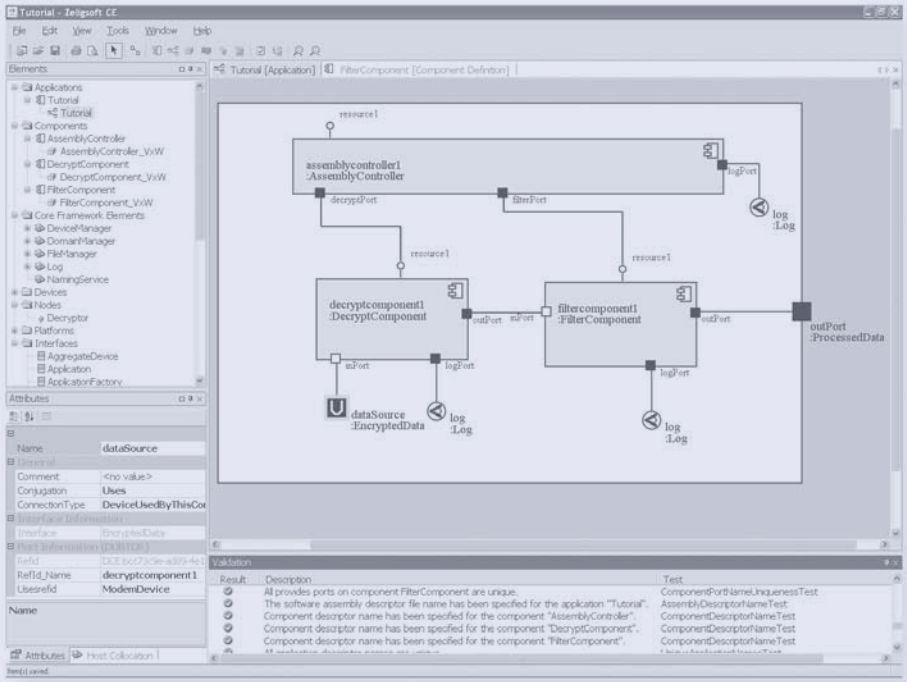
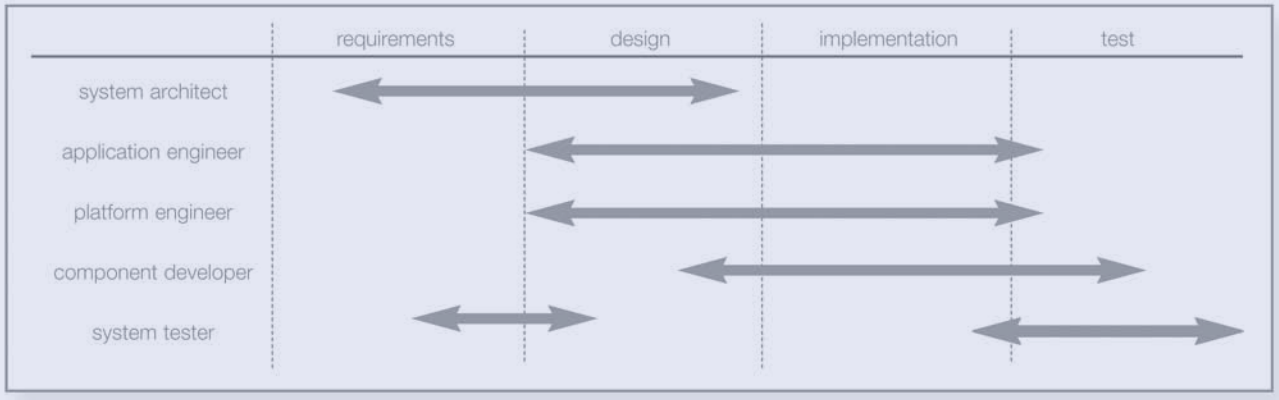
Code Generation for SCA Components

Mark Hermeling

zeligsoft


Tutorial - Zeligsoft CE
File Edit View Tools Window Help
Elements
Applications
Tutorial
Components
AssemblyController
DecryptComponent
FilterComponent
Core Framework Elements
DeviceManager
DomainManager
FileManager
Log
NamingService
Devices
Nodes
Decryptor
Platforms
Interfaces
AggregateDevice
Application
ApplicationFactory
Tutorial [Application]
FilterComponent [Component Definition]
resource1
assemblycontroller1
:AssemblyController
logPort
decryptPort
filterPort
log
:Log
decryptcomponent1
:DecryptComponent
outPort
inPort
filtercomponent1
:FilterComponent
outPort
:ProcessedData
dataSource
:EncryptedData
Attributes
Name
dataSource
Comment
Conjugation
Uses
Validation
Result
Description
Test
All provides ports on component FilterComponent are unique.
ComponentPortNameUniquenessTest
The software assembly descriptor file name has been specified for the application "Tutorial".
AssemblyDescriptorNameTest
Component descriptor name has been specified for the component "AssemblyController".
ComponentDescriptorNameTest
Component descriptor name has been specified for the component "DecryptComponent".
ComponentDescriptorNameTest
Component descriptor name has been specified for the component "FilterComponent".
ComponentDescriptorNameTest


requirements
design
implementation
test
system architect
application engineer
platform engineer
component developer
system tester

Code Generation for SCA Components

Mark Hermeling

The definition and coding of a component that makes up part of an SCA system requires detailed knowledge of the SCA standard, and continuous attention to the external interface of the component. SCA specific coding tools and automation used in the development process can make component development much more manageable and can thus accelerate SCA development, and improve software quality.

This paper provides an overview of the source code requirements for an SCA component, and the design considerations that should be taken by the developer. This paper then describes how SCA artifacts can be automatically generated from a component definition built with a UML 2.0 compliant visual modeling tool, and how this implementation can be customized for use with different SCA Core Frameworks, CORBA ORBs, coding patterns, and target operating systems. Without customization, few projects can benefit from automation. Domain specific tools and customizable code generation allow developers, both experienced and inexperienced with the SCA, to formulate and produce SCA compliant radios quickly, and with high quality.

1 INTRODUCTION

The Software Communications Architecture (SCA) is a powerful framework for realizing flexible, reusable component-based applications. SCA components are composable artifacts that can be independently deployed, configured and connected together.

A component provides functionality to other components through *provides* ports, and uses functionality from other components through *uses* ports. Components can be assembled into larger entities by connecting their interfaces together through connectors. Components can be individually configured according to the role they play in an application.

This paper provides an overview of the code that needs to be authored for SCA compliant components and how automated code generation for SCA artifacts can benefit projects. Section 2 provides a detailed description of the parts of an SCA component. Section 3 discusses component design considerations, and section 4 describes the advantages and challenges designers face when coding SCA components. Once these challenges have been outlined, section 5 describes how a code generation tool needs to be customizable to accommodate different coding patterns, SCA Core Frameworks, and implementation languages. Without customization, few projects can benefit from automation. Zeligsoft Component Enabler is a tool that provides customizable code generation of SCA artifacts. Section 6 summarizes this paper and explains how Zeligsoft CE can be used today.

2 SCA COMPLIANT COMPONENTS

This paper focusses on SCA compliant components only. SCA compliant components are CORBA-capable components that typically execute on a general purpose processor. A typical application consists of a combination of SCA compliant and non-SCA compliant components. The latter are usually executed on non-CORBA capable processors such as FPGAs and DSPs. Non-SCA compliant components will be covered in a later paper.

An SCA compliant component has an external interface, consisting of ports and supports-interfaces, for communication with other components. A component has user-defined properties and one or more implementations. An SCA compliant component is deployed to the platform and started by the SCA Core Framework (CF).

The elements of an SCA component are described below.

2.1 Component Implementation

A component implementation is written in a specific language and typically compiled for a specific operating system and processor. Multiple implementations can exist for a component, which makes it possible to run the application on different physical platforms (radios).

The implementation has to implement the elements that make up the definition: the ports and the attributes. The SCA defines certain rules and behavior that the implementation must adhere to. These rules enable the SCA Core Framework to deploy and configure the component and assemble a set of components into a complete application. The following need to be present in a component implementation:

2.2 Component Entrypoint

An implementation needs an entrypoint function. This is the function that the CF needs to start the component. This function will be called with a number of SCA required, as well as user-defined parameters. The SCA required attributes provide, among others, the CORBA naming context and name that the implementation uses to register itself. The user-defined parameters (see section 2.5) are the *exec-param* parameters as defined on the component definition or implementation. User-defined attributes can be passed to the entrypoint function (to provide information) to initialize the ORB (the –ORBInit parameter for example).

The entrypoint function needs to start the CORBA object for the implementation and register it in the naming service.

Some component implementations might not have an entrypoint. Typically these are dynamic link libraries or other non-CORBA capable components.

2.3 SCA Interfaces

At the very least the component needs to implement the *CF::Resource* interface. This is the main control interface the SCA CF uses to initialize, connect, query and configure the component. Figure 1 shows the SCA *CF::Resource* interface.

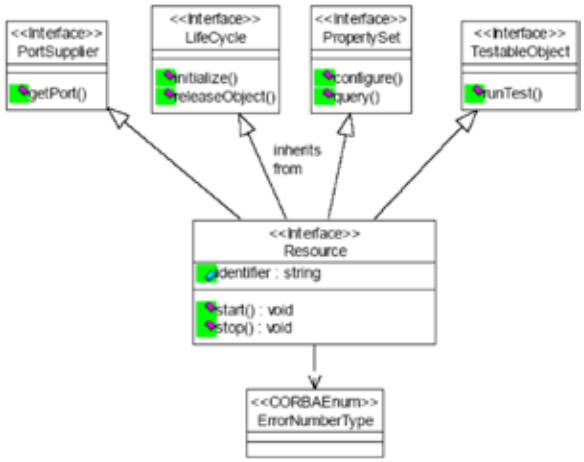


Figure 1: SCA Base Application Interfaces

2.4 Component Ports

Components communicate with other components through ports defined by an IDL interface. The component implementation needs to create the port objects and provide them to the CF when requested through the *getPort()* operation.

Ports are separate CORBA objects that completely isolate component internals from the environment. These ports provide incoming as well as outgoing encapsulation (two-way encapsulation).

Ports have a conjugation and can be either *uses* or *provides* ports. A provides port is the server (servant in CORBA terminology) and a uses port is the client. A uses port must implement the *CF::Port* interface. This interface is used by the CF to create connections between different ports. Both *uses* and *provides* ports must implement the interface used on the port. The *uses* port forwards calls on that interface from the inside of the component to the servant. The *provides* port forwards CORBA calls from the environment to the internals of the component.

The responsibility of a port is to provide the separation between the internals of a component and the environment of the component. Component internals constitute the functional behavior of the component, for example signal processing behavior or control behavior. The port translates (marshalls) data, when necessary, between the internal format and the format expected by the environment (CORBA calls), and passes the data on to the receiver.

2.5 User-Defined Attributes

The user-defined attributes of a component are the configurable properties of a component. They can, for example, be used to configure modulation rates, frequencies and so forth.

Properties can be one or more of the types: *allocation, configure, execparam, factoryparam* and *test*. For SCA compliant components only *configure, execparam* and *test* are relevant. *Allocation* is relevant for devices and *factoryparam* is relevant for resource factories. They are beyond the scope of this paper.

The *execparam* parameters are used in the entrypoint function mentioned in section 2.2. The *test* parameters provide input and expected output to the *runTest* operation of the *TestableObject* interface.

The *configure* parameters are attributes that can be used as configuration values for the component. The application that uses a component can override the default values for the configuration parameters and in that way modify the behavior of the component for this specialized usage. The parameters can be configured and requested through the *configure* and *query* operation of the *CF::PropertySet* interface.

3 DESIGN CONSIDERATIONS FOR COMPONENT IMPLEMENTATIONS

The SCA defines what the component needs to implement, but does not specify in detail how the implementations should be designed. The SCA covers the external interface of components. The designer has a lot of freedom in the actual implementation of the interface. The designer decides how a port or a configure attribute should be implemented. The only thing the SCA defines is the external behavior of these.

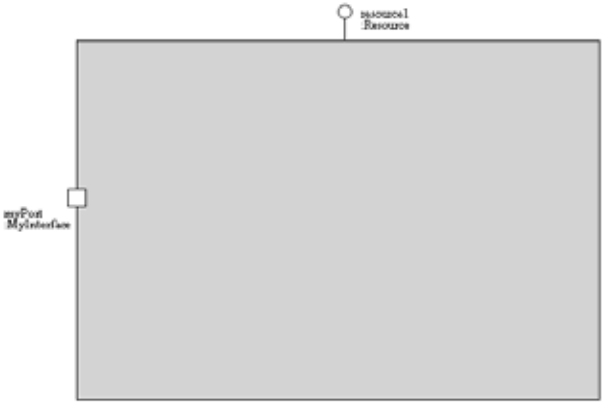


Figure 2: MyComponent

3.1 Examples of Component Implementations

As an example, consider a component *MyComponent* with a single provides port *myPort* of interface *MyInterface* as in Figure 2. This component can be implemented in a single class *MyComponent* that implements the *Resource* interface. A class *MyComponent_start* together with a *main()* function can provide the entrypoint function. Class *MyComponent* can also implement *MyInterface* and hence play the role of the port. See Figure 3.

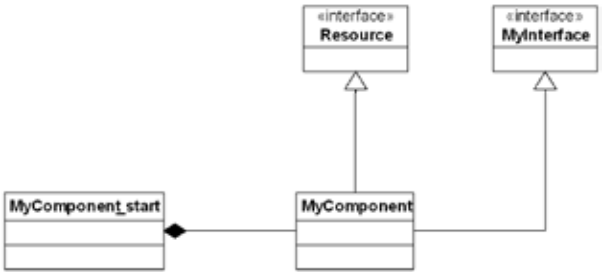


Figure 3: Design One

An alternate implementation can instead implement the port *myPort* in a completely different class *MyInterfaceProvides*. *MyComponent* has an attribute of *MyInterfaceProvides* by the name of *myPort*. *myPort* forwards all calls to *MyComponent* for processing by the functional aspect of the component. See Figure 4.

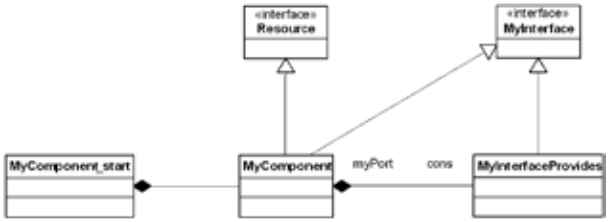


Figure 4: Design Two

A third implementation modifies this last scenario except that *myPort* does not forward calls to *MyComponent*, but instead handles the data itself. This is the same design as in Figure 4, with more logic in *MyInterfaceProvides*.

All three implementations are SCA compliant, but they differ in their properties. The first implementation is efficient, but has two disadvantages. The first disadvantage is that all functionality is in one class. The second disadvantage is that it is not possible to distinguish between multiple ports with the same interface.

The second implementation is better because it does have a separation of concern in the handling of the port. But, this comes at the expense of one extra function call. This is not a great expense if the data-copy can be avoided (and it can), as the function call is a simple intra-process call.

The third implementation does provide the separation of concern by having a port, but it performs the data processing in the port. This is a violation of the concept of encapsulation.

Since all designs are compliant with the requirements of the SCA, which design to choose depends largely on the preference of the user. A typical project team will decide on a particular implementation pattern and then commit to using that pattern for all component implementations. This makes the component implementations standard across the waveform and easy to understand for the team members (architects, designers, testers) working on them.

The examples above are not exhaustive, but they illustrate that while component implementations will be standard within projects, implementations could vary greatly across different projects.

3.2 Component Development Kit

The implementation of a component is executed within an environment controlled by an SCA Core Framework (CF). This CF is responsible for starting the component, configuring and connecting it. SCA compliant core frameworks are available privately, from commercial third parties and research institutes. These Core Frameworks contain base implementations of the SCA required interfaces (*CF::Resource, CF::Port* and others). They frequently also contain other classes that make it easier for a user to implement components.

An example of this is a class that can manage a property with all the different settings that a property can have according to the SCA. While strictly not required by the CF, a component will need to deal with its properties and hence needs to manage them.

These types of classes are often referred to as a "Component Development Kit" (CDK). A CDK is a utility library provided by the Core Framework vendor in order to make development easier. The CDK is comparable to the standard C++ library that is used by many development projects. The source code for a component uses the Component Development Kit and therefore depends on it. This does somewhat limit the portability of the component from one CF implementation to the other as the CDK might not be available for the other CF, however, the code that needs to be modified is limited, and modifications should be trivial.

Alternatively the project could decide to implement its own CDK instead of using the CDK provided by the CF supplier. This endeavor, if undertaken, would require some effort, but will ensure that the application can easily be ported to other platforms.

The choice of CDK has a direct impact on the source code written for component implementations. Every project has to make this decision based on their needs and preferences. Hence the code written will differ between projects.

4 IMPLEMENTING AN SCA COMPONENT USING AUTOMATED CODE GENERATION

The code that needs to be written for a component can be divided into three pieces: functional code, SCA wrapper code and glue code.

The functional code performs the main job of the component. This is typically some form of signal processing or high level control code.

The SCA wrapper code is the code that handles the interfaces, ports and properties that the component is required to implement. Elements such as the entrypoint interfaces and ports discussed in Section 2 make up the wrapper code. The SCA wrapper code is also responsible for handling the CORBA aspects such as starting of the ORB and registration in the naming service.

Finally, glue code ties functional and wrapper code together. The glue code is responsible for forwarding the data from the SCA wrapper to the functional processing, and performing any data conversion necessary. Similarly, the glue code takes information from the functional processing part and passes it on to the wrapper code after performing data conversion if necessary. As an example, consider a component that has a user-defined configure property called *frequency.* The property controls how the component performs some of its functional behavior. The property can be given a new value through a Human Computer Interface. The CF will write this value to the component and the SCA wrapper code will receive this value and process it. However, the functional code needs to be informed of this new value. Informing the functional code of the change is part of the glue code.

Figure 5 graphically depicts the the pieces that make up the component. The functional code makes up the center. It is wrapped by the dark line representing the glue code. The SCA wrapper code completes the component appearing on the outside of the glue code.

Both the wrapper code and glue code are SCA-specific artifacts. The functional code is not. This paper is concerned with code generation of SCA artifacts, and therefore, the rest of this section will focus on wrapper code and glue code.

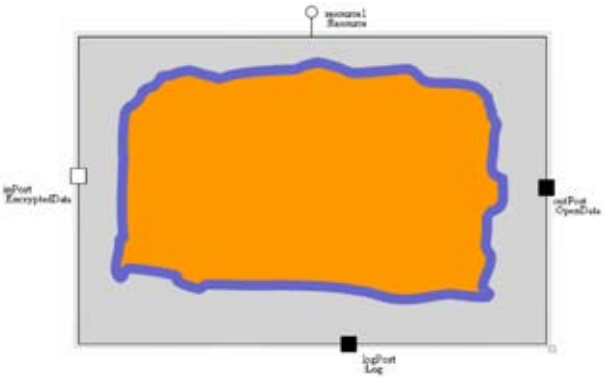


Figure 5: Make up of a component

4.1 SCA Code Generation

Coding SCA artifacts is not particulary difficult, but it requires detailed knowledge of the SCA standard, and sizable attention to details related to the external interface of the component. For example, the names in the source code need to match the names in the component descriptors (XML files) exactly to prevent run-time problems from occuring. The human brain is a limited device, and is not suited to keep track of this detail. Automation however is perfectly suited to keep everything synchronized.

Code generation technology has been used in software development for almost two decades now. Different projects have had different experiences with code generation. Some project teams trust in code generation as it provides great benefits. Other project teams do not want to give control of their source code to an automated piece of logic. Reasons frequently given for not using code generation are the fear that code generation will increase code size, negatively impact efficiency and, diminish the flexibility of the code.

With the right provisions though, code generation stands to benefit SCA projects substantially. Code generation of SCA artifacts diminishes the reliance on hard to find SCA expertise, it abstracts the intricacy of this type of code from developers, and it can maintain synchronization between interfaces of components in a component application — a task very difficult to accomplish manually. Lastly, code generation reduces the amount of code developers have to write and hence improves efficiency.

What provisions does an SCA code generation solution need to posess in order for project teams to truly reap the benefits of automation, without compromising on their implementation intentions? This next section answers this question.

4.2 Customizable SCA Code Generation

In order for an automated tool to successfully generate source code for the implementation of a component, it must take into account all of the following:

- The code patterns adopted by the project
- The Core Framework selected
- The CDK selected (if applicable)
- The CORBA ORB (Object Request Broker)
- The operating system selected

All of the items in the list above will influence the code. For example, the Windows operating system has a different way of starting a component than VxWorks. Also, the ACE ORB is different in a number of ways from ORBexpress. The source code needs to account for these differences.

Generated code must depend on the choices the project team has made and hence code generation needs to be customizable.

Another important benefit of customizable code generation is that it allows a project team to generate code based on their own proven and efficient patterns.

Tested code can be used to form a template which can be applied to the generation of all other component code. This way, components will adhere to the proven coding paterns of the project, thus significanlty reducing risk.

Customizable code-generation provides the project team with all the benefits of code generation, while allowing them to retain full control of the code that is generated.

5 ZELIGSOFT COMPONENT ENABLER

Component Enabler™ (CE) provides a visual modeling environment for the definition of components, component implementations and applications built out of components. CE allows the designer to validate their model for SCA compliance. CE then generates correct-by-construction SCA compliant artifacts based on the visual definition.

Component Enabler forms the backbone of the development process as it allows a project team to capture all SCA related information in an easy to understand UML 2.0 based model. The code generation feature then takes this information and translates it into code, completing the cycle from architectural design to implementation.

Zeligsoft CE code generation is controlled by user-customizable templates. The user can change the code generation template and adapt it to their project's needs. The code generated is always synchronized with the visual model, thereby eliminating "finger-trouble". CE code generation can be used to generate wrapper code in Java, C, C++ or ADA. The Zeligsoft Professional Services team can adapt Component Enabler's code generation feature to any programming language, any CDK, any Core Framework, any CORBA ORB and any OS.

Most software development projects use an iterative approach to build software. This means that the visual model is elaborated in several different steps. When the model changes the user can easily generate new wrapper code. The wrapper code is separated from the function code through the glue code. The interfaces are standardized, which allows for convenient iterative workflows.

An SCA compliant component also requires functional code and glue code. Functional code can be written in a tool of the user's choice such as the visual modeling tools distributed by IBM Rational [3] and I-Logix [4].

Other options include pure command-line based development or integrated development environments like GreenHills Multi [5], WindRiver Tornado [6] or Eclipse [7]. CE provides scripting interfaces so that functional code builds can be part of the automated process.

The glue code serves to connect the SCA wrapper code and the functional code and is typically written by hand. This code is added to the wrapper code and the functional code and makes information and control flow from one to the other.

6 SUMMARY

SCA components have a standard external interface. The component developer needs to implement that interface. Different patterns can be used for this.

The code needed for an implementation depends on the code patterns mandated by the project. The code also depends on the Core Framework, Component Development Kit, CORBA ORB and the operating system of choice.

Automated code generation increases the efficiency of the developers and reduces their dependency on SCA expertise. In order to realize these benefits though, code generation must be customizable. Customizable code generation allows the project team to benefit from automation while retaining full control over their code.

Zeligsoft Component Enabler can provide automated code generation that is customizable to the specific needs of a project.

7 REFERENCES

1. Zeligsoft Component Enabler
   http://www.zeligsoft.com/

2. WinMerge
   http://winmerge.sourceforge.net/

3. IBM Rational
   http://www-306.ibm.com/software/rational/

4. I-Logix
   http://www.ilogix.com/

5. Green Hills
   http://www.ghs.com/

6. WindRiver
   http://www.windriver.com/

7. Eclipse
   http://www.eclipse.org/

8. Component Enabler Best Practices: SCA
   http://www.zeligsoft.com

9. Developing SCA Compliant Systems
   http://www.zeligsoft.com/

Contact Information

Website: www.zeligsoft.com

Email: info@zeligsoft.com

Toll-free (North America): 1-800-ZELIGSW (1-800-935-4479)

Direct dial: +1 819-684-9639

Version 1.1
Printed in Canada