Understanding the CCA Standard Through Decaf

(Updated for CCA 0.6.1 and Babel 0.8.4)

GARY KUMFERT

Disclaimer

This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

Release Information

Understanding the CCA Standard Through Decaf (this document)	UCRL-MA-148390
Babel Source Code (associated software)	UCRL-CODE-2000-048
Babel Users' Guide	UCRL-MA-145991

Understanding the CCA Standard Through Decaf

(Updated for CCA 0.6.1 and Babel 0.8.4)

GARY KUMFERT

Center For Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808
Livermore, California, USA

April 30, 2003

Preface

Babel in a Nutshell

Babel is a tool that enables software written in different languages to communicate. It accomplishes this task by using an Interface Definition Language (IDL) similar to COM and CORBA. Babel relies on the Scientific Interface Definition Language (SIDL) that is specifically tuned for scientific applications. By expressing software interfaces, or APIs[1], in SIDL the appropriate glue code stubs and skeletons can be generated to facilitate language interoperability. Features unique to SIDL are:

- Dynamic multi-dimensional arrays
- Complex numbers (e.g. $2 + 3i$)
- In-process optimizations
- Special directives for large-scale parallel distributed programming (future)
- Syntax for specifying interface behavior (future)

Babel enables true object-oriented techniques even in non object-oriented languages. The object model that SIDL supports is similar to Java and Objective C where a class can extend at most one class, but implement many interfaces. In C++ speak, an interface is simply a class of all pure-virtual methods. Furthermore, if library developers want object-oriented features but are required to be 100% ANSI C compliant, Babel can meet those constraints. Although the Babel code generator is implemented in Java, the runtime libraries and generated files for C bindings are 100% ANSI C compliant.

Babel can be used as the basis for a component framework, but it is *not* a complete framework by itself. We've added a tiny CCA-compliant framework, called *Decaf*, in our examples/ directory. Decaf demonstrates how Babel can be used to implement a component framework.

SIDL is also a useful communications tool for code development teams since it only expresses the public API. That is, implementation details, which often prove distracting during collaborative design, can be safely avoided by restricting discussions to the interfaces described in SIDL. Furthermore, since SIDL is simple and clean it can be used by Computer Scientists, Math Programmers, and Application Scientists to debate APIs even using only email.

[1] Application Programming Interfaces

Scope of this Manual

This document is a tutorial on the CCA Standard as realized through the Decaf implementation. Decaf does not equal the CCA standard much in the same way that Microsoft Visual C++ is not ANSI/ISO C++.

This document was created because the CCA standard is evolving and still too fluid to nail down in a tutorial document. Because of its fluidity, and that it represents a hotbed of research and development, beginners can only start learning CCA by choosing one of the frameworks (warts and all). Decaf has just enough functionality to be a useful tool for beginners in the CCA to get started on. Though it lacks many features of the bigger CCA frameworks (CCAFE [3], XCAT [10], and SciRUN [8]) where the heavy-duty research is still going on, it is the first CCA framework that is underpinned by Babel, which provides its language interoperability features.

This document can also serve the dual-purpose of providing a reasonable-sized example of building an application using Babel. The entire source for Decaf is included in the examples/ subdirectory of the Babel code distribution.

This manual assumes the reader is a programmer who has a conceptual understanding of the Babel Language Interoperability Tool. They should be proficient in two or more of the following languages: Fortran77, C, C++, Java, or Python. Furthermore, this manual assumes the reader is familiar with the SPMD[2] programming model that pervades the scientific computing community. Knowledge of and experience with MPI programming is helpful, but not strictly required.

Getting the Software

Babel source is available free of charge on the web. Developed by the Components Project at the Lawrence Livermore National Laboratory Center for Applied Scientific Computing (CASC), it is licensed under the Lesser GNU Public License (LGPL). See the source distribution for details.

The Babel distribution is published on Alexandria along with software components available for use with Babel. Alexandria is a software component repository that is also built by the Components Project at CASC. You can access Alexandria on the web from the following URL:

```
http://www-casc.llnl.gov
```

Readers may also be interested in viewing the Components Project home page at

```
http://www.llnl.gov/CASC/components
```

Conventions

The following typographic conventions are used throughout this manual.

[2] Single Program Multiple Data

Italic	is used for file and command names. It is also used to highlight comments in examples and to define terms the first time they appear in a document.
`Constant Width`	is used in examples to show the text that is generated, and in regular text to show operators, variables, and the output from commands or programs.
`Constant Slanted`	is used for displaying for SIDL source code. We use a separate font to distinguish SIDL code from generated code.
`Constant Bold`	is used to show user's modifications to generated code and in examples to show user's actual input at a terminal.
Sans Serif Slanted	is used in examples to show variables for which a context-specific substitution should be made. The variable *filename*, for example, would be replaced by the actual filename.

Additionally, we may use specific blocks of text as sidebars to call the readers attention to particular information. Here's one kind.

> **Rationale:** *Often when listing restrictions or requirements, we find it helpful to also explain and document the rationale behind a design decision. In time, the context in which the rationale was based may become irrelevant, making the rationale blocks very useful for understanding when to change a decision.*

We Appreciate Your Feedback

We have tested and verified the information in this manual. Nonetheless, features may have changed or oversights may exist. Please contact us with any issues, corrections, or suggestions for future versions of this manual through snail mail at:

Components Project
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
P.O. Box 808, L–365
Livermore, CA 94551

or through email to:

`components@llnl.gov`

To find out more about Babel, feel free to subscribe to one or more of the associated distribution lists given below.

- `babel-announce@llnl.gov` is a moderated email forum to which anyone can subscribe (though no-one can post). This is a low-volume alternative for people who want to know about releases and major announcements.
- `babel-dev@llnl.gov` is an open discussion forum about Babel for serious babel users who want to talk about the internal workings of the tools. Anyone can subscribe or send email to this list.
- `babel-users@llnl.gov` is an open discussion forum about Babel for users. Anyone can subscribe or send email to this list.

To subscribe, simply send email to `majordomo@lists.llnl.gov` with the appropriate line(s):

`subscribe babel-announce` *[email-address]*
`subscribe babel-dev` *[email-address]*
`subscribe babel-users` *[email-address]*

where you can explicitly state your email address in *email-address* or, if you leave *email-address* blank, majordomo will use your email ReplyTo: field.

Acknowledgments

Project Alumni

- Melvina Blackgoat
- Nathan Dykman
- Scott Kohn
- Brent Smolinski

Alpha Testers

- Andy Cleary
- Jeff Painter
- Cal Ribbens

Contents

Chapter 1

Introduction

Before the beginning of great brilliance, there must be chaos.
Before a brilliant person begins something great, they must look foolish in the crowd.
— From the I Ching (a.k.a. Yi Jing)

Contents

1.1 Brief Introduction to the CCA

The Common Component Architecture Forum (CCA Forum) [4] is a grassroots organization that since 1998 have tried to improve reusability in scientific computing software by developing and deploying component technology. It is largely comprised of various U.S. Department of Energy (DOE) [9] Laboratories, NASA, some industry and other government researchers, and Universities. Membership is free and open to all researchers. Voting rights are acquired by attending two out of the last three quarterly meetings, which are alternately hosted by various members around the nation.

In 2001, a subset of the CCA Forum gained funding under the U.S. Department of Energy as an Integrated Software Infrastructure Center (ISIC) under the Scientific Discovery through Advanced Computing (SciDAC) program [7]. This funded subgroup has a separate formal name: the Center for Component Technology for Terascale Simulation Software (CCTTSS). In practice, "the SciDAC CCA" or "CCA ISIC" has proven more pronouceable in informal circles. Organizations represented in the CCTTSS include participants from Argonne, Livermore, Los Alamos, Oak Ridge, Pacific Northwest, and Sandia National Laboratories; Indiana University and the University of Utah. The CCTTSS is dedicated to the development of a component-based software development model suitable for the needs of high-performance scientific simulation, particularly the Common Component Architecture (CCA).

This architecture developed by the CCA Forum takes a minimalist view of components. Two primary motivations for this approach are (1) that anything too complicated would never be used in practice, and (2) to maximize the number of existing scientific codes amenable to retrofit or "componentization."

The specification for the CCA, prescribes how a CCA component interacts with a CCA framework. At the time of this writing, there is no one "official" implementation of a CCA framework. The major efforts in this arena are CCAFFEINE from Sandia, XCAT in Indiana University, and SCIRun2 from University of Utah. Each of these implementations have their relative strengths and weaknesses, and all of them are actively used as research tools to flesh out the dark corners of this new technology. A CCA component can be any piece of software — including wrapped legacy software — that interacts with a CCA framework in the prescribed manner.

1.2 The SIDL Grammar

Another important technology to understand is the Babel language interoperability tool, and the SIDL grammar that it uses. Babel is a stand-alone product that can (and increasingly is) being used to provide language interoperability to CCA frameworks. Decaf is the first CCA framework to use (as well as be implemented using) Babel. It was originally written by the Babel developers as a proof of concept to the hard-core CCA framework developers. Decaf's mission then grew to serve as a stopgap for early CCA adopters until the full-featured frameworks complete their Babel integration. Now that CCAFFEINE is "Babelized", Decaf is being maintained to demonstrate how the same CCA components can be loaded into different CCA-compliant frameworks. For more background on the SIDL grammar or the Babel tool — including getting the Babel software which includes Decaf as an example — we recommend looking at the Babel Website [1] and the Babel User's Manual [5].

Babel's technology is immediately important to the CCA because SIDL is the official language in which the CCA specification is written. It is also the language that component interfaces must be written in. The word "interface" is so dangerously overloaded, it is worth a moment to explain its uses here. In general terms, an interface is simply how one interacts with a "black box" of software. The acronym API, which originally was a marketing buzz-word for *advanced programming interface*, has now become the vernacular for any programming interface to a software library. In the SIDL grammar, an interface is a named group of function calls. In practice, a SIDL interface is identical to an "interface" in the Java language, or a "pure abstract base class" in C++. The SIDL interface itself has no implementation associated with its member functions (a.k.a. methods). It only specifies the contract between the caller and whatever "black box" is provided under the hood. We go into this detail because the CCA (and most standards using SIDL) prescribe a collection of SIDL interfaces, which collectively form the API that adherents of the standard must conform to. A CCA component is an implementation of one or more SIDL interfaces.

The CCA specification in SIDL is entirely expressed in interfaces[1] meaning that the component

[1] CCAException is currently not an interface. This is a temporary concession to Babel's exception handling infrastructure and may change in future Babel releases. The CCA spec also defines two enumerations but these are equally implementation independent.

interacting with the CCA framework is completely decoupled from which implementation of the CCA framework is running. By specifying CCA components in SIDL and using Babel, component developers are free to implement their CCA components in any Babel supported language (currently C, C++, Fortran77, Fortran90, or Python[2]).

1.3 Author's Comment on the Learning Curve

As the title suggests, the document was written with the understanding that the reader may well be learning two or more technologies simultaneously: some combination of SIDL, CCA, Decaf, Babel, OOP, etc. As such, this document labors a bit to keep readers from diverse backgrounds engaged.

The material is presented in an iterative tutorial fashion, mirroring the learning process. To help in this process, I've formalized levels of understanding from 1–5. The first level of understanding is purely conceptual; what are the different technologies and what's the relation between them. This level also glosses over lots of details (including some rather important ones!) but is a necessary first level. The second level includes the major details and brings the different technologies into sharper focus. At this level of understanding concepts become distinct entities and not just blobs with some degree of overlap. Level 3 includes some of the more nit-picky details that are all but the greenest of users would eventually learn. Level 3 is the hump of the learning curve. Level 4 is advanced details that casual users can do without. Level 5 is guru-dom, which almost certainly won't be found anywhere in this tutorial document.

Speaking of these levels 1–5 in the actual text becomes tedious, so I'll also use the some form of the words *conceptual*, *beginner*, *intermediate*, *advanced*, and *expert*, respectively.

In these terms, the goal of this document is to boot-strap people new to the CCA and/or Decaf up to intermediate understanding. Chapter 3 starts from nothing and presents up to beginner information on the Common Component Architecture and the CCA Forum. Chapter 4 deals with beginner information of Decaf and pushes CCA information to intermediate. Chapter 5 shows coding examples using Decaf which takes us to intermediate for Decaf. This tutorial assumes the reader is starting with a beginner understanding of Babel and SIDL. (If you've never heard of Babel or SIDL before, consider yourself Level 0, and come back to this document after you've at least read the first two chapters of the Babel User's Manual.)

As the sole author of Decaf, I suppose it behooves me to claim expert understanding of Decaf. I suspect, however, that one can only teach at a maximum of one less than their own level of understanding. This constraint has the desirable implication that guru-dom cannot be taught and must be experientially achieved. Now that I think about it, I think understanding, particularly of software, suffers at least another notch if it is gained academically and not experientially. So even by presenting advanced details, the reader is stuck on the side of the learning curve with positive slope until time is expended on working with the code directly.

[2] Babel also supports calling from Java into any of the other supported languages, but development of the converse is under development.

Chapter 2

CCA Forum's Vision

See first that the design is wise and just:
that ascertained, pursue it resolutely;
do not for one repulse forego
the purpose that you resolved to effect.
— William Shakespeare (1564–1616)

2.1 Terminology

Here we explain the concepts of a *component*, *port*, and *framework*; providing a functional descriptions and clearing up the hype and buzz surrounding component technology in general.

2.1.1 Definition of "Component"

There is no universally accepted definition of *component* in contempory computer science. There's still no single definition of what *object oriented* is either — and it's been around for a lot longer. As with OOP, definitions of component will invariably congeal around particular implementations, forming different camps. The SmallTalk camp's defintion of OO will be different than the C++ camp's. Similarly, CCA components are different than CORBA, .NET, or EJB — not in the primary goals or fundamental principles, but in the final design and execution.

The core principle of component technology is to produce chunks of code that are composable. The goal is to make programmers more efficient by increasing the amount of reusable chunks and reducing the need for reinventing the wheel. In my personal research, I often observe that 90% of the code I produce is infrastructure to support the 10% which is novel research. If I could get that 90% from stuff others wrote and concentrate on my own 10%, then I'd be a lot more productive.

The idea of composability is not new. UNIX programmers have long enjoyed a certain level of composability at the commandline by connecting processes using pipes and file redicrection. With simple programs such as `echo`, `cat`, `find`, `grep` and `test` and some redirection a powerful range of possible tools can be constructed quickly.

As a composability model, file redirection has severe limitations. Data flow and execution order is unidirectional; from one process to the next, linearly. The shared interface by which processes are connected is the most general and low performant: text I/O. While this composability model works well for system administration tasks, it is not well suited for high-performance scientific simulation.

CCA components allow for data to flow back and forth between caller and callee, control flow is passed from caller to callee and returned when callee completes. Furthermore, component developers can design arbitrarily complex interfaces by which components connect on a function call level. So there is no need for I/O, IPC, or data conversion overheads. CCA components are designed to be plugged into an existing application without recompiling it. Much as web-brower plug-ins can be added without forcing a recompile of the browser.

2.1.2 Definition of a "Port"

CCA components are often said to have well-defined interfaces. It is hard to write software without some kind of interface — whether commandline, GUI, or API — so the "well-defined" adjective is at best inadequate and more likely downright confusing. It doesn't help that the word "interface" is used with a different meaning than most new comers expect. A stronger and more descriptive restatement is: interfaces to components are source code, completely separate from implementation source. The practical importance of this is easily underestimated.

There are two places programmers typically look for information about the calling interface when using a software library: the documentation and the source code. The documentation tends to be the first place to look. It is a text-based representation, it should have some narrative to help people understand and begin to anticipate design features, but is often incomplete, mistaken, or simply out of sync with the actual software. Source code is the less desirable, but authoritative source for calling interface information. Here, users must winnow out implementation details and search for calling interfaces.

A CCA Port is similar in function to a Java interface, a C++ pure abstract base class, or even a struct of function pointers in C. It is a specification that is shared between the user and provider. Hardware people often relate port specifications to socket specifications for chips on circuit boards. As long as the circuit board and the chip manufactures agree on a socket spec (physical layout as well as electrical signals) the two should work together.

CCA Ports are specified in SIDL; a programming language and platform independent grammar for describing interfaces. SIDL is sourcecode for Babel, which will generate sourcecode for various languages to connect to each other seamlessly. So, when people talk about CCA components having well defined interfaces... they mean that their interfaces are formally defined in SIDL; simultaneously hiding the implementation details of the component from view and enabling that component to be accessed from all Babel-supported languages.

2.1.3 Definition of a "Framework"

The ability to compose simple programs in UNIX through file redirection relies specifically on UNIX facilities. Component composability models require additional services beyond what an OS can do. These services are provided by a component *framework*. It is essentially an application that manages the creation, manipulation, and destruction of components.

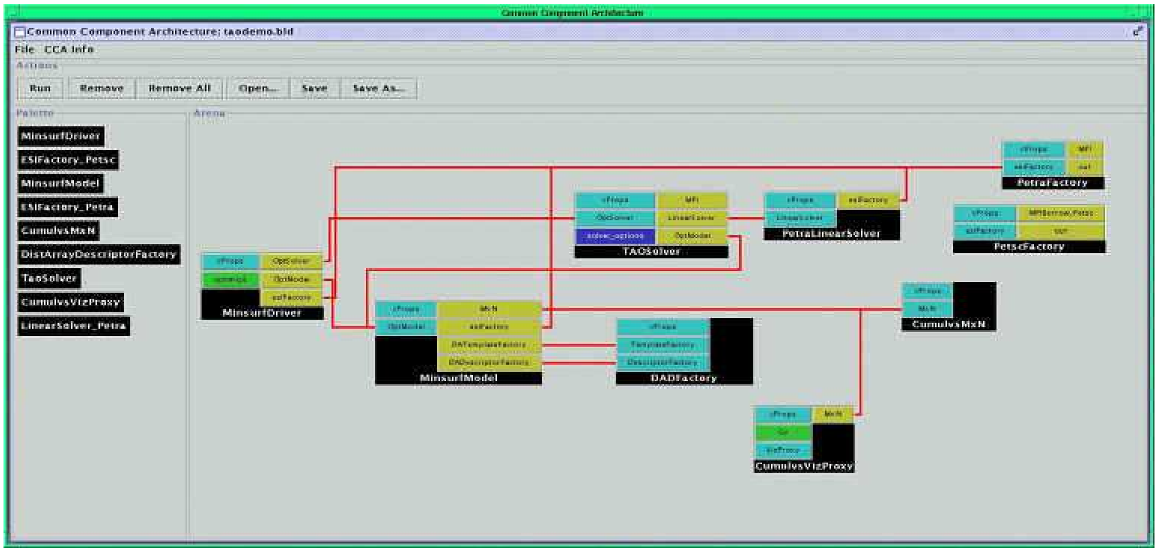

Figure 2.1: A Screenshot of the Ccaffeine Framework's GUI in action.

2.2 Example of a CCA Framework in operation

Figure 2.1 shows a screenshot of the Ccaffeine Framework's GUI in action. Building and running applications using CCA components does not require use of a GUI, but it is helpful for demonstrating code structure, debugging, and rapid prototyping.

The main window has two parts: a *palette* on the left hand side and an *arena* on the right. The palette lists all the component types that are available, therefore each entry is unique. An instance of a component of a particular type is created by clicking on a component type in the palette and dragging it into the arena. At that point, the framework asks for a suggested name for the instance. Although many instances of the same component type are allowed, each component instance must have a unique name regardless of type. Typically a framework will append digits to the suggested name to force uniqueness if need be, though the exact process for insuring uniqueness is not formally specified.

Instantiated components are rerpesented as large black boxes with the symbolic name at the bottom center. Starting from the top left corner going down the left edge are *provides ports*. Starting from the top right corner going down the right edge are the *uses ports*. Ports also have a type and a name. The name of each port must be unique to that component. To connect a provides port to a uses port, one simply clicks on the appropriate port, the framework highlights all the candidate ports that can be connected to it, and the programmer clicks on one of those. If a connection is succesfully established, the framework draws a line between the two ports.

Because of the geometric layout of ports, it is common to have a driver component leftmost in the arena, and arrange components in tree-like fashion. A special port type call the `gov.cca.ports.GoPort` serves as the equivalent to `main` and is usually used as a provides port. The Ccaffeine framework displays the GoPort as a green button which the programmer presses to launch the application.

2.3 Common Questions

2.3.1 Performance

The issue everyone raises with new-fangled software technology is performance. The CCA specification focuses on creating and assembling components. Once the application is assembled, the user clicks the "go" button and the application runs without CCA intervention. Ccaffeine is also *embarassingly parallel*, meaning that if a component needs to run on a thousand processors, a thousand Ccaffeine frameworks can trivially be launched to host them. The Ccaffeine framework can provide a MPI communicator, but has no real internal need to use it. The parallelism is an implementation detail of the component, not really the framework.

What does get in between caller and callee of components is Babel. It is responsbile for maintaining the langauge interoperable layer between languages. Usually, this layer has a overhead of 2-3 Fortran 77 call overheads (about same as virtual C++ methods). Most data is either small and passed by value (e.g. integers), or large and passed by reference (e.g. arrays and objects). One notable exception is strings which are represented differently in almost every language and therefore copied ad nauseum. The neccessity of optimizing the case of strings is dubious, however, since HPC codes rarely pass large blocks of text back and forth.

Detailed analysis of Babel overheads are published in [2]. Remember that these overheads are constant overheads per subroutine. . . on the order of nanoseconds on contemporary workstations. In complete parallel applications, this small constant overhead can be completely lost in the noise, as was found in [6].

2.3.2 Components vs. Object-Oriented

To distinguish components from object-oriented programming (OOP), it is helpful to consider what is involved in converting an object-oriented library into a package of CCA components.

First, there is a taxonomy exercise where the author(s) of an object-oriented library must classify their objects appropriately. First they must distinguish which objects are to be advertised to the user, and which are infrastructure that they don't want exposed. Then, of the objects that are to be exposed to the public, they must be further subdivided into components and *ports*. CCA uses the term *ports* to denote the "well-defined" interfaces that components exhibit. Concrete (and therefore instantiable) classes in the public interface tend to become components. Base classes, abstract classes, or interfaces (depending which OOP language's vernacular) tend to become ports.

Second, a good conversion will reexamine relationships between components. For an object to invoke a method on another, there generally is a "has-a" relationship between the two. In component programming, components should eschew direct "has-a" relationships with other components in preference for "has-a" relationships with ports. The rationale is weaker in a single package of components context, but becomes very important later when multiple packages are involved.

To be a CCA component, one must implement the `setServices()` method. This method is the main entry point for each component when it is created. In this method, components typically make a personal copy of the `Services` object passed to it. This object is the component's primary communicator to the framework. Components will also typically register what ports they can provide and what ports they require. Then the method generally returns without having done any computationally intensive tasks, other than general initialization. By registering a port that they provide, a component is allowing unknown and component(s) make calls on that port that it must service. By convention, an invocation of `setServices` with a Null `Services` object is the framework's mechanism for shutting down the component for reclamation.

Lastly, a good conversion to CCA should wrap components and ports in Babel. Babel is a language interoperability tool whose input grammar is called SIDL (Scientific Interface Definition Language). Though one can program in the "CCA Design Pattern" without using Babel, the official CCA specification is written in SIDL and alternative bindings are largely historical artefacts or speculative research. The first benefit is that by wrapping in Babel, all the language specifc details about the component are hidden from other components or the framework itself. Secondly Babel allows the framework to hide its implementation language specific details from the component. Babel wrapped components can even be used without modification on CCA frameworks from different institutions.

Chapter 3

CCA Specification

The significant problems we have cannot be solved
at the same level of thinking with which we created them.
— Albert Einstein (1879–1955)

Contents

3.1 General Comments

This encyclopedic chapter lists every interface in the CCA spec and explains each line of SIDL code, the history, and the intent... as best I understand them. I've ordered the sequence of interfaces and methods therein for cleaner presentation purposes as well as stripping the comments out of the code and providing an overarching narrative instead.

The following section presents all the SIDL interfaces, abstract classes, and enumerated types that make up the CCA standard, but they are presented one at a time, with narrative interspersed. A comment is added in each fragment of the SIDL file to remind the reader that the definition of the SIDL type should be appropriately scoped in a package declaration.

```
package gov {
  package cca version 0.6.1 {

    // CCA Core Spec goes here

    package ports {

      // CCA default Ports go here.

    }
  }
}
```

According to SIDL, all classes and interfaces gain the same version number as their most immediate versioned package. The outter most package, `gov`, is unversioned so it cannot directly contain any classes or interfaces only other packages. The `package` statement is used in this manner to create nested scopes to minimize naming conflicts. The second line creates the nested, versioned package `gov.cca`. All classes and interfaces declared therein are assigned the same version number as the enclosing package. Nested inside is the `gov.cca.ports` package, which looks unversioned and is implicitly versioned by `gov.cca`. The SIDL grammar allows nested packages like `gov.cca.ports` to be explicitly versioned to be different from parent versioned packages.

By design, the entire CCA specification is SIDL interfaces[1] with the so-called *Core Specification* in `gov.cca` and *Default Ports* in `gov.cca.ports`. The expectation is that the core spec will be small and evolve very slowly. The list of default ports will more likely expand over time and with growing users.

[1] There are two exceptions due to technical limitations of Babel/SIDL, but these are expected to be fixed in the near future.

3.2 The CCA Core Specification

The CCA core specification is pretty minimal by design. The interesting functionality comes from the default ports in Section 3.3. All of the SIDL types that make up the CCA core are in the `gov.cca` package.

3.2.1 `gov.cca.Port`

The idiom of "ports" and how they're used is the single most distinctive feature of the CCA. Conceptually, ports serve a similar function as to SIDL interfaces: a typed collection of method signatures. All CCA ports must (directly or eventually) inherit from the `gov.cca.Port` interface.

```
// in package gov.cca

interface Port { }
```

The way `gov.cca.Port` is defined in SIDL seems to indicate that CCA ports are exactly SIDL interfaces, but in truth ports are only implemented as SIDL interfaces. Every beginner soon learns that CCA Ports are used very differently.

Using raw Babel — or any other object-oriented paradigm for that matter — the typical pattern would be to instantiate an object that inherits the interface, cast it to a pointer or reference to that interface (a.k.a. abstract base class) and then hand it off to other code that knows nothing about the underlying concrete type.

The CCA has a looser coupling between the user and provider of a computational service. Connecting the two involves multiple steps.

1. The user of a service registers its request to the framework.
2. The provider of a service registers its availability to the framework.
3. The user and provider are connected (by various means).
4. The user does what it needs to with the service it has been granted.

This conceptual description will be refined later, but is a good start. We will go into more details about this process after a few more interfaces have been presented.

3.2.2 `gov.cca.Component`

If conceptually a CCA port can be mapped to a SIDL interface, then a CCA component is mapped to a SIDL class... one that implements the `gov.cca.Component` interface.

```
// in package gov.cca

interface Component {
    void setServices( in Services svcs );
}
```

By definition, any piece of software that implements this interface is a CCA component. The idea here is that the framework itself will create a unique `Services` object, create the component, and then call this method on the component... giving it an opportunity to do whatever communication it needs to with the outside world through this `Services` object.

During the CCA component's life-span, it will hopefully have its port requests satisfied, and itself be used to satisfy other requests. In this way, its ability to influence its surroundings grows beyond the simple Services object, but it always starts here.

To be concrete: for a component writer to implement this interface using Babel and a Babel-enabled framework (such as Decaf), they can do the following steps:

1. Define a SIDL class that inherits one or more SIDL interfaces, including the `gov.cca.Component` interface.
2. Run Babel's code generator over the SIDL file, specifying what language they want to implement the class in.
3. Fill out the implementation in the files Babel generates, or just add enough to connect it to existing code elsewhere.
4. Package all the files together in some static, shared, or dynamically loadable library[2].

Whereas much of the interfaces specified in the CCA must have an underlying implementation in the Framework. The SIDL class(es) that implement this interface are entirely the design of the component writer, not the framework.

3.2.3 *gov.cca.ComponentID*

A component never gets to interact with another component directly in the CCA. Instead, it deals with this proxy interface, `gov.cca.ComponentID`. It is interesting to note that this interface is essentially *read-only*, meaning that component users and developers can get information from a `gov.cca.ComponentID`, but can never change its internal state.

```
// in package gov.cca

interface ComponentID {

    string getInstanceName() throws CCAException ;

    string getSerialization() throws CCAException ;
}
```

The instance name of a `gov.cca.ComponentID` is a unique string identifier for all live component instances in the framework. The actual format of this string is undefined. The serialization string of a `gov.cca.ComponentID` is intended to be a string sufficient in detail for a component to save its state to disk and restart at a different time. The XCAT framework is the only one known to implement this serialization at the time of this writing. XCAT is a Java-only implementation and relies on the serialization facilities built into the Java language.

[2]Shared libraries (a.k.a. dynamically loadable librarys or DLLs) have the added advantage of not having to relink the entire framework before using the component. Babel supports statically linked libraries for added performance at the expense of this flexibility.

3.2.4 *gov.cca.Services*

It should be emphasized, that most components will cache the Services object they're given and use it to register/unregister ports throughout its lifetime, not just in response to the `setServices()` call. A component may choose not to provide ports until all of its user port requests are met. CCAFE is CCA framework with built-in scripting and GUI development tools. Users can see new provides ports created and destroyed as they draw lines to connect components.

It may be that all software has to do to be a CCA component is implement the `setServices()` method, but that's like saying all it takes to write a valid C++ code is to put

```
int main() { }
```

in a file. To write a useful CCA component, you must understand the nine methods of the `gov.cca.Services` object, and use them appropriately.

```
// in package gov.cca

interface Services {

    ComponentID getComponentID();

    TypeMap createTypeMap() throws CCAException;

    void registerUsesPort(in string portName,
                          in string type,
                          in TypeMap properties )
        throws CCAException ;

    void unregisterUsesPort(in string portName)
        throws CCAException ;

    void addProvidesPort(in Port inPort,
                         in string portName,
                         in string type,
                         in TypeMap properties )
        throws CCAException ;

    void removeProvidesPort(in string portName)
        throws CCAException ;

    gov.cca.TypeMap getPortProperties( in string portName );

    Port getPort(in string portName)
        throws CCAException;

    Port getPortNonblocking(in string portName)
        throws CCAException;
```

```
        void releasePort(in string portName)
            throws CCAException;
    }
```

Discussion of each of these ten methods follows in order.

Each component can get access to its own *gov.cca.ComponentID* through the *gov.cca.Services* interface. Because a framework assigns a unique string name to each component instance, getting this name from the *gov.cca.ComponentID* is sometimes useful and cannot be statically predicted.

The ability of *gov.cca.Services* to create a *gov.cca.TypeMap* may at first seem a bit wierd. Why not create a *gov.cca.TypeMap* directly? The answer is that *gov.cca.TypeMap* is an interface, not an implementation. Interfaces cannot be instantiated. The *gov.cca.Services.createTypeMap()* method is the CCA Forum's way of saying "use whatever kind the framework hands you."

The CCA created the *gov.cca.TypeMap* interface as an extensible mechanism for passing miscellaneous configuration information. SIDL does not support variable length argument lists[3], default argument values[4], or key-value pairs[5] and *gov.cca.TypeMap* can be used to provide a similar capability. We'll save the details for *gov.cca.TypeMap* for later because its a very basic thing — think typesafe hash table or Microsoft registry.

Now we get to the real meat of the CCA, manipulating ports to the framework. The four methods *registerUsesPort()*, *unregisterUsesPort()*, *addProvidesPort()*, and *removeProvidesPort()* are the main mechanism by which this is achieved.

The *registerUsesPort()* method is how a component notifies the framework that it requires a port of some specified type. This connection request is satisfied by some mechanism external to the component that we'll describe later. To *registerUsesPort()*, the requesting component must give a string name to this request, called the *portName*. This name must be unique for all ports that this component registers to this *Services* object. The second string argument is the type of the port. If and when this request for a port is satisfied, the framework guarantees that the type of the port provided is compatible with the request. The final argument is a *TypeMap* for any additional meta-data about the port's properties. The *unregisterUsesPort()* method simply removes the request for this port based on the port's name. This means that the *portName* in the removal of the port must be identical to the *portName* given when registering it.

The *addProvidesPort()* method is how the component notifies the framework that it is making available a port of a specific type.

The corresponding methods to add and remove "provides ports" are almost identical to the "uses ports" methods. The one difference is that *addProvidesPort()* also requires a *gov.cca.Port* as its first argument. In this call, the framework will save the port it was given and forward it to any uses port that it gets connected to.

The *getPortProperties()* method returns the *TypeMap* of properties associated with

[3]e.g. `printf(...)` like in C

[4]e.g. `foo(int i=0)` like in Java or C++

[5]e.g. `foo( color='red')` like in Python

the port.

The recipient of this provided port, the component which registered the corresponding uses port, is also called the *using component*. The using component gets access to this `gov.cca.Port` by calling after their posted using port has been satisfied. Strictly speaking, `getPort()` blocks until the call is satisfied, an alternative `getPortNonblocking()` returns whether or not the request can be satisfied. Decaf has no multi-threading capabilities so `getPort()` doesn't block either. When using components no longer need the ports that were provided to them, they signal this to the framework by calling `releasePort()`.

Back in Section 3.2.1, we listed the four steps needed to connect two components via the ports they expose. Now that we've covered the `gov.cca.Services` interface, we can rewrite that four step list with a bit more specificity.

1. A component that requires a port, registers that need to the framework via an invocation of `registerUsesPort()` on its `gov.cca.Services` object. This invocation specifies to the framework the type of port needed, and the string name that the component uses to uniquely identify this port.
2. A component that provides a port, notifies the framework that it can be used to provide that port via an invocation of `addProvidesPort()` on its `gov.cca.Services` object. This invocation specifies to the framework the type of port provided, the string name that this component uses to uniquely identify this port, and a reference to the live object itself.
3. The using component and providing component are connected by various means (which have yet to be defined). The framework guarantees that the port type requested and port type provided are compatible. The string port names used in step 1 and 2 are not involved in the connection mechanism. Details on how this is done won't be covered until Section 3.3.2.
4. The using component gains access to the live `gov.cca.Port` object by calling `getPort()` (or `getPortNonblocking()`) using the same string name it declared back in step 1.

3.2.5 `gov.cca.AbstractFramework`

The motivation behind the methods in `gov.cca.AbstractFramework` may be too subtle for the beginner. I will explain them here, but it is probably better to put a bookmark on this issue and wait til it gets revisited in discussing the implementation of Decaf, and some of the example codes (Section 5.2.3).

```
// in package gov.cca

interface AbstractFramework {

    TypeMap createTypeMap() throws CCAException;

    AbstractFramework createEmptyFramework()
        throws CCAException;

    Services getServices(in string selfInstanceName,
                         in string selfClassName,
```

```
                            in TypeMap selfProperties)
            throws CCAException ;

        void releaseServices(in Services svc)
            throws CCAException ;

        void shutdownFramework() throws CCAException;
    }
```

Most developers will only care about this interface if they are writing "`main()`" in their application. The first two methods of this interface simply create empty *gov.cca.TypeMap*s and *gov.cca.AbstractFramework*s. The next two allow the owners of main to create *gov.cca.Services* objects that are not associated with code that implements the *setServices* method.

3.2.6 *gov.cca.ComponentClassDescription*

```
// in package gov.cca {

interface ComponentClassDescription {
    string getComponentClassName() throws CCAException ;
}
```

This interface is specially created to handle when working with *gov.cca.AbstractFramework* directly. Remember that each component instance in a framework must have a unique name. This interface provides the ability to get the unique name based on the requested name in *gov.cca.AbstractFramework.getServices()*.

3.2.7 *gov.cca.ConnectionID*

```
// in package gov.cca

interface ConnectionID {

    ComponentID getProvider() throws CCAException ;

    ComponentID getUser() throws CCAException ;

    string getProviderPortName() throws CCAException ;

    string getUserPortName() throws CCAException ;

}
```

The *gov.cca.ConnectionID* interface is another read-only interface to provide access to the *gov.cca.ComponentID*s of both the user and provider in the connection. It also provides the respective port names of both the user and provider components.

3.2.8 *gov.cca.Type*

This enumeration lists all of the types supported in *gov.cca.TypeMap*, which follows. This means that one can't add user defined types, ports, or components to a *gov.cca.TypeMap*.

```
// in package gov.cca

enum Type {
    None, Int, Long, Float, Double, Fcomplex, Dcomplex,
    String, Bool, IntArray, LongArray, FloatArray,
    DoubleArray, FcomplexArray, DcomplexArray, StringArray,
    BoolArray
}
```

Advanced: Some versions of the CCA spec add explicit values to the members of the *gov.cca.Type* enumeration. Such a practice is prone to abuse. Babel will generate an include file so Fortran77 programmers don't need to know the real values of an enumerated type. There no need in a Babel-enabled CCA framework to ever hard-code enumerated values.

3.2.9 *gov.cca.TypeMap*

Conceptually, the *gov.cca.TypeMap* is just an interface to a typesafe hash table.

```
// in package gov.cca

interface TypeMap {

    // get single type
    int getInt(in string key, in int dflt)
        throws TypeMismatchException;
    long getLong(in string key, in long dflt)
        throws TypeMismatchException;
    float getFloat(in string key, in float dflt)
        throws TypeMismatchException;
    double getDouble(in string key, in double dflt)
        throws TypeMismatchException;
    fcomplex getFcomplex(in string key, in fcomplex dflt)
        throws TypeMismatchException;
    dcomplex getDcomplex(in string key, in dcomplex dflt)
        throws TypeMismatchException;
    string getString(in string key, in string dflt)
        throws TypeMismatchException;
```

```
bool getBool(in string key, in bool dflt)
    throws TypeMismatchException;

// get array of type
array<int> getIntArray(in string key,
                       in array<int> dflt)
    throws TypeMismatchException;
array<long> getLongArray(in string key,
                         in array<long> dflt)
    throws TypeMismatchException;
array<float> getFloatArray(in string key,
                           in array<float> dflt)
    throws TypeMismatchException;
array<double> getDoubleArray(in string key,
                             in array<double> dflt)
    throws TypeMismatchException;
array<fcomplex> getFcomplexArray(in string key,
                                 in array<fcomplex> dflt)
    throws TypeMismatchException;
array<dcomplex> getDcomplexArray(in string key,
                                 in array<dcomplex> dflt)
    throws TypeMismatchException;
array<string> getStringArray(in string key,
                             in array<string> dflt)
    throws TypeMismatchException;
array<bool> getBoolArray(in string key,
                         in array<bool> dflt)
    throws TypeMismatchException;

// put single type
void putInt(in string key, in int value);
void putLong(in string key, in long value);
void putFloat(in string key, in float value);
void putDouble(in string key, in double value);
void putFcomplex(in string key, in fcomplex value);
void putDcomplex(in string key, in dcomplex value);
void putString(in string key, in string value);
void putBool(in string key, in bool value);

// put array of types
void putIntArray(in string key, in array<int> value);
void putLongArray(in string key, in array<long> value);
void putFloatArray(in string key, in array<float> value);
void putDoubleArray(in string key, in array<double> value);
void putFcomplexArray(in string key, in array<fcomplex> value);
void putDcomplexArray(in string key, in array<dcomplex> value);
void putStringArray(in string key, in array<string> value);
```

```
    void putBoolArray(in string key, in array<bool> value);

    // others
    TypeMap cloneTypeMap();

    TypeMap cloneEmpty();

    void remove (in string key);

    array<string> getAllKeys(in Type t);

    bool hasKey(in string key);

    Type typeOf(in string key);

}
```

Most of the get and put methods are boilerplate, considering the types supported in `gov.cca.Type`. There are a few methods near the end that deserve special comment.

One of the unusual side-effects of writing language-independent software using Babel, is that developers soon learn to care about languages they would never before consider. Babel will catch reserved words in all of its supporting languages, but there are extra situations that are harder to catch. For example: arguments to the `getXXX()` methods name the default argument as `dflt` because `default` is a reserved word in C, C++, and Java. The Babel compiler will throw exceptions when using reserved words in one of its supported languages in a SIDL file. The more subtile problem that Babel won't catch is the following: The `gov.cca.TypeMap.cloneTypeMap()` method used to be just called `gov.cca.TypeMap.clone()`, but that collided with the Java have a built-in `clone()` method; generating a second one breaks things in interesting (and vendor specific) ways. Even though `clone` is not a reserved word in Java, using it as a method name has implications. For these kinds of issues, Babel doesn't provide a lot of guidance.

3.2.10 *gov.cca.CCAExceptionType*

This is an enumeration used in `gov.cca.CCAException`.

```
// in package gov.cca

enum CCAExceptionType {
    Unexpected = -1,
    Nonstandard = 1,
    PortNotDefined = 2,
    PortAlreadyDefined = 3,
    PortNotConnected = 4,
    PortNotInUse = 5,
    UsesPortNotReleased = 6,
    BadPortName = 7,
```

```
        BadPortType = 8,
        BadProperties = 9,
        BadPortInfo = 10,
        OutOfMemory = 11,
        NetworkError = 12,
    }
```

This enum has explicit values voted into the standard. Don't ask me why.

3.2.11 *gov.cca.CCAException*

Almost all methods in the CCA specification potentially throw a *gov.cca.CCAException*.

```
// in package gov.cca

abstract class CCAException extends SIDL.BaseException {
    abstract CCAException getCCAExceptionType();
}
```

It is interesting to note that *gov.cca.CCAException* is a SIDL abstract class and not an interface. This is due to a technical constraint in Babel rather than a design decision in the CCA. Babel requires that all exceptions inherit from the *SIDL.BaseException* class. The CCA specification intends this exception to be another read-only type, so its only method is declared *abstract*, meaning that there's no implementation of this method, only a declaration. By virtue of being a class and having an unimplemented method, *gov.cca.CCAException* must be an abstract class.

3.3 CCA Default Ports

In addition to the core spec, there are a few important ports that have made it through the CCA approval process. All of these SIDL types are in the *gov.cca.ports* package.

3.3.1 *gov.cca.ports.GoPort*

The first and simplest of all ports. This one also usually has a special relationship with the CCA frameworks that are actually implemented. The graphical CCAFE, for instance, renders a *gov.cca.ports.GoPort* as a green button. After hooking up all the ports using the graphical editor, it is this port which serves as the trigger to make things go. For example, this could make the simulator start calculating results.

```
// in package gov.cca.ports

interface GoPort extends Port {
    int go();
}
```

Note that `gov.cca.ports.GoPort` extends the `gov.cca.Port` interface. This is true of all ports in the CCA, whether part of the CCA specification, or application specific. The integer return value of `goPort.go()` is limited to the following: 0 means everything went well, -1 indicates that an internal error occurred, but the component can be used further, and -2 indicates an error so severe that the component cannot be safely used anymore.

3.3.2 *gov.cca.ports.BuilderService*

Now we get to the interesting ports. Until here, there's no specification in the CCA on how to actually create an instance of a component or to associate a uses port on one component to a provides port on another. The `gov.cca.ports.BuilderService` interface provides all of this capability and more.

```
// in package gov.cca.ports

interface BuilderService extends gov.cca.Port {

    gov.cca.ComponentID
    createInstance(in string instanceName,
                   in string className,
                   in gov.cca.TypeMap properties)
        throws gov.cca.CCAException ;

    gov.cca.ComponentID
    getDeserialization(in string s)
        throws gov.cca.CCAException ;

    gov.cca.ConnectionID
    connect(in gov.cca.ComponentID user,
            in string usingPortName,
            in gov.cca.ComponentID provider,
            in string providingPortName)
        throws gov.cca.CCAException ;

    void
    disconnect(in gov.cca.ConnectionID connID,
               in float timeout)
        throws gov.cca.CCAException ;

    void
    disconnectAll(in gov.cca.ComponentID id1,
                  in gov.cca.ComponentID id2,
                  in float timeout)
        throws gov.cca.CCAException ;

    void
    destroyInstance(in gov.cca.ComponentID toDie,
```

```
                in float timeout )
    throws gov.cca.CCAException ;

gov.cca.TypeMap
getComponentProperties(in gov.cca.ComponentID cid)
    throws gov.cca.CCAException ;

void
setComponentProperties(in gov.cca.ComponentID cid,
                       in gov.cca.TypeMap properties)
    throws gov.cca.CCAException ;

gov.cca.TypeMap
getPortProperties(in gov.cca.ComponentID cid,
                  in string portName)
    throws gov.cca.CCAException ;

void
setPortProperties(in gov.cca.ComponentID cid,
                  in string portName,
                  in gov.cca.TypeMap properties)
    throws gov.cca.CCAException ;

gov.cca.TypeMap
getConnectionProperties(in gov.cca.ConnectionID connID)
    throws gov.cca.CCAException ;

void
setConnectionProperties(in gov.cca.ConnectionID connID,
                        in gov.cca.TypeMap properties)
    throws gov.cca.CCAException ;

gov.cca.ComponentID
getComponentID(in string componentInstanceName)
    throws gov.cca.CCAException ;

array<gov.cca.ComponentID>
getComponentIDs()
    throws gov.cca.CCAException ;

array<string>
getProvidedPortNames(in gov.cca.ComponentID cid)
    throws gov.cca.CCAException ;

array<string>
getUsedPortNames(in gov.cca.ComponentID cid)
    throws gov.cca.CCAException ;
```

```
    array<gov.cca.ConnectionID>
    getConnectionIDs(in array<gov.cca.ComponentID> componentList)
        throws gov.cca.CCAException ;

}
```

Weighing in at 17 methods, this interface is one of the more complicated ones in the CCA specification (second in size only to *gov.cca.TypeMap* which has lots of boilerplate methods). Taken in small chunks, everything here is very straightforward to explain. The first six methods are the most important to understand, they deal with creating, connecting, disconnecting, and destroying components. The next six are get/set methods for component, port, and connection properties. The last five are all get methods to replicate internal data of the framework.

Component create/destroy, connect/disconnect

Components are created by *BuilderService* by only two methods: *createInstance()* and *getDeserialization()*. The first takes three arguments: a string *instanceName* (which may or may not be reflected in the resulting *gov.cca.ComponentID*, a string representation of the component's type (unfortuately called *className*), and a *gov.cca.TypeMap* of additional properties to associate with the created instance. The second, takes only a string representation of the class instance. This string should be identical to one returned by *gov.cca.-ComponentID.getSerialization()* previously.

In processing this request to create an instance, the underlying framework will invariably also create a *gov.cca.Services* instance and call *setServices()* on the new component. This is the typical opportunity for a component to register what ports it uses and provides.

The only restriction on port names are that they must be unique to the component instance associated with them, uses and provides ports are connected by the *connect* method. Notice that the inputs in the *BuilderService.connect()* method are exactly the same data that can be obtained by the read-only interface *gov.cca.ConnectionID* that is returned.

WARNING:

It is a common mistake to assume that uses and provides ports are connected automatically based on string matching. This is not correct. The types of the ports must match, but the names need not. The port names are strictly for distinguishing different ports within the context of a single component instance.

Once connected, the using component is free to call *gov.cca.Services.getPort()* (or *gov.cca.Services.getPortNonblocking()*) and use the port that has been provided by it. After it is complete, some cleanup may be required.

Connections can be broken individually by the *BuilderService.disconnect()* call, or wholesale between two components with *BuilderService.disconnectAll()*. Component instances themselves can be destroyed using the *BuilderService.destroyInstance()* method. All of these destruction methods take a *timeout* argument which, in threaded frameworks, should return within that number of seconds, regardless.

get/set properties of components, ports, and connections

The CCA specification includes `gov.cca.TypeMap`s being associated with every component instance, every port on each component, and each connection between two ports. These are intended to communicate miscellaneous properites and meta-data. There is a method in `BuilderServices` suitable to get and set properties in all three cases. To explicitly change an attribute in the properties `TypeMap`, it is not sufficient to simply get the properties object and change it in place. All these `get`***XXX***`Properties()` methods return a clone of the internal `TypeMap`. One would have to modify the important entries and then invoke the appropriate `set`***XXX***`Properties()` to commit the change.

Get info stored in framework

These final five methods in `gov.cca.ports.BuilderServices` are included to explore the information known inside the framework. `BuilderServices.getComponentID()` returns a `gov.cca.ComponentID` based on the unique name assigned to the component instance. `BuilderServices.getComponentIDs()` returns a 1-dimensional array of all components currently instantiated in the framework.

Given any `gov.cca.ComponentID`, one can use `BuilderServices` to query all the registered using ports associated with that component using `BuilderServices.getUsedPortNames()`. Similarly, one can get a 1-dimensional array of all the port names registered with the component as providing ports using `BuilderServices.getProvidedPortNames()`. Finally, a 1-dimensional array of all connections associated with a `gov.cca.ComponentID` is available by calling `BuilderServices.getConnectionIDs()`.

3.3.3 `gov.cca.ports.ConnectionEvent`

Sometimes, a CCA component may not register a provides port until all of its using ports have been satisfied. To accomplish this, the component needs to be savvy enough to detect when its uses port requests are satisfied, and when those ports have been rescinded.

This is done using `gov.cca.ports.ConnectionEvent`s. The next few types implement a simple publish/subscribe model for handling connect/disconnect events.

```
// in package gov.cca.ports

enum EventType {
    Error = -1,
    ALL = 0,
    ConnectPending = 1,
    Connected = 2,
    DisconnectPending = 3,
    Disconnected = 4,
}

interface ConnectionEvent {
```

```
    EventType getEventType();

    gov.cca.TypeMap getPortInfo();
}
```

Above is the simple *gov.cca.ports.ConnectionEvent* interface which is another read-only interface. It has an associated *gov.cca.ports.EventType* enumerated type and can be used to get the properties associated with the port.

3.3.4 *gov.cca.ports.ConnectionEventListener*

For a component to receive *gov.cca.ports.ConnectionEvents*, it must itself implement the *gov.cca.ports.ConnectionEventListener* interface, and subscribe itself using the *gov.cca.ports.ConnectionEventService* port. The interface the component must implement is pretty minimal.

```
// in package gov.cca.ports

interface ConnectionEventListener {
    void connectionActivity(in ConnectionEvent ce);
}
```

3.3.5 *gov.cca.ports.ConnectionEventService*

The *gov.cca.ports.ConnectionEventService* is a port like any other. That means that before a component can receive *ConnectionEvents*, it must register that it uses *gov.cca.ports.ConnectionEventService*, the request must be satisfied, and the component call *Services.getPort()* to get access to the *ConnectionEventService*.

```
// in package gov.cca.ports

interface ConnectionEventService extends gov.cca.Port {

    void
    addConnectionEventListener(in EventType et,
                               in ConnectionEventListener cel);

    void
    removeConnectionEventListener(in EventType et,
                                  in ConnectionEventListener cel);
}
```

Once a component has a live connection to the *ConnectionEventService*, it is free to add and remove *ConnectionEventListeners*. The CCA spec doesn't associate any naming strategy with *ConnectionEventListeners*, so its unlikely that the same listener can be added to the same even multiple times.

3.3.6 *gov.cca.ports.ComponentRepository*

```
// in package gov.cca.ports

interface ComponentRepository  extends gov.cca.Port {

    array<gov.cca.ComponentClassDescription>
    getAvailableComponentClasses()
        throws gov.cca.CCAException ;

}
```

The goal of this port is to return a list of CCA component types available to instantiate.

3.4 Summary

3.4.1 Reiterating the Levels of Understanding

Conceptual Understanding: The goal of the CCA is to increase programmer performance not necessarily program performance. The CCA strategy is to promote code reuse through a loosely coupled system of software.

The key abstraction in the CCA model is the use of "ports" (roughly equivalent to SIDL interfaces), which are both used and provided by CCA components. The CCA model also prescribes the `gov.cca.Services` interface which is (initially) the component's only view of the outside world.

Beginner Level: For a component instance to use a port, it must register its need through its `gov.cca.Services` object, another component must similarly post a provides port of a compatible type, somehow the framework is driven to connect the uses port with the provides port, and then the using component must call some version of `gov.cca.Services.getPort()` to gain access to the live port on the other side of the connection.

There is a special CCA port called the `GoPort` that eventually gets triggered once everything is assembled. Once the `GoPort` is triggered, the components act out in their prescribed manner without intervention of the CCA Framework. The CCA Framework regains control either if the components signal the Framework internally, or when the `GoPort.go()` call returns.

Intermediate Level: Almost every component needs to have access to its `gov.cca.Services` instance throughout its life-span: cache a copy of it in your `gov.cca.Component.setServices()` implementation.

There is currently no agreement on whether or not a provides port can be simultaneously connected to multiple using ports. (Decaf's internals only allow point-to-point connections, not one to all. Personally, I was floored that this was even an option.)

Advanced Details: Be wary about threading and serialization features. Many implementations of the CCA (including Decaf) are planning to implement these things later.

Expert Details: I said there wouldn't be any Level 5 information presented here. What'd you expect?

3.4.2 Personal Opinion

Having implemented the last two versions of the CCA standard, I've developed a few personal opinions of what I like and don't like. I tread carefully because many of my friends and collaborators have put serious time into this standard. At the same time, however, a little background and personal preference can provide important insight to the reader who would otherwise be unaware of why some things are the way they are.

Decaf being the largest piece of software I've written with Babel to date, I'll also include criticsm about my own project to be fair. Actually, since there's just SIDL here and no real implementation, I'll have criticms for the SIDL grammar here, and then more about Babel when I get to Decaf in the next chapter.

What I'm fond of:

1. CCA spec being mostly interfaces. As more frameworks become Babel enabled, this should help components be compatible with different frameworks without recompiling[6]
2. CCA's use of read-only interfaces.

What other people seem to like:

1. Frameworks with GUI builders. Let's face it, the whole register user, register provider, connect, get port model is cumbersome for simple scripting applications. However, this model makes perfect sense for a GUI tool where you instantiate components by dragging instances onto a canvas from a palette and draw the connections using a mouse. Its surprizing (to me) how much the GUI tools are used... and not just for toy applications.

What I'm not so fond of:

1. CCA specification not being all interfaces. Basically, its polluted by a few enumerated types and abstract classes for exceptions.
   (a) `gov.cca.TypeMap` or something like it probably should be in a SIDL standard library. Babel's current runtime library has just enough to support its object model and dynamically loading libraries.
   (b) `gov.cca.CCAExceptionType`. The whole idea behind exceptions as classes is that they can be extended as need arizes. By requiring all methods to throw a single exception class (namely `gov.cca.CCAException` which then users query for the value of an enumerated type, defeats the whole extensiblity thing.
2. Lack of locality. For instance to serialize of an object, the `getSerialization()` method is bound to the `gov.cca.ComponentID` interface. The matching `getDeserialization()` method is bound to `gov.cca.ports.BuilderService`.

[6]Assuming the same platform, of course.

3. *gov.cca.ComponentClassName*. I would rather it be called *gov.cca.ComponentType*. Names are for arbitrary strings, types are not as flexible.
4. Explicit values assigned to enums. (Hey, I said this was my opinion. You're free to disagree.)
5. Not all ports are equal. A CCA framework doesn't work at all without *GoPort* and *BuilderService*. Without these two ports, there's no way to instantiate, connect, activate, disconnect, or destroy component instances and their ports. (It turns out that *ConnectionEventService* tends to have a rather intimate relationship to the framework too.) It wasn't clear to me when I was learning all this just how important these few ports are, and that the so-called "core" CCA spec doesn't provide enough standards to make anything useful.
6. Not enough access to *BuilderService* and *ComponentRepository*. For instance, there's no way to specify where these things should look (e.g. *setComponentSearchPath*) to find components to instantiate.
7. *ConnectionEventService*. Removing the listener based on its instance instead of some name is inconsistent with the flavor of the rest of the CCA.

Chapter 4

Decaf Implementation

When I'm working on a problem, I never think about beauty.
I think only how to solve the problem. But when I have finished,
if the solution is not beautiful, I know it is wrong.
— R. Buckminster Fuller (1895 - 1983)

Contents

4.1 Overview

The primary purpose of this chapter is to provide insight into the CCA by discussing the Decaf implemenation. The secondary purpose is to provide an example of using Babel on a non-trivial piece of code. If the reader's goal is only to use Decaf (or another Babel-enabled CCA framework, when they come online), then this chapter is not required and the reader can safely skip to Chapter 5.

Given the CCA specification, the next logical step to build the framework is to actually implement the interfaces. Since the CCA specification is expressed in SIDL, the framework can express its implementation as SIDL classes that implement the appropriate interfaces.

The framework need not implement every interface in the CCA Spec. `gov.cca.Component` and `gov.cca.ports.ConnectionEventListener` are two interfaces in particular that are intended for the component developer to implement. SIDL enumerated types are complete as is. In addition to the remaining interfaces, there are two CCA exceptions which are abstract classes and need to be finished up.

We haven't discussed versioning in SIDL yet. This is an often misunderstood feature of Babel, owing in part to the fact that early releases of Babel simply said "add more here" to the relevant section. This is how the Decaf SIDL file begins.

```
require gov.cca version 0.6.1;

package decaf version 0.6.1 {

  // Decaf classes go here

}
```

The require statement specifies the version of the CCA spec. Unlike `import`, it does not put all the types in the current scope, hence they are still referenced by fully qualified names. The package statement opens up a new scope where all the decaf classes go.

4.2 Low Hanging Fruit

Several of the CCA interfaces can be implemented as is, their definition in Decaf's SIDL file is trivial.

4.2.1 *decaf.TypeMap*

```
// in package decaf

class TypeMap implements-all gov.cca.TypeMap { }
```

4.2.2 *decaf.ComponentClassDescription*

```
// in package decaf

class ComponentClassDescription
      implements-all gov.cca.ComponentClassDescription { }
```

4.2.3 *decaf.ports.ComponentRepository*

This is not implemented in the current release of Decaf.

```
// in package decaf.ports

class ComponentRepository
      implements-all gov.cca.ports.ComponentRepository { }
```

4.3 Implementing the Read-Only Interfaces

The read-only interfaces need two things: a class to implement them and an additional method to actually set the internal state. This is easily handled by Babel.

4.3.1 *decaf.ComponentID*

```
// in package decaf

class ComponentID implements-all gov.cca.ComponentID {
    void initialize( in string name );
}
```

Since `decaf.ComponentID` is a concrete class, it must implement all unimplemented methods it inherits. Furthermore, it can override any number of non-`final` methods it inherits. The usual method for indicating which methods are overridden in a class is to explicitly list all the methods in the class definition. The `implements-all` SIDL keyword serves two purposes: first it means interface inheritance just like the SIDL `implements` keyword, additionally it indicates that the class will implement all of the methods in the interface, so there's no need to write out each inherited method in the interface.

This paragraph is a self-check to help understand the details. If you run Babel to generate the code, and wanted to implement the `decaf.ComponentID` class yourself, how many methods would be generated for you to add an implementation to? If your answer is "one," you probably failed to understand the difference between the SIDL `implements` and `implements-all` keywords, reread the paragraph above (and pull out the Babel User's Guide if necessary). If you had to check back to Section 3.2.3 to count its two inherited methods, added the one declared here, and answered "three," you're on the right track. If you immediately included the `_ctor` and `_dtor` methods (Babel's name for user implementations of constructors and destructors) and said "five," you get extra credit.

Implementation of this class is straightforward, its state consists of a single string. Since Babel objects are created with no arguments, it is necessary to explicitly have a method that set's the string. In Decaf, the *getInstanceName* and *getSerialization* methods return the same string. Remember, that Decaf does not implement component serialization at this time. Babel has plans to implement RMI in the future and such a mechanism necessarily includes serialization.

4.3.2 *decaf.ConnectionID*

```
// in package decaf

class ConnectionID implements-all gov.cca.ConnectionID {

    void initialize( in gov.cca.ComponentID provider,
                     in string providerPortName,
                     in gov.cca.ComponentID user,
                     in string userPortName,
                     in gov.cca.TypeMap properties );

    void setProperties( in gov.cca.TypeMap properties );

    gov.cca.TypeMap getProperties();

}
```

4.3.3 *decaf.ports.ConnectionEvent*

```
// in package decaf


class ConnectionEvent implements-all gov.cca.ports.ConnectionEvent {
    void initialize( in gov.cca.ports.EventType eventType,
                     in gov.cca.TypeMap portProperties );
}
```

This shows a very useful paradigm that was used repeatedly in Decaf. Whenever something non-standard is done, the object in question must be downcast from a CCA object to a Decaf object.

4.4 Implementing CCA Exceptions

4.4.1 *decaf.CCAException*

```
// in package decaf

class CCAException extends gov.cca.CCAException {
    gov.cca.CCAExceptionType getCCAExceptionType();
    void setCCAExceptionType( in gov.cca.CCAExceptionType et );
}
```

4.4.2 *decaf.TypeMismatchException*

```
// in package decaf

class TypeMismatchException extends gov.cca.TypeMismatchException {

    void initialize( in gov.cca.Type requestedType,
                     in gov.cca.Type actualType );

    gov.cca.CCAExceptionType getCCAExceptionType();

    gov.cca.Type getRequestedType();

    gov.cca.Type getActualType();

}
```

4.5 Hard Part

Four interfaces remain: `AbstractFramework`, `BuilderServices`, `Services`, and `ConnectionEventService`. The first two are so tightly coupled, it made sense to actually implement them in a single object: `decaf.Framework`. Similarly, the latter two interfaces are implemented by `decaf.Services`. This section describes the details of how/why this was done this way.

4.5.1 *decaf.Framework*

```
// in package decaf

class Framework implements-all gov.cca.AbstractFramework,
                               gov.cca.ports.BuilderService
{

    gov.cca.Port lookupPort(in gov.cca.ComponentID componentID,
                            in string portName);

    void provideRequestedServices( in gov.cca.ComponentID componentID,
                                   in string portName,
                                   in string type );
```

```
}
```

In addition to implementing all of the methods in *gov.cca.AbstractFramework* and *gov.cca.ports.BuilderService*, I've added two additional methods. After including the *_ctor* and *_dtor* methods, that adds up to 26 methods that are implemented by this one class. We'll discuss the reasons for these two additional methods here, and postpone the sticky implementation details after introducing *decaf.Services*.

The *lookupPort()* method returns the providing *gov.cca.Port* associated with a particular *gov.cca.ComponentID* and *portName*. This method will return provides ports regardless of whether they're connected or not. This can be useful from `main()` when you need to get a *goPort* out of the component, and don't want to go through the contortions of creating an explicit *gov.cca.Services* object, registering your need for a *goPort* and then using *gov.cca.BuilderServices* to connect the ports. (Skip ahead to the code samples in Section 5.2.3 to see what I mean.)

The *provideRequestedServices()* method is intended for the *decaf.Services* object to invoke when its components register a need for either *gov.cca.ports.BuilderServices* or *gov.cca.ports.ConnectionEventService*. Since these two services are provided directly by Decaf, *registerUsesPort()* calls on these two types are satisfied immediately by the framework, which is notified by this call. More details after we discuss the last class in the Decaf SIDL file.

4.5.2 *decaf.Services*

```
// in package decaf

class Services implements-all gov.cca.Services,
                              gov.cca.ports.ConnectionEventService
{

    void initialize( in gov.cca.AbstractFramework fwk,
                     in gov.cca.ComponentID componentID,
                     in gov.cca.TypeMap properties );

    gov.cca.TypeMap getInstanceProperties();

    void setInstanceProperties( in gov.cca.TypeMap properties );

    void setPortProperties( in string portName,
                            in gov.cca.TypeMap properties );

    array<string> getProvidedPortNames();

    array<string> getUsedPortNames();

    void bindPort( in string portName, in gov.cca.Port port );
```

```
        gov.cca.Port getProvidesPort( in string name );

        void notifyConnectionEvent( in string portName,
                                    in gov.cca.ports.EventType event );
    }
```

In addition to implementing all of the methods in `gov.cca.Services` and `gov.cca.ports.ConnectionEventService`, there are 9 additional methods added to make it all work. This class has a total of 22 methods to implement including `_ctor` and `_dtor`.

The `initialize()` method should be straightforward. A `decaf.Services` object is created with a reference to the framework that created it, the component instance associated with it, and the properties associated with that component instance.

Accessor methods to get and set properties associated with either the entire component instance, or ports associated with that instance are `getInstanceProperties()`, `setInstanceProperties()`, and `setPortProperties()` (to complement the `getPortProperties()` method in the `gov.cca.Services` interface). These methods are exposed in the standard as being associated with `gov.cca.ports.BuilderService`. Since Decaf keeps the data in question associated with the `decaf.Services` object, and since `decaf.Framework` implements the `gov.cca.ports.BuilderService` interface, corresponding calls through `gov.cca.ports.BuilderServices` to get and set component and port properties are simply delegated by the framework to the responsible `decaf.Services` instance. This same arrangement holds true for `getProvidedPortNames()` and `getUsedPortNames()`.

The `bindPort()` method is invoked internally by `decaf.Framework.connect()` to connect the `portName` on the services object of the component using the port, to the `gov.cca.Port` instance providing the port.

The `getProvidesPort()` method is invoked internally by `decaf.Framework.lookupPort()` to satisfy requests for a named port.

Finally, the `notifyConnectionEvent()` method is called by `decaf.Framework` before and after each connection is made and broken between two components.

4.6 Implementation Details

This section contains advanced details about how Decaf is implemented. It also provides an example of Babel being used in a non-trivial application. Readers can skip ahead to Section 4.7 if disinterested in both of these issues.

Since `decaf.Framework` and `decaf.Services` are really the two most interesting objects, discussion of implementation details here will be restricted to these two classes. This is *not* a prescription for how to properly implement a CCA framework. The only such prescription is to use the SIDL interfaces in the previous chapter. How a CCA framework is implemented is entirely up to the implementor. I am discussing my implementation of Decaf for educational purposes, and reserve the freedom to change it at a later date as the CCA specification — or my understanding of it — evolves.

4.6.1 Internal Data Structures

Decaf is implemented in C++ and uses the C++ standard library, including the parts that were once known separately as the Standard Template Library (STL).

`decaf::Framework` The `decaf::Framework` class maintains three `std::maps`.

The first, `d_instance`, is a map from `std::string` unique instance names to a `std::pair` of `cca::Component/cca::Services` tuples. This contains *bona fide* instances as well as alaises, although aliases have a null `cca::Component`. "Aliases" in this discussion are created by `decaf::Framework::getServices()` method, which creates Services objects that are not associated with a particular component instance.

The second map, `d_connection`, is actually a map of maps, or conceptually a 2-D map. It stores the `cca::ConnectionID` based on the `std::string` unique component instance name and the portName.

Lastly, `d_aliases` is a map of instance names to purported class names. This map is only accessed and modified by `decaf::Framework_impl::getServices()`. This association is currently not used by anything inside of Decaf, but it is recorded in case a need is uncovered later (and to use all the arguments in the method signature so compilers don't complain.)

`decaf::Services` This class keeps around a bit of information internally. First it keeps references to all the information passed to it in its initialize function: `cca::AbstractFramework` — the framework that it belongs to, `cca::ComponentID` — the component instance associated with it, and `cca::TypeMap` — the properties associated with that component. Additionally, it keeps `std::maps` from `std::string` port names to a `std::pair` of `cca::Port` ports and `cca:TypeMap` port properties. One such map for uses ports and another for provides ports.

To satisfy all the needs of the *`gov.cca.ports.ConnectionEventService`* that this class implements, there's an additional `std::map` (called `d_listeners`) which given a `cca::ports::EventType`, returns a `std::list` of interested `cca::ports::ConnectionEventListeners`.

4.6.2 `decaf::Framework::connect()`

The following is an exact copy of how Decaf implements its connect method. This is one of the more complicated details for the advanced user. When connect is called, the uses port should already be registered and the provides port added, and that both of the ports must have identical types (based on string matching).

```
gov::cca::ConnectionID
decaf::Framework_impl::connect (
  /*in*/ gov::cca::ComponentID user,
  /*in*/ std::string usingPortName,
  /*in*/ gov::cca::ComponentID provider,
  /*in*/ std::string providingPortName )
throw (
  gov::cca::CCAException
){
```

```
      // DO-NOT-DELETE splicer.begin(decaf.Framework.connect)
  1.  decaf::ConnectionID connectID;
  2.  std::string userName = user.getInstanceName();
  3.  std::string provName = provider.getInstanceName();
  4.  if ( ( d_instance.find( userName ) != d_instance.end() ) &&
       ( d_instance.find( provName ) != d_instance.end() ) ) {
  5.    decaf::Services userSvc = d_instance[ userName ].second;
  6.    decaf::Services provSvc = d_instance[ provName ].second;
  7.    provSvc.notifyConnectionEvent( providingPortName,
                                       gov::cca::ports::ConnectPending );
  8.    userSvc.notifyConnectionEvent( providingPortName,
                                       gov::cca::ports::ConnectPending );
  9.    gov::cca::Port port = provSvc.getProvidesPort(providingPortName);
 10.    if ( port._not_nil() ) {
 11.  userSvc.bindPort( usingPortName, port );
 12.  connectID = decaf::ConnectionID::_create();
 13.  connectID.initialize( user, usingPortName,
                            provider, providingPortName, 0);
 14.  d_connection[ userName ][ usingPortName ] = connectID;
 15.  provSvc.notifyConnectionEvent( providingPortName,
                                       gov::cca::ports::Connected );
 16.  userSvc.notifyConnectionEvent( providingPortName,
                                       gov::cca::ports::Connected );
        }
      }
 17.  return connectID;
      // DO-NOT-DELETE splicer.end(decaf.Framework.connect)
    }
```

This code snippet includes the so-called *splicer blocks* where developers are free to insert their own implementation. The bounds of the splicer blocks include the comments with the prominent `DO-NOT-DELETE` warnings. We'll march through the body of the code, line by line.

1. Create a nil `ConnectionID` reference. This will be properly initiallized if everything works and returned as an empty object if the arguments are wrong.
2. Get the unique name for the instance of the Component that will have its uses port request satisfied.
3. Get the unique name for the instance of the Component that will provide the port needed elsewhere.
4. If both the using Component and the providing component are found in the framework, continue.
5. Get the services object associated with using component. Note that this is also being downcast from a `cca::Services` object to a `decaf::Services` object to give us access to Decaf specific methods.
6. Ditto with the services object associated with the providing component.

7. Notify the pending connection to the provider component first.
8. Notify the pending connection to the user component.
9. Extract the actual provides port from the providing component's name.
10. If the reference to the providing port is not nil, continue
11. Bind the providing port to the using port name on the user services object.
12. Create a full-fledged connection ID instance.
13. Initialize the connection ID instance.
14. Add the connection ID to the framework's `d_connect` map.
15. Notify the completed connection to the provider component first.
16. Notify the completed connection to the user component next.
17. Return the connection ID.

Note that error handling is minimal in this code. Decaf is intended to be an example code, not production. Babel's business is the language interoperability layer it provides to all CCA frameworks, and not the business of building and maintaining CCA frameworks.

4.7 Summary

4.7.1 Broken into Levels of Understanding

Conceptual Understanding. Decaf is an implementation of the CCA specification, and the first implementation using Babel. It was intended as a proof-of-concept and example to other CCA framework developers. It has since grown to be a fully compliant CCA framework (albeit without a GUI).

Beginner Level. CCA components that can be loaded by Decaf are binary compatible with CCAFFEINE, the CCA framework from Sandia National Labs. Other frameworks such as XCAT and SciRUN have yet to fully integrate Babel technology.

Average Level. Although Babel only has explicit support for public methods — all methods in SIDL are public — there are techniques for achieving the equivalent of private methods, and package access only methods. For private methods, simply add the methods in the implementation and don't even mention them in the SIDL. Package level access is achieved by putting the interfaces in one package (namely *`package gov.cca`*) and putting the classes in another package (*`decaf`*). The Decaf implementation adds a handful of new methods to these classes that aren't prescribed by the CCA interfaces. To gain access to these decaf-specific methods, the objects must be downcast from their interfaces in the *`gov.cca`* package to the concrete classes in the *`decaf`* package.

This strategy makes it very clear when Decaf is adhering to the standard, and when it is doing its implementation-specific/non-standard business. It should be noted that the existence of these additional methods does not imply that Decaf *breaks* the CCA standard. Components can be written having access only to the CCA SIDL file and subsequently used by the Decaf implementation. The existance of read-only objects in the CCA shows that it was expected for framework implementations to do these kinds of things, since the only way to get information into the object is to do something not prescribed by the standard.

Advanced Details. CCAFFEINE is a much larger project that also has a "classic," C++-only way of specifying components that predates its Babelization. CCAFFEINE can connect Babelized components with its own "classic" kind. Decaf has no support for CCAFFEINE's classic components. There are a few outstanding technical warts that make mixing components a little tricky for a novice, but these should settle down by the time of this publication.

4.7.2 Personal Opinion

What I'm fond of:

1. The Decaf implementation in general. Although seasoned C++ programmers may initially find Babel a bit restrictive (like I did at first), a little experience has convinced me that its also less error prone. The smart-pointer classes make all Babel objects fully reference counted, which is a pleasure to work with. Casting up and down is done with simple assignment.

What other people seem to like: No data here. Not many people have seen Decaf yet.

What I'm not so fond of:

1. Weakened typesafety. Babel's smart-pointer classes do not mirror the inheritance heirarchy in the SIDL file: they only emulate it. This has an important side effect that if you pass the wrong type as an argument, these classes will attempt to cast, the cast will fail, and the implementation gets the argument as a null pointer. As the implementor of Babel's C++ bindings, this was a disheartening side-effect. There has been some suggestions to adopt BOOST's smart-pointer classes, but others refuse to introduce a dependency on `pthread.h` that that implementation apparently has.

Chapter 5

Example Components

Few things are harder to put up with
than the annoyance of a good example.
— Mark Twain (1835 - 1910)

Example is not the main thing in influencing others.
It is the only thing.
— Albert Schweitzer

Contents

5.1 `strop.sidl`: three string manipulation ports

This is the silly string operation port standard that I just threw together. I'll use it for all the examples below.

```
package strop version 0.6 {

    interface StringProducerPort extends gov.cca.Port {
        string get();
    }

    interface StringDisplayPort extends gov.cca.Port {
        void display( in string msg );
    }

    interface StringTransformPort extends gov.cca.Port {
        string transform( in string msg );
    }

}
```

Note two things. First, the version of `strop.sidl` is different than the CCA spec. For such a simple example as this, one would expect it to not be modified as often. Second there are no *`using`* or *`require`* statements regarding *`gov.cca`*. The compiler first finds out about this when it sees interfaces extending *`gov.cca.Port`* and simply tries to find a symbol to resolve. Since no version is specified, it will try to find the most recent version available.

5.2 Hello World

There are lots of ways to write the famous “Hello World!” program. Using CCA components, is not the recommended one. Its kindof like swatting a housefly with heavy artillery. It will work, but you’re working to hard for such a trivial application.

That said, hello world does serve the purpose of doing something trivial so you can see the complexity of using the framework. So on we go with the hello world example. . .

5.2.1 Hello World Component (in F77)

If you have a fresh Babel tarball and successfully configured and built it, look at `babel-x.x.x/-examples/cca/hello-server/hello-server.sidl`. It should look like this:

```
package HelloServer version 0.6 {
  class Component implements-all strop.StringProducerPort,
                                 cca.Component {}
}
```

We’ve now described a CCA component. It implements the *`cca.Component`* interface and even implements a port. There are two methods of interest that this thing must implement: *`set-Services()`* and *`get()`* and here’s the relevant Fortran77 code that implements this.

```
            subroutine HelloServer_Component_setServices_fi(self, services)
            implicit none
            integer*8 self
            integer*8 services

C           DO-NOT-DELETE splicer.begin(HelloServer.Component.setServices)
            integer*8 port
            integer*8 properties
            integer*8 exception

            call gov_cca_Port__cast_f(self, port)
            call gov_cca_Services_createTypeMap_f( services,
     &         properties,
     &         exception )
            call catch( exception )
            call gov_cca_Services_addProvidesPort_f(
     &         services,
     &         port,
     &         'HelloServer',
     &         'strop.StringProducerPort',
     &         properties,
     &         exception)
            call catch( exception )
            call gov_cca_TypeMap_deleteRef_f(properties)
C           DO-NOT-DELETE splicer.end(HelloServer.Component.setServices)
            end


            subroutine HelloServer_Component_get_fi(self, retval)
            implicit none
            integer*8 self
            character*(*) retval

C           DO-NOT-DELETE splicer.begin(HelloServer.Component.get)
            retval = 'Hello World!'
C           DO-NOT-DELETE splicer.end(HelloServer.Component.get)
            end
```

Earlier drafts looked different than whats in the sample above. Babel 0.8 changed the way it generated implementation subroutine names, so the symbols look different. Babel 0.8.2 also changed the casting in Fortran 77 to be more consistent with the C bindings. Both changes are relatively minor to the developer.

There's really not any state associated with this implementation. Since F77 doesn't have any equivalent to a C struct, one would expect that adding state to a F77 component might be tricky. In fact, there's a nifty trick to implementing state using SIDL arrays which I'll mention in advanced topics at the end of this chapter.

5.2.2 Printf component (in C)

What I'm calling "the Printf component" here, is actually called `hello-client` in the Babel directory for historical reasons. It's function has always been the same; print a string and implement *GoPort*. Here's the SIDL:

```
package HelloClient version 0.5 {

  class Component implements cca.ports.GoPort, cca.Component {

    int go();

    void setServices(in cca.Services services);
  }

}
```

In this example, I used the *implements* keyword instead of *implements-all* in the previous section. This means that I have to list the methods explicitly that I intend to override. There's no reason for doing it differently other than to make readers aware that this is a perfectly valid alternative.

The implementation of this component is pretty trivial in C. In the "Impl" header, we augment the *_data* struct to have a reference to the *cca.Services* object.

```
/*
 * Private data for class HelloClient.Component
 */

struct HelloClient_Component__data {
  /* DO-NOT-DELETE splicer.begin(HelloClient.Component._data) */
  gov_cca_Services services;
  /* DO-NOT-DELETE splicer.end(HelloClient.Component._data) */
};
```

In the "Impl" source, we add some things to the *_includes* splicer block, do some Babel/C specific boilerplate to implement the *_ctor()* and *_dtor* methods, then finally the *go()* and *setServices()* methods inherited from the *cca.ports.GoPort* and *cca.Component* interfaces respectively. In this example, I've included the entire source file, without any editing (except a few linebreaks to fit within margins). It is just under 140 lines, including comments.

```
/*
 * File:          HelloClient_Component_Impl.c
 * Symbol:        HelloClient.Component-v0.5
 * Symbol Type:   class
 * Babel Version: 0.8.2
 * Description:   Server-side implementation for HelloClient.Component
 *
 * WARNING: Automatically generated; only changes within splicers preserved
```

```
 *
 * babel-version = 0.8.2
 */

/*
 * DEVELOPERS ARE EXPECTED TO PROVIDE IMPLEMENTATIONS
 * FOR THE FOLLOWING METHODS BETWEEN SPLICER PAIRS.
 */

/*
 * Symbol "HelloClient.Component" (version 0.5)
 *
 * The component uses the hello port and provides a go port.
 */

#include "HelloClient_Component_Impl.h"

/* DO-NOT-DELETE splicer.begin(HelloClient.Component._includes) */
#include <stdio.h>
#include "strop_StringProducerPort.h"
#include "SIDL_String.h"
#include "SIDL_string_IOR.h"
/* DO-NOT-DELETE splicer.end(HelloClient.Component._includes) */

/*
 * Class constructor called when the class is created.
 */

#undef __FUNC__
#define __FUNC__ "impl_HelloClient_Component__ctor"

void
impl_HelloClient_Component__ctor(
  HelloClient_Component self)
{
  /* DO-NOT-DELETE splicer.begin(HelloClient.Component._ctor) */
  struct HelloClient_Component__data* data =
    (struct HelloClient_Component__data*) malloc(
      sizeof(struct HelloClient_Component__data));

  data->services = NULL;

  HelloClient_Component__set_data(self, data);
  /* DO-NOT-DELETE splicer.end(HelloClient.Component._ctor) */
}

/*
```

```
 * Class destructor called when the class is deleted.
 */

#undef __FUNC__
#define __FUNC__ "impl_HelloClient_Component__dtor"

void
impl_HelloClient_Component__dtor(
  HelloClient_Component self)
{
  /* DO-NOT-DELETE splicer.begin(HelloClient.Component._dtor) */
  struct HelloClient_Component__data* data =
    HelloClient_Component__get_data(self);

  if (data->services != NULL) {
    gov_cca_Services_deleteRef(data->services);
  }

  free((void*) data);
  HelloClient_Component__set_data(self, NULL);
  /* DO-NOT-DELETE splicer.end(HelloClient.Component._dtor) */
}

/*
 * The following method starts the component.
 */

#undef __FUNC__
#define __FUNC__ "impl_HelloClient_Component_go"

int32_t
impl_HelloClient_Component_go(
  HelloClient_Component self)
{
  /* DO-NOT-DELETE splicer.begin(HelloClient.Component.go) */
  struct HelloClient_Component__data* data =
    HelloClient_Component__get_data(self);
  struct gov_cca_CCAException__object *err = 0;

  gov_cca_Port port =
    gov_cca_Services_getPort(data->services, "HelloServer", &err);
  strop_StringProducerPort hello = strop_StringProducerPort__cast(port);

  char* saying = strop_StringProducerPort_get(hello);
  printf("%s\n", saying);
  SIDL_String_free(saying);
```

```
  gov_cca_Services_releasePort(data->services, "HelloServer", &err);
  gov_cca_Port_deleteRef(port);

  return 0;
  /* DO-NOT-DELETE splicer.end(HelloClient.Component.go) */
}

/*
 * Method <code>setServices</code> is called by the framework.
 */

#undef __FUNC__
#define __FUNC__ "impl_HelloClient_Component_setServices"

void
impl_HelloClient_Component_setServices(
  HelloClient_Component self, gov_cca_Services services)
{
  /* DO-NOT-DELETE splicer.begin(HelloClient.Component.setServices) */
  struct HelloClient_Component__data* data =
    HelloClient_Component__get_data(self);
  SIDL_BaseException ex;

  gov_cca_Services_registerUsesPort(services, "HelloServer",
    "HelloServer.HelloPort", 0, &ex );

  gov_cca_Services_addProvidesPort( services, gov_cca_Port__cast(self),
    "GoPort", "gov.cca.ports.GoPort",
    0, &ex );

  data->services = services;
  gov_cca_Services_addRef(services);

  /* DO-NOT-DELETE splicer.end(HelloClient.Component.setServices) */
}
```

The implementation in the `_ctor` method is boilerplate; so much so that we considered folding it into Babel's generated code. But there's legitimate reasons for doing it this way, and so C developers using Babel will have to get comfortable with this. The state information in the Babel class is always in a `_data` struct[1] in the C header file. Generally in the `_ctor`, you want to `malloc` the struct, initialize its variables, and call `_set_data()` which will insert that pointer into the Babel managed class.

The `_dtor` method, does the opposite. It first calls `_get_data()` to get the pointer managed by the Babel class. It frees up resources held by the struct, then it frees up the struct itself. Since the Babel managed memory reference is free'd, it also makes good programming style to `_set_data()` to `NULL`, so Babel doesn't try any funny business.

[1]As in `struct package_class_data{ }`

5.2.3 Two C++ Drivers

There are two C++ drivers included in Babel's Decaf example: the easy one, and the official one. The easy one drives everything directly from *decaf.Framework* and hence has access to nonstandard capabilities. The official one makes use of only official CCA APIs as soon as the framework is created.

Easy one first:

```
#include "decaf_Framework.hh"
#include "gov_cca_ComponentID.hh"
#include "gov_cca_ports_GoPort.hh"

int main() {
  decaf::Framework fwk = decaf::Framework::_create();
  gov::cca::ComponentID server =
    fwk.createInstance( "HelloServerInstance",
"HelloServer.Component", 0 );
  gov::cca::ComponentID client =
    fwk.createInstance( "HelloClientInstance",
"HelloClient.Component", 0 );

  fwk.connect( client, "HelloServer", server, "HelloServer" );
  gov::cca::ports::GoPort go = fwk.lookupPort( client, "GoPort" );

  go.go();
}
```

Now the official way of handling things:

```
#include <iostream>
#include "decaf_Framework.hh"
#include "gov_cca_AbstractFramework.hh"
#include "gov_cca_ComponentID.hh"
#include "gov_cca_ports_GoPort.hh"
#include "gov_cca_ports_BuilderService.hh"


void main_setServices( gov::cca::Services svcs );
void main_go( gov::cca::Services svcs );

int main() {
  try {
    gov::cca::AbstractFramework fwk = decaf::Framework::_create();
    gov::cca::TypeMap properties = fwk.createTypeMap();
    gov::cca::Services svcs =
      fwk.getServices( "me", "myOwnType", properties );

    // \begin{main() masquerading as a component}
```

```
    main_setServices( svcs );
    main_go( svcs );
    // \end{main as a component}

    fwk.releaseServices( svcs );
    fwk.shutdownFramework();
  } catch (gov::cca::CCAException ex ) {
    std::cout << "Caught Exception\n"
      << ex.getNote() << '\n'
              << ex.getTrace() << std::endl;
  }

}

void main_setServices( gov::cca::Services svcs ) {
  gov::cca::TypeMap properties = svcs.createTypeMap();
  svcs.registerUsesPort("builder", "gov.cca.ports.BuilderServices",
properties );
  svcs.registerUsesPort("go", "gov.cca.ports.GoPort", properties );
}

void main_go( gov::cca::Services svcs ) {
  // get my builder service
  gov::cca::ports::BuilderService bs = svcs.getPort( "builder" );

  // create and connenct hello server and client
  gov::cca::ComponentID server =
    bs.createInstance( "HelloServerInstance", "HelloServer.Component", 0 );
  gov::cca::ComponentID client =
    bs.createInstance( "HelloClientInstance", "HelloClient.Component", 0 );
  bs.connect( client, "HelloServer", server, "HelloServer" );

  // now connect client's go point to mine
  bs.connect( svcs.getComponentID(), "go", client, "GoPort" );
  gov::cca::ports::GoPort go = svcs.getPort( "go" );
  go.go();

  svcs.releasePort( "builder" );
  svcs.releasePort( "go" );
}
```

Now remember way back in Section 3.2.5, when I indicated it might be best to bookmark explanation for later? Well, the time has come. The whole reason for *`cca.AbstractFramework.-getServices()`* is so that `main()` can masquerade as a component to the framework. Generally speaking, one of the tenents of component development is that components don't get `main()`. This religious view is borne out of years of experience trying to get two codes that both require `main()` to work together. . . they don't let go easily.

5.2.4 Java Driver

```
//
// File:        HelloDriver.java
// Copyright:   (c) 2001 The Regents of the University of California
// Release:     $Name:  $
// Revision:    @(#) $Revision: 1.3 $
// Date:        $Date: 2003/04/02 18:56:05 $
// Description: Simple CCA Hello World Java driver
//

public class HelloDriver {
  public static void main(String args[]) {
    try {
      decaf.Framework decaf = new decaf.Framework();
      gov.cca.TypeMap properties = decaf.createTypeMap();

      gov.cca.ComponentID server =
          decaf.createInstance("HelloServerInstance",
                               "HelloServer.Component",properties);
      gov.cca.ComponentID client =
          decaf.createInstance("HelloClientInstance",
                               "HelloClient.Component", properties);
      decaf.connect(client, "HelloServer", server, "HelloServer");

      gov.cca.Port port = decaf.lookupPort(client, "GoPort");
      gov.cca.ports.GoPort go =
          (gov.cca.ports.GoPort) port._cast("gov.cca.ports.GoPort");
      go.go();

      decaf.destroyInstance(server,0.0F);
      decaf.destroyInstance(client,0.0F);
      Runtime.getRuntime().exit(0); /* workaround for Linux JVM 1.3.1 bug */
    } catch (Throwable ex) {
      ex.printStackTrace(System.err);
      System.exit(-1);
    }
  }
}
```

5.2.5 F77 Driver

```
c
c     File:       HelloDriver.f
c     Copyright:  (c) 2001 The Regents of the University of California
c     Release:    $Name:  $
c     Revision:   @(#) $Revision: 1.3 $
c     Date:       $Date: 2003/04/02 18:56:05 $
```

```
c     Description:Simple CCA Hello World F77 client
c
c
      program HelloDriver
      integer *8 decaf
      integer *8 server
      integer *8 client
      integer *8 port
      integer *8 go
      integer *4 retval
      integer *8 ex
      integer *8 properties
      integer *8 connectionID

      call decaf_Framework__create_f(decaf)

      call decaf_Framework_createTypemap_f( decaf, properties, ex )

      call decaf_Framework_createInstance_f(
     &  decaf,
     &  'HelloServerInstance',
     &  'HelloServer.Component',
     &  properties,
     &  server,
     &  ex)
      call decaf_Framework_createInstance_f(
     &  decaf,
     &  'HelloClientInstance',
     &  'HelloClient.Component',
     &  properties,
     &  client,
     &  ex)
      call decaf_Framework_connect_f(
     &  decaf,
     &  client,
     &  'HelloServer',
     &  server,
     &  'HelloServer',
     &  connectionID,
     &  ex )

      call decaf_Framework_lookupPort_f(decaf, client, 'GoPort', port)
      call gov_cca_ports_GoPort__cast_f(port, go)
      call gov_cca_ports_GoPort_go_f(go, retval)

      call decaf_Framework_destroyInstance_f(decaf, server, 0.0, ex)
      call decaf_Framework_destroyInstance_f(decaf, client, 0.0, ex)
```

```
        return
        end
```

5.2.6 Python Driver

```
#!/usr/local/bin/python
#
# File:        HelloDriver.py
# Copyright:   (c) 2001 The Regents of the University of California
# Release:     $Name:  $
# Revision:    @(#) $Revision: 1.3 $
# Date:        $Date: 2003/04/02 18:56:05 $
# Description: Simple CCA Hello World Python client
#

import decaf.Framework
import gov.cca.ports.GoPort

if __name__ == '__main__':
  dec = decaf.Framework.Framework()

  server = dec.createInstance( "HelloServerInstance",
                               "HelloServer.Component", None )
  client = dec.createInstance( "HelloClientInstance",
                               "HelloClient.Component", None )
  dec.connect(client, "HelloServer", server, "HelloServer")

  port = dec.lookupPort(client, "GoPort")
  go = gov.cca.ports.GoPort.GoPort(port)
  go.go()

  dec.destroyInstance(server,0.0)
  dec.destroyInstance(client,0.0)
```

5.3 Summary

In this chapter we saw the most complex "Hello World!" example I've ever seen. But it is important to note that we did it by arbitrarily mixing Fortran77, C, C++, Java, and Python code. The key is that with Babel, any one of these parts could have been swapped out with a replacement written in another language without affecting any of the existing parts.

5.3.1 Broken into Levels of Understanding

Conceptual Understanding. Lots of code and details. Not much in new concepts here.

Beginner Level. Ditto.

Average Level. Ditto.

Advanced Details. To implement state in components in general, Babel generally manipulates a pointer to a struct in C, or a class in C++. To implement state in F77 components, that pointer can be made to reference to a SIDL array of `opaque`. Then each of those entries in that array can be a reference to another array of a specific type. It may look gross, but it gets the job done... besides, its Fortran 77.

Chapter 6

Conclusion

It is impossible to design a system so perfect that no one needs to be good.
— T. S. Elliot (1888–1965)

6.1 Future Changes

CCA and Babel are subject to constant research and development. This document is expected to be maintained through those changes for the forseeable future. Here's a sampling of things coming on the horizon.

6.1.1 Framework Interoperability

Ccaffeine supports components written with SIDL as well as two other types: classic and Chasm. Classic Ccaffeine is a C++ only binding that is being maintained for backward compatability, but deprecation status is expected. Chasm is an experimental Fortran 90 binding that predates Babel's support for Fortran 90. Chasm and Babel teams are working to cement a unified binding using SIDL and Chasm's advanced array descriptor manipulations.

SCIRun2 has recently announced that it can load and run a SIDL/Ccaffeine component without modifying the source. Ccaffeine earlier demonstrated this same level of compatability with Decaf.

XCAT emphasizes distributed component computing and bridges the gap between CCA, the Grid, and Web Services. Coupling between it and the other frameworks has been demonstrated via a messaging connection, but only as a feasiblity study. A robust communication protocol between framework developers, and Babel is expected in the next year or so.

6.1.2 Model Coupling

A harder issue for scientific codes is not just language interoperability, or reusing parts with low surface to volume ratios (such as solver or equation of state libraries); but coupling physics modules... most likely with different meshes. A full quarter of CCTTSS's effort is on handling exactly this issue in a robust way.

6.1.3 Component Repositories

A prototype web repository has been created and deployed. Babel can be configured to access this repository directly. Unfortunately it hasn't yet been substantially populated or used for various non-technical reasons. We expect this activity to pick up as developers switch from playing with components to actually deploying and exchanging them.

Bibliography

[1] Babel homepage. `http://www.llnl.gov/CASC/components/babel.html`.

[2] David E. Bernholdt, Wael R. Elwasif, James A. Kohl, and Thomas G. W. Epperly. A component architecture for high-performance computing. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages (POHLL-02)*, New York, NY, June 2002.

[3] CCAFE homepage. `http://www.cca-forum.org/~baallan/ccafe`.

[4] Common Component Architecture (CCA) Forum homepage. `http://www.cca-forum.org`.

[5] Tammy Dahlgren, Tom Epperly, and Gary Kumfert. *Babel User's Guide*. CASC, Lawrence Livermore National Laboratory, version 0.8.4 edition, April 2003.

[6] Scott Kohn, Gary Kumfert, Jeff Painter, and Cal Ribbens. Divorcing language dependencies from a scientific software library. In *10th SIAM Conference on Parallel Processing*, Portsmouth, VA, March 2001.

[7] SciDAC: Scientific Discovery through Advanced Computing. `http://www.science.doe.gov/scidac`.

[8] SCIRun homepage. `http://www.sci.utah.edu`.

[9] U. S. Department of Energy (DOE) homepage. `http://www.energy.gov`.

[10] XCAT homepage. `http://www.extreme.indiana.edu/xcat`.