Practical Software Design & Style

Practical Software Design & Style

'Computational science has to develop the same professional integrity as theoretical and experimental science', Douglas Post, LANL

Software Design - what and who?

- Requirements: what (not how)
- Users: You, others in the Group, others in the field...
- Longevity: quick project? Your PhD? The next major code for... Remember RCUK requirements!

Design

Start with a blank piece of paper, not a blank file

Philosophy

- Decide what your program will do
- Design it to be tested
- Design the data flow
- Write the broad structure
    - High-level (physics)
    - Medium-level (data)
    - Low-level (infrastructure)
- What exists already?

Lessons from Unix (from Eric Steven Raymond)

Core design principles

- Modularity: simple parts connected by clean interfaces.
- Clarity: clarity is better than cleverness.
- Simplicity: design for simplicity; add complexity only where you must.
- Transparency: design to be comprehensible (helps reading & debugging).
- Robustness: robustness is the child of transparency and simplicity.
- Least Surprise: code should always do the least surprising thing.
- Silence: when a program has nothing surprising to say, it should say nothing.
- Repair: when you *must* fail, fail noisily and as soon as possible.
- Extensibility: design for the future; it's sooner than you think!
- Representation: fold knowledge into data so logic can be stupid and robust.

Design considerations (more debatable!)

- Composition: design programs to be connected to other programs.
- Separation: separate policy from mechanism; interfaces from engines.
- Economy: your time is expensive, conserve it (in preference to machine time).
- Generation: where feasible, write programs to write programs.
- Optimisation: prototype before polishing - get it working first!
- Diversity: distrust all claims for “one true way”.

Algorithms

'When in doubt, use brute force', Ken Thompson (Unix creator)

Me vs the world

- Does my program do something new?
- If a good implementation exists, use it

  Portable? Robust? Fast?

Fancy vs plain; fancy is. . .

- Often only better for large problems
- Complex to code
- Fewer reference implementations
- Prone to bugs

Personal philosophy

Showing off

Don't! Code to be readable. Think about 'reading age' - beware:

- New language features
- Golfing (be expressive)
- Overloading operators
- Confusing syntax

  E.g. Fortran arrays vs functions
- Object orientation is a double-edged sword
  - encourages good encapsulation
  - can simplify code & coding greatly
  - is inherently complex
  - hides operations
  - may have hidden performance & storage costs

Don't be a 'Real programmer'

Real programmers?

- Real Programmers don't write specifications
  Users should consider themselves lucky to get any programs at all, and take what they get.
- Real Programmers don't comment their code
  If it was hard to write, it should be hard to read.
- Real Programmers don't do documentation
  Documentation is for numpties who can't figure it out from the source code.
- Real Programs never work right the first time
  Just throw them on the machine; they can be patched into working in "just a few" all-night debugging sessions.

Language

New vs Old

- Small and quick: write what you know.
- Longer: think about best language.
  Speed of writing, speed of running, no. of bugs, complexity, maintainability...
- ASCI Complexity metric,

$$FP = \left(\frac{C{+}{+}}{53} + \frac{C}{128} + \frac{F77}{107}\right)$$

Duration $= 1.6 \times FP^{0.5}$
Team required $\frac{FP}{150}$
Bugs as $FP^{1.25}$
Documentation as $FP^{1.15}$

Naming is important

'[God] brought [the animals] to the man to see what he would name them; and whatever the man called each living creature, that was its name.' (Genesis 2:19b)

Consistency

- There are lots of different conventions to naming things
- Pick something and stick to it (i.e. be consistent)
  If you use a particular synonym or abbreviation (e.g. "calc" for "calculate") then stick to it. Try to avoid mixtures like:
  - calc_density
  - velocity_calculate
  - flux_computation
- Generally: nouns for variables, verbs for functions.

Variables

- Think about what you need to know about a variable; perhaps:
  - What is it physically (e.g. particle density)?
  - What is it computationally (e.g. array of reals, derived-type, Object. . .)?
  - Where is it defined?
- Often end up with names comprised of several words, e.g. “particle density”.
  - **snake case**: particle_density (Perl & Python; C & C++ standard libraries)
  - **camel case**: particleDensity (lower, camelCase; Microsoft) or ParticleDensity (upper, CamelCase; Pascal case)
  - **train case**: particle-density (not supported by many languages; Lisp case)
- Sometimes use different naming style for different things, e.g. functions use one style and variables use another.
- Avoid cryptic abbreviations (e.g. cptwfp).

Data

Separation

- Keep code and data separate
- Read from input, don't hard-code

Access control

- Think: who 'owns' this data?
- Try not to change data you don't 'own'
- Consider restricting access (private data)

Encapsulation

- Keep related data together (derived types, Objects)

```
type, public :: wavefunction
    complex(kind=dp), dimension(:,:,:,:), allocatable :: coeffs
    integer                                           :: nbands
    integer                                           :: nkpts
    integer                                           :: nspins
end type wavefunction
```

Functions and subroutines

Operation

- Clear purpose
- No side-effects
  (Or minimise and *document*)
- Error checking and propagation
  Check for errors in inputs, optionally return error status.
- Single entry and exit points
  (Except for trivial checks with early exit?)
- Clear API
  ... and *consistent*
- Document it

Lessons from projects

Accelerated strategic computing initiative (ASCI)

- Create predictive simulation codes for nuclear weapons research.
- ~ $6B from 1996-2004.
- Successful projects tended to emphasise:
  - Building on successful code development history and prototypes
  - User focus
  - Better physics/mathematics more important than better “computer science”
  - Modern *but proven* Computer Science techniques,
  - They don’t make the code project a Computer Science research project
  - Software Quality Engineering: Best Practices rather than Processes
  - Validation and Verification
- Unsuccessful projects… didn’t.

Lessons from projects

'Employ modern computer science techniques, but don't do computer science research' Douglas Post, LANL

Accelerated strategic computing initiative (ASCI)

- Main value of the project is improved science (e.g. physics and maths)
- LANL spent over 50% of its code development resources on a project that had a major computer science *research* component. It was a massive failure (~$100M).
- "Best practices" better than "Good processes"

CASTEP Design History

Aim: Quantum mechanical simulation of materials

Ancient history

- Written in F77 in 1980s by Mike Payne; added to by PhDs & postdocs
  - F90 fork by Matt Probert
  - Metals simulation fork by Nicola Marzari
- Parallelised by Lyndon Clarke
  - CETEP (F77 + MPI)
  - F90 fork by Matt Segall
  - Metals F90 fork by Phil Hasnip
- 20 kLOC F77
- Very difficult to maintain
- Separate commercial codebase (100 kLOC F77 + F90 + C + MPI)

CASTEP Design History

Not-so-ancient history

- End of 1990s:
  - difficult to maintain
  - ‘impossible’ to add new features
- 1999 form CASTEP Developers Group (6 people)
- Write a Design Specification
  - F90 + MPI
  - Metals and insulators
- 2000 start coding low-level modules
- 2001 commercial release

CASTEP Design History

Then

- About 250 kLOCs
- ASCI metrics (actual): - FP = 2800 - Team size 16 (6) - Duration 77 PYs (12)

Now

- F2003 (with some F2008)
- Single codebase (serial/parallel, academic/commercial)
- 600 kLOC
- Actively maintained and developed

CASTEP Design

Style

- Derived-types and encapsulation, but not Objects
- Allocatable arrays, not pointers
  (Performance and readability)
- No hand-optimisations
  If the compiler should do it, let it (and file bug reports when it doesn't!)

Naming

- Modules defined in file of same name
- Main derived data types defined in modules
- Operations on main derived data types in modules
- Functions & subroutines start with module name

CASTEP Design

Code blocks

- Functions
  For short, well-defined operations that could in principle be in-lined
- Subroutines
  Single entry point, single main exit point though early exit allowed if arguments mean no work is required (e.g. data length of zero).
- Argument lists always ordered: "inputs, outputs, optional"
- Standard header to say:
  - What it does
  - What the arguments are
  - Key modules it uses
  - Any known shortcomings
  - Who wrote it and when

CASTEP Design Failings

Naming inconsistencies

- Different orderings:
  - `calculate_stress`
  - `popn_calculate`
- Different abbreviations:
  - `calc_molecular_dipole`
  - `phonon_calculate_dos`

CASTEP Design Failings

Missing low-level types and methods

- Defined physical objects, e.g. potentials and densities
- Implemented as arrays, e.g. complex values on grid
  - No low-level ‘grid’ types
  - Duplicate operations for potential and density grids
  - New ‘grid’ things need a completely new type and code, or misuse potential or density

CASTEP Design Failings

Encapsulation

- Some data is strongly related to more than one physical object

Where should it live?

- E.g. eigenvector equation $H\Psi_b = E_b\Psi_b$
  - Is E_b a property of H, Ψ or a separate object?
  - What about something which depends *upon* E_b?

Summary

- Think before coding!
  - What, why, who for, and *then* how
  - How much is new?
  - How will you know it's working?
- Stick to your design
- Be consistent
- What about surprising input?

References

http://www.catb.org/~esr/writings/taoup/html/ch01s06.html

http://www.csm.ornl.gov/meetings/SCNEworkshop/Post-IV.pdf

http://www.castep.org