Software Design and Development

Introduction and Principles

Nick Maclaren

nmm1@cam.ac.uk

October 2019

Apologia

There is too much for one afternoon, or even two
So it is a selected subset of just the principles
I need to assume too much background for more

- See the notes for a lot more information

Please say if more details would be useful
And WHAT sort of details you want!

Purpose

Not a complete software design course
That would occupy the whole MPhil on its own!

Too complicated to be directly examinable

- But you are expected to use it appropriately

- Remember that the MPhil has a purpose

You are also learning skills that you will need later
Whether writing research code or commercial

Software Engineering

Software engineering is a bit of a catch−all term
The skills you need to write high−quality code

A good quote:

The difference between theory and practice
Is less in theory than it is in practice

- This course is practical software engineering

You will get occasional references to the theory

Computer science courses are often poor on this
Partly explains why so much software is so unreliable

Saving Time

- Scientific Computing in a year is tight
A main purpose is to minimise your wasted time
Learning from mistakes would take too long

- Most of the techniques often save time
The course will describe how, why and when
- But all of them will waste time if over–used

- It is your task to select what to use
That is one of the objectives of a graduate course

Please Note

Assumes fairly experienced programmers
This course does not teach basic programming

May use examples from several languages
However, you need be able to program in only one

- Please interrupt if you don't understand

It mentions techniques, but has few details
Those depend on language and requirements

- Contact your supervisor if you have trouble

Languages

Most principles are the same for all of them

From assembler to Pascal to Fortran to C++
to Matlab to Excel to LaTeX to XML to . . .

The details vary immensely . . .
This course is about the principles

Python and Matlab are the safest languages
C/C++, Perl and TeX/LaTeX the least safe

More Information (1)

Not covering everything – full materials are in:
Full course materials are in:
https://www-internal.lsc.phy.cam.ac.uk/nmm1/
Handouts are fairly complete

Includes other relevant courses, some mentioned
They are all "transferrable skills" courses
Not part of this MPhil, so get no credit
Relevant mainly if you need to learn the skill

More Information (2)

Few books are much good, and some are ghastly
May push some dogma or even be provably wrong
The following is one of the best (despite its flaws):

McConnell, Steve (2004). Code complete: a practical handbook of software construction, 2nd edition.
Do NOT use the 1st edition – it's badly flawed

Most of it is good advice, and it covers a lot
But its coding conventions are merely one of many

I checked it fairly briefly, and noted the following:

More Information (3)

- Far too kind to C and derived languages
You need to defend yourself against the languages

- Book implies most debugging stops with shipping
But bugs found in actual use take up most time
Some aspects of this are described later in this course

- Chapters 25, 26 on tuning are out of the ark
Don't hack code by hand – increase the optimisation!
To improve that, you simplify and clean up your code

Overview of Course

The development cycle and design principles

Documentation, consistency and interfaces

Checking, validation, tracing and debugging

Computer arithmetic (integer and floating–point)

Languages, and parallel models and design

KISS

KISS means Keep It Simple and Stupid
Kelly Johnson, lead engineer at The Skunkworks
Often misquoted as Keep It Simple, Stupid

- Ancient engineering principle of great worth

The simplest workable solution is usually best

C.A.R. Hoare has coined similar aphorisms, too

Debugging? What's That?

Best solution is not to make mistakes

- Careful design/coding helps – little else

Will cover some of this aspect

Finding errors automatically before use

- Stricter languages can help with this

But most debugging needs testing on data
Or is when the program goes wrong in use

- Course concentrates on this aspect

Run-Time Debugging

Can design in semi–automatic debugging

- Maximise chance of catching errors early
- Produce helpful diagnostics on error

Can help with (tedious!) manual debugging

- Produce targetted, comprehensible tracing
- Checking/diagnostic functions when needed

Much of the course will target these aspects
Aim is to improve debugging effectiveness

Aside: Optimisation

- Always try to debug with target optimisation
Some checks are done as part of optimisation
Many bugs show up only in optimised code

- Particularly true for C and C++
Most ‘optimiser bugs’ are breaches of standard

You sometimes have to drop optimisation
Some compilers don’t support it with –g at all

- Avoid running unoptimised more than necessary

Development in Academia

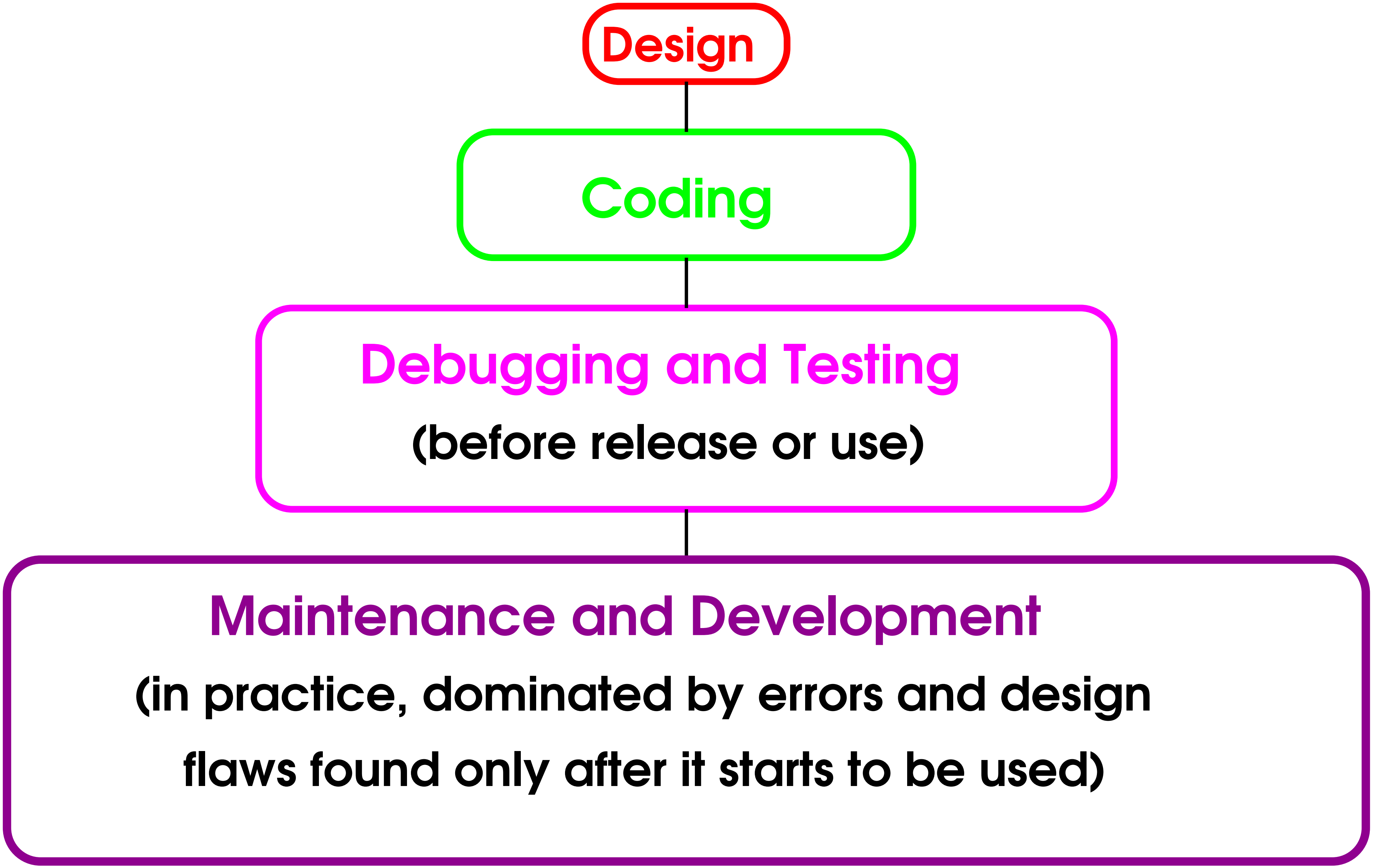
Typically design phase is neglected
Coding begins at the keyboard

- But debugging takes longer than either

And most debugging occurs in actual use

Has been measured at 10–100 times as much
The next slide is not an exaggeration
The effort is proportional to the area

Effort Involved (1)



Managed Development

- Not talking toolkits – see later for them

Ditto for make and source control systems

More effort spent in design phase

- Typically 3–10 times as much

Code includes internal checks/diagnostics

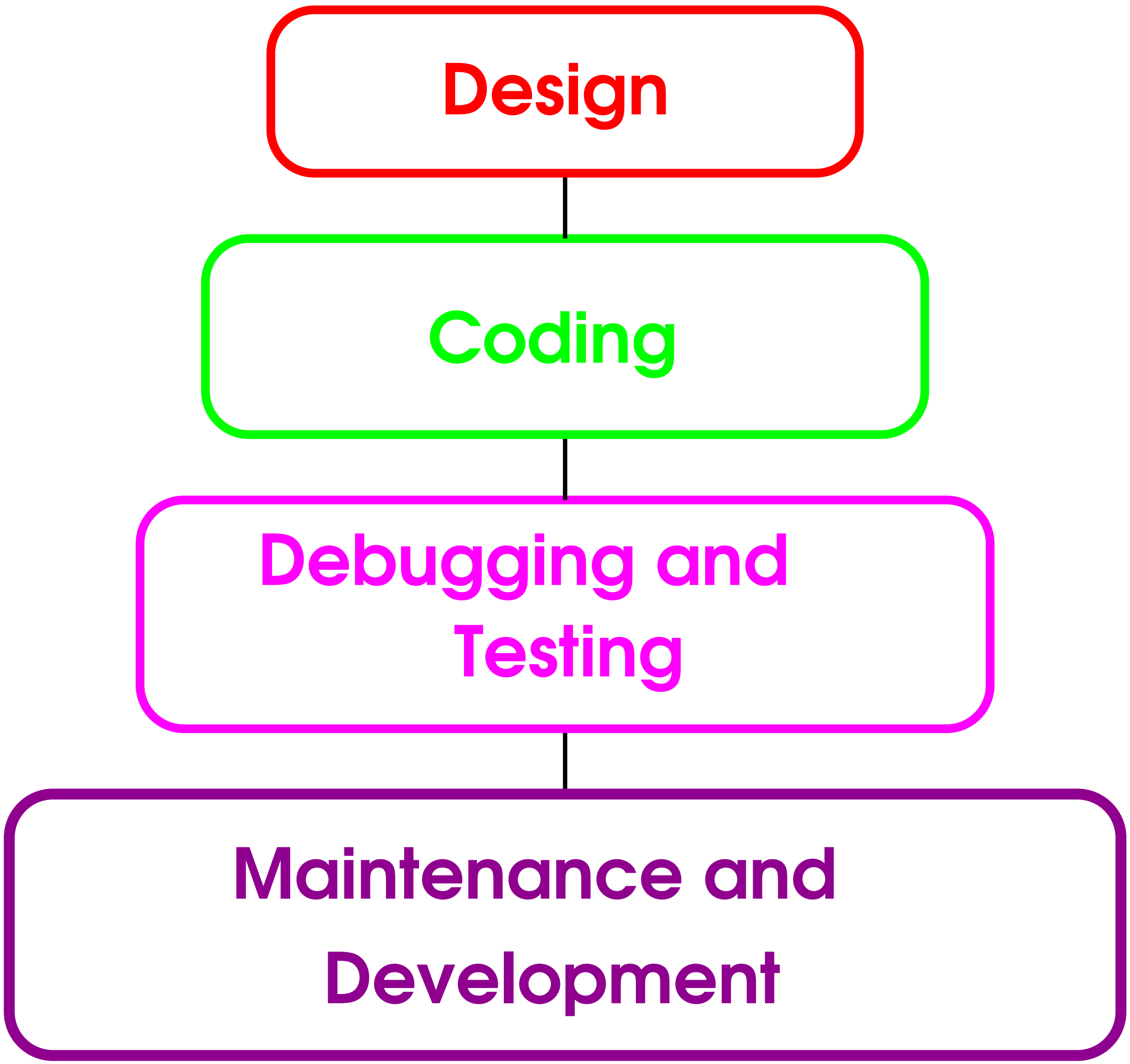
- Takes perhaps 50% longer to write

Initial debugging is often much slower

- You have to debug the internal checks!

Overall effort can be 2–5 times less

Effort Involved (2)



Taken to Extremes

Can prove the correctness of the design
Can almost prove code implements the design
IBM Hursley used Z to do that for CICS
Following figures are from (20 year) memory:

Design took 3 times longer than average
Completed development ahead of schedule
Bug reports were 5 times below average
Total project cost was 30% below target

Let’s Get Real

You and I very often code at the keyboard
Fix the syntax errors, and . . . Oops!
Then fix simple errors, tediously
Problem occurs with first difficult error

- Often worthwhile to go back a step

Code and insert proper checking features
The difficult error often becomes easy

Checking may double time to first complete run
AND halve total time until it mostly works!

Software Reuse

A.k.a. “Don’t reinvent the wheel”

Means using existing libraries etc., not writing own

Currently almost a mantra, especially in C++ area

- A very good principle, but a very bad dogma

You are adding a dependency on what you use

- Generally, start by reusing and change if needed

Makes program development quicker and easier

But think about it for production code

Questions

The following are some of the questions to ask

- Will it be simpler and cleaner, or not?
- Will it be more reliable, or less so?
- Will it be more portable, or less so?
- Will it be more maintainable, or less so?
- Will it be more efficient, or less so?

Which ones depend mainly on your requirements
Your skill is a secondary consideration – seriously

When to Reuse (1)

In these cases, you should almost always reuse
But don't include the source in your program
Use the latest, most improved version when building

- When there is a standard and stable interface

Usually choice of software, and no changes needed
E.g. BLAS, LAPACK, simple use of C++ library, ...

- Or reliable, portable and stable software

E.g. NAG, FFTW, PCRE, ...

When to Reuse (2)

In these cases, you should usually reuse
But watch out for maintenance and reliability etc.

- When your system has a library that does the job

Or a reasonably well–managed software project
E.g. MKL, ACML, Boost, ...
Advanced use of C++ library also comes here

- When there is suitable open source to include

Provided that the copyright conditions are OK
E.g. most of Netlib, some of the above, ...

When NOT to Reuse

$\Rightarrow$ Even here, start by trying to reuse
It's a good way to get a first version running

- When the software doesn't do what you need to do

AND extending it is more complicated than coding it

- When you need a high level of portability
AND the software is too system specific

- When it simply doesn't work on your data
AND you are sure it isn't a bug in your code

Consistency of Style

- A consistent style is a very important tool

One purpose of NAGWare, GNU indent etc.

Tell what code does at first glance

What it will NOT do – and can trust that

- Almost–consistency can be worse than none

You can use more than one style in a program

- Provided that the boundaries are clear

Instrumentation

- Consistency of style helps instrumentation

E.g. can add tracing code automatically
Or can put wrappers around library calls

Roughly parsing Fortran is almost trivial

Minimal C/C++ parser is gcc's front–end

- But can be very simple on consistent code

Best tools are Python and (if you know it) Perl
For simple tasks, awk and even grep/sed

Consistency of Semantics

- Biggest gain is consistency of semantics

Same construct means the same everywhere

E.g. what does positive definite matrix mean?
Does it include approximately semi–definite ones?
Or that min(eigenval) > eps*max(eigenval)?

If components A and B interact,
they had better assume the same meaning

- Failure is major cause of hard problems

Documentation and Specifications

- Do not underestimate their importance

Rarely help when shaking initial bugs out

- Benefit comes from then onwards

Will your program be in use a decade hence?
Or will you get a collaborator/assistant?

And examiners don't like analysing code!

- Make it clear what you are doing and why

Basic Guidelines

- Sole criteria are complete and correct

When you update code, fix the documentation

- If you can't, then SAY so!

/* WARNING: comments are for release 1.3 */

But please try to avoid doing that

Separate specifications or block comments?

- Latter are a little easier to keep in step

More on this in notes

Integrated Documentation

Methods to integrate source and documentation
I don't like them, but some people do
Technique dates from 1960s, in many forms

Look at doxygen, CWEB and others
See also Wikipedia "literate programming"

- If you find one suits you, why not use it?
If you don't, why add to your difficulties?

Reverse Engineering

Without documentation, have little option
Even on your own code, years later

- Can be incredibly time−consuming
Often increases debugging time tenfold

Obviously, good documentation takes time
- Generally, best balance is more of it
In many cases, it should be longer than code!

No, I am not exaggerating there

Top-Level Specifications

Can use block comments or separate file

- What program is supposed to do
- References to algorithms/formulae/etc.
- Possibly the resource usage and complexity

- Its input format and constraints
- Its output and its guarantees
- Roughly what it intends to diagnose
- What it assumes but does not check

And anything else of that nature

Why is it Critical?

When (not IF) a program fails, obscurely

Reminder of which assumptions to check

- Half of failures are false assumptions

Decide between simple bug and data error

Helps to know where/how to fix the problem

- Fixing bug wrongly wastes a lot of time

E.g. is performance problem a bug or feature?

Detailed Commenting

Helps to keep your own mind clear

- Absolutely critical a decade later
- Or when someone else modifies the code

Component A creates a symmetric real matrix
Assumes a variance matrix, so always positive
Component B divides by its determinant

This then fails on an indefinite matrix

- Which of A or B needs fixing?

Low-Level Commenting

Simple code needs very little of this

Old assembler rule was comment every line
Exercise in futility set by born bureaucrats
ADD R1,=1 Add one to register one

Dogma said Pascal etc. are self–commenting

- Complete and utter twaddle, too

- Think of what you want comments FOR

Finding Your Way Around

- Introduce significant blocks of code
Describe purpose of procedures and data

Call tree information can be very useful, too
Where procedures are called from and go
- Pain the neck to maintain, so rarely done

Use for locating code or data more easily
Details are entirely a matter of taste
- Do whatever speeds up your debugging

Describing Pitfalls

MUCH the most important low–level comments

- Reminds you not to make same mistake twice
- Documents assumptions that may break later
- Documents horrible and unobvious hacks

```
! This code assumes binary floating-point
C = P*A+(1.0-P)*B
```

```
/* Casts added (and needed!) for C99 - sigh */
A = (double)((double)((B*C)+D))-D;
```

Identifier Names

- Remember to use appropriate identifier names
Especially for ones used by separate components

- Longer names help to avoid name clashes
A common cause of obscure errors

- And make your code much easier to read
E.g. use same names as in referenced paper
Also velocity usually clearer than V
Or Cholesky_solver() instead of solution()

Keep simple names (e.g. A) for local scratch use

My Experience

Good commenting can slow coding by 25%
Rarely speeds up initial debugging much

- Even when I was 30, it helped a month later
- Often speeded up by 2+ times a year later
- And even more on other people's code

Overall, in research, effort repays (say) 3:1

- It can pay 10:1 for production code

Program Components

All large programs should be subdivided
Even if language has no formal modules

- Document components as for programs!

- All of the above advantages and more
- Critical for designing internal interfaces

If it is too complex to document,
will you be able to use it correctly?

- You will NEVER manage to debug it!

Program Structure

- Break programs up into components

Simple and small enough to understand

But mincing into hundreds of tiny pieces is also bad

- Use modules if language supports them

And some sort of equivalent if not

- Do the same to data structures and types

If they are independent, then separate them

Keep closely related things together

Lower-level Components

At least for the major procedures:

- Document purpose and interface, precisely

Sometimes obvious from context, usually not

- Its input format and constraints
- Its output and its guarantees
- Roughly what it intends to diagnose
- What it assumes but does not check

Exactly like programs, at a lower level

Trivial Example

```
FUNCTION DET (MAT)
USE MATRIX
DOUBLE :: DET
TYPE(SYMMAT), INTENT(IN) :: MAT

! MAT must be positive semi–definite
! Returns –1.0 for invalid matrix
! Returns BIGNUM on overflow
```

Objects and Sets of Data

- Treat object types as components
Also any set of data handled together

- Document what their function is, precisely
Sometimes obvious from context, usually not

- Any limits or constraints assumed
- Any invariants that are preserved
- What interface procedures are provided
- Any other forms of access allowed

Trivial Example

```
typedef struct {
        int size;
        double sum, *values;
} vector;

/* A basic vector of reals
size must be >= 0
fabs(value[i]) < BIGNUM
sum is total of values, within rounding
See tools.h for access functions
*/
```

What Are Interfaces?

- Any way of passing data or control
Network interfaces, routine calls, files
Specification of data structures or objects
Anywhere component A meets component B

- Guidelines apply at all levels

Commonly objects or modules or procedures
Also suites of programs operating on files
And other levels, higher and lower

Data Interfaces

Procedural languages make actions primary
You pass data to procedures to act on it

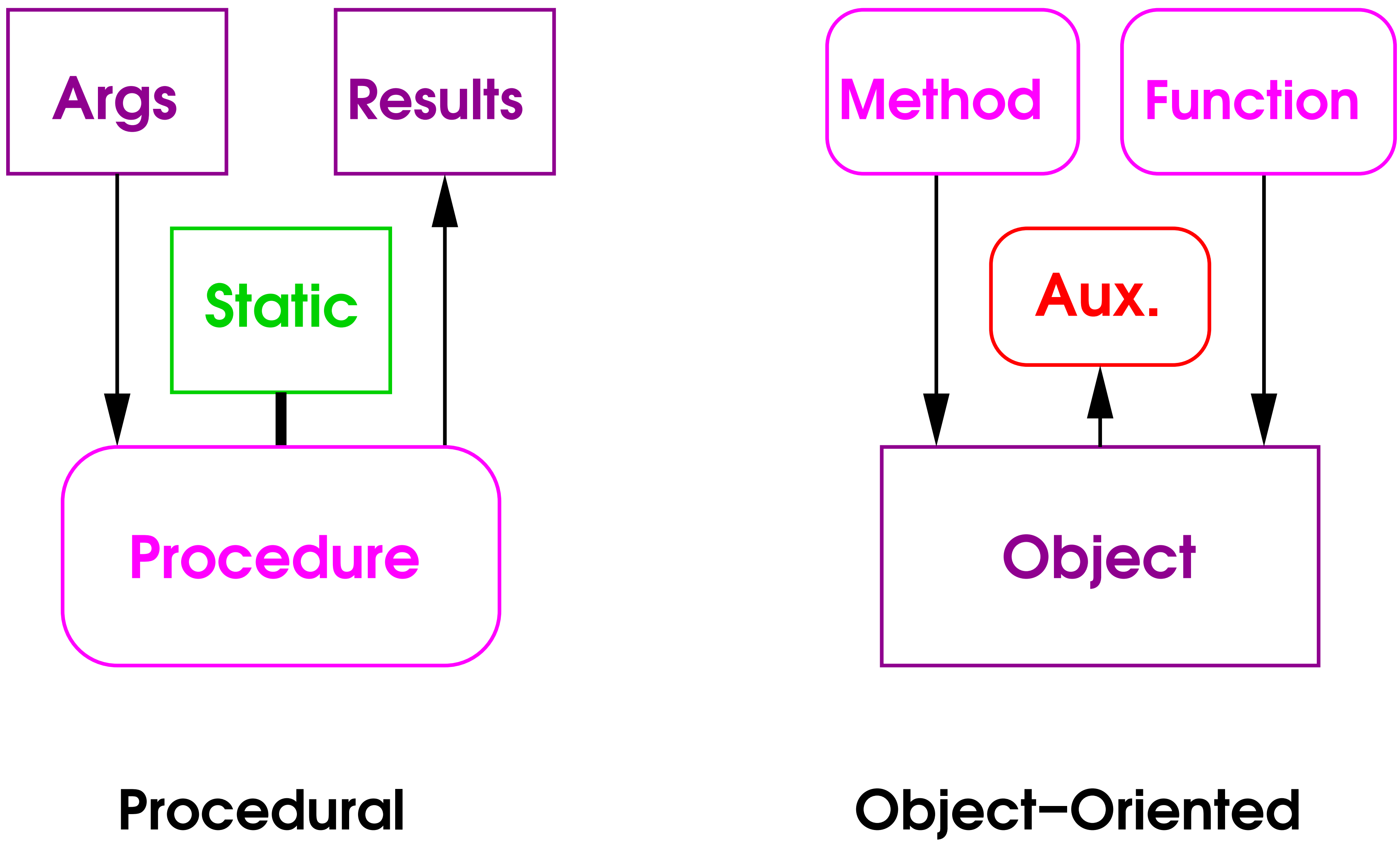
'Object–oriented' ones do the converse
You apply actions to data structures

- Think in terms of interfaces to data/code

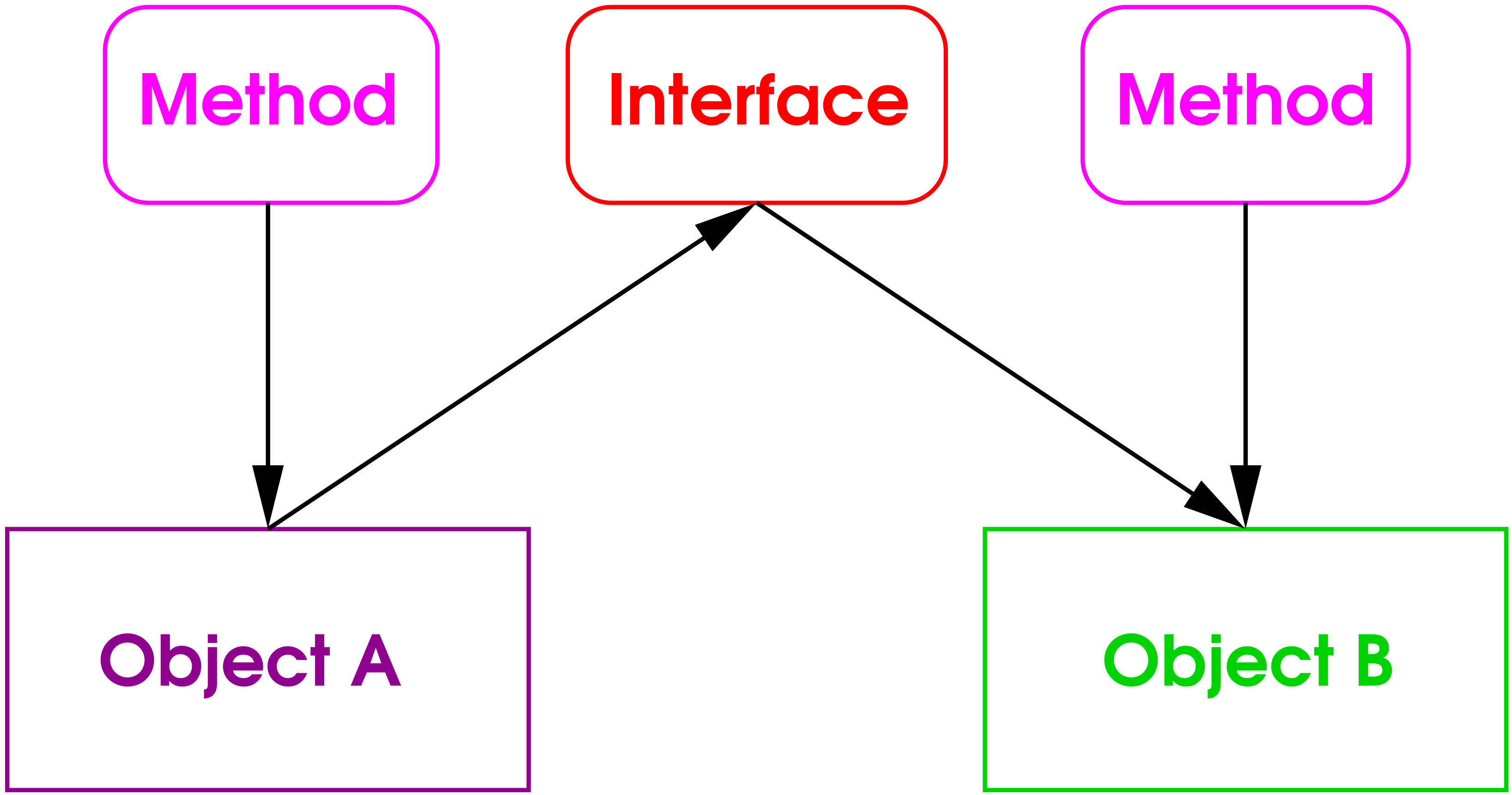
Write and use conversion or access functions

Avoid exporting internals for other uses

Difference Between Models



O–O Data Interfaces



Debugging is About Interfaces

But . . . WHY?

- Most serious bugs occur at interfaces

People forget what they were assuming earlier
Most common error made by experienced coders

- Languages may not allow interface checking
- Compilers rarely do it even if they could

NAG Fortran, Python are best common ones

- KISS is more relevant here than anywhere else

Much more on this in notes

Interfaces, Generally

- KISS is more relevant here than anywhere else

- Keep interface concepts simple
Clever designs often hide "gotchas"
Are you sure no inconsistencies lurking?

- Minimise complicated interactions
Especially multi–way, and long–term

Much more on this in notes

Avoid Updatable Data Objects

- Ideally, keep data simplex – input OR output
What was value before the action failed?

- Updatable data is a real pain
Say, procedure fails during millionth use
Need to know value that triggered failure

- Same issue for updatable files
Better to separate input and output files

- But sometimes you need to update data
Many matrix algorithms are like that

Avoid Context-Dependence

- Don’t make interpretation context–dependent

E.g. using one matrix for two different purposes
Or use several unit systems – a classic mistake

- Avoid unobvious/unspecified side–effects

Not *just* updating of global data

- Includes updating of environment files

- More complicated ≡ harder to debug

State Changes (1)

State changing is rarer, but can be EVIL
POSIX signal handling is one example
IEEE 754 floating–point handling is another

- Reset properly before leaving module

Not just returning, but calling others

- Implicit calls in C++ etc. are nasty

- Same applies in suite of programs

They often keep their state in files
Think of CVS, Web browsers, GUIs etc.

State Changes (2)

- Avoid global state changes if you can
Debugging them is usually a foul task

- Remember problems caused by failure
Often won't have cleaned–up correctly

- Need a "restore clean state" primitive
Most such primitives are too half–hearted
A few always destroy too much data

- Easiest to change only at start and finish

Encapsulation (1)

- Most useful technique of all

Can speed up debugging by a large factor

ALL access is through defined interfaces
Usually via procedures kept in a module
May provide extraction/insertion primitives

- Can also encapsulate only updates to data

Data exported read–only, using ‘direct’ methods

Encapsulation (2)

- You know where to start when data goes bad
- Provides place to add checking/tracing

Also allows changing internals easily

- Can be applied to a data type (class)

Basic principle of object–orientation

- Object internals known to few components

All other code uses exported interfaces

Encapsulation (3)

- Can apply to any data or interface

Objects, global/static data, system state
File I/O, user interface, memory management
Application–specific components or state
Device control, networking, GUI use

Above approach helps with hard problems

- It will not solve all such problems

Unset indices/pointers can trash anything
So can using subtly wrong command on a file!

Procedure/Module Interfaces

Multiple simple ones better than complex
There is a very relevent acronym: TANSTAAFL
There Ain't No Such Thing As A Free Lunch
More components mean more interactions

- Interactions are part of interface, too!

POSIX (all of them) get this very badly wrong

- Don't think just in terms of global data
- Any interacting constraints and assumptions

Such as A guarantees what B assumes

- Remember application's own state changes

Arguments and Globals

- Most languages use very poor data model

Properties of structure etc. apply at one level
Not helpful for debugging or parallelism

- Properties should apply recursively

Read–only args refer only to read–only data
If this is true, debugging is much simpler

Not always possible, unfortunately

- Minimise places where it is not so
- Make them explicit and document them

Global/Static Data

Not as unclean as traditional dogma claims

- Worst problems are pointer aliasing and scoping

Very common causes of hard–to–locate problems

- Not safe to cache any argument pointer

Applies even in languages like Python

Exceptions do exist, but be very careful

Remember that means everything referred to

- If in doubt, copy data – if possible

Watch out for shallow/deep copying problems

Procedure Interfaces

- Use pure functional when possible

NO side–effects, NO updated arguments
Includes all data pointed to by arguments

- Use pure output arguments when needed

Can copy/alias pure input into them, safely

- Use encapsulated static data if needed

Beyond that, debugging becomes rapidly harder

Argument Properties

Best to keep to single purpose

- Read–only input, not updated during use
- Pure output, written only at end
- Workspace, undefined at entry and exit

Document which component allocates their space
Similarly for deallocation, extension
Remember copying can be shallow or deep
Details language–specific, outside course

- Make it VERY clean and clear

Object Orientation

- Like code, data should be structured
Think in terms of 'objects' and 'object types'

May be defined as an object type, need not be
Any related group of data (structures, arrays etc.)

- Use modules if language supports them

- If data are independent, then separate them

- Keep closely related things together

Fortran Example

```
MODULE LIST
    USE PRECISION
    INTEGER, PARAMETER :: SIZE=1000
    REAL(FP) :: DATA(5,SIZE)
    INTEGER :: PARAMS(42), USED
    LOGICAL :: FLAG(SIZE)
END MODULE LIST
```

Or even the same in a COMMON block

C++ Example

```
class mydata {
public:
	static const int size = 1000;
	double data[size][5];
	int params[42], used, flag[size];
};
```

It's not essential to use a structure or class
Using a single header is better than nothing

- Most of the benefits come from disciplined coding

Basic Actions (1)

All object types need the following primitives
Generally, one of each, but sometimes alternatives
There are more details on these in the next lecture

- A constructor to initialise them

This should always be used to create objects
It doesn't need to be a formal constructor

Use of uninitialised data causes foul bugs
Can hide for decades, especially when zero is OK
Completely unrelated factor alters that – BOOM!

Basic Actions (2)

- A destructor to destroy them

This should always be called to release them

Using ‘dead’ values is almost equally bad
Exactly the same problems, but rarer

- A display method to show their contents

This is for diagnosis, not printing results

- A checker to check their validity

This is the principal debugging tool

Other Actions (1)

There are some that are very often needed
Not covered further in this course

- One to copy or move an object

memcpy etc. can work, but are dangerous
Add one pointer and shallow copying fails

- ‘Binary’ dump and restore methods

Either to and from memory or a file
These should preserve the value exactly

Other Actions (2)

- One to print in a suitable format

Displays the value for use in output
Often very different from the diagnostic method

- One to read in a suitable format

Needed when the object is an input value
Quite complicated if humans input the data

Warning: remember to include thorough checking!

You may also need to import from other programs