CCA
Common Component Architecture

Welcome to the Common Component Architecture Tutorial

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

Active Members:

Rob Armstrong, David Bernholdt, Wael Elwasif, Lori Freitag, Dan Katz, Jim Kohl, Gary Kumfert, Lois Curfman McInnes, Boyana Norris, Craig Rasmussen, Jaideep Ray

Lawrence Livermore National Laboratory · Los Alamos National Laboratory · ornl Oak Ridge National Laboratory · Sandia National Laboratories

Acknowledgements

- Thanks to all members of the CCA Forum
  - Originated in 1998; open to everyone interested in HPC components
  - See http://www.cca-forum.org
  - Active CCA Forum participants include
    - ANL - Lori Freitag, Kate Keahey, Jay Larson, Ray Loy, Lois Curfman McInnes, Boyana Norris, …
    - Indiana University - Randall Bramley, Dennis Gannon, …
    - JPL – Dan Katz, …
    - LANL - Craig Rasmusse, Matt Sotille, …
    - LLNL - Tom Epperly, Scott Kohn, Gary Kumfert, …
    - ORNL - David Bernholdt, Wael Elwasif, Jim Kohl, Torsten Wilde, …
    - PNNL - Jarek Nieplocha, Theresa Windus, …
    - SNL - Rob Armstrong, Ben Allan, Curt Janssen, Jaideep Ray, …
    - University of Utah - Steve Parker, …
    - And others as well …
- Thanks to the Office of Mathematical, Information, and Computational Sciences (MICS), U.S. Department of Energy
  - Support of the Center for Component Technology for Terascale Simulation Software (CCTTSS), which is funded through the SciDAC Initiative
    - CCTTSS team is a subset of the CCA Forum; leader is Rob Armstrong (SNL)
    - See http://www.cca-forum.org/ccttss

CCA
Common Component Architecture

Introduction to Components

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

CCA
Common Component Architecture

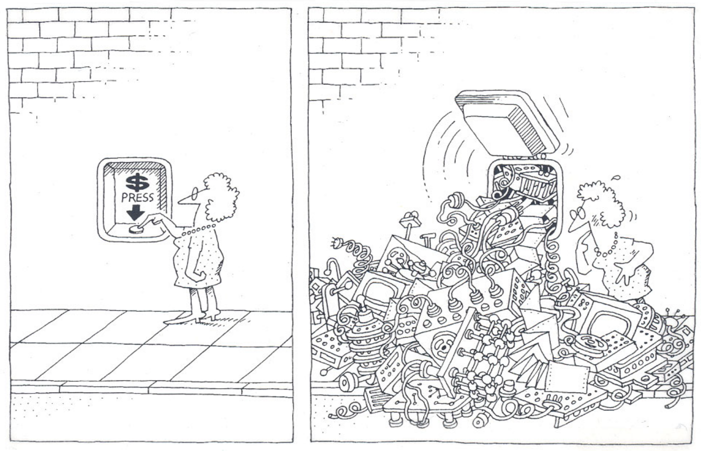
Overview

- **Why** do we need components?
- **What** are components?
- **How** do we make components?

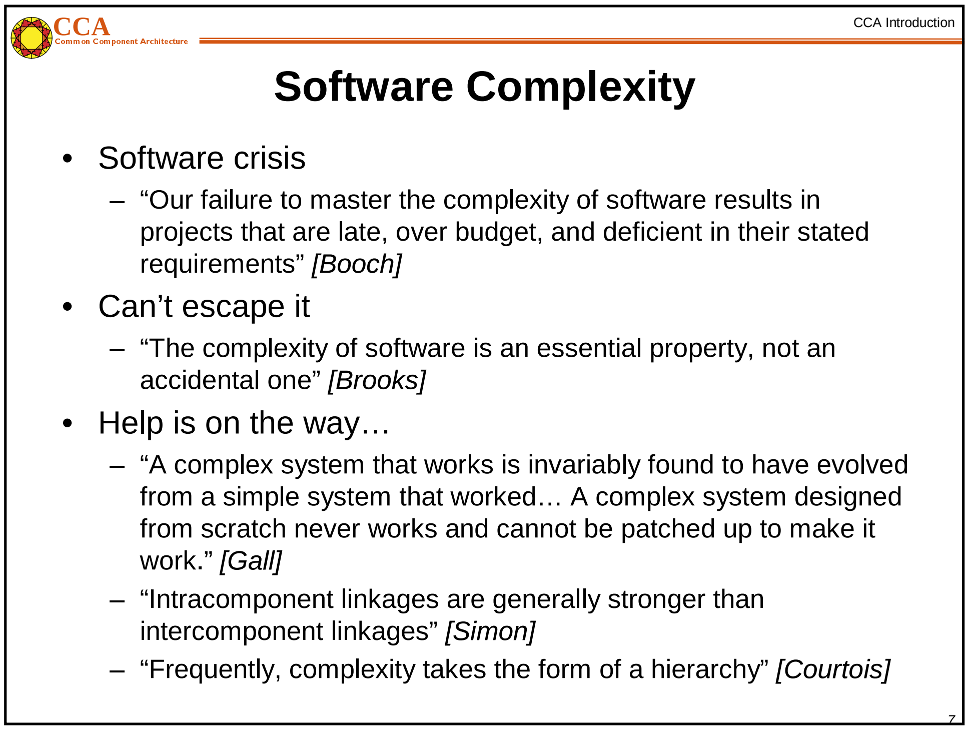
Why Components

- In “Components, The Movie”
  - Interoperability across multiple languages
  - Interoperability across multiple platforms
  - Incremental evolution of large legacy systems (esp. w/ multiple 3rd party software)
- Complexity

Why Components

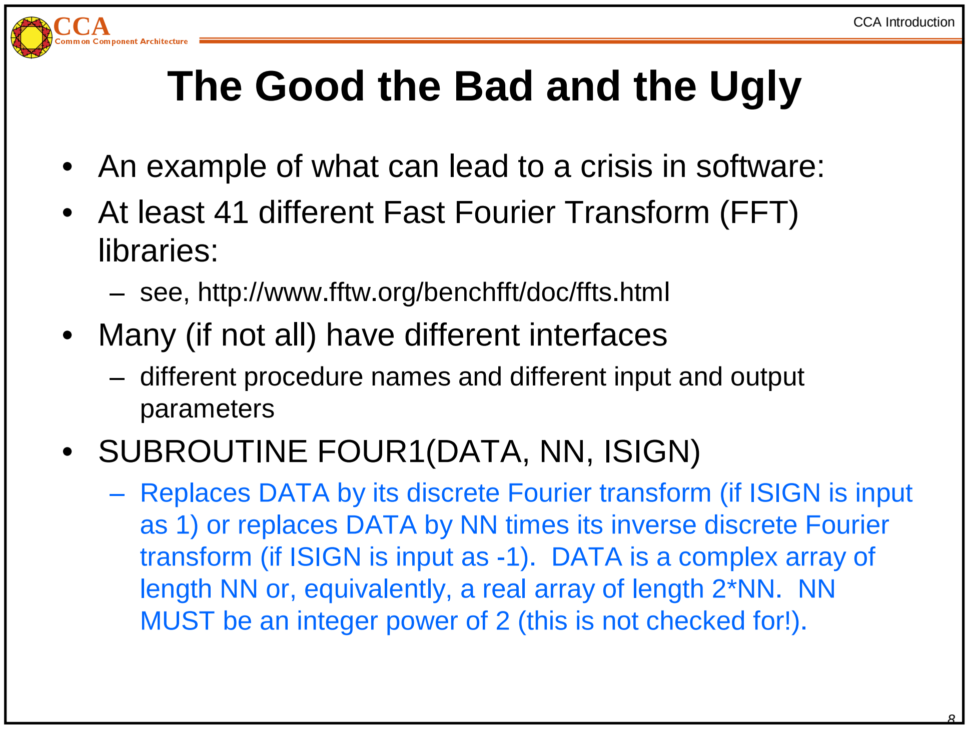


The task of the software development team is to engineer the illusion of simplicity *[Booch]*.



Software Complexity

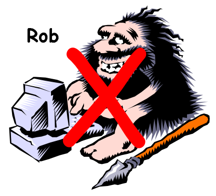
- Software crisis
  - “Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements” *[Booch]*
- Can’t escape it
  - “The complexity of software is an essential property, not an accidental one” *[Brooks]*
- Help is on the way…
  - “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]*
  - “Intracomponent linkages are generally stronger than intercomponent linkages” *[Simon]*
  - “Frequently, complexity takes the form of a hierarchy” *[Courtois]*



The Good the Bad and the Ugly

- An example of what can lead to a crisis in software:
- 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
- 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!).

Components Promote Reuse

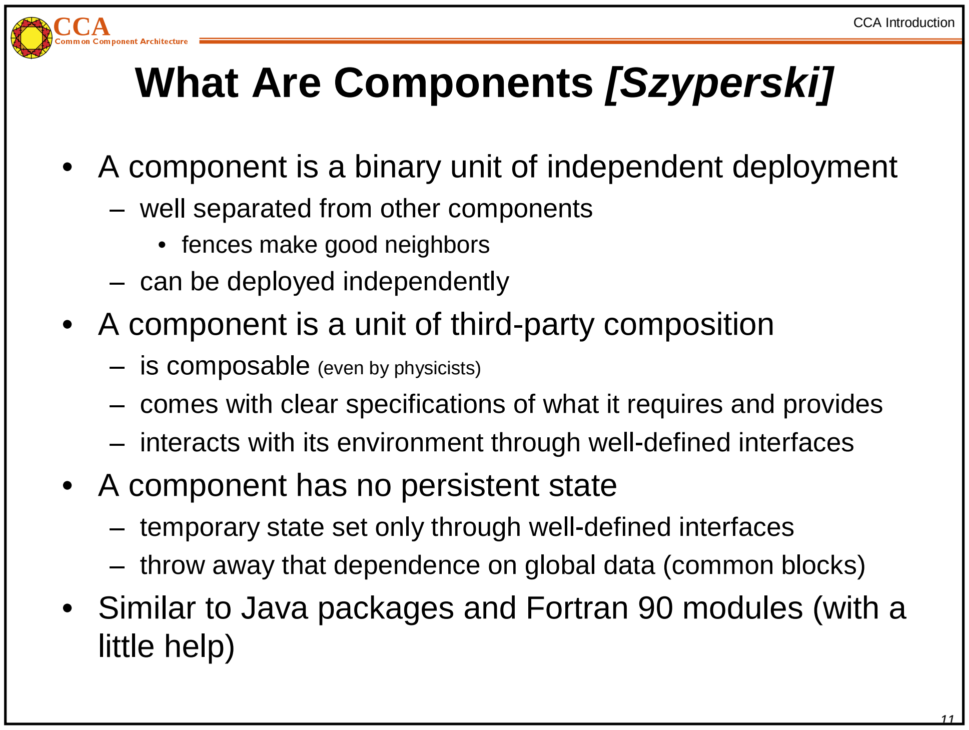


Hero programmer producing single-purpose, monolithic, tightly-coupled parallel codes

- Components promote software reuse
  - “The best software is code you don’t have to write” *[Steve Jobs]*
- Reuse, through cost amortization increases software quality
  - thoroughly tested code
  - highly optimized code
  - improved support for multiple platforms
  - developer team specialization

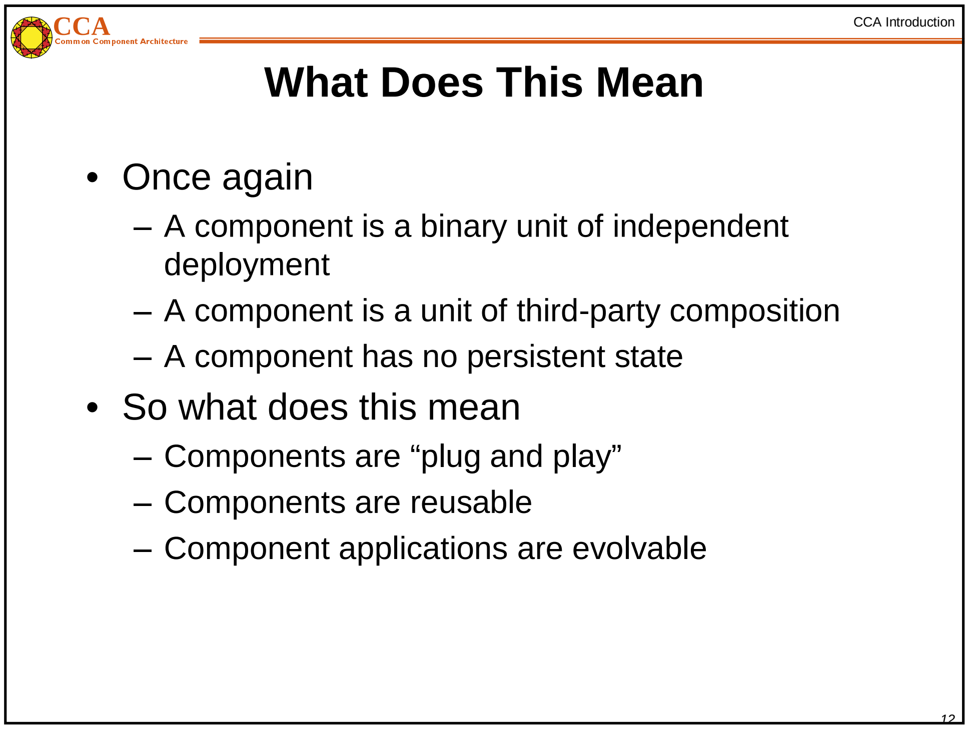
What Are Components

- **Why** do we need components?
- **What** are components?
- **How** do we make components?



What Are Components *[Szyperski]*

- A component is a binary unit of independent deployment
  - well separated from other components
    - fences make good neighbors
  - can be deployed independently
- A component is a unit of third-party composition
  - is composable (even by physicists)
  - comes with clear specifications of what it requires and provides
  - interacts with its environment through well-defined interfaces
- A component has no persistent state
  - temporary state set only through well-defined interfaces
  - throw away that dependence on global data (common blocks)
- Similar to Java packages and Fortran 90 modules (with a little help)



What Does This Mean

- Once again
  - A component is a binary unit of independent deployment
  - A component is a unit of third-party composition
  - A component has no persistent state
- So what does this mean
  - Components are “plug and play”
  - Components are reusable
  - Component applications are evolvable

What Are Components II

- Components live in an environment and interact with the environment through a framework and connections with other components.
- Components can discover information about their environment from the framework.
- Components must explicitly publish what capabilities they provide.
- Components must explicitly publish what connections they require.
- Components are a runtime entity.

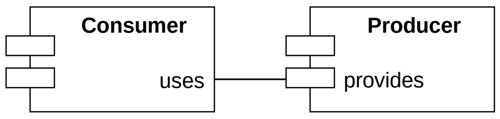
Components Are Different From Objects

- Think of a component stereo system:
  - You buy a new, super-cool CD player, bring it home, wire it up, turn on the power, and it works!
- A software component system:
  - You buy (or download) a new, super-fast FFT component, wire the connections, click on the go button, and it works!
  - (remember, a software component is a binary unit)
- A software class library:
  - You buy it, install it, do a little programming (or a lot), compile it, link it, and then run it, and hopefully it works.

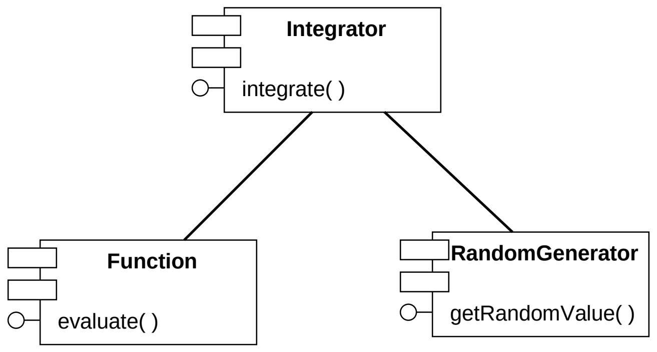
Components, Different From Objects II

- You can build components out of object classes.
- But a component is more than just an object.

Pictorial Example



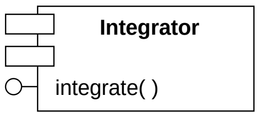
Three Components



How Do We Make Components

- **Why** do we need components?
- **What** are components?
- <u>**How** do we make components?</u>

Interface Declaration



Integrator.h

```
class Integrator
{
   virtual void
    integrate( double lowBound,
               double upBound,
               int count ) = 0;
};
```

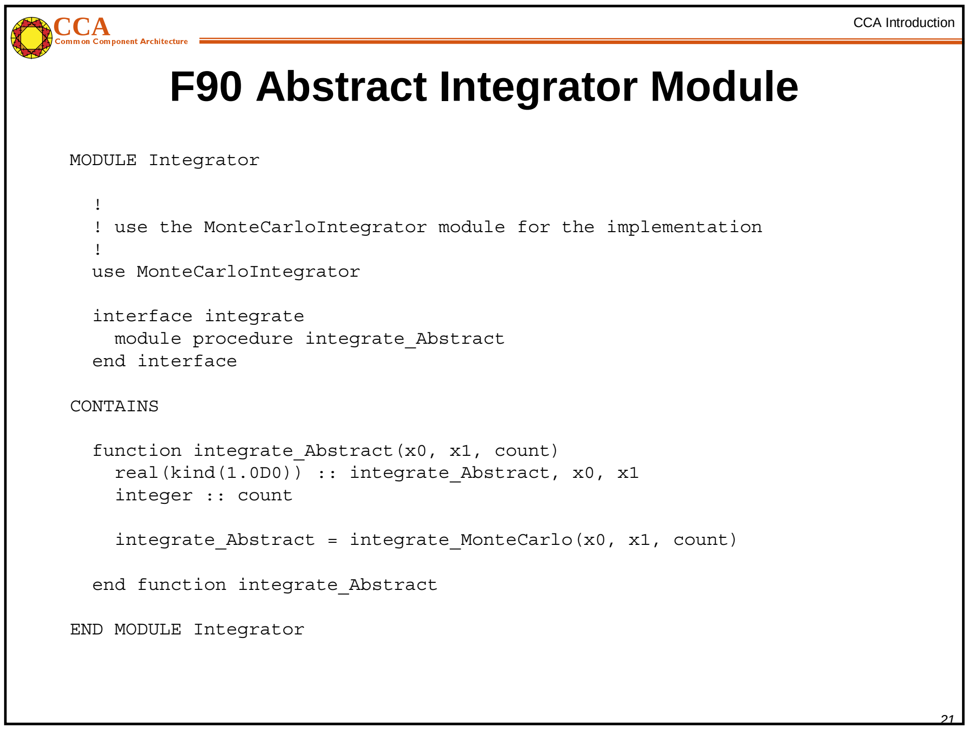
Integrator.f90

```
MODULE Integrator
  interface integrate
    module procedure integrate_Abstract
  end interface
END MODULE
```

Publish the Interface in SIDL

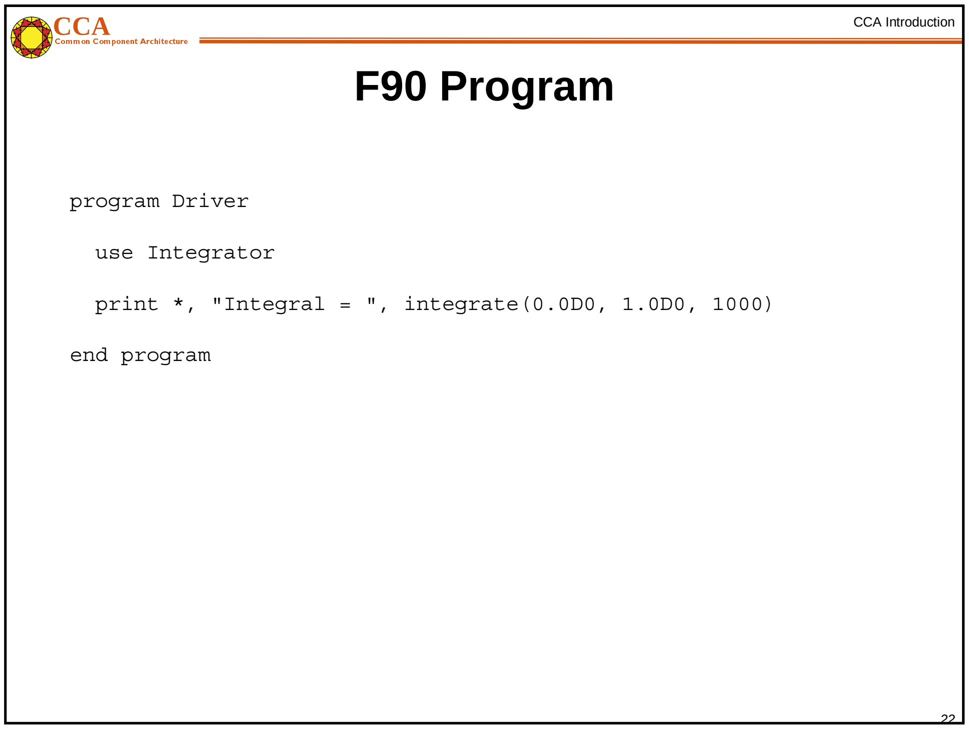
- Publish the interface
  - interfaces are published in SIDL (Scientific Interface Definition Language)
  - can’t publish in native language because of language interoperability requirement
- Integrator example:

```
interface Integrator extends cca.Port
 {
   double integrate(in double lowBound,
                    in double upBound,
                    in int count);
 }
```



F90 Abstract Integrator Module

```
MODULE Integrator

  !
  ! use the MonteCarloIntegrator module for the implementation
  !
  use MonteCarloIntegrator

  interface integrate
    module procedure integrate_Abstract
  end interface

CONTAINS

  function integrate_Abstract(x0, x1, count)
    real(kind(1.0D0)) :: integrate_Abstract, x0, x1
    integer :: count

    integrate_Abstract = integrate_MonteCarlo(x0, x1, count)

  end function integrate_Abstract

END MODULE Integrator
```



F90 Program

```
program Driver

  use Integrator

  print *, "Integral = ", integrate(0.0D0, 1.0D0, 1000)

end program
```

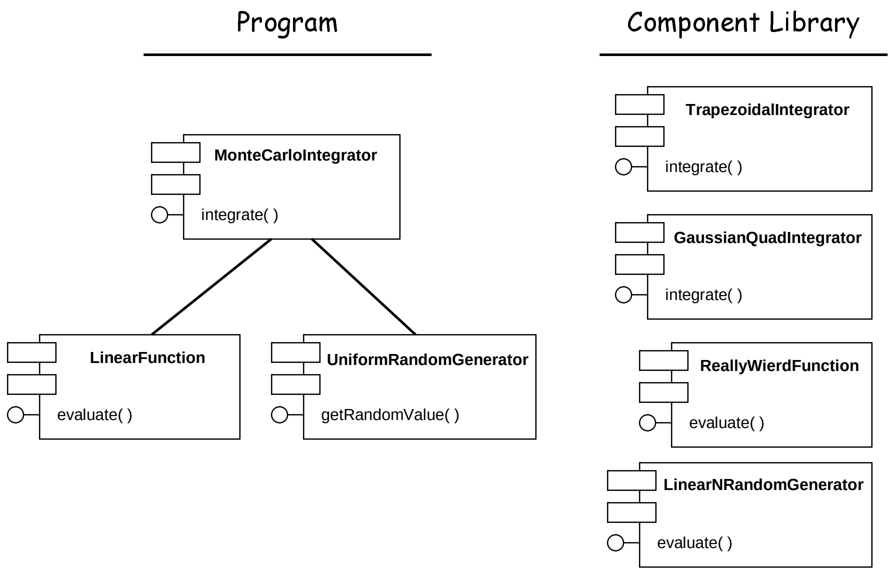
C++ Abstract Integrator Class

```
/**
 * This abstract class declares the Integrator interface.
 */

class Integrator
{
  public:
    virtual ~Integrator() { }

    /**
     * Sets the function, which will be evaluated by the integration.
     */
    virtual void setFunction(Function* function_to_integrate) = 0;

    /**
     * Returns the result of the integration from lowBound to upBound.
     *
     * lowBound - the beginning of the integration interval
     * upBound  - the end of the integration interval
     * count    - the number of integration points
     */
    virtual double integrate(int count, double x0, double x1) = 0;
};
```

C++ Object-Oriented Program

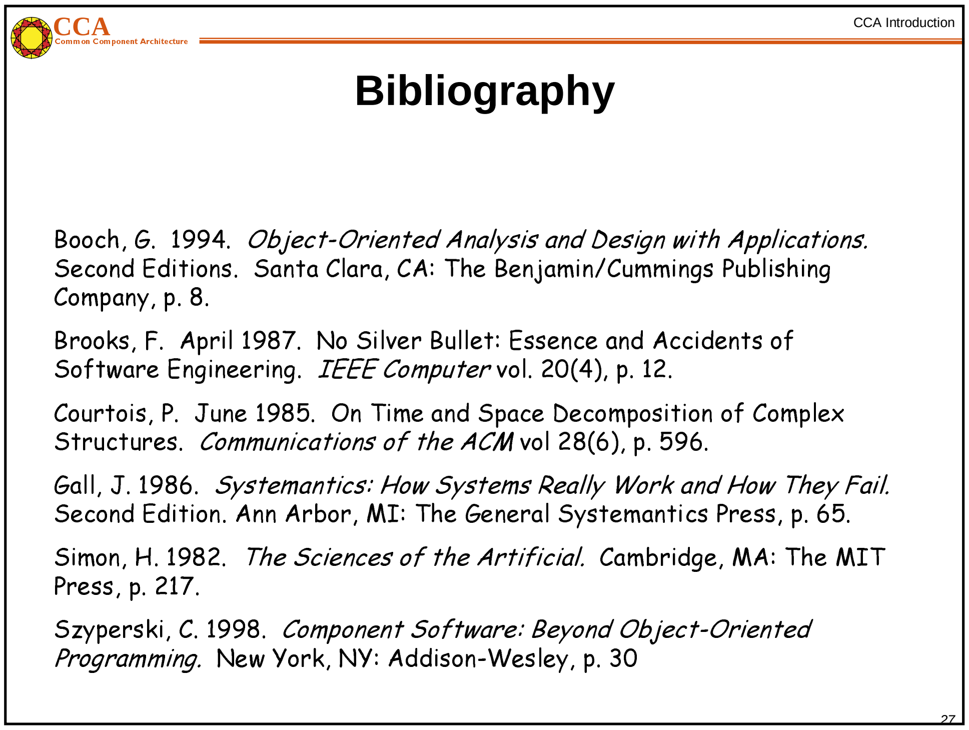
```
#include <iostream>
#include "Function.h"
#include "Integrator.h"
#include "RandomGenerator.h"

int main(int argc, char* argv[])
{
   LinearFunction* function = new LinearFunction();
   UniformRandomGenerator* random = new UniformRandomGenerator();
   MonteCarloIntegrator* integrator = new MonteCarloIntegrator();

   integrator->setFunctionInstance(function);
   integrator->setRandomGeneratorInstance(random);

   cout << "Integral = " << integrator->integrate(100) << endl;

   return 0;
}
```

Component Program



Final Thought

- Components are reusable assets. Compared with specific solutions to specific problems, components need to be carefully generalized to enable reuse in a variety of contexts. Solving a general problem rather than a specific one takes more work. In addition, because of the variety of deployment contexts, the creation of proper documentation, test suites, tutorials, online help texts, and so on is more demanding for components than for a specialized solution. *[Szyperski, p. 14]*



Bibliography

Booch, G. 1994. *Object-Oriented Analysis and Design with Applications.* Second Editions. Santa Clara, CA: The Benjamin/Cummings Publishing Company, p. 8.

Brooks, F. April 1987. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer* vol. 20(4), p. 12.

Courtois, P. June 1985. On Time and Space Decomposition of Complex Structures. *Communications of the ACM* vol 28(6), p. 596.

Gall, J. 1986. *Systematics: How Systems Really Work and How They Fail.* Second Edition. Ann Arbor, MI: The General Systemantics Press, p. 65.

Simon, H. 1982. *The Sciences of the Artificial.* Cambridge, MA: The MIT Press, p. 217.

Szyperski, C. 1998. *Component Software: Beyond Object-Oriented Programming.* New York, NY: Addison-Wesley, p. 30