



# Service Component Architecture (SCA)

---

Doug Tidwell  
IBM Corporation

[dtidwell@us.ibm.com](mailto:dtidwell@us.ibm.com)





# Agenda

---

- A brief history of SOAs
- Why SCA makes life simpler
- SCA application diagrams
- Code and other details
- Resources – Next steps



# Today's quote

---

- *Anyone who's interested in the future of application development should also be interested in SCA.*
  - David Chappell, *Introducing SCA*  
[davidchappell.com/articles/  
Introducing\\_SCA.pdf](http://davidchappell.com/articles/Introducing_SCA.pdf)



# The big picture

---

- There are four technologies that are the future of SOA:
  - All of the services involved are accessed with SCA.
  - The data sources are accessed with SDO.
  - *The interface is based on XForms.*
  - The process definition is based on BPEL.
- **We'll cover SCA today.**



# A brief history of SOAs

---





# Using a component

---

- When dealing with a component (in an SOA or not), there are three important pieces of information:
  - The **interface** of the component
  - The **implementation** of the component
  - The **access method** to invoke the component
- We'll consider how we use this information to invoke components.



# The bad old days

---

- Originally, most components were hardwired into an application:
  - The application knew the details of the component's interface at build time.
  - The application accessed the component's implementation at build time.
  - The application knew the details of the component's access method at build time.



# The bad old days

---

- This worked (and still does), but the application is relatively brittle.
  - If the implementation or access method changes, we have to modify our code, rebuild it, retest it and redeploy it.



# Web services: The early days

---

- SOAP introduced a way to invoke a remote service with an XML envelope.
- The SOAP infrastructure built the envelope and sent it to a particular URL; the SOAP service's host invoked a service and sent XML back to us.
  - The application knew the details of the component's interface at build time.
  - *The application did not access the component's implementation at build time; the component is invoked at run time by the SOAP infrastructure.*
  - The application knew the details of the component's access method at build time (usually SOAP/HTTP).



# Web services: The early days

---

- Things have gotten more complicated since then:
  - Protocols other than HTTP
  - Document-style SOAP services instead of RPC
  - Asynchronous invocation with JMS
  - Encryption, conversations, reliable messaging, WS-\*
  - *etc.*



# More flexibility with SCA

---

- An SCA application is even more dynamic:
  - The application knows the details of the component's interface at build time.
    - *The application does not access the component's implementation at build time; the component is invoked by the SCA invocation framework.*
    - *The application does not know the details of the component's access method at build time; this is also handled by the SCA invocation framework.*
- SCA moves the implementation and access method details out of your application and into the middleware.



# More flexibility with SCA

---

- We mentioned the three important pieces of information:
  - The **interface** of the component
  - The **implementation** of the component
  - The **access method** to invoke the component
- With SCA, the implementation and the access method are determined at runtime by the infrastructure.
  - Our code isn't involved in this determination, so we don't have to write it or maintain it.

# Why SCA makes life simpler

---





# Why SCA makes life simpler

---

- One way to look at SCA is that it takes all of the details of implementations, access methods, encryption, authentication, *etc.* and moves them into the middleware layer.
  - Application developers write business logic, code that actually builds value for your organization.
  - The details of using services are handled by SCA.
  - As the details change, your applications (and the developers who wrote them) aren't affected.



# SCA simplifies your apps

---

- With SCA, you can change the definition of a component without changing the applications that use it.
- The original code:
  - `myService.doSomething(x,y);`
- After changing the component so that every request requires a digital signature:

- `myService.doSomething(x,y);`



# Everything's a POJO

---

- SCA gives your developers a single programming model for using services.
- As your SOA gets more complicated, your developers have to learn more and more interfaces.
  - EJBs, RMI, JCA, JAX-WS, JAX-RPC (and that's just Java!)
- With SCA, you're using a POJO:
  - **`myService.doSomething(x,y);`**
  - We don't know where **`myService`** is, how it's accessed, how it's written, what policies apply to it, and so forth. To us, it's just a POJO.



# Everything's a POJO

---

- Invoking an actual POJO:
  - `myService.doSomething(x,y);`
- Invoking a Web service:
  - `myService.doSomething(x,y);`
- Invoking an RMI service:
  - `myService.doSomething(x,y);`
- Invoking a Web service with digital signatures:
  - `myService.doSomething(x,y);`
- And so forth....



# Freedom of choice

---

- Because every SCA component works like a POJO, you have more freedom when you choose a service provider.
- For example, say you discover a great service based on RMI.
  - With SCA, your developers don't have to know anything about RMI to use the service.
  - Without SCA, if your developers don't know RMI, you have to balance the training costs and risk factors against the benefits of the new service.



# SCA simplifies admin

---

- SCA gives you a single declarative way to establish policies.
  - "Requests to this service must be authenticated."
  - "Requests to this service must be digitally signed."
- For example, here's how you require digital signatures:
  - **`sca:requires="integrity"`**
- The SCA runtime implements the policy, the application does not.



# SCA simplifies governance

---

- With SCA, you define a component one time, in one place, then point your applications to that definition.
  - If all of the applications use the same definition, you know what components your organization uses.
  - If you need to change the component or how it works, you make that change one time, in one place.
- SCA makes it easier to track what components are being used, and SCA lets you change those components without changing your applications.



# What SCA *is*

---

- An **executable** model for assembling services
- A simplified **component programming model** for implementing services
  - Write BPEL processes, Java POJOs, EJBs, COBOL apps, PHP scripts, C++ apps, ...



# SCA is *not*...

---

- ...tied to a specific programming language, protocol, technology, runtime, *etc.*
- ...a workflow model
  - Use BPEL for that
- ...Web services
  - SCA can access local objects, avoiding the overhead of Web services.





# SCA and ESBs

---

- Because SCA and an ESB both do mediations, there is some confusion about the differences between the two.
- SCA and ESBs work at different levels. SCA is concerned with building SOA applications, while an ESB is concerned with deploying the services underneath those applications.



# SCA and ESBs

---

- **SCA** is an executable model for describing composite services applications and a model for building individual service components.
- An **ESB** is an infrastructure upon which service-oriented applications can be deployed.



# SCA and ESBs

---

- **You can view SCA as a programming model for ESBs.**
  - All of the Components that do the mediations are part of the model, making them easy to manage.
  - The ESB provides the infrastructure to deploy, run and manage those Components.



osoia.org



- SCA and SDO were developed by the Open Service Oriented Architecture group ([osoia.org](http://osoia.org)):



The logo for OASIS Open CSA features the word "OASIS" in a bold, dark blue sans-serif font, followed by a stylized icon of three overlapping shapes (yellow, red, blue) and the words "Open CSA" in a lighter blue sans-serif font. A thick black horizontal line runs across the slide below the logo.

## OASIS Open CSA

- The specifications work of OSOA has been turned over to OASIS.
- The Open Composite Services Architecture group is being formed now.
  - See [oasis-opencsa.org](http://oasis-opencsa.org) for more details.



# The SCA specs

---

- There are four parts to the specs:
  - The **Client and Implementation** specifications: How to use SCA in different languages
  - The **Assembly Model**: How to define composite applications
  - **Binding** specifications: How to use access methods
  - **Policy Framework**: How to add security, transactions, conversations, reliable messaging, *etc. declaratively*



# Client & Implementation

---

- There are different specs for different languages:
  - Java
  - Spring (very short)
  - WS-BPEL
  - C++
  - C
  - COBOL



# Bindings

---

- There are three specific binding specifications:
  - **Web Service Binding Specification**
  - **JMS Binding Specification**
  - **EJB Session Bean Binding**
- Others are on the way.



# SCA Application Diagrams

---









# The SCA development model

---

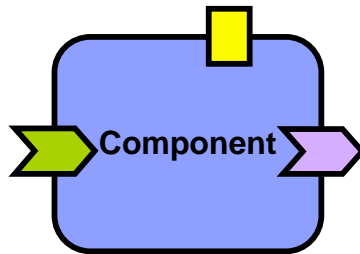
- SCA has a consistent model:
  - A simple piece of code in SCA is called a *component*.
  - Components can be grouped into *composites*.
    - *Components are atoms, composites are molecules. (David Chappell)*
  - Components and composites are hooked together with *wires*.
  - Components and composites can have *properties*.

# Assembly diagram symbols

- Here are the symbols used in SCA assembly diagrams:
  -  A **green chevron** represents a **service**. This is an entry point to the SCA component or composite.
  -  A **purple chevron** represents a **reference**. This points to a service provided by something else.
  -  A **line** represents a **wire**. This is the connection between a service reference and the service itself.
  -  A **yellow rectangle** represents a **property**. This is a value you can set when you invoke the component or composite.

# Assembly diagram symbols

- More symbols:

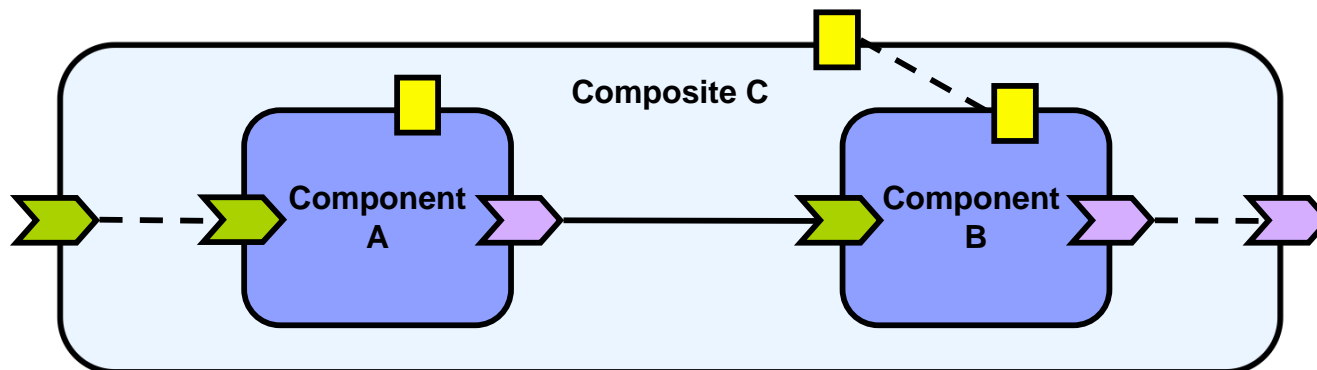


A **rounded rectangle** represents a **component**. A component can have services, references and properties.

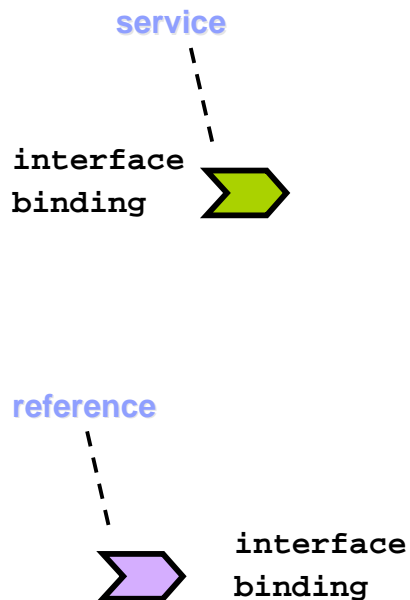
# Assembly diagram symbols

- More symbols:

A **large rounded rectangle** represents a **composite**. A composite contains one or more components. Like a component, it can have services, references and properties. A composite can also contain a composite.

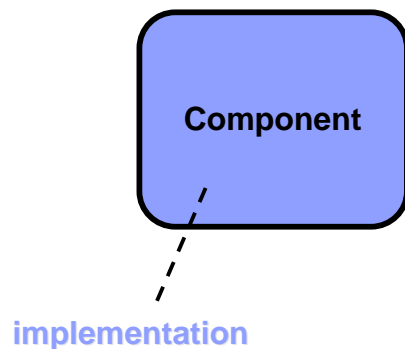
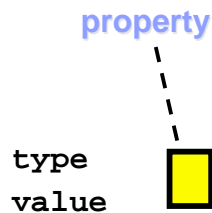


# Services and references



- A service or a reference has an **interface** and a **binding**.
- The interface might be a Java interface, a WSDL port type, a BPEL partner link, a C++ class, *etc.*
- The binding defines the access method. It might be SOAP/HTTP, JMS, JSON, RMI-IIOP, SCA, *etc.*

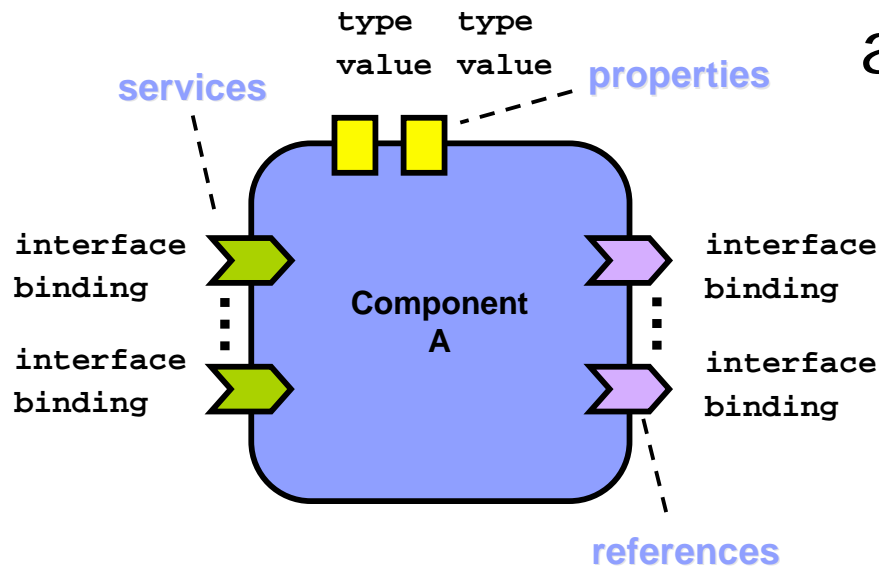
# Properties & implementations



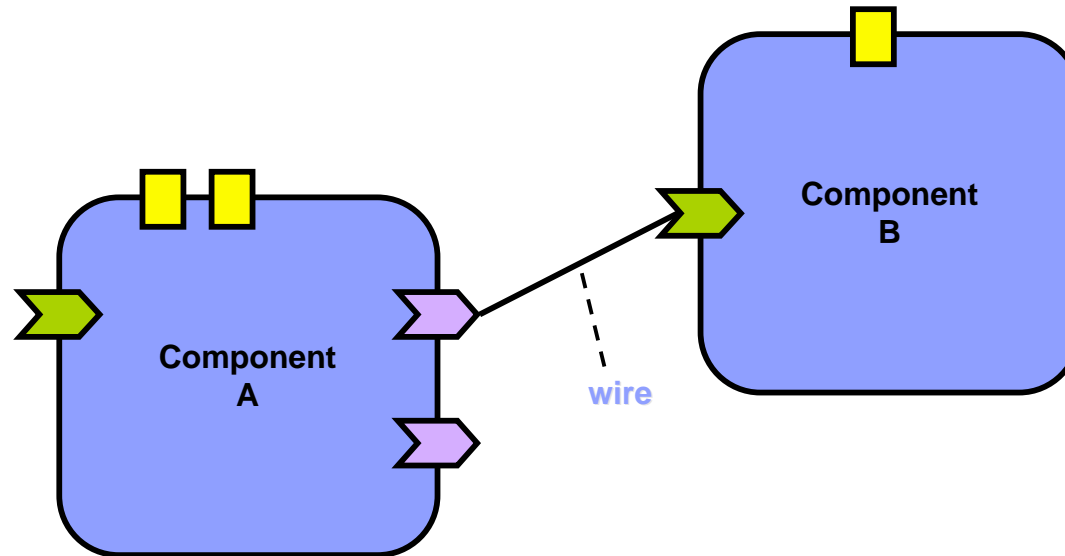
- A **property** has a type and a value.
- A component has an **implementation**; that's the code that actually provides the service.
- The implementation might be PHP, BPEL, Java, C++, Spring, *etc.*
- We won't show the implementation in most of our diagrams; we don't care what it is.

# A component

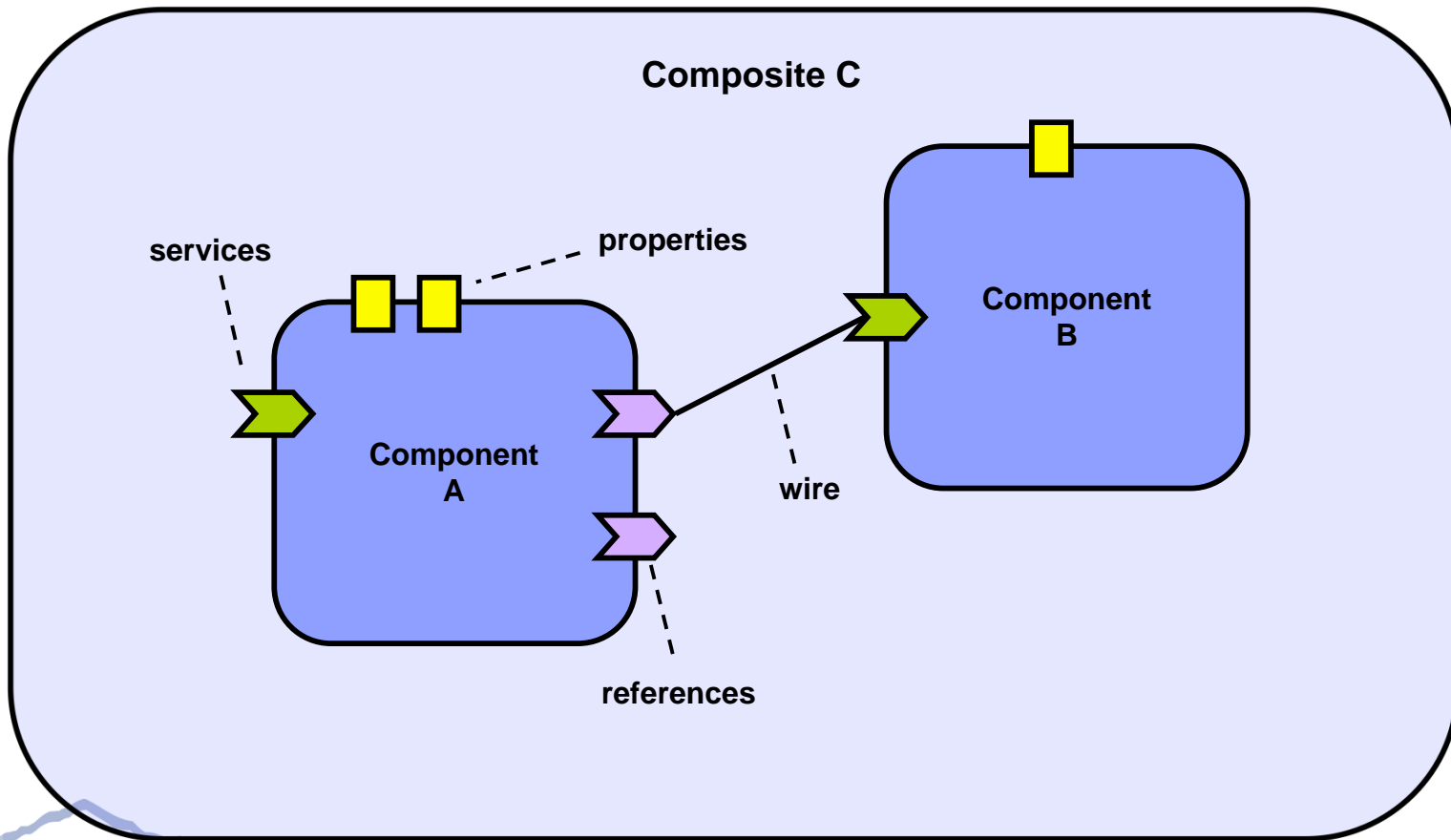
- This diagram is a component with services, references and properties.



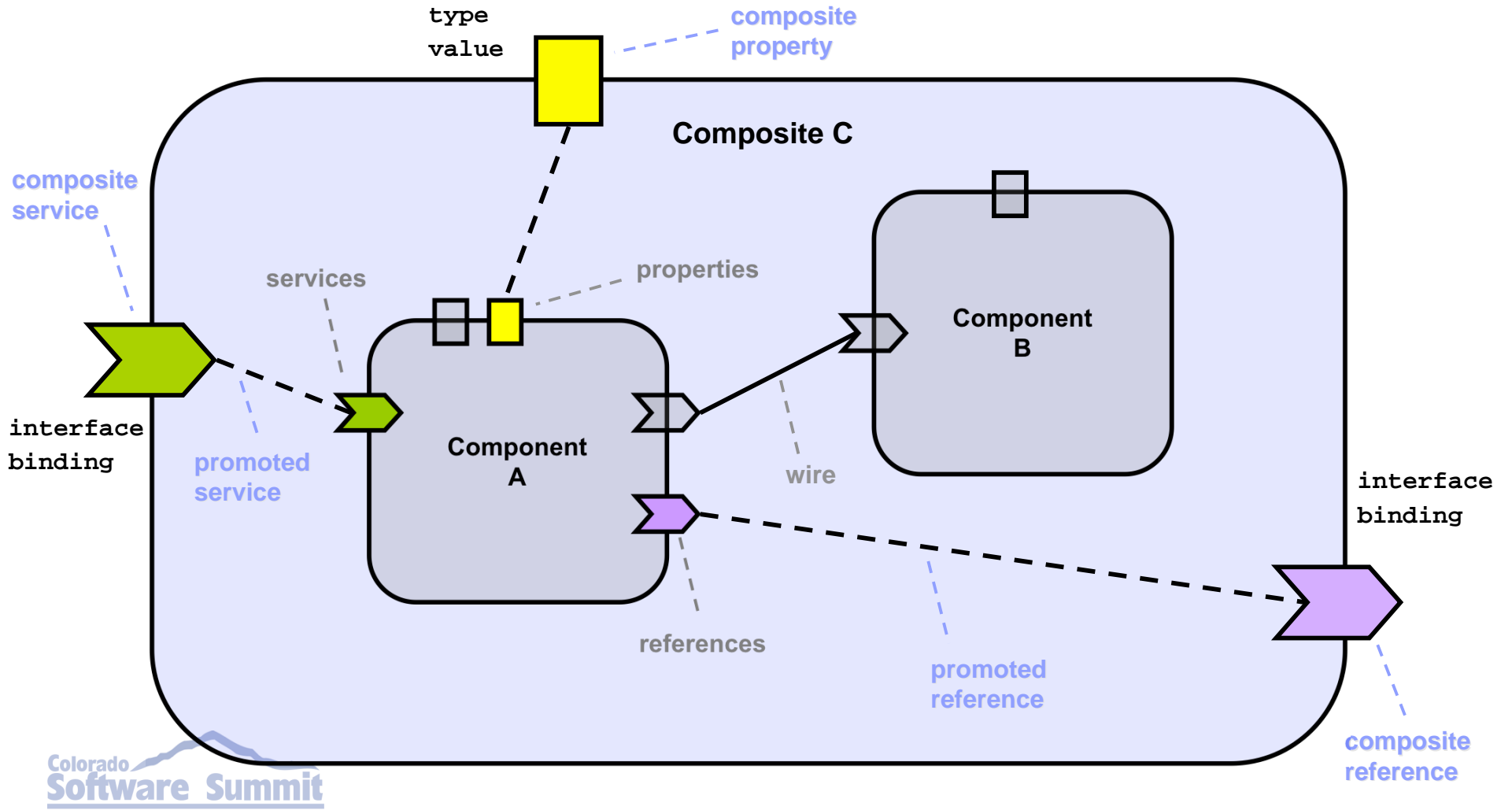
# Wiring



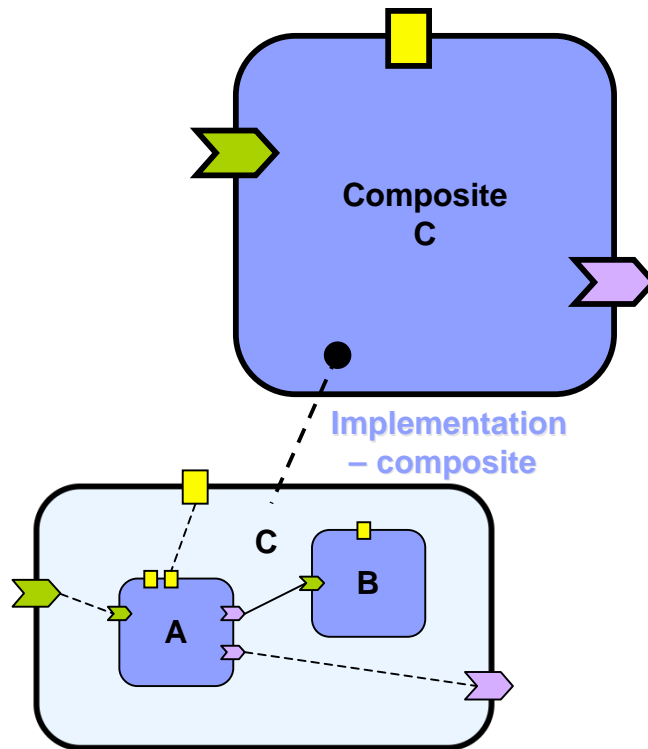
# A composite



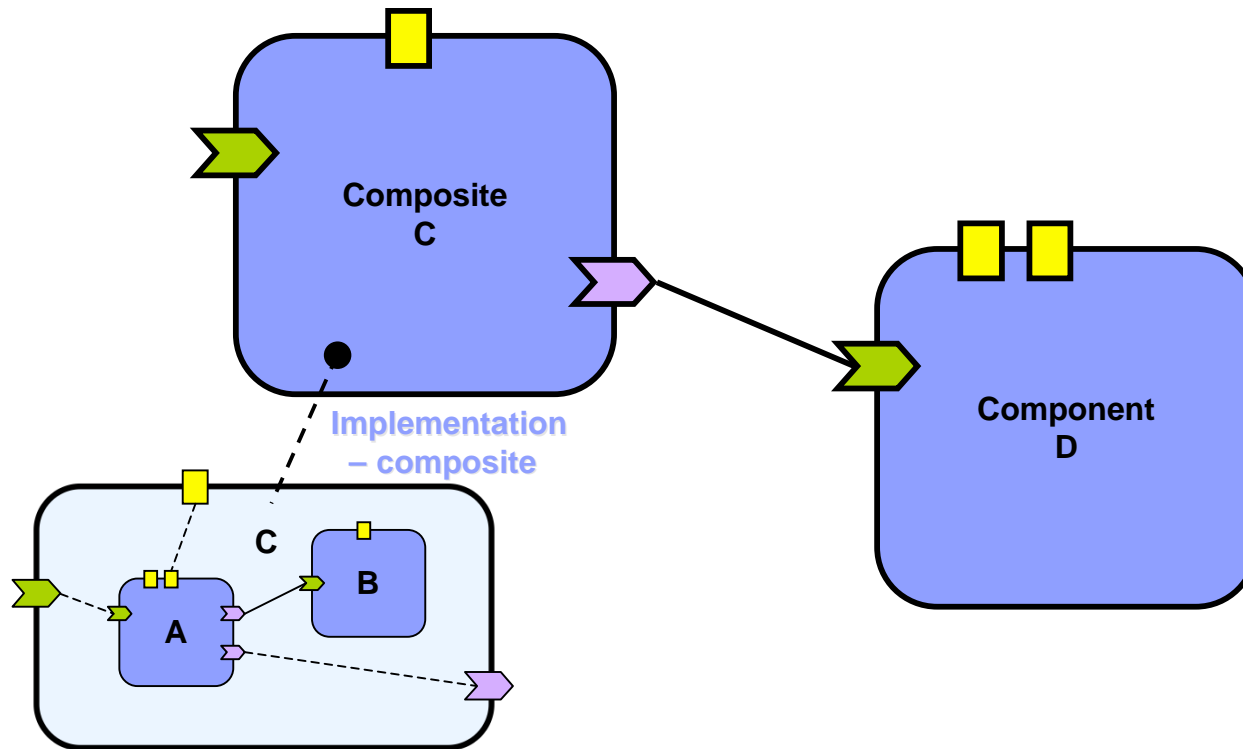
# Promotion



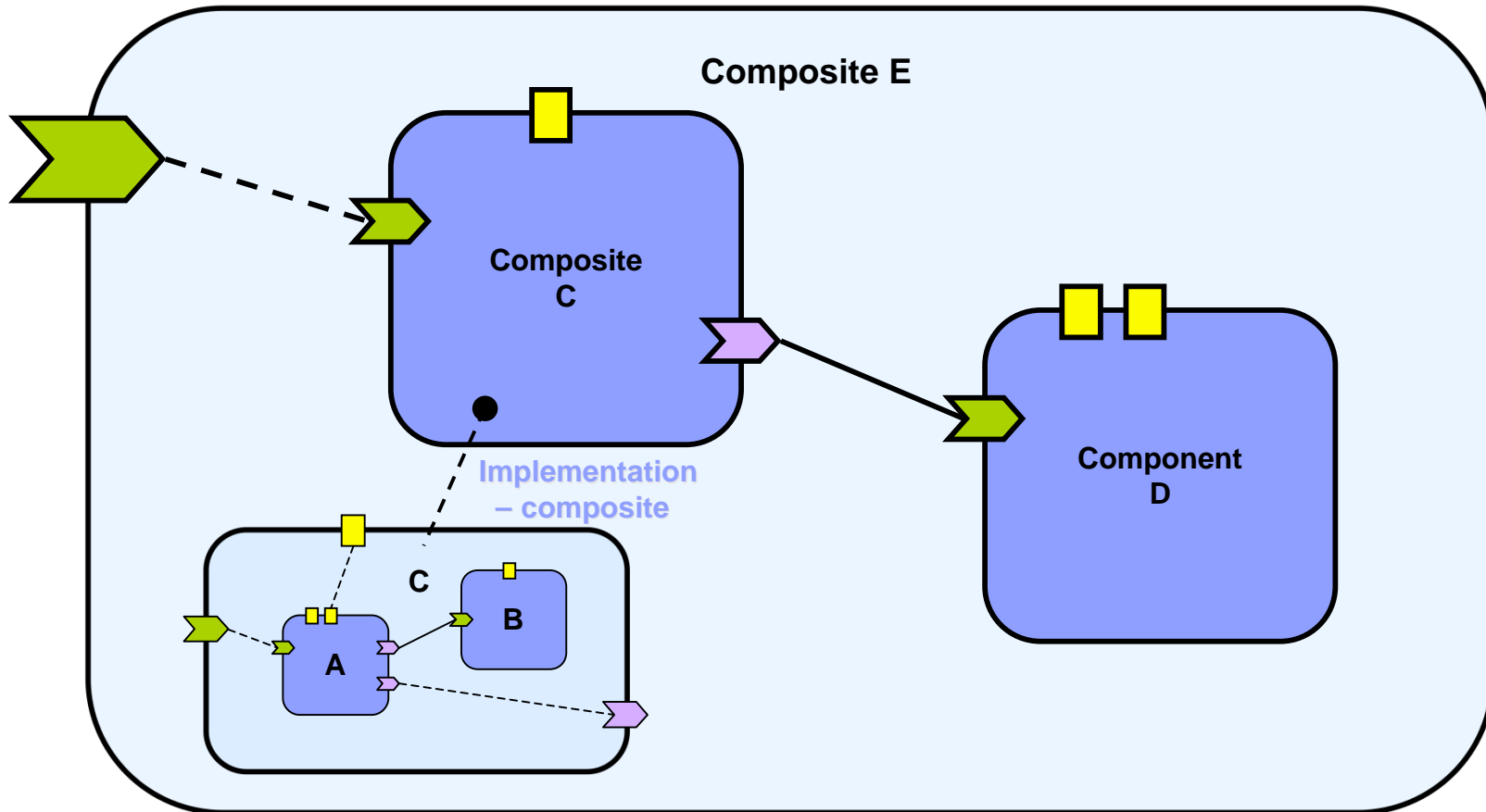
# A composite implementation



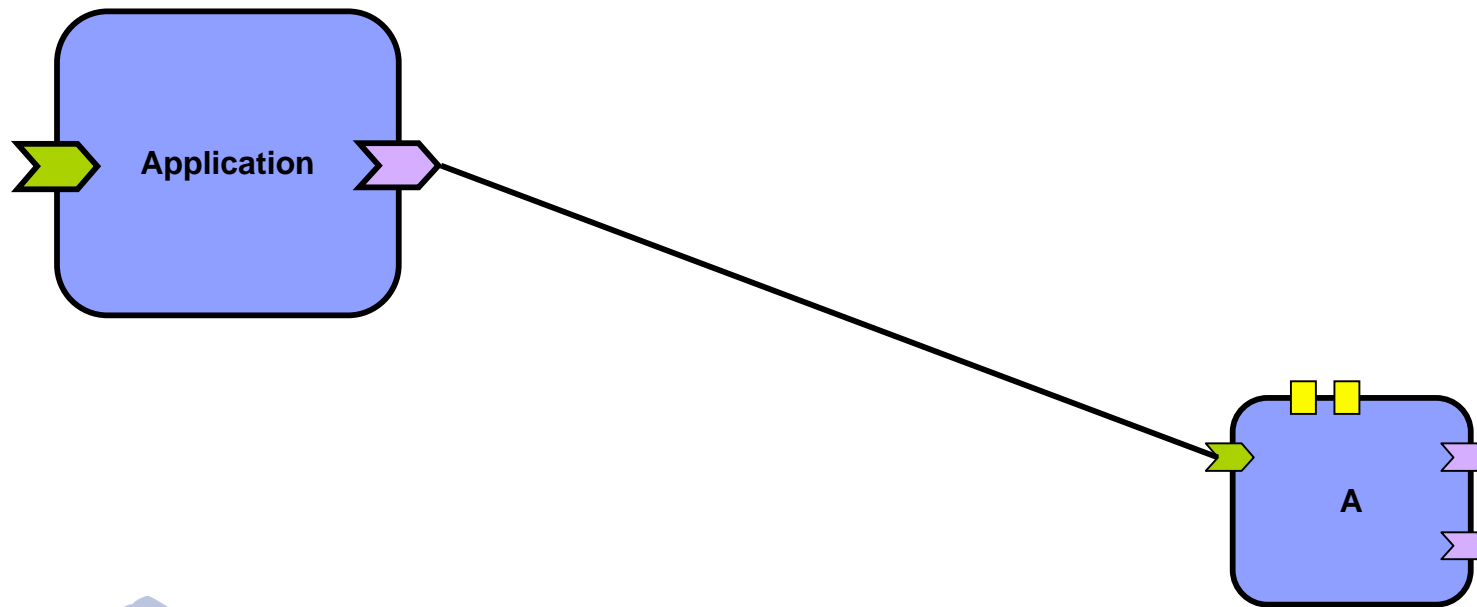
# A composite using a component



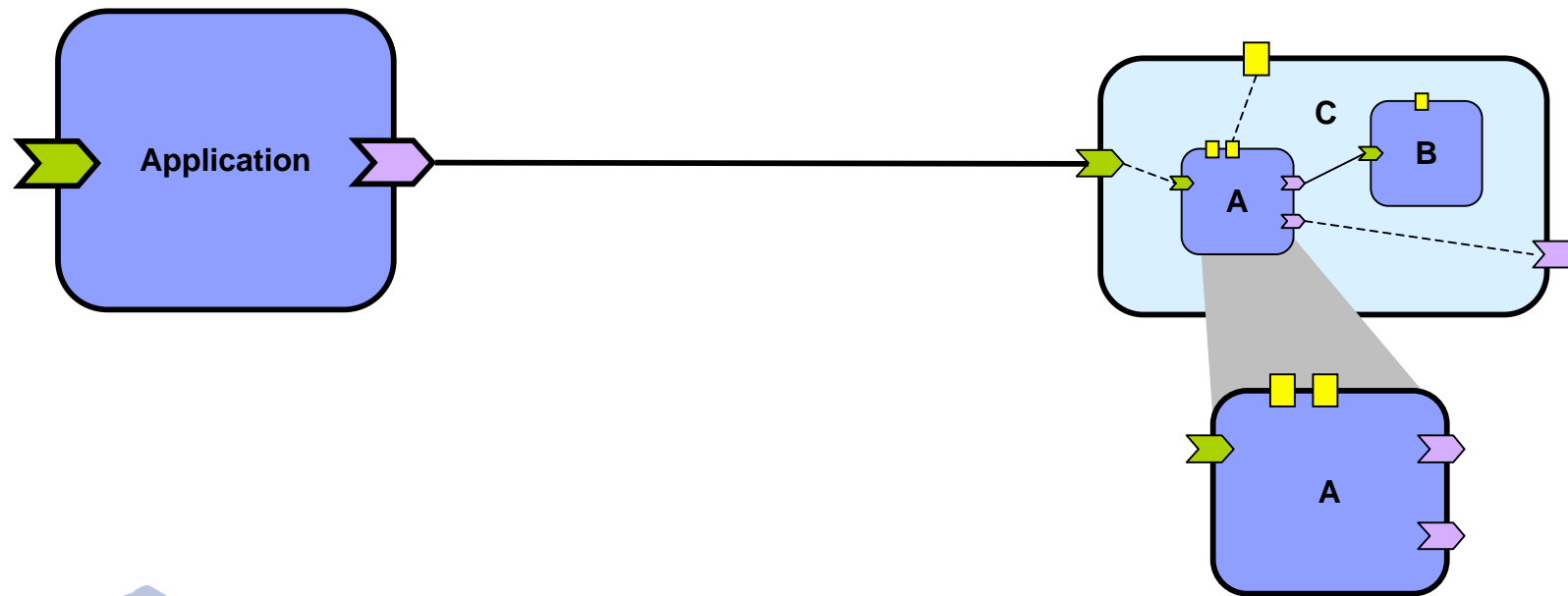
# A composite using another composite



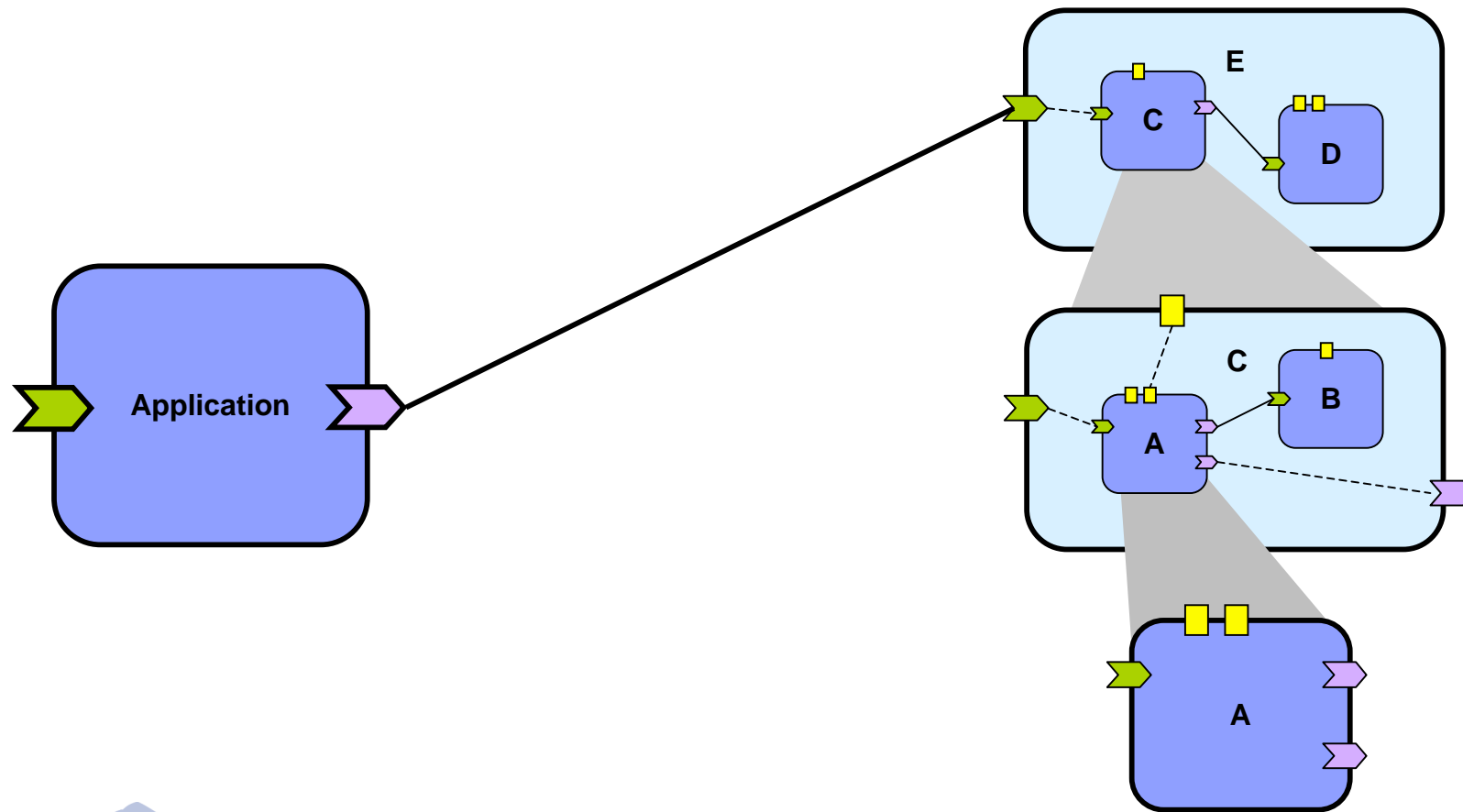
# Recursive assembly model



# Recursive assembly model



# Recursive assembly model





# Code and other details

---



# What's in a WSDL file?

---

- The data structures
  - *Defined with XML Schema*
- The interface
  - *There's a method called **getStockQuote**, it takes a string as input and returns a string as output*
- The binding(s)
  - *SOAP over HTTP*
- The endpoint(s)
  - ***http://xyz.com:8080/myService***
- Ideally the bindings and endpoints are in a separate file.



# What can I do with WSDL?

---

- Send a SOAP envelope to a particular service over a particular protocol.
  - That's it.
- *A Service-Oriented Architecture needs a far more sophisticated way of working with services.*



# More sophisticated things

---

- “Every request sent to this service must be digitally signed.”
  - **The WSDL file won't tell you that.**
- “Every message sent to this service must be encrypted.”
  - **The WSDL file won't tell you that.**
- “Everyone using this service must be authenticated.”
  - **The WSDL file won't tell you that.**
- “This service is asynchronous. Give the service a callback interface and don't wait on a response.”



# The problem

---

- Without this information, **nothing works.**
  - That's a big problem.
- **SCA solves this problem in an elegant way.**
  - The details are handled by the SCA runtime.
  - Those details can be changed without any changes to the client application or the service.

# Apache Tuscany



- The Tuscany project has implementations of the SCA and SDO specs.
  - There are Java and C++ implementations of the core runtime components.
  - For SCA, there is support for components written in BPEL, Java and C++; other languages are being added.
  - Tuscany Java also supports the Bean Scripting Framework. That means you can write components in JavaScript, Ruby, Python, Groovy, Haskell, *etc.*
  - There are also PHP implementations of SCA and SDO.
- Our examples today are based on the Tuscany project.



# The calculator demo

---

- Our demo application is a calculator. The class we'll run is **VisualCalculatorClient**. This class gets a **CalculatorService** from the SCA runtime, then it calls the **add**, **subtract**, **multiply** and **divide** methods.
  - **CalculatorService**, **AddService**, **SubtractService**, **MultiplyService** and **DivideService** are all Java *interfaces*.
- The SCA runtime is reloaded and a different service is used whenever we select a menu item in the **VisualCalculatorClient** class.



# The calculator demo

---

- We'll use `visualCalculatorClient` to invoke:
  - A POJO
  - A set of files written in various scripting languages (JavaScript, Groovy, Ruby and Haskell)
  - An RMI service
  - A Web service
  - A service with an incompatible interface
  - A Web service with authentication tokens in the SOAP header
  - A Web service with a digitally-signed request
- **The `visualCalculatorClient` never changes.**



# An SCA client

---

- Here's an SCA client loads a composite:

```
// Menu item clicked, so load a .composite  
file
```

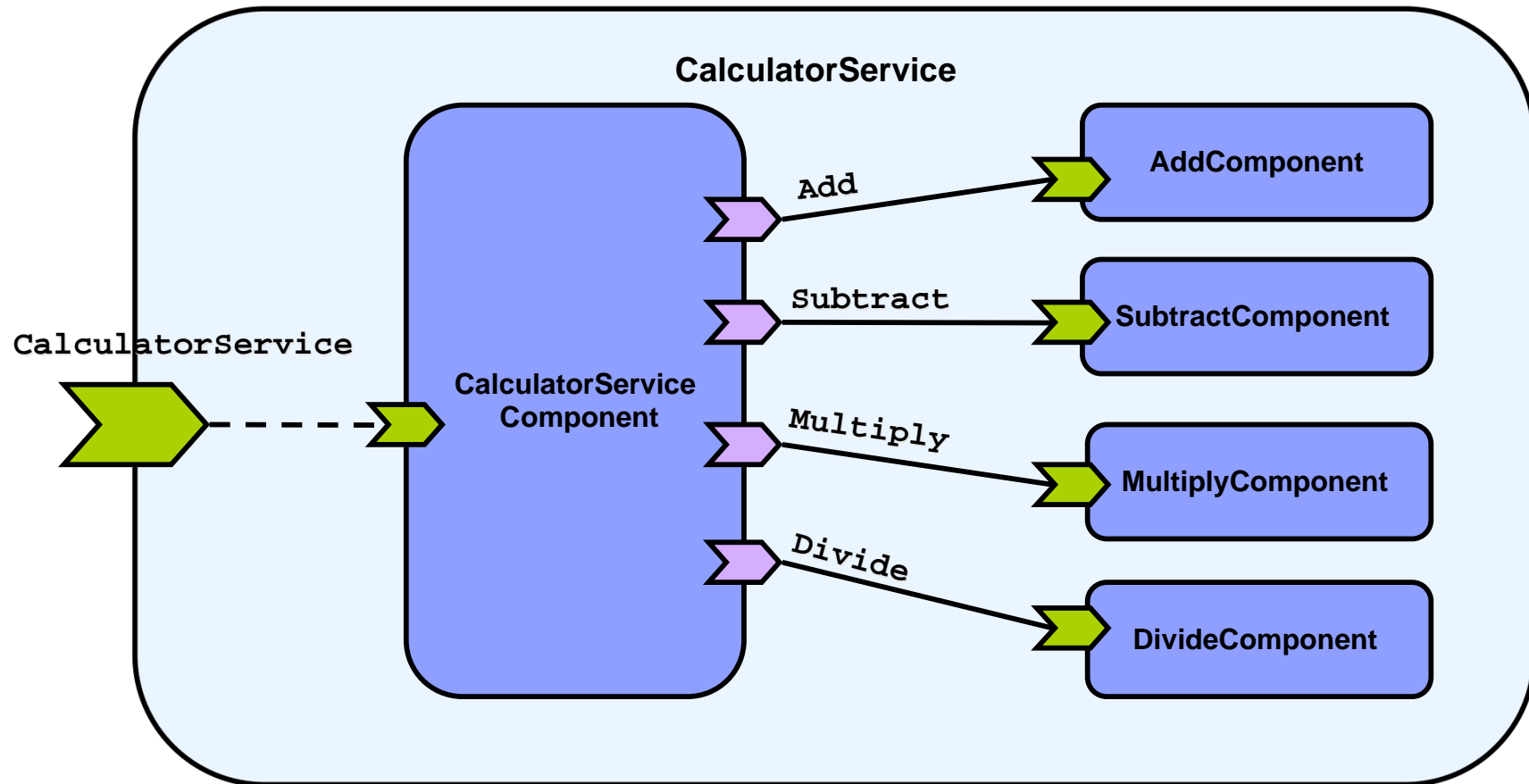
```
SCADomain scaDomain = SCADomain.  
    newInstance( "xxx.composite" );
```

```
CalculatorService calcServ =  
    scaDomain.getService(  
        CalculatorService.class,  
        "CalculatorServiceComponent" );
```

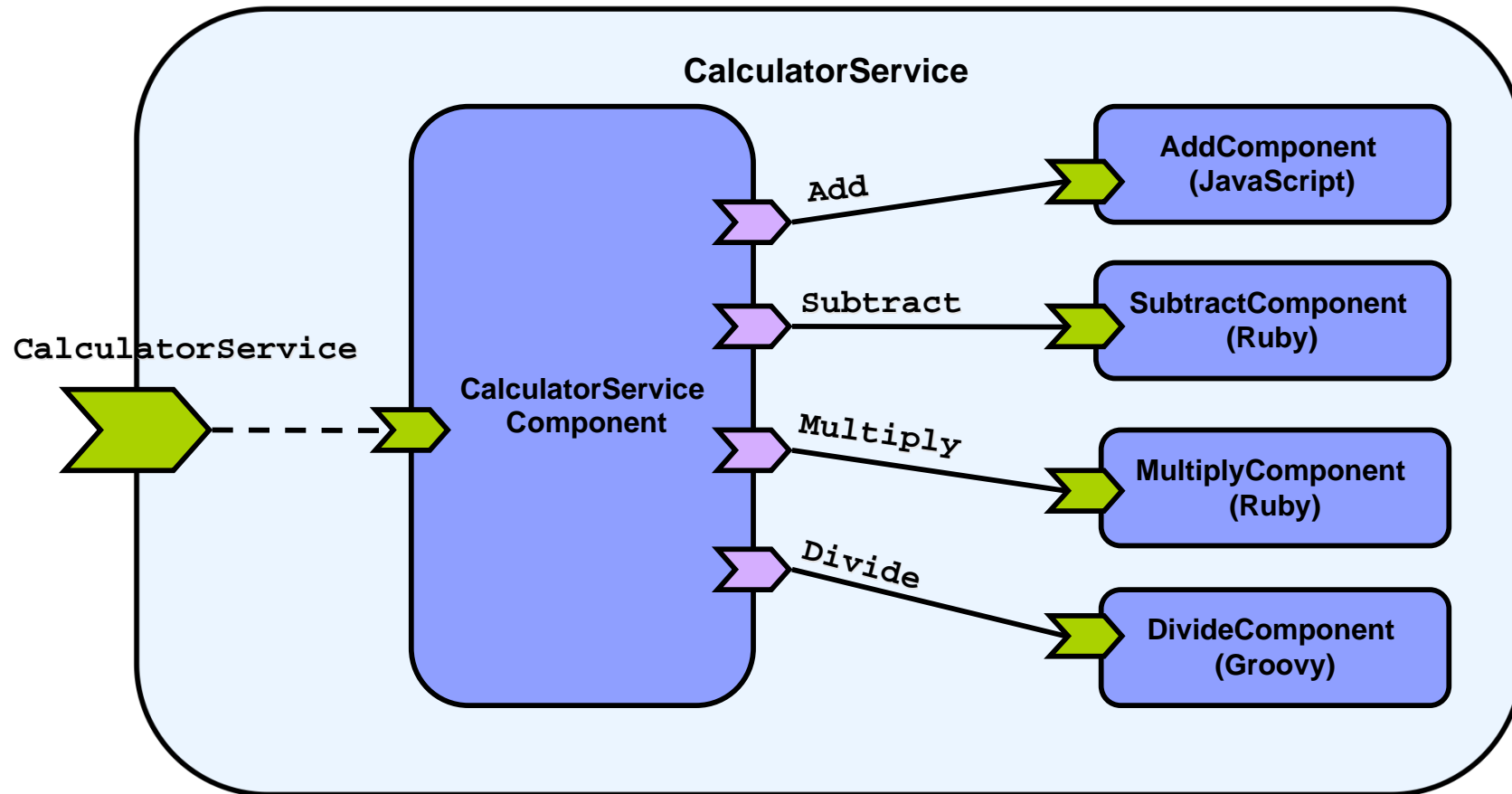
```
. . .
```

```
// Once calcServ is set, we can call it:  
System.out.println( calcServ.add(3,2) );
```

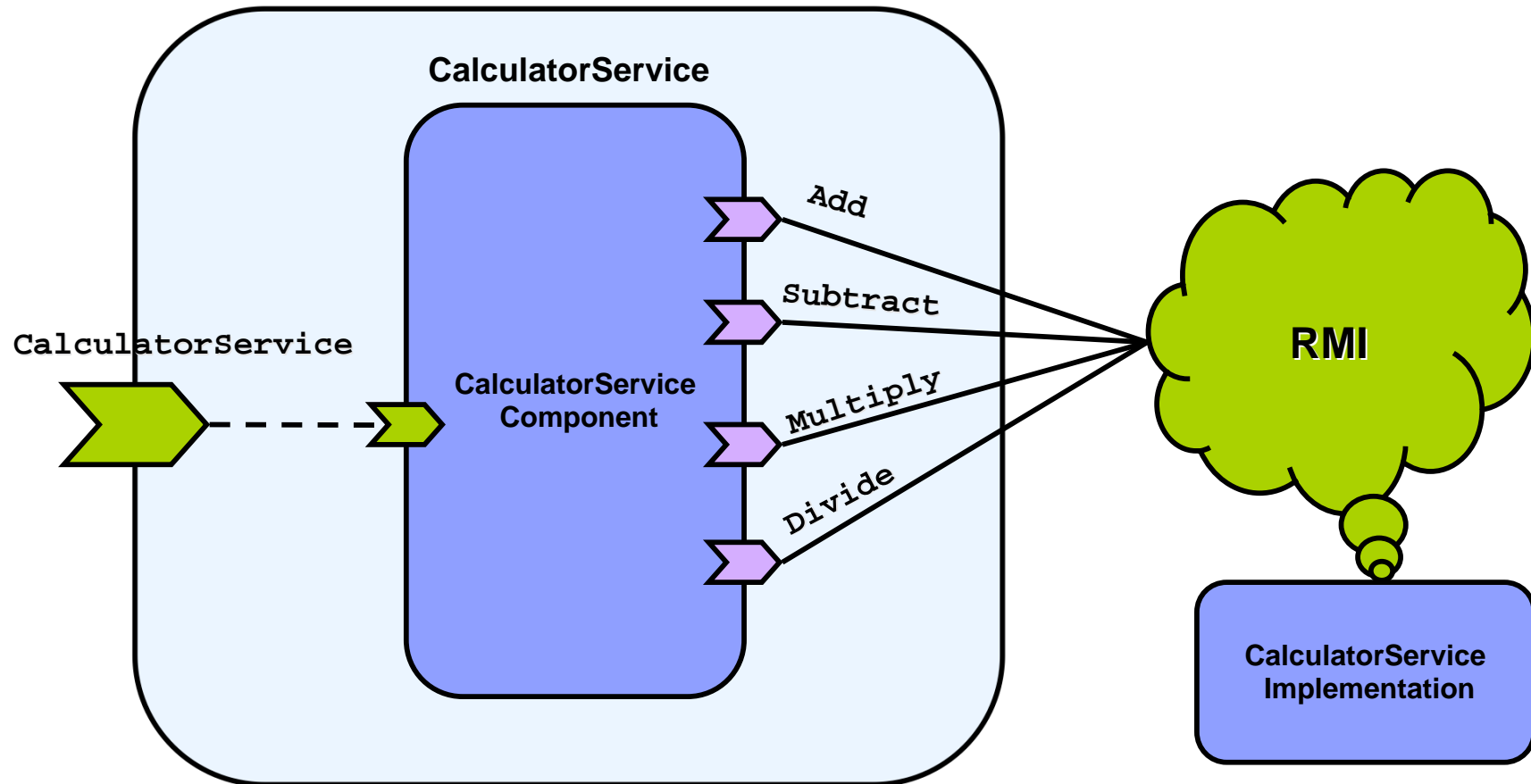
# The POJO component



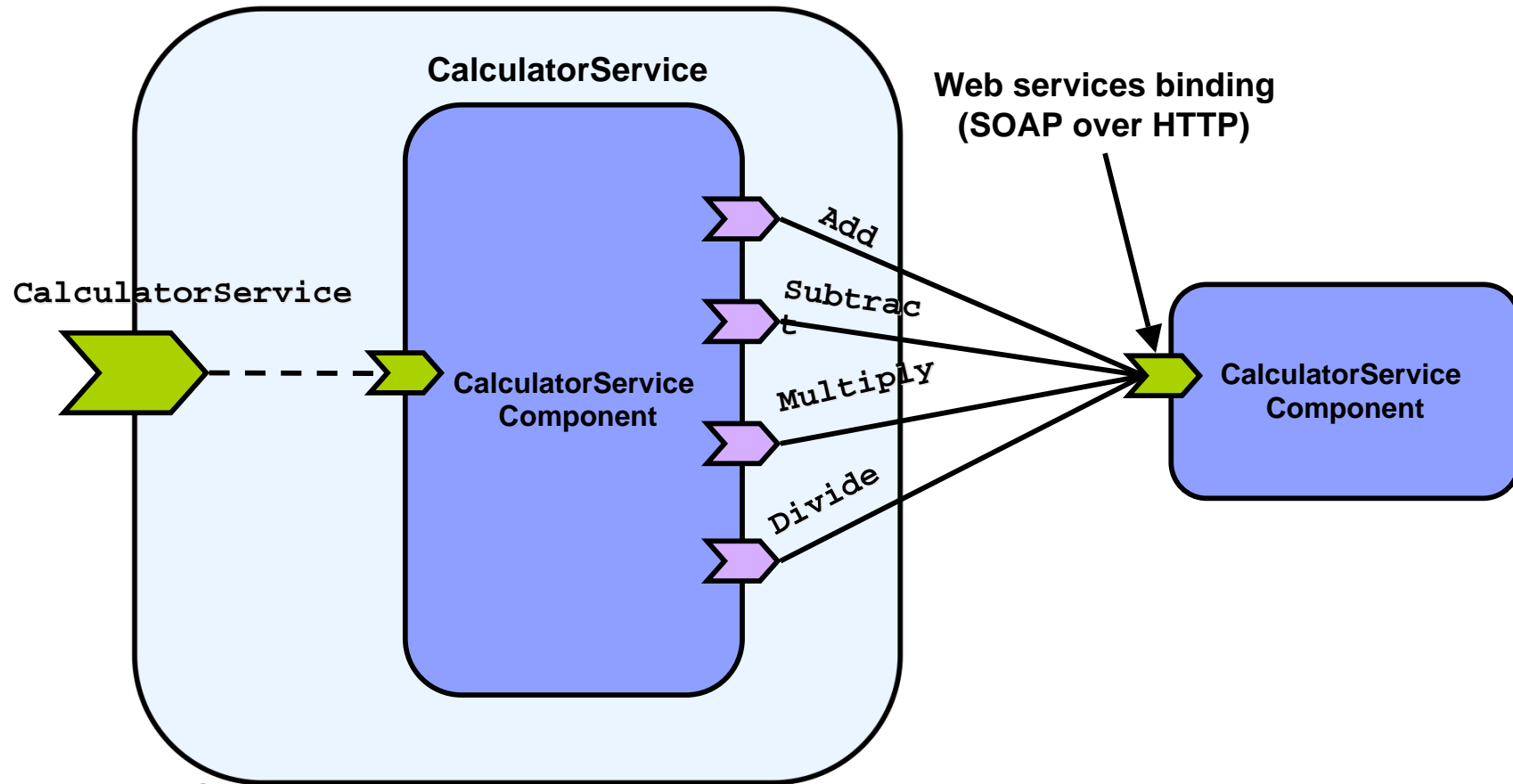
# The script component



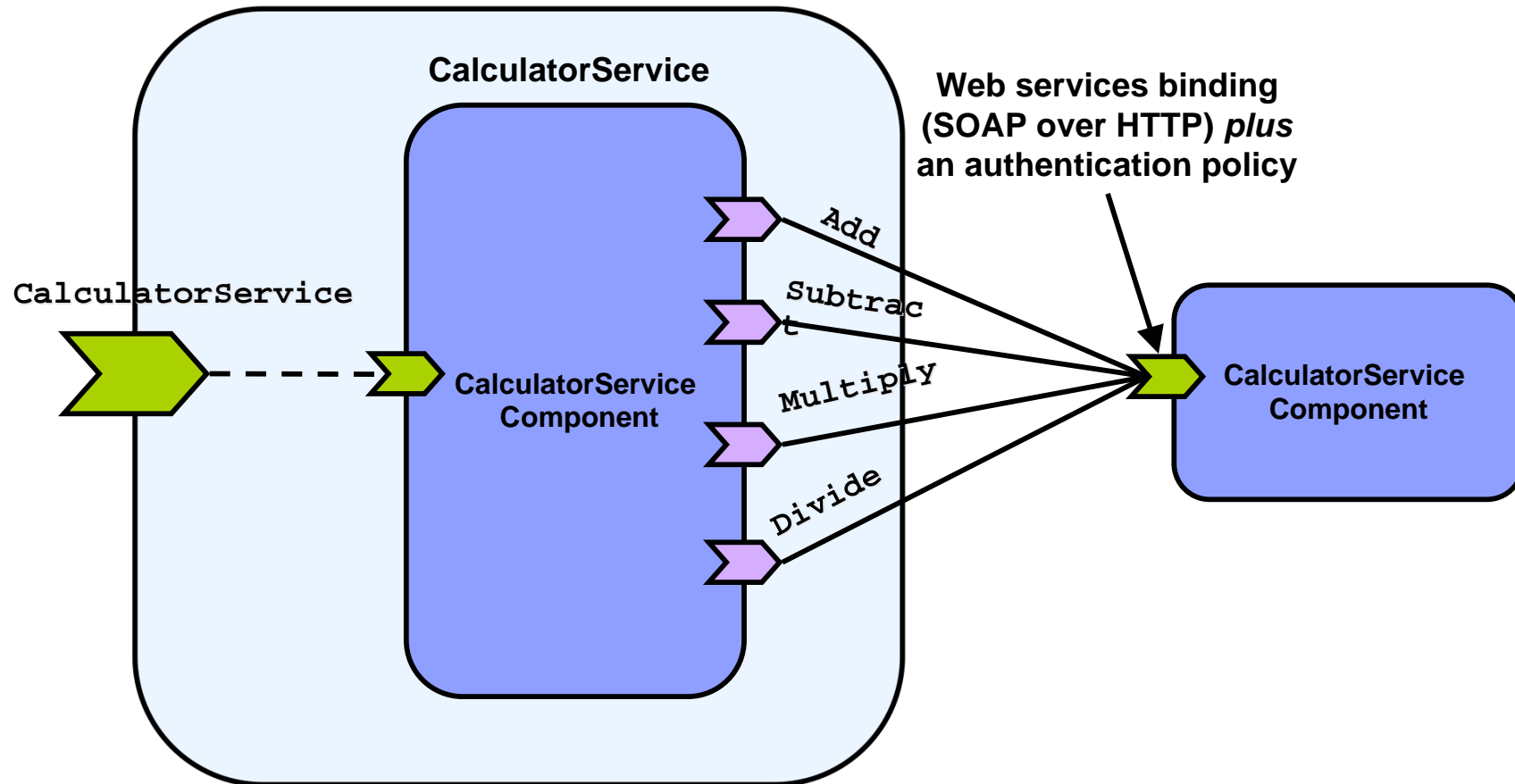
# The RMI component



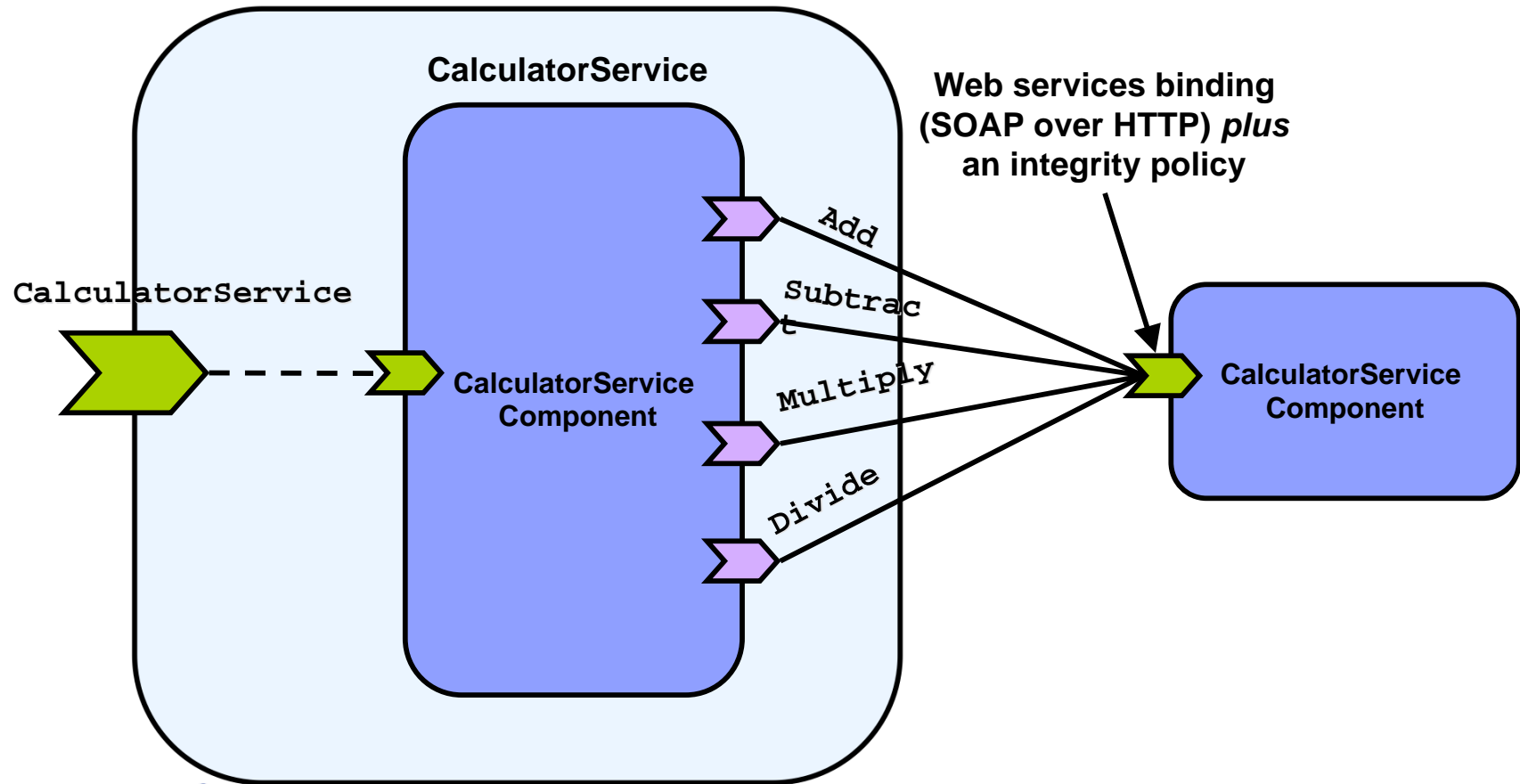
# The Web service component



# The Web service component



# The Web service component



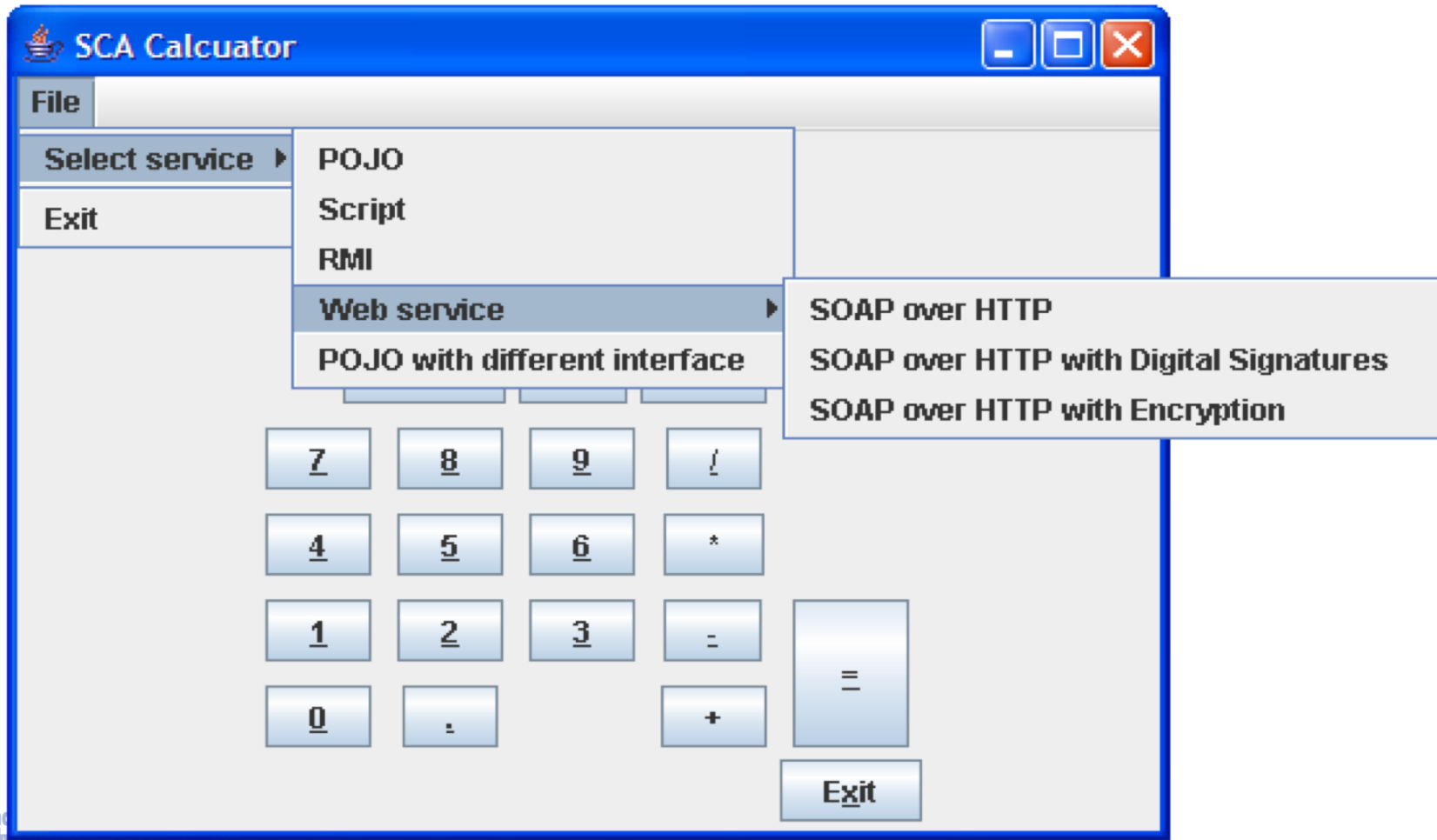


# The calculator demo

---

- **The `VisualCalculatorClient` class and the interfaces it calls never change.**

# The calculator demo





# Finding the service definition

---

- We asked for two things in our code:
  - A new **SCADomain** created from the file **x.composite**
  - A service named **CalculatorServiceComponent**
- To simplify the example, all the **.composite** files define components that have the same name.



# POJO component definition

---

- In the `PojoCalculator.composite` file:

...

```
<component
  name="CalculatorServiceComponent">
  <implementation.java class="..." />
  <reference name="addService"
    target="AddServiceComponent" />
```

...

```
</component>
```

...

```
<component name="AddServiceComponent">
  <implementation.java class="..." />
</component>
```



# RMI component definition

---

- In the `RMICalculator.composite` file:

...

```
<component
  name="CalculatorServiceComponent">
  <implementation.java class="..." />
  <reference name="addService">
    <binding.rmi host="localhost"
      port="8099"

      serviceName="CalculatorRMIService" />
  </reference>
```



# Script component definition

---

- In the `ScriptCalculator.composite` file:

...

```
<component
```

```
  name="CalculatorServiceComponent">
```

```
    <implementation.java class="..." />
```

```
    <reference name="addService"
```

```
      target="AddServiceComponent" />
```

...

```
</component>
```

...

```
<component name="AddServiceComponent">
```

```
  <implementation.script script="..." />
```

```
</component>
```



# Web service component definition

---

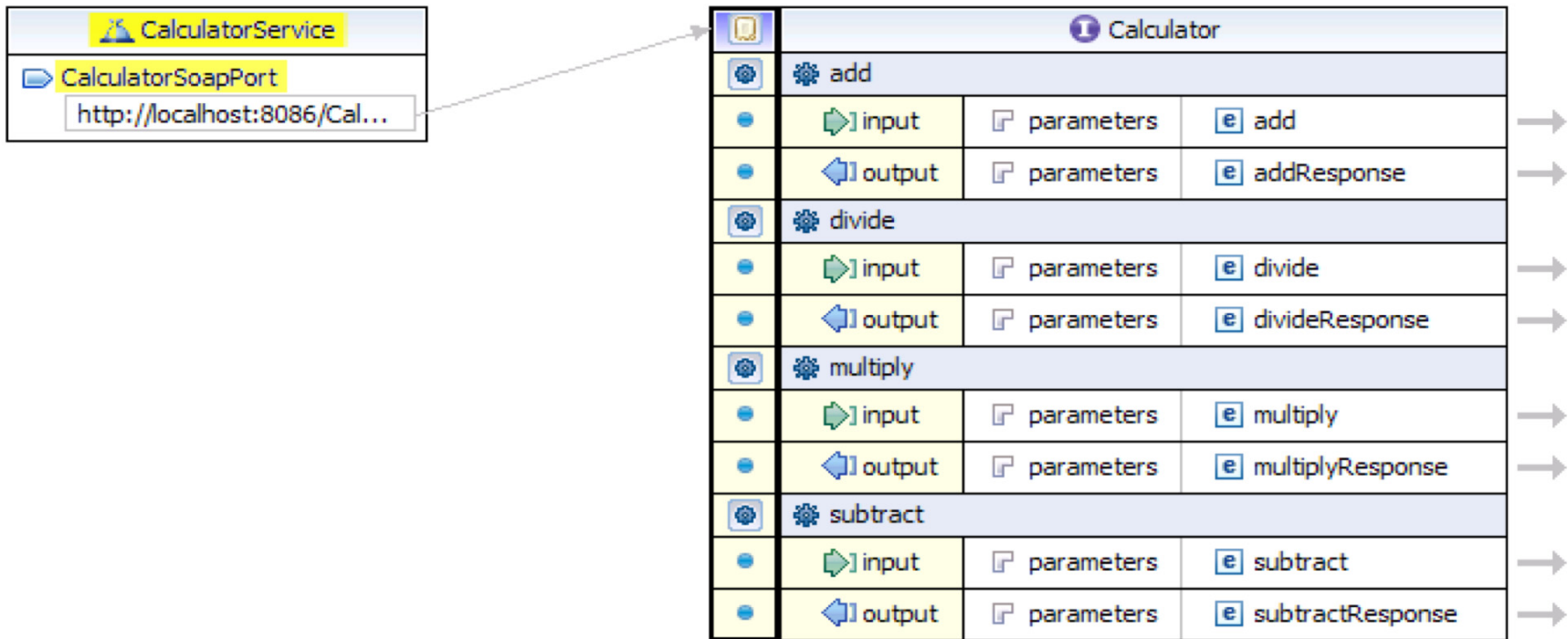
```
<component name="CalculatorServiceComponent">
  <implementation.java
    class="wsCalculatorClient.
      WSCalculatorServiceComponent" />
</component>
<reference name="WsCalculatorService"
  promote="CalculatorServiceComponent/
calculatorService">
  <interface.java
    interface="calculator.
      CalculatorService" />
  <binding.ws
    wsdlElement="http://calculator#
      wsdl.port(CalculatorService/
        CalculatorSoapPort" />
</reference>
```

# Web service component definition

---

- The SCA runtime handles uses the WSDL file and the Java interface generated from it.
- The `<interface.java>` element refers to the same Java interface used for all the other examples.
- The `wsdlElement` attribute of the `<implementation.wsdl>` element consists of:
  - The `targetNamespace` from the WSDL file
  - A hash mark (#)
  - `wsdlPort()`, and in the parentheses:
    - The `name` of the `<wsdl:service>`
    - A slash (/)
    - The `name` of the `<wsdl:port>`

# The WSDL file



# The promoted reference

- The SCA domain uses the promoted reference in the `<component>` definition for dependency injection.
  - In the Java code:
    - `public class WsCalculatorServiceComponent`
    - `...`
    - `// This has public get and set methods`
    - `CalculatorService calculatorService;`
    - `...`
  - In the `.composite` file:
    - `<reference`
    - `promote="CalculatorServiceComponent/calculatorService">`
- Matches exactly; this is the field name, not the class name
- Name of the component as defined in the `.composite` file



# Dependency injection

---

- The SCA runtime uses dependency injection to resolve the reference.
- The class that implements the actual service (effectively a proxy that handles the SOAP calls) is generated.
- The SCA runtime creates an instance of that class and calls **setCalculatorService()** to set the class object.
- Once the object is defined, the service looks like a call to a POJO.



# Properties

---

- Another feature of SCA is the ability to define **properties** for components.
- We've found a new calculator service that lets us define the number of decimal places in the result. The methods look like this:

```
double add(double n1, double n2,  
           int precision);
```
- The **precision** parameter to these methods doesn't exist in our **CalculatorClient** application or the **CalculatorService** interface.

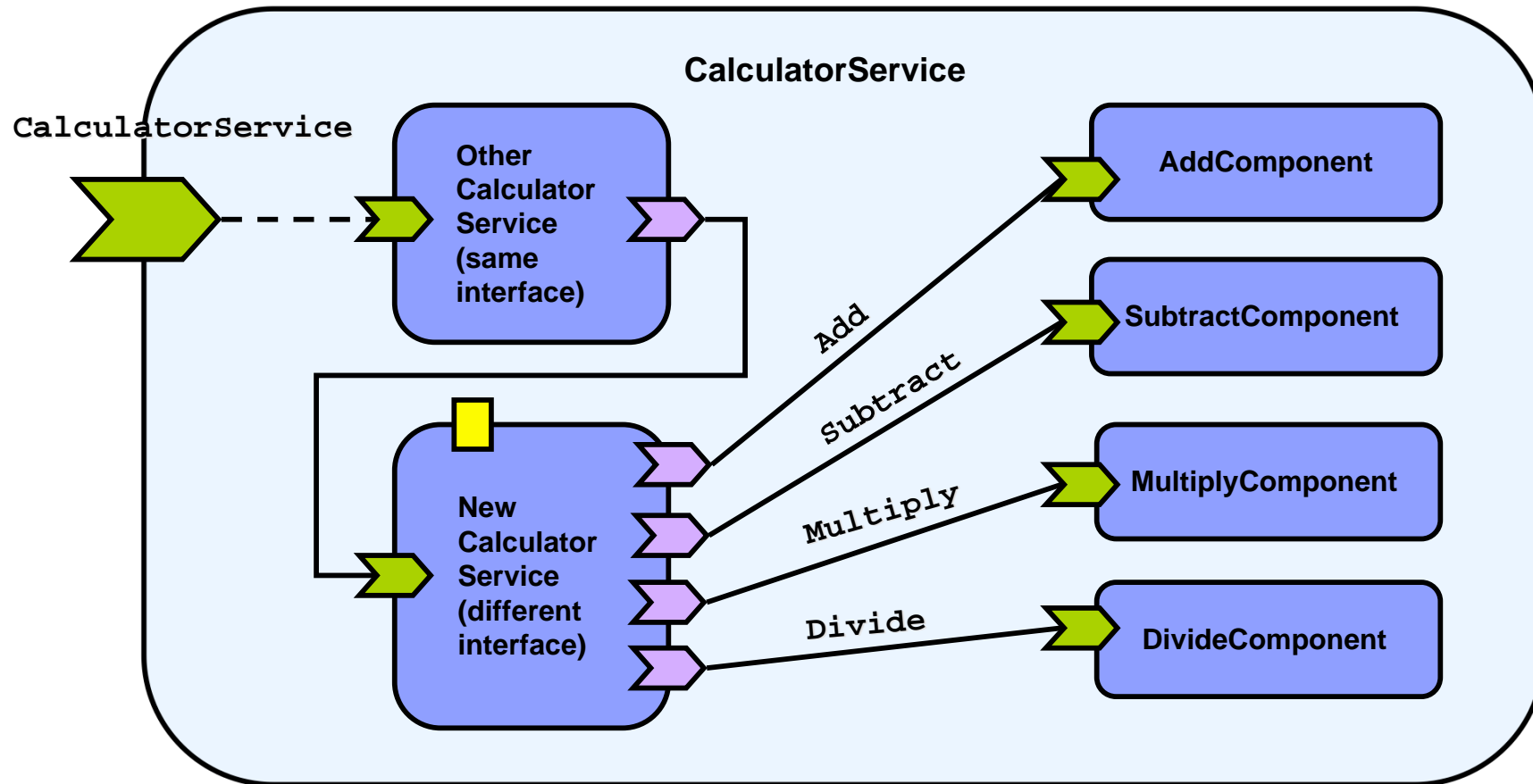


# Properties

---

- We can define a property to handle the difference between the interfaces:
  - We'll set up a component that has the same interface as the original calculator class.
  - That component will have a reference to the new calculator class, the one with the incompatible interface.
  - The new calculator class will use a property for the precision parameter.

# Properties





# Properties

---

- Here's how the property looks in the service code:

```
private int precision = 2;
@property(name="precision")
public void setPrecision
    (int precision)
{
    if (precision >= 0)
        this.precision = precision;
```



# Properties

---

- When the SCA domain loads the service, the property from the **.composite** file is passed to the **setPrecision** method.
- If the property isn't defined in the **.composite** file, the code uses the default value (**2**) that we coded in the service class.
  - If there's no default value and no property in the **.composite** file, the SCA runtime initializes the value to zero.



# Finding the `.composite` file

---

- There are different ways to find the file:
  - Use the `.composite` file name as a parameter.
  - Use the same name for every XML configuration file that you might want to use.
    - Change the Java `CLASSPATH` so that the SCA infrastructure finds one `.composite` file instead of another.
  - Put the name of the `.composite` file into a Java `.properties` file. Load the string at runtime.
  - Change the `.composite` file to point to a different service.



## Finding the `.composite` file

---

- Clearly this is a management issue; you'll want the `.composite` files in some sort of registry and repository.
- You'll also want some sort of management software to track what services are being used, who's calling the, how they're performing, *etc.*
- None of these concerns are addressed by the SCA specs.



# Bindings

---

- In SCA, a *binding* specifies how to access a service.
  - Current bindings include WSDL, RMI, JMS, JCA and EJBs.
  - More bindings are coming all the time at **osoa.org**.
  - Like all of SCA, the binding specification is open, so you can create your own.



# Policies

---

- Previous standards efforts, WSDL in particular, didn't include how to define *policies* for services.
- SCA gives you a single declarative way to establish policies.
  - "This component must provide this level of QoS."
  - "All traffic on this wire must be digitally signed."
- To add authentication declaratively:
  - `sca:requires="authentication"`
- The SCA runtime implements the policy, the application does not.



# Without SCA

---

- **It's more difficult to maintain your applications.** If you need to use a different service provider, you have to change the code.
- The application programmer also has to know the details about the service endpoint and the access method.



# Resources

---





# Resources **OASIS** **Open CSA**

---

- The OSOA's work is moving to OASIS. For more information on the Open CSA project, visit [oasis-opencsa.org](http://oasis-opencsa.org).
- The [oasis-opencsa.org/specifications](http://oasis-opencsa.org/specifications) page is another great way to find the specifications and other technical resources.

# Apache Tuscany



- The Tuscany project is hosted at [tuscany.apache.org](http://tuscany.apache.org).
- The latest Java versions are:
  - SCA – V1.3.2 (October 2008)
  - SDO – V1.1-1 (July 2008)
- C++ and PHP versions are available as well.



# Fabric3

---



- Fabric3 is another open-source implementation of SCA.
- You can find it at [fabric3.codehaus.org](http://fabric3.codehaus.org).

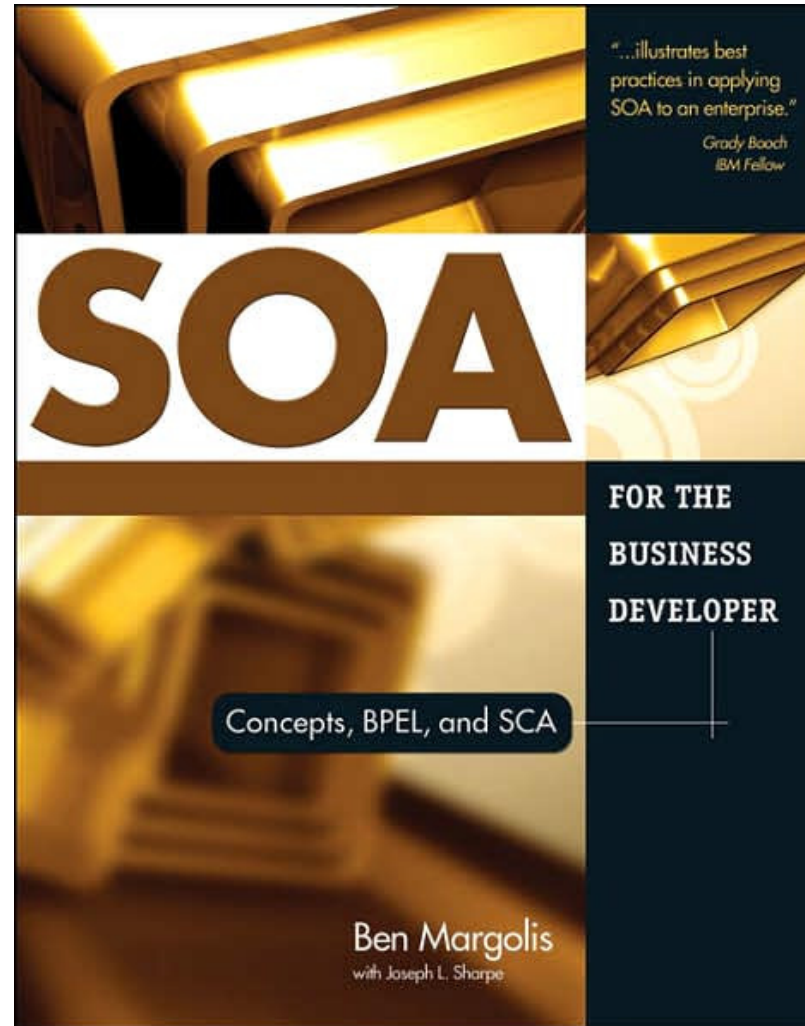
# SCA/SDO in PHP



- Visit [pecl.php.net/package/SCA\\_SDO](http://pecl.php.net/package/SCA_SDO).
- You can use the PECL package manager to install the SCA and SDO extensions automatically.
- The ability to use annotations in PHP is a major step forward for using PHP scripts in an SOA.

# The best book on SCA

- Get Ben Margolis' book ***SOA for the Business Developer: Concepts, BPEL and SCA.***
- *This is the best book on SCA.* It covers both the architecture and the technical details.
- Many of the best minds in the SCA world reviewed this book.
- ISBN 1-58347-065-4
- **[www.mc-store.com/5079.html](http://www.mc-store.com/5079.html)**





# Getting started

---





# Getting started

---

- Visit the Apache Tuscany project and download the latest Java or C++ code.
  - All of the packages contain sample applications and services.
- If you're a PHP developer, use PECL to install SCA and SDO support. The PHP package includes sample applications as well.
- Visit [osoa.org](http://osoa.org) for the latest SCA and SDO resources.
- **Start a pilot project** – Find an application that requires a great deal of flexibility.



# Summary

---





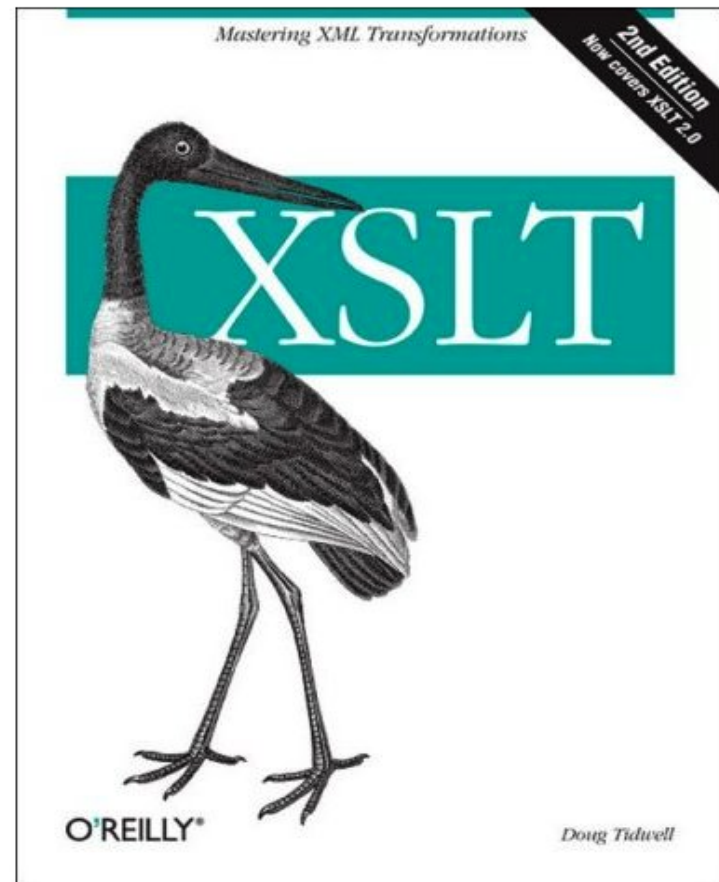
# Summary

---

- SCA is a much more elegant way of building composite applications in a Service-Oriented architecture.
  - With SCA, every kind of service is invoked the same way.
  - With SCA, you can change the infrastructure without changing the app.
  - With SCA, your application developers focus on business logic, not plumbing.

# A serious lapse in taste...

- The second edition of O'Reilly's XSLT is available! (ISBN 0-59652-721-7)
- A great gift for:
  - Freemasons
  - Expectant mothers
  - Supermodels
  - Disaffected loners
  - Your parole officer





# Thanks!

---

Doug Tidwell  
IBM Corporation

[dtidwell@us.ibm.com](mailto:dtidwell@us.ibm.com)

