



Joint Program Executive Office Joint Tactical Radio System

Component Model Introduction



24-25 August 2010
JTRS SCA Working Group

JPEO JTRS



Task Overview

- **Objective**
 - Formalize **Component** concept into the SCA
 - SCA has numerous references to **Components**, but does not define the term
 - SCA lacks a clear representation that captures the runtime characteristics of instantiated elements
- **Benefits**
 - Improves the clarity and consistency of the specification
 - provides concrete bridge from interface to implementation which will be more important as optional capabilities are introduced
 - highlights implementation expectations, responsibilities and requirements
 - Provides a platform that establishes a convention for the use of Modeling and of Model Driven Development techniques in the development of SCA compliant products
- **Impact**
 - Should be negligible for most existing implementations
 - Will necessitate changes in current SCA object models
 - SCA document will look significantly different, with general behavioral requirements being allocated to a **Component** rather than attempting to be assigned to an **Interface**



Core Definitions & Ramifications

- **Interface**

1. “A shared boundary or connection between two entities ...”
2. Specifies a well-defined – and limited – “**Role**” which needs to be fulfilled, which may be:
 - “*functional*” – definition of specific behavior to be performed; “**to DO**”
 - “*non-functional*” – criteria used to judge the qualities of operation: “**to BE**”
3. Defines **WHAT** needs to be done, **WHY** it needs to be done, but **NOT HOW** to do it – as-such, most pure **Interfaces** tend to be stateless
4. Since a ‘good’ **Interface** needs to define a limited **Role**, and complex system elements generally need to fulfill multiple **Roles**, often multiple separate **Interfaces** are required to fully-define the set of *functional* and *non-functional* requirements:
 - quite often, there are interactions between multiple **Interfaces** – i.e., only certain sequences of use provide useful functionality
 - therefore, it is often-useful to package these interactions between multiple Interfaces into an integrated unit of defined behavior – a **Component**



Core Definitions & Ramifications

• Component

1. *“An autonomous unit within a system or subsystem ...”*
 - provides one or more **Interfaces** which users may access
 - internals are hidden and inaccessible other than as provided by its **Interfaces**
2. Encapsulates a modular, replaceable part of a system within its defined environment
 - implements its own self-contained lifecycle, which may include sequential interaction requirements which exist between multiple provided **Interfaces**
 - presents a complete and consistent view of its execution requirements (MIPS, memory, etc) to its physical environment
 - serves as a **Type** definition, whose conformance is defined by both the ‘*provided*’ as well as the ‘*required*’ **Interfaces**
 - encompasses both static and dynamic semantics (behaviors)
 - represents how developments may implement their software elements when multiple **Roles / Interfaces** must be fulfilled – e.g., a “**CF::Resource**” **Component**, which must fulfill the **Roles / Interfaces** prescribed by “**CF::PortSupplier**”, “**CF::PropertySet**”, “**CF::LifeCycle**”, and “**CF::TestableObject**”



Interfaces .vs. Components – Why A Component Model Is Required

Interface Characteristic	Component Characteristic
Role -oriented à best suited as problem domain / analysis-level abstractions	Service -oriented à best suited as solution domain / functional-level abstractions
Conceptual / Abstract / Unbounded Responsibilities	Practical / Concrete / Constrained Responsibilities
Have no implementation mechanisms	Can – and often do – provide prototype or default implementations
A NECESSARY – though NOT SUFFICIENT – element of Portability and Detailed Architecture / Design Reuse	Properly-developed, Components improve – but do not guarantee – prospects of Portability and Detailed Architecture / Design Reuse
Interfaces are generally SYNTAX without an underlying SEMANTIC definition, and are generally seen as STATELESS as a result	Components MUST HAVE well-defined SEMANTIC baselines because they fulfill multiple Roles within a Framework à Components are MUCH-MORE than the sum of the Interfaces which they implement
Preferred in THEORY	Preferred in PRACTICE



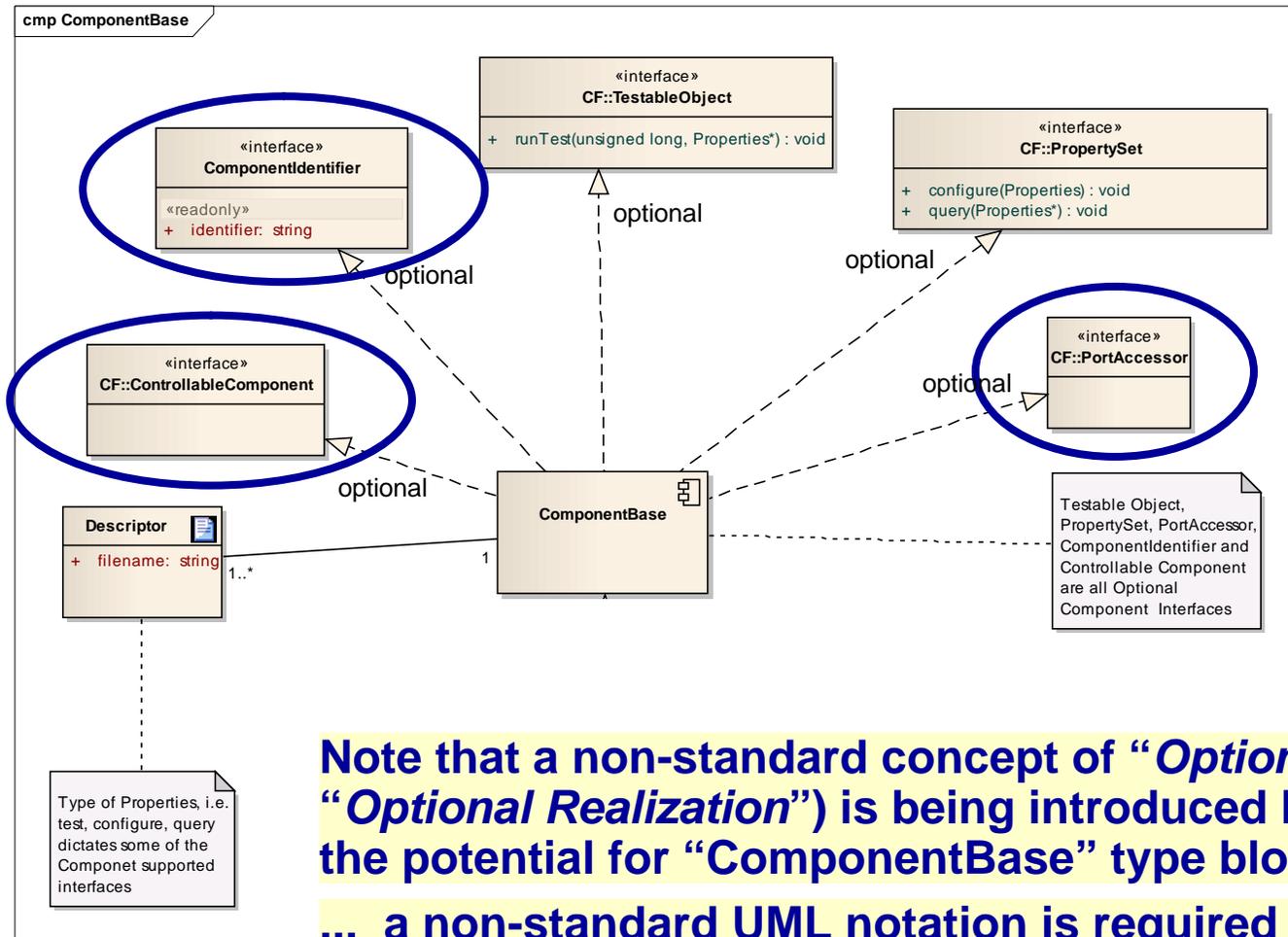
Current Working Set Of Candidate Components

- Resource
- Resource Factory
- Device
- Loadable Device
- Executable Device
- Aggregate Device
- Device Manager
- Domain Manager
- Application Resource
- Application
- Application Manager
- Service
- File
- File System
- File Manager
- Application Factory

Not Surprisingly, The Initial Set Of Candidate **Components Being Contemplated Follow The Current SCA Core Framework **Interface** Definitions ...**



Conceptual Definition Of The “ComponentBase”



Additional Interfaces Defined:

- ComponentIdentifier
- PortAccessor
- ControllableComponent

Note that a non-standard concept of “*Optional Inheritance*” (aka: “*Optional Realization*”) is being introduced here in order to reduce the potential for “ComponentBase” type bloat ...

... a non-standard UML notation is required here to indicate this concept, which is simple to accomplish within the IDL definitions of the Interfaces inherited / realized but currently lacks a standard OMG UML notation

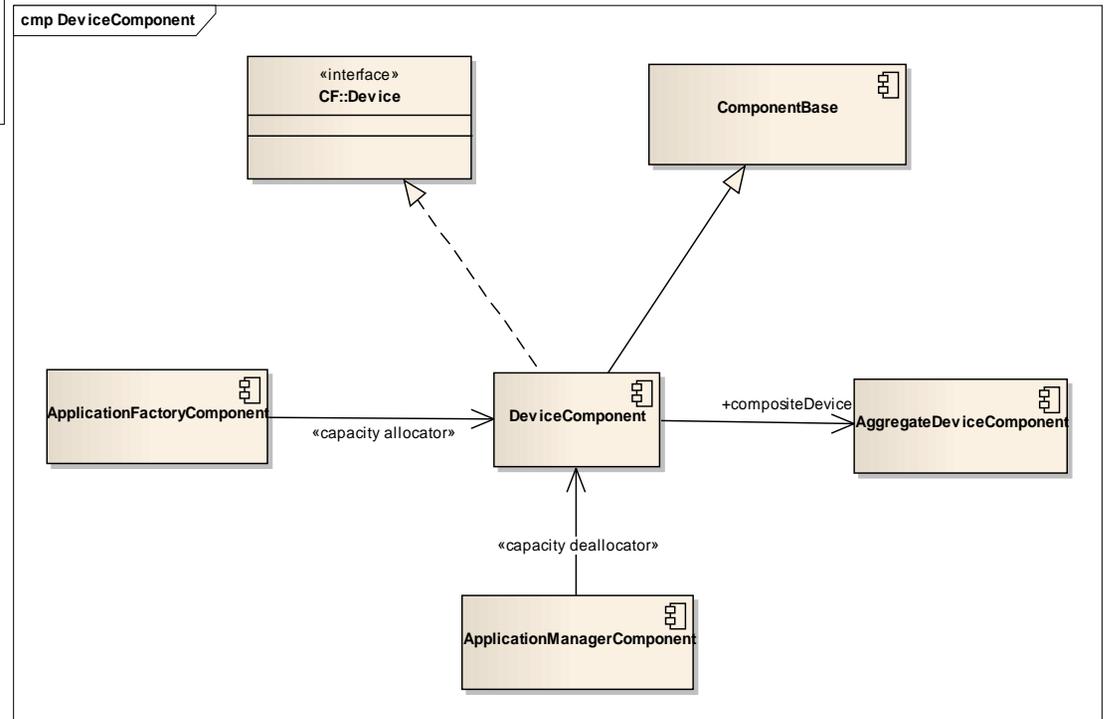
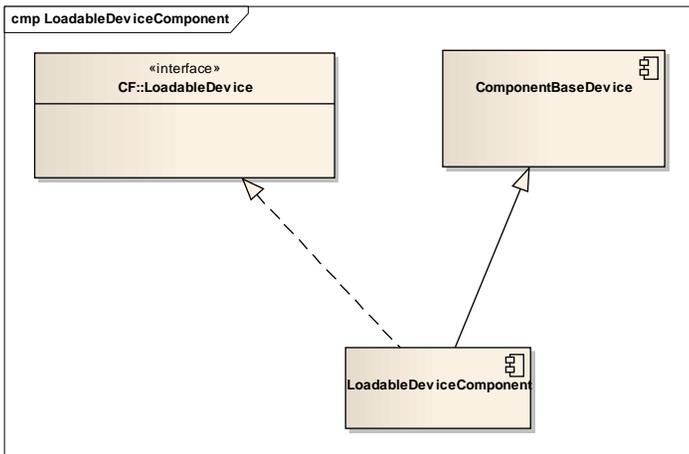
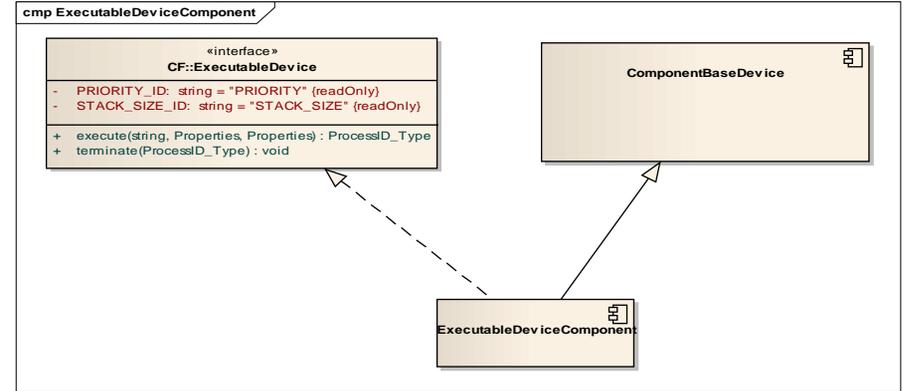
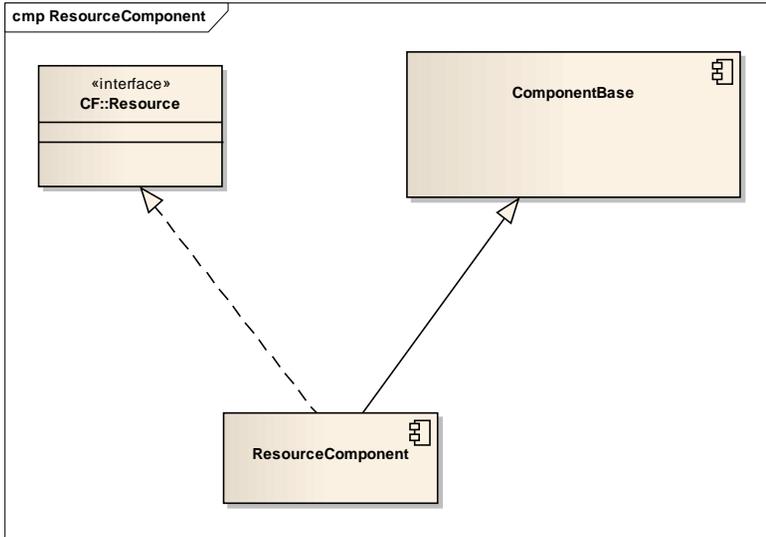


Sample of “ComponentBase” Associated Requirements

- A **Component** may realize the **ComponentIdentifier** interface.
- A **Component** shall be associated with at least one descriptor file.
- If a **Component** has test properties, then the component shall realize the **TestableObject** Interface.
- If a **Component** has configure and/or query properties, then the component shall realize the **PropertySet** interface.
- If a **Component** has dynamic provides **Ports**, then the component shall realize the **PortAccessor** interface.
- If a **Component** has uses **Ports**, then the component shall realize the **PortAccessor** interface.
- A **Component** that realizes the **TestableObject** interface shall support all the **TestProperty** types and values as stated in the component’s descriptor.
- A **Component** that realizes the **PropertySet** interface shall support all the configure and/or query properties as stated in the component’s descriptor.
- The **PortAccessor** interface shall support all the uses or dynamic provides **Ports** as specified in the component’s descriptor that realizes this interface.
- The **connectPort** operation shall support all of the uses **Ports** identified in the component’s descriptor.
- **Components** shall implement a **ConfigureProperty** with a name of PRODUCER_LOG_LEVEL when connected to a log service. The PRODUCER_LOG_LEVEL **ConfigureProperty** provides the ability to “filter” the log message output of a Component. This property may be configured via the **PropertySet** interface to output only specific log levels.
- **Components** shall output only those log records to a **LogService** that correspond to enabled log level values in the PRODUCER_LOG_LEVEL attribute. Log levels that are not in the PRODUCER_LOG_LEVEL attribute are disabled.
- **Components** shall use their identifier in the log record output to the **LogService**.
- **Components** shall operate normally in the case where the connections to a **LogService** are nil or an invalid reference.
- A **Component** may also take on additional behavior for life cycle management by realizing the **Lifecycle** interface that would be used during deployment and teardown of a component.
- A **Component** may also realize the **ControllableComponent** interface to provide overall management control of the component.
- The **PropertySet::configure()**, **PropertySet::query()**, and **Testable::runTest()**, and **Resource::start()** operations are not inhibited by the **Resource::stop()** operation.
- A fundamental state model relationship exists between **Lifecycle** and **ControllableComponent** interfaces.



Some Example Component Types – A General Pattern Emerges ...





Component Description and Documentation Requirements

- A format for the **Component** descriptions (documentation) is currently being established:
 - The established SCA precedent for Interface definitions (description, UML, types, attributes, operations) is not applicable for **Component** definitions
 - Proposal involves a 4 part structure:
 - **description** – summary of what the **Component** is
 - **associations** – identifies and explains relationships between the **Component** and other system elements
 - **constraints** – this encompasses restrictions on the structural features of the **Component**
 - **semantics** – explanation of the expected behavior of the **Component**, this may contain supporting text, rationale and requirements



SCA Wide Implications

- Existing SCA v2.2.2 convention of italicized text (e.g. “*resource*” will be replaced by its “**xComponent**” counterpart
- CORBA neutral PIM operation signatures will be replaced with “**xComponent**” when referring to an SCA defined interface type – these references will revert back to the interface name in the PSM representation



Backup Slides



Component Hierarchy

