adapters to work, two essential requirements need to be met. First, all object interfaces need to be described in a common language. Second, all languages used must have bindings to the common language. The first condition enables construction of generic marshaling and unmarshaling mechanisms. The second allows calls from or to a particular language to be related to the common language. This common language formed an essential part of CORBA from the beginning and is called OMG interface definition language (OMG IDL). Here is an example of an OMG IDL specification:

```
Language/implementation/platform

obj.m(args)

Invocation interface
ORB interface
ORB interface
Object adapter

Language/implementation/platform barrier

Method 1
... Method n

Object
```
OMG Interface Definition Language

- Compiled and deposited in interface repository
- Used for automatic stub and skeleton generation

```plaintext
module Example { struct Date {
  unsigned short Day; unsigned short Month;
  unsigned short Year;
}
  interface Ufo {
    readonly attribute unsigned long ID; readonly attribute string Name; readonly attribute Date FirstContact; unsigned long Contacts ();
    void RegisterContact (Date dateOfContact);
  }
}
```
preferable to read relevant parts of Chapter 14 before proceeding.) Individual components are shipped in component packages that contain an XML document detailing their contents, which can include binaries for multiple platforms. CCM assemblies contain an XML document describing the set of component packages they refer to and the deployment configuration of these.

A CCM component itself can consist of multiple segments. CCM runtimes load applications at the granularity of segments (see Figure 13.5). As CCM requires special server-side support, CCM applications can only be executed on CORBA 3-compliant ORBs. Perhaps surprisingly, a CORBA 3 ORB is also required at the client end to achieve full CCM fidelity. CCM does support so-called component-unaware clients on pre-CORBA 3 ORBs, but such clients will not have access to several CCM features, such as navigation operations.

A CCM component is classified into one of the four categories of service, session, entity, and process components. (The session and entity categories correspond to stateful session and entity beans in EJB, respectively.) A CCM application provides declarative information on component categories and component factories. A POA uses this information to create and assign servants to component instances. Service components are instantiated per
Interoperable Object Reference (IOR)

• A unique ID used to locate the object

    module CORBA {
        pseudo interface ORB {
            string object_to_string (in Object obj);
            Object string_to_object (in String ior);
        }
    }

• Object reference shipped by server to clients and stored in the proxy

• Contains
  – Type name
  – Protocol and address details
  – Object key
Java beans

- Reusable software components
- Written in Java

Flowchart:
- Client
- Server
- Stubs
- Skeletons
- Remote Reference Layer
- Transport
Java beans – main aspects

• **Events** Beans can announce that their instances are potential sources or listeners of specific types of events. An assembly tool can then connect listeners to sources.

• **Properties** Beans expose a set of instance properties by means of pairs of getter and setter methods.

• **Introspection** An assembly tool can inspect a bean to find out about the properties, events, and methods that a particular bean supports.

• **Customization** Using the assembly tool, a bean instance can be customized by setting its properties.

• **Persistence** Customized and connected bean instances need to be saved for reloading at the time of application use.
Properties

A bean defines a property $p$ of type $T$ if it has accessor methods that follow these patterns:

**Getter**

- public $T$ get$P()$

**Setter**

- public void set$P(T)$

**Boolean getter**

- public boolean is$P()$
Events

- Event (MouseOver)
- Event source – generates an event (mouse hovers over a button)
- Event listener - triggers some behavior when an event is detected (button is highlighted)

```java
public void add<Event>Listener(<Event>Listener a)
```
Events - issues

- Exceptions—an exception in a bean can propagate up to unrelated beans that just happened to be on the call stack.
- Security—JVM grants any function the least privileges of all functions on the call stack.
- Concurrency—JavaBeans neither enforces nor encourages any global synchronization
A property can be...

- Indexed: an array instead of a single value, additional index specific getter and setter are defined

  ```java
  public int[] getTestGrades() { return mTestGrades; }
  ```

- Bound: changes to a bound property trigger the firing of a property change event.

- Constrained: the property setter method is declared to throw PropertyVetoExceptions.
Introspection

- Introspector()
- Can be used by builder tool and other tools to provide information about a bean

```java
BeanInfo info = Introspector.getBeanInfo(SimpleBean.class);
    for ( PropertyDescriptor pd : info.getPropertyDescriptors() )
        System.out.println( pd.getName() );
```
Working with Beans: BeanInfo
Persistence

Object serialization – converting object into byte stream
  – Serializable interface: serialization and deserialization methods
  – Mark fields transient to avoid serialization
  – Information can be sent across platforms

Long-term persistence – XML
  – XML encoder and decoder

Serialization is Not Secure!

Updates can be split over several steers
Consistent state?
Virtualization: Portability on machine level

Managing resource virtualization and distribution
Virtualization

• Abstract the hardware and infrastructure
• Allow a unified user experience
• Energy saving
• Secure

But:
• Overhead
• Compatibility limitations
• Shared resources

VMWare, VirtualBox …
Containerization: Portability on OS/application level

- App A
- App B
- App C

Container Engine

Host OS

Hardware
Containerization

• Lightweight
• Portability
• Less overhead
• Breakdown into smaller chunks

But

• Less isolation
• No hardware virtualization
• Still... overhead

Complementary approaches?

Docker, Rocket…
Some of the current trends - how do they impact component based systems?

- Open-source
- Embedded and real time systems
- Safety and security guarantees
- Requirement driven software development
- Model based development
- Green computing
Questions?

www.liu.se