CCA
Common Component Architecture

Components for Scientific Computing: An Introduction

CCA Forum Tutorial Working Group
http://www.cca-forum.org/tutorials/
tutorial-wg@cca-forum.org

Goals of This Module

- Introduce basic concepts and vocabulary of component-based software engineering
- Highlight the special demands of high-performance scientific computing on component environments
- Provide a unifying context for the remaining talks
  - And to consider what components might do for your applications

Motivation: Modern Scientific Software Engineering Challenges

- **Productivity**
  - Time to first solution (prototyping)
  - Time to solution (“production”)
  - Software infrastructure requirements (“other stuff needed”)
- **Complexity**
  - Increasingly sophisticated models
  - Model coupling – multi-scale, multi-physics, etc.
  - “Interdisciplinarity”
- **Performance**
  - Increasingly complex algorithms
  - Increasingly complex computers
  - Increasingly demanding applications

Motivation: For Library Developers

- People want to use your software, but need wrappers in languages you don’t support
  - Many component models provide language interoperability
- Discussions about standardizing interfaces are often sidetracked into implementation issues
  - Components separate interfaces from implementation
- You want users to stick to your published interface and prevent them from stumbling (prying) into the implementation details
  - Most component models actively enforce the separation

Motivation: For Application Developers and Users

- You have difficulty managing multiple third-party libraries in your code
- You (want to) use more than two languages in your application
- Your code is long-lived and different pieces evolve at different rates
- You want to be able to swap competing implementations of the same idea and test without modifying any of your code
- You want to compose your application with some other(s) that weren't originally designed to be combined

Some Observations About Software…

- "The complexity of software is an essential property, not an accidental one." *[Brooks]*
  - We can't get rid of complexity
- "Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements." *[Booch]*
  - We must find ways to manage it

More Observations…

- "A complex system that works is invariably found to have evolved from a simple system that worked… A complex system designed from scratch never works and cannot be patched up to make it work." *[Gall]*
  - Build up from simpler pieces
- "The best software is code you don't have to write" *[Jobs]*
  - Reuse code wherever possible

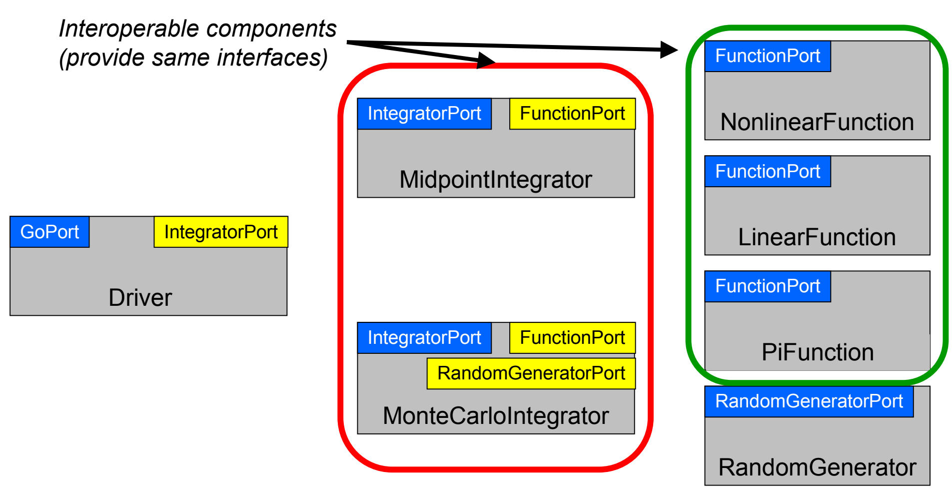
Not All Complexity is "Essential"

- An example of how typical development practices can exacerbate the complexity of software development…
- At least 41 different Fast Fourier Transform (FFT) libraries:
  - see, http://www.fftw.org/benchfft/doc/ffts.html
- Many (if not all) have different interfaces
  - different procedure names and different input and output parameters
- Example: SUBROUTINE FOUR1(DATA, NN, ISIGN)
  - "Replaces DATA by its discrete Fourier transform (if ISIGN is input as 1) or replaces DATA by NN times its inverse discrete Fourier transform (if ISIGN is input as -1). DATA is a complex array of length NN or, equivalently, a real array of length 2*NN. NN MUST be an integer power of 2 (this is not checked for!)."

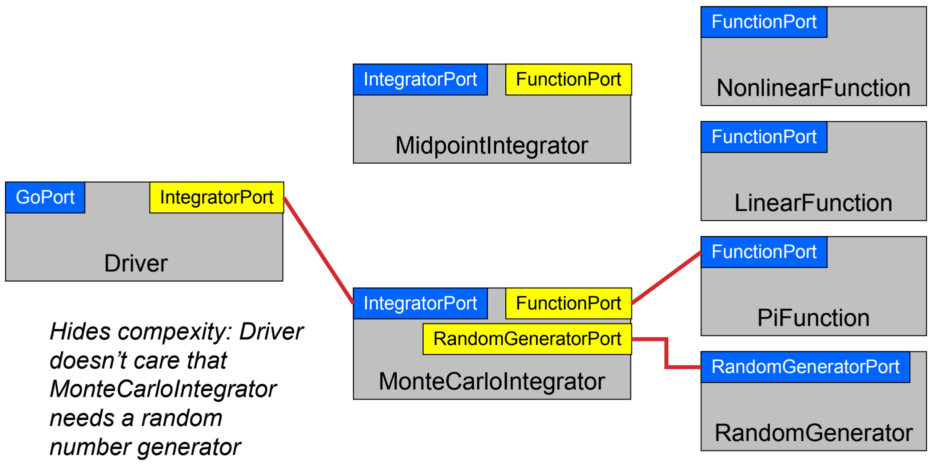
Component-Based Software Engineering

- CBSE methodology is emerging, especially from business and internet areas
- **Software productivity**
  - Provides a "plug and play" application development environment
  - Many components available "off the shelf"
  - Abstract interfaces facilitate reuse and interoperability of software
- **Software complexity**
  - Components encapsulate much complexity into "black boxes"
  - Plug and play approach simplifies applications
  - Model coupling is natural in component-based approach
- **Software performance** (indirect)
  - Plug and play approach and rich "off the shelf" component library simplify changes to accommodate different platforms

A Simple Example: Numerical Integration Components

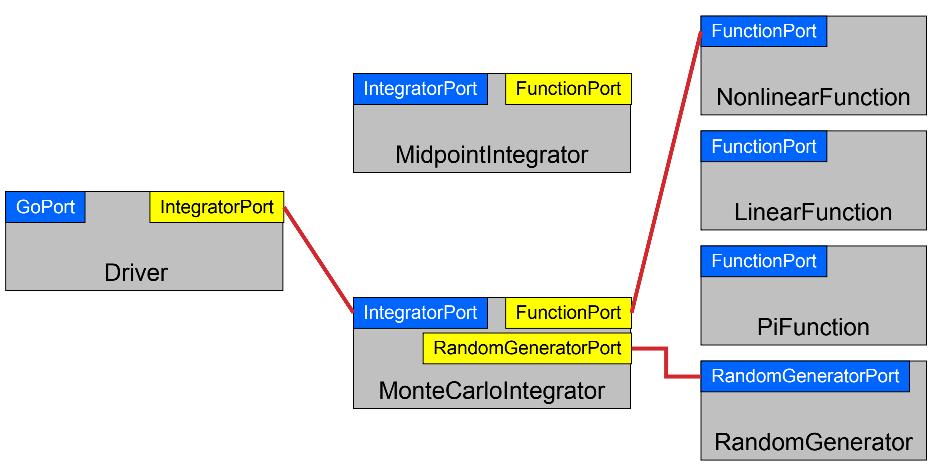


An Application Built from the Provided Components

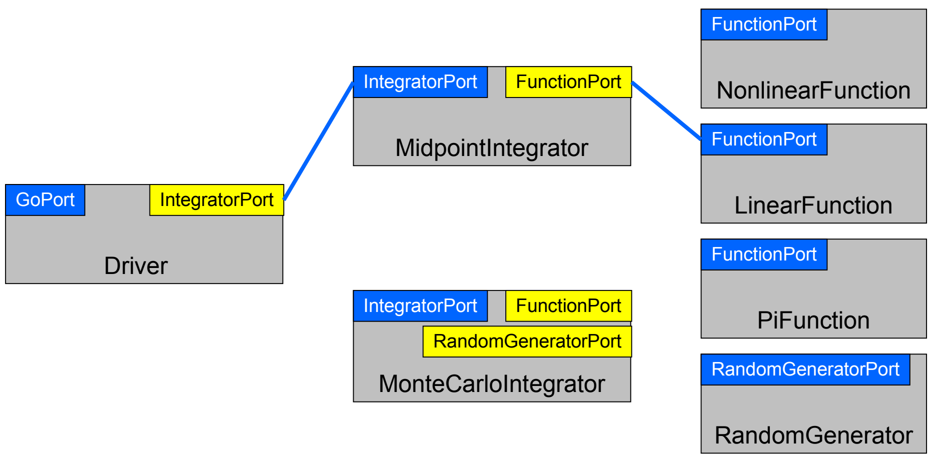


Hides compexity: Driver doesn't care that MonteCarloIntegrator needs a random number generator

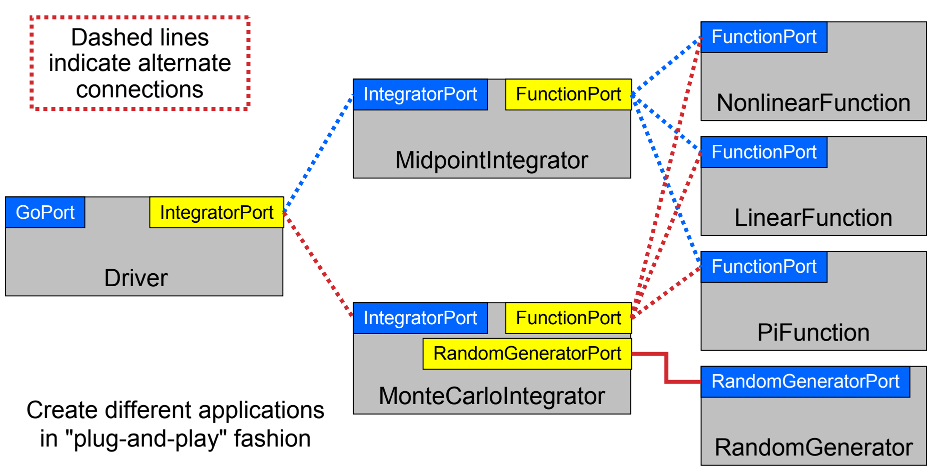
Another Application…



Application 3…



And Many More…



What *are* Components?

- No universally accepted definition…yet
- **A unit of software development/deployment/reuse**
  - i.e. has interesting functionality
  - Ideally, functionality someone else might be able to (re)use
  - Can be developed independently of other components
- **Interacts with the outside world *only* through well-defined interfaces**
  - Implementation is opaque to the outside world
  - Components *may* maintain state information
  - But external access to state info must be through an interface (*not a common block*)
  - File-based interactions can be recast using an "I/O component"
- **Can be composed with other components**
  - "Plug and play" model to build applications
  - Composition based on interfaces

What is a Component Architecture?

- A set of standards that allows:
  - Multiple groups to write units of software (components)…
  - And have confidence that their components will work with other components written in the same architecture
- These standards define…
  - The rights and responsibilities of a component
  - How components express their interfaces
  - The environment in which are composed to form an application and executed (framework)
  - The rights and responsibilities of the framework

Interfaces, Interoperability, and Reuse

- Interfaces define how components interact…
- Therefore interfaces are key to interoperability and reuse of components
- In many cases, “any old interface” will do, but…
- General plug and play interoperability requires multiple implementations providing the same interface
- Reuse of components occurs when they provide interfaces (functionality) needed in multiple applications

Designing for Reuse, Implications

- Designing for interoperability and reuse requires “standard” interfaces
  - Typically domain-specific
  - “Standard” need not imply a formal process, may mean “widely used”
- Generally means collaborating with others
- *Higher* initial development cost (amortized over multiple uses)
- Reuse implies longer-lived code
  - thoroughly tested
  - highly optimized
  - improved support for multiple platforms

Typical Component Lifecycle

- **Composition Phase**
  - Component is instantiated in framework
  - Component interfaces are connected appropriately
- **Execution Phase**
  - Code in components uses functions provided by another component
- **Decomposition Phase**
  - Connections between component interfaces may be broken
  - Component may be destroyed

In an application, individual components may be in different phases at different times

Steps may be under human or software control

Relationships: Components, Objects, and Libraries

- Components are typically discussed as objects or collections of objects
  - Interfaces generally designed in OO terms, but…
  - Component internals need not be OO
  - OO languages are not required
- Component environments can enforce the use of published interfaces (prevent access to internals)
  - Libraries can not
- It is possible to load several instances (versions) of a component in a single application
  - Impossible with libraries
- Components *must* include some code to interface with the framework/component environment
  - Libraries and objects do not

Domain-Specific Frameworks vs Generic Component Architectures

Domain-Specific

- Often known as "frameworks"
- Provide a significant software infrastructure to support applications in a given domain
  - Often attempts to generalize an existing large application
- Often hard to adapt to use outside the original domain
  - Tend to assume a particular structure/workflow for application
- Relatively common

Generic

- Provide the infrastructure to hook components together
  - Domain-specific infrastructure can be built as components
- Usable in many domains
  - Few assumptions about application
  - More opportunities for reuse
- Better supports model coupling across traditional domain boundaries
- Relatively rare at present
  - Commodity component models often not so useful in HPC scientific context

Special Needs of Scientific HPC

- Support for legacy software
  - How much change required for component environment?
- Performance is important
  - What overheads are imposed by the component environment?
- Both parallel and distributed computing are important
  - What approaches does the component model support?
  - What constraints are imposed?
  - What are the performance costs?
- Support for languages, data types, and platforms
  - Fortran?
  - Complex numbers? Arrays? (as first-class objects)
  - Is it available on my parallel computer?

Commodity Component Models

- CORBA, COM, Enterprise JavaBeans
  - Arise from business/internet software world
- Componentization requirements can be high
- Can impose significant performance overheads
- No recognition of tightly-coupled parallelism
- May be platform specific
- May have language constraints
- May not support common scientific data types

The “Sociology” of Components

- Components need to be shared to be truly useful
  - Sharing can be at several levels
    - Source, binaries, remote service
  - Various models possible for intellectual property/licensing
    - Components with different IP constraints can be mixed in a single application
- Peer component models facilitate collaboration of groups on software development
  - Group decides overall architecture and interfaces
  - Individuals/sub-groups create individual components

Who Writes Components?

- "Everyone" involved in creating an application can/should create components
  - Domain scientists as well as computer scientists and applied mathematicians
  - Most will also use components written by other groups
- Allows developers to focus on their interest/specialty
  - Get other capabilities via reuse of other's components
- Sharing components within scientific domain allows everyone to be more productive
  - Reuse instead of reinvention
- As a unit of publication, a well-written and –tested component is like a high-quality library
  - Should receive same degree of recognition
  - Often a more appropriate unit of publication/recognition than an entire application code

Summary

- Components are a software engineering tool to help address software productivity and complexity
- Important concepts: components, interfaces, frameworks, composability, reuse
- Scientific component environments come in "domain specific" and "generic" flavors
- Scientific HPC imposes special demands on component environments
  - Which commodity tools may have trouble with