The CÆSAR Code: Software Design Issues (Extended Version)

Michael L. Hall
Radiation Transport Methods Group (XTM)
Los Alamos National Laboratory
Email: **hall@lanl.gov**

X-Division External
Review Committee Presentation
3 / 10 / 99

Available on-line at
http://www.lanl.gov/Caesar/

Outline

- Background
  - CÆSAR Description
  - Diffusion Discretization References
- Documentation
  - Why Document A Program?
  - Levels of Documentation
  - Literate Programming
  - Simplified Approach: **`Document`**
  - Examples
  - CÆSAR Documentation Features
- Unit Testing / Levelized Design
  - Basic Ideas
  - Preliminary CÆSAR Levelized Design
  - Unit Testing Implementation
- Design By Contract / Verification
  - Basic Ideas
  - Verification Implementation
  - Design By Contract Implementation
- Summary

CÆSAR Description

- 3-T Photonics Diffusion (P_1) Code
- Multiple Dimensionality (1-D, 2-D, 3-D)
- Unstructured Hexahedral Cells in 3-D
- Second-Order Convergent Diffusion Discretizations
- Parallel, written in Fortran 90
- Based on earlier Augustus (P-1) and Spartan (SP_N) codes
- Future: Polyhedral Meshes, Multigroup, Tensor Diffusion, Mixed Cells, Transport

Diffusion Discretization References

- Morel-Hall Asymmetric Method
  - Described in

    Michael L. Hall, and Jim E. Morel. A Second-Order Cell-Centered Diffusion Differencing Scheme for Unstructured Hexahedral Lagrangian Meshes. In *Proceedings of the 1996 Nuclear Explosives Code Developers Conference (NECDC), UCRL-MI-124790*, pages 359–375, San Diego, CA, October 21–25 1996. LA-UR-97-8.

    which is an extension of

    J. E. Morel, J. E. Dendy, Jr., Michael L. Hall, and Stephen W. White. A Cell-Centered Lagrangian-Mesh Diffusion Differencing Scheme. Journal of Computational Physics, 103(2):286-299, December 1992.

    to 3-D unstructured meshes, with an alternate derivation.

- Support Operator Symmetric Method:
  - Described in

    Michael L. Hall, and Jim E. Morel. Diffusion Discretization Schemes in Augustus: A New Hexahedral Symmetric Support Operator Method. In *Proceedings of the 1998 Nuclear Explosives Code Developers Conference (NECDC)*, Las Vegas, NV, October 26–30 1998. LA-UR-98-3146.

    which is an extension of

    Mikhail Shashkov and Stanly Steinberg. Solving Diffusion Equations with Rough Coefficients in Rough Grids. Journal of Computational Physics, 129:383-405, 1996.

    to 3-D unstructured meshes, with an alternate derivation.

Why Document A Program?

For Others:

- To Demonstrate Progress in Coding
- To Encourage Use of the Package
- To Reduce “Hit-By-A-Bus” Syndrome
- To Facilitate Technical Review

For Yourself:

- To Understand Global Logical Code Structure
- To Facilitate Computer Code “Re-Entry” For Debugging, Maintenance, and Enhancement
- To Explain Things Once, not Multiple Times, to Users
- To Allow Quick Code Access via Hypertext
- To Be Proud of Your Work

Levels of Documentation

A code can be rated according to where it falls on this sequential list:

0. Layout
   - 0-a. Consistency
   - 0-b. Logical Block Structure (Few or No Branches)
   - 0-c. Indentation to Show Logical Structure
   - 0-d. Blank Lines and Spaces for Readability
   - 0-e. Statements Grouped Semantically
1. Descriptive Variable and Routine Names
2. Comments throughout the Code
3. Routine Headers with
   - 3-a. Purpose
   - 3-b. Input/Output Variable Descriptions
   - 3-c. Internal Variable Descriptions
   - 3-d. Methods Employed

Levels of Documentation (cont)

4. Hardcopy Documentation
   - 4-a. Code Listing
   - 4-b. Code Manual
   - 4-c. User's Manual
   - 4-d. Method Discussion
5. Hypertext Documentation
   - 5-a. Code Listing
   - 5-b. Code Manual
   - 5-c. User's Manual
   - 5-d. Method Discussion
   - 5-e. External Links
6. Literate Programming:
   Source Code and Documentation are Generated from the Same File

Articles on methods constitute supporting, but ancillary, documentation of a code. They should be included in the references of the hardcopy version and the external links of the hypertext version.

Literate Programming

- Basic Idea: Combine Documentation and Source Code
- WEB (Donald Knuth, of TeX fame)
  - Weave: web file ⟶ documentation (TeX)
  - Tangle: web file ⟶ source code (Pascal)
- Many others, most based on WEB:

Program	Source Language	Formatting Language	Availability
APLWEB	APL	TeX	MSDOS
AWEB	Ada	?	?
CLiP	Any	Any	written in Pascal
CWEB	C/C++	TeX/LaTeX	Unix/DOS/Amiga
mCWEB	C/C++	TeX	Unix
FunnelWeb	Any	TeX/Any	Many
FWEB	Many/Any	LaTeX	written in C
IMPACT	C/C++	TeX	Macintosh Only
LPW	C++/Pascal	WYSIWYG	Macintosh Only
MWEB	Modula-2	?	?
noweb	Any	TeX/LaTeX/HTML	Unix/DOS
nuweb	Any	LaTeX	Unix/DOS/Amiga
ProTeX	Any	TeX/LaTeX	written in TeX
RWEB	?	?	written in awk
SchemeWEB	Lisp	LaTeX	Unix/DOS
SpideryWEB	C/Ada/Pascal	TeX/LaTeX	Unix/DOS
WEB	Pascal	TeX	?
WinWordWEB	Any	Word	DOS

- My opinion: most are too complex or don't support my situation (F90, LaTeX, Unix)

The `Document` Package:

A Simplified Approach to Literate Programming

- Eliminate "tangle" step – files are compilable source
- Documentation is included in comments
- Small set of commands to direct output
- Formatting language independent
- Source code language independent (almost — just need to know comment characters)
- Implementation via a perl script: **`Document`**
- **`Document`** (1000 lines of documented source) is much smaller than WEB (10,000 lines)
- Source and documentation for the **`Document`** Package are available online at:

  `http://www.lanl.gov/Document`

A Simple Example

This input file:

```
! Begin_Doc
! Some documentation for standard out.
! End_Doc
!
! This line doesn't get output by Document.
! Begin_Doc file.tex
! This output goes to the file named file.tex.
! Comment characters are stripped by default.
!
! Begin_Verbatim
! Comment characters are included in verbatim
! environments, which are often used for code:
  do i = 1, 100
    j = j+1
  end do
! End_Verbatim
! End_Doc
```

when processed by **`Document`**, outputs this to standard out:

```
Some documentation for standard out.
```

and this to **`file.tex`**:

```
This output goes to the file named file.tex.
Comment characters are stripped by default.

    ! Comment characters are included in verbatim
    ! environments, which are often used for code:
      do i = 1, 100
        j = j+1
      end do
```

Other Document Features

If your formatting language supports it, you can modify input order:

```
! Begin_Doc main.tex
! % Note that the order of files a.tex
! % and b.tex has been switched.
! \input{b}
! \input{a}
! End_Doc
!
! Begin_Doc a.tex
! This line is in file a.tex.
! End_Doc
!
! Begin_Doc b.tex
! This line is in file b.tex.
! End_Doc
!
! Begin_Doc a.tex
! This line is appended to file a.tex.
! End_Doc
```

`Document` also has a self-document (or self-test) option:

```
! Begin_Self_Documentation (or Begin_Self_Test)
!  % mv file1 file2
!  % f90 file.f90
!  % Document file.f90
! End_Self_Documentation (or End_Self_Test)
```

which executes commands included in the file itself.

CÆSAR Documentation

Making use of the capabilities of **`Document`**, LaTeX and LaTeX2**`HTML`**, the CÆSAR Code documentation has these features:

- Hardcopy *and* HTML versions from a single source, which is collocated with the source code
- Multiple output files and source languages (f90, gm4)
- Graphics, equations, code listings easily included
- Automatic table of contents (hyperlinked in HTML)
- Semi-automatic indexing (hyperlinked in HTML)
- Items included in only LaTeX or HTML version
- Automatic navigation tools for HTML (Next, Up, Previous, Contents, and Index links on every page)
- Hyper references (e.g. "see Section 3.2" becomes a link)
- External HTML links (e.g. to related presentations, papers, packages or projects)
- Level 6 Documentation — User's Manual, Code Manual, Methods Discussion and Code Listing in Hardcopy and Hyperlinked HTML via Literate Programming

Unit Testing / Levelized Design

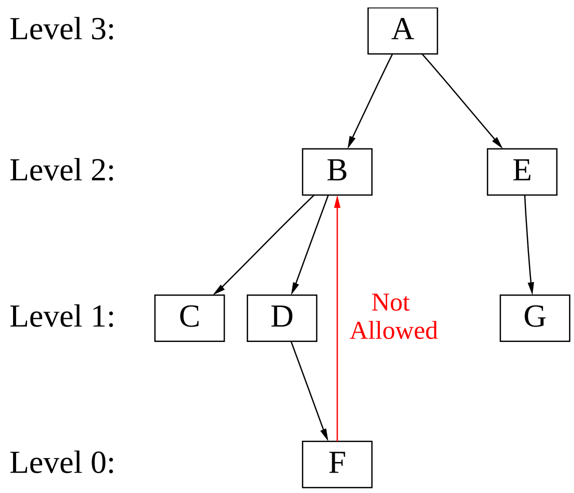
Basic Idea of Unit Testing:

Each component is tested in isolation – only components that have been previously tested may be included.

Basic Idea of Levelized Design:

Each component depends only on components that are at a lower level – no feedback or circular designs.

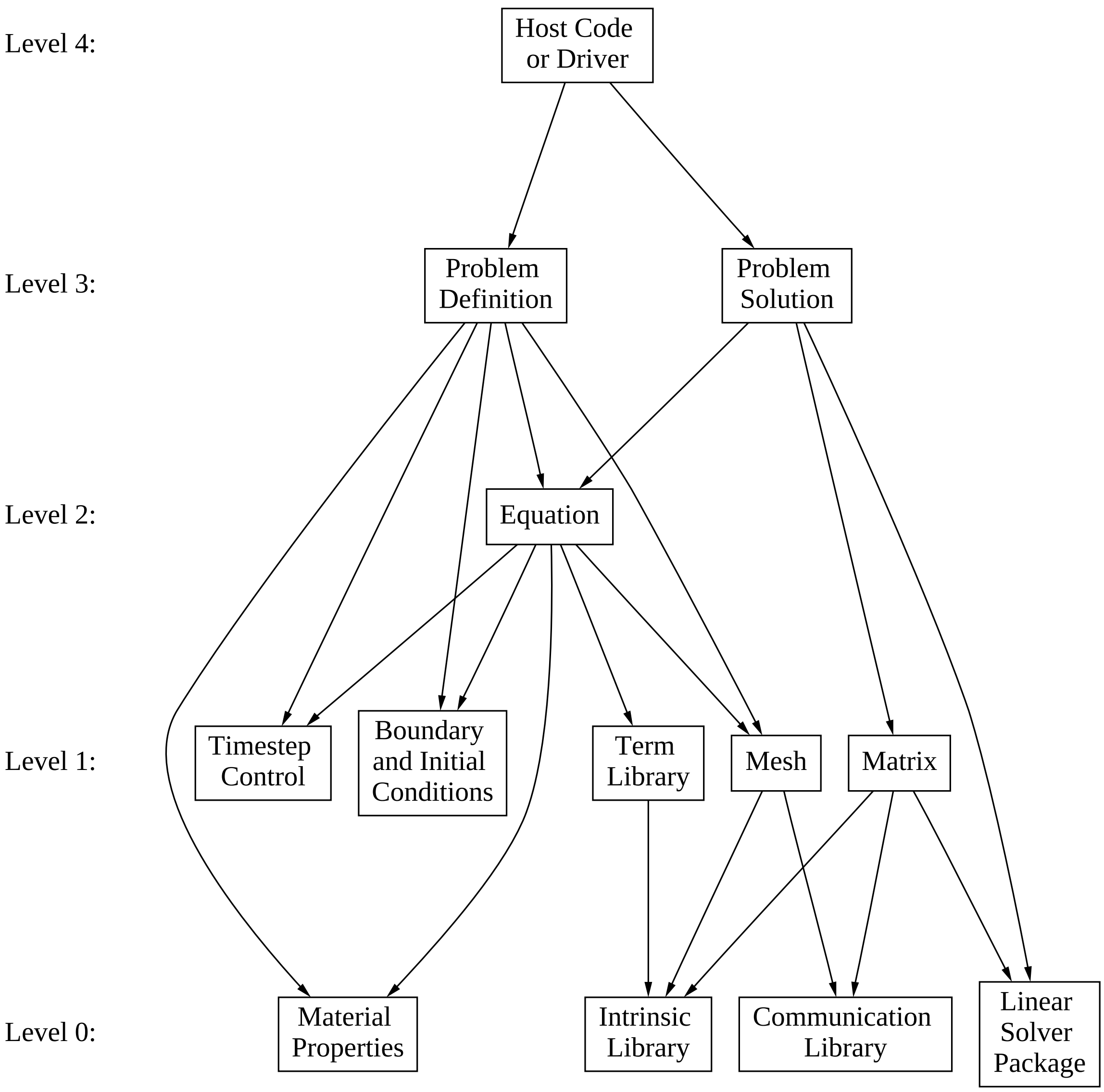
Example:



Why is a Levelized Design desirable?

- Necessary for incremental compilation in F90 if dependency is via "use association"
- Makes Unit Testing possible

Preliminary Levelized Design for CÆSAR



Unit Testing Implementation

- Every component contains its own specific driver routine for unit testing.
- All CÆSAR files are filtered through the **gm4** macro preprocessor.
- Unless the UNITTEST flag is set, the Unit Test driver routine is filtered out.

Example:

```
module Template_Class

  ! Module data and routines.

end module Template_Class

ifdef([UNITTEST],[

program Unittest

  ! Testing code.

end

])
```

Unit Testing Implementation (cont)

- Each component to be unit tested must be compiled and linked with a unique subset of CÆSAR.
- The **`make`** utility is not well suited to this task.
- CÆSAR uses **`Document`** to extract and run a unit test script imbedded in each component.

Example:

```
! To test this module,
!
! Begin_Self_Test
!   % echo "Preprocessing unit test on Template..."
!   % m4 -P -I../include ../constants/numbers.F90 >  unittest.f90
!   % m4 -P -I../include   ../constants/flags.F90 >> unittest.f90
!   % m4 -P -I../include      ../debug/verify.F90 >> unittest.f90
!   % m4 -P -I../include              logical.F90 >> unittest.f90
!   % m4 -P -DUNITTEST -I../include  template.F90 >> unittest.f90
!   % echo "Compiling unit test on Template..."
!   % f90 unittest.f90 -w -o unittest
!   % echo "Running unit test on Template..."
!   % unittest > battery/template.test.new 2>&1
!   % rm -f unittest*
!   % echo "Diffing Template results with saved version..."
!   % diffnewold battery/template.test.new battery/template.test
! End_Self_Test
```

Design By Contract / Verification

Basic Idea of Verification:

Statements that verify that specified conditions are true are conditionally compiled into the code, allowing error checking that can be turned off completely for fast execution.

Basic Idea of Design by Contract:

Routines satisfy a contract when they are called – input requirements are verified upon entry and output guarantees are verified prior to exit.

These are very simple, but very powerful ideas. Unfortunately, the main proponents of these ideas (Eiffel and C++) use bad nomenclature.

Here's a translation table, so you'll recognize these ideas in other venues:

Eiffel or C++	English
assert	verify
precondition	requirement
postcondition	guarantee
class invariant	valid state
require	verify (on routine entry)
ensure	verify (on routine exit)

Verification Implementation

Verification is implemented via `gm4` macros.

Command syntax is:

```
VERIFY(<logical expression>, <activation level>)
```

where

`<logical expression>` is the test to be satisfied.

`<activation level>` is the value of the `gm4` variable `DEBUG_LEVEL` which is necessary to activate the verification.

For example, if a file named `example.F90` contains:

```
VERIFY(i < 1, 1)                  ← on line 46
VERIFY(Valid_State(matrix), 5)    ← on line 92
```

and it is processed by `gm4 -DDEBUG_LEVEL=3`, then:

```
 if (.not.(i < 1)) &
   call Verify_Out ("i < 1", &
   "example.F90", 46, .true.)
! if (.not.(Valid_State(matrix))) &
!   call Verify_Out ("Valid_State(matrix)", &
!   "example.F90", 92, .true.)
```

Aside: Valid_State is an F90 logical function which is defined for every variable type and dispatched polymorphically (both at compile time and dynamically).

Verification Implementation (cont)

If the **`Verify_Out`** routine is called, it prints

```
Verification failed: i < 1, file example.F90, line 46.
```

and terminates the program.

A similar **`gm4`** macro is called **`WARN_IF`**. It is controlled by the **`WARNING_LEVEL`** **`gm4`** variable. In contrast to the **`VERIFY`** macro, **`WARN_IF`** prints

```
Warning - test failed: i < 1, file example.F90, line 46.
```

and *continues execution*.

Note that this implementation of the verification idea allows for extreme error checking if the tests are compiled in and unfettered execution speed if they are commented out.

Design By Contract Implementation

Design by Contract does nothing more than specify where and what to verify. For example:

```
subroutine Quadratic_Roots (a, b, c, root1, root2)

  ! Input variables.
  type(real), intent(in) :: a, b, c       ! Equation coefficients.

  ! Output variables.
  type(real), intent(out) :: root1, root2 ! Roots of the equation.

  ! Internal variable.
  type(real) :: determ                    ! Determinant of the equation.

  ! Verify requirements.

  VERIFY(Valid_State(a),1)  ! The equation coefficients can
  VERIFY(Valid_State(b),1)  ! take on any real value, but
  VERIFY(Valid_State(c),1)  ! we can check for NaNs & Infs.

  ! Calculate roots.

  determ = b**2 - 4.d0*a*c
  VERIFY(determ>=0.d0,1)
  determ = sqrt(determ)
  root1 = (-b + determ)/(2.*a)
  root2 = (-b - determ)/(2.*a)

  ! Verify guarantees.

  VERIFY(Valid_State(root1),1)  ! The roots can take on any real
  VERIFY(Valid_State(root2),1)  ! value, so only test Valid_State.

  return
end subroutine Quadratic_Roots
```

Aside: type(real) is a `gm4` macro for the F90 intrinsic real type.

Summary

The CÆSAR 3-T photonics package employs many of the latest ideas in software design:

- Literate Programming - source and documentation stored together.
- The **`Document`** Package is used to extract documentation from code source, which is processed by LaTeX into hardcopy and LaTeX2**`HTML`** into hyperlinked HTML.
- A Levelized Design is used to facilitate Unit Testing, which is accomplished using the **`gm4`** preprocessor and the self-test feature of the **`Document`** Package.
- Verification **`gm4`** macros are used to implement Design By Contract.