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


  which is an extension of


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

- Support Operator Symmetric Method:
  - Described in


  which is an extension of


  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:

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

- 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:

```plaintext
! 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:

```latex
\begin{Doc}
\% Note that the order of files a.tex and b.tex has been switched.
\input{b}
\input{a}
\end{Doc}

\begin{Doc}
This line is in file a.tex.
\end{Doc}

\begin{Doc}
This line is in file b.tex.
\end{Doc}

\begin{Doc}
This line is appended to file a.tex.
\end{Doc}
```

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

```latex
\begin{SelfDocumentation}
% mv file1 file2
% f90 file.f90
% Document file.f90
\end{SelfDocumentation}
```

which executes commands included in the file itself.
CÆSAR Documentation

Making use of the capabilities of Document, LATEX and LATEX2HTML, 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)
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

Level 4:
- Host Code or Driver

Level 3:
- Problem Definition
- Problem Solution
- Equation

Level 2:
- Timestep Control
- Boundary and Initial Conditions
- Term Library
- Mesh
- Matrix

Level 1:
- Material Properties
- Intrinsic Library
- Communication Library

Level 0:
- Linear Solver Package
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:

```plaintext
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 Caesar.

- The make utility is not well suited to this task.

- Caesar uses Document to extract and run a unit test script imbedded in each component.

Example:

```bash
! 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:

<table>
<thead>
<tr>
<th>Eiffel or C++</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>assert</td>
<td>verify</td>
</tr>
<tr>
<td>precondition</td>
<td>requirement</td>
</tr>
<tr>
<td>postcondition</td>
<td>guarantee</td>
</tr>
<tr>
<td>class invariant</td>
<td>valid state</td>
</tr>
<tr>
<td>require</td>
<td>verify (on routine entry)</td>
</tr>
<tr>
<td>ensure</td>
<td>verify (on routine exit)</td>
</tr>
</tbody>
</table>

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

Verification is implemented via gm4 macros.

Command syntax is:

\[ \text{VERIFY}(\text{<logical expression>}, \text{<activation level>}) \]

where

\text{<logical expression>} is the test to be satisfied.

\text{<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:

\[ \text{VERIFY}(i < 1, 1) \quad \text{← on line 46} \]
\[ \text{VERIFY}(\text{Valid}_\text{State}(\text{matrix}), 5) \quad \text{← on line 92} \]

and it is processed by \text{gm4} -DDEBUG_LEVEL=3, then:

\[ \text{if (.not.}(i < 1)) \text{ &} \]
\[ \quad \text{call Verify\_Out}("i < 1", \text{ &} \]
\[ \quad \quad \text{"example.F90", 46, .true.}) \]
\[ \text{! if (.not.}(\text{Valid\_State}(\text{matrix}))) \text{ &} \]
\[ \quad \text{! call Verify\_Out}("\text{Valid\_State}(\text{matrix})", \text{ &} \]
\[ \quad \quad \text{"example.F90", 92, .true.}) \]

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

```fortran
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 \LaTeX{}2\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**.