Software Components
-An Introduction-

Jean-Charles Tournier
tournier@cs.unm.edu

SSL Lab - 10/19/2005
Objectives

• At the end of the talk, you should know
  – What is a component
  – Why and when components are useful
  – How to classify/identify a component
  – What are the actual issues in components
Context

- Software Engineering
  - Better code, to reduce development time
  - Mastering software complexity (structuration)

- Solutions
  - Functions/procedures
  - Libraries/modules
  - Objects
  - Components
Why components?

- Because of object's drawbacks
  - No explicit specification of required resources (e.g. Objects or services)
  - No global view of an application/system
  - Hard to separate functional and non functional concerns
  - Do not address deployment and administration issues
What is a component?

- No global agreement about component definition
  - Each component model provides its own definition
  - Basic idea: component = software brick

- But, usually a component is defined as:

  « *A component is a unit of composition with contractually specified interfaces and fully context dependencies that can be deployed independently and is subject to third-party composition »

Component model and component framework

• To use components, we need:
  – A component model
    • Define (formally) what is a component, its interaction modes and its composition modes
  – A component framework
    • Is an implementation of a model
    • Allows to build, deploy, execute, manage components conform to a specific model
Component model

- Encapsulation
  - Interfaces = unique access points
  - Configurable properties = tunes a component (improve re-usability of components)

- Composition
  - Dependencies description (provided and required resources)
  - Hierarchical composition (a component made of components)

- Global description
  - Description language (e.g. ADL)
  - As formal as possible in order to prove some properties (e.g. type checking) or to automate deployment/reconfiguration
Component framework

• Covers all the life-cycle
  – From design phase to execution phase

• Non-functional properties
  – Persistence, transactions, security, QoS
Expected benefits of components

- Clear description
- Global view
- Properties V&V

- Less development time thanks to components reuse

- Automatic deployment
- Component grain deployment

- Dynamic properties V&V
- Dynamic reconfiguration or adaptation
A non-exhaustive list of components models

- UML 2.0, SDL, Fractal, EJB, CCM, COM/COM+/DCOM, .NET, OSGi, PECOS, VEST, Koala, Rubus, Choices, OS-Kit, Coyote, PURE, 2K, MMLite, Pebble, eCos, ..... 

- But, how to classify them, what are the differences?
Some classification criteria

- Programming language
  - specific/independent
- Support of non-functional properties
- Flat/hierarchical
- Open/closed
- Full/partial life-cycle supports
- Targeted domain
  - Application level/operating system level/independent
A preliminary classification

- Application independent
  - UML 2.0, SDL, Fractal
- User application level
  - EJB, CCM, DCOM, .NET, OSGi
- Operating system level
  - PECOS, VEST, Koala, Rubus, Choices, OS-Kit, Coyote, PURE
Characteristics of independent component models

- All of them are hierarchical
- UML 2.0 and SDL can only be used at design stage
- UML2.0 is the most complete, and complex, one
  - Notion of ports (group of interface), interface protocols
  - But no implementation and no tools
- Fractal covers design, development and execution stages
  - Several implementations: Julia for application level and Think for OS level
  - It's an open model (e.g. extend the ADL, add non functional properties, deployment support, etc.)
Characteristics of application level component models

- Heavy framework (e.g. Jonas is more than 160 MB)
- Deals with distributed application and classical non-functional properties (persistence, transactions, security)
- Flat and non-extensible models
- Components at development and execution stages
- EJB and OSGi are only for Java programmers
- .NET is for Windows users (Mono must be tested...)
- CCM is language and OS independent

- Interests of these models
  - Containers approach: programmers focus only on the functional part
  - Dynamic deployment
Characteristics of OS level component models

- Target embedded systems
- Low level languages (C/assembly)
- Some of them deals statically with real-time constraints or memory consumption
- Components are only available at development stage, at runtime the system is monolithic
- Most of them are based on a micro-kernel
Some actual issues

- No standard, no common agreement on a component model
- Dynamic adaptation for OS level component model
- Composition from a functional point of view is achieved (almost), but what about non-functional composition?
- Composition of several non-functional properties (QoS, security, persistence, RT, etc.)
Conclusion

• What is a component?
  – A unit of composition with well described resources (provided and required)

• Why and when components are useful?
  – To structure the code and to reduce development time
  – At each stage of the software life-cycle
    • Execution stage can be omitted if no need for dynamic evolution....

• How to classify/identify a component model?
  – A primary classification is on application domains

• What are the actual issues in CBSE?
  – No standard, need for dynamic support and non-functional properties composition
Next

- A more detailed presentation of the Fractal component model and its implementation for OS development (called Think)
- A detailed (and formal?) comparison of OS level component models