Software Design and Development
Introduction and Principles

N.M. Maclaren
nmm1@cam.ac.uk

October 2018

1.1 Introduction

1.1.1 Purpose

This is **not** a complete software design course, because one of those would occupy the whole MPhil on its own - indeed, there are just such courses in some universities. It is also too complicated to be directly examinable, but you are expected to use it appropriately in your projects.

You should remember that the MPhil has a purpose beyond merely collecting a degree, and you are also learning skills that you will need later, when writing real scientific applications. These include both research applications and commercial ones, and the difference is only one of emphasis. The objectives of what you will be taught in this course are to improve the quality of your code (e.g. the reliability of the results) and to save you time getting the program working.

Software engineering is a bit of a catch-all term, and came into use only about 40 years ago, though the subject is older. It covers most of the skills you need to write high-quality code, whether based in theory or just practical guidelines. There is a relevant quote:

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

This course is practical software engineering, though you will get occasional references to the theory. It is not intended to turn you into computer scientists, though it will teach some of what is now called computer science! Unfortunately, modern computer science is often very bad on this aspect, which partly explains why so much software is so unreliable.

Also, learning scientific computing in a year is tight, and a major purpose of this course is to minimise your wasted time – learning from mistakes would take too long. Most of the techniques can be used to save time, and the course will describe how, why and when. But, obviously, 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, as distinct from an undergraduate one.

1.1.2 More Information

Note that it assumes fairly experienced programmers, and this course does not teach basic programming. It may use examples from several languages, but they are all simple to follow even if you do not know that language – you need to be able to program in only one language. For the same reason, it mentions techniques, but has few details, which depend on the language and your requirements. Most principles are the same for all of the languages you are likely to use.

- Contact your supervisor or Director of Studies if you have any trouble with these aspects. I am happy for them to contact me if they feel that I could help.

There are several other relevant courses that I used to give, and some other UCS courses, which are **not** part of this MPhil, and you will **not** get credit for attending them. However, if you need the relevant skill, then looking at them could be useful and save you time. Some will be mentioned in passing, and mine are all in the Web pages:

`https://www-internal.lsc.phy.cam.ac.uk/nmm1/`

Few books are much good, and some are truly ghastly (including several that are widely recommended). They are often written to push some dogma, or may even be provably erroneous in their claims. 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 first edition, because it is badly flawed. Most of the second edition is good advice, and it covers a very wide range of relevant topics, but its coding conventions are merely one of many, and not necessarily the right way to code. I checked it fairly briefly, and noted the following fairly major flaws:

- It is far too kind to C and similar languages, because many important coding conventions are to defend yourself against the languages, rather than against your mistakes.

The signed/unsigned morass will be mentioned later, but there are many other *gotchas* in C and C++. That term means anything where there is a subtle constraint hidden behind an apparently simple description, and where breaking it causes failure. For example, the following *gotcha* is a major problem, but is not mentioned in the long section on taming nesting:

```
if (test_1)
    if (test_2) action_2();
else
    action_1();
```

- The book seems to imply that most debugging stops when a product is shipped, but all software engineering experts agree that bugs found in actual use take up most time (even if there are fewer of them).

It also gives poor advice on unpredictable errors, and ones which are changed by unrelated code changes, which are described later.

- Chapters 25, 26 on tuning are something that could have come from a book of 1970, and are best ignored.

Almost nobody has unrolled loops by hand for 20 years, or done many of the other optimisations – that is what optimising compilers are there for! To improve the optimisability, you simplify and clean up your code, rather than complicating it, which is the converse of the advice given.

1.1.3 KISS

This stands for *Keep It Simple and Stupid*, often misquoted as *Keep It Simple, Stupid.* According to Wikipedia, that acronym was first coined by Kelly Johnson, lead engineer at the Lockheed Skunk Works, and it means that the simplest design is the best – i.e. fastest and easiest to get working and most reliable in use. That is an engineering principle that has proven to be reliable since prehistory! Hoare has also coined a couple of aphorisms along the same lines.

1.1.4 Debugging

The best solution is not to make mistakes but, unfortunately, all humans do. Careful design and coding helps to reduce the error rate, but little else does.

Failing that, the next best solution is to find errors automatically before use; stricter languages can help with this, but most people do cannot choose such a language (even when they exist!) So most debugging involves testing the program on real data, or is when the program goes wrong in actual use, and the course concentrates on this aspect.

- One of the best ways to improve run-time debugging is to design in semi-automatic checking.

This maximises the chance of catching errors early (i.e. before they have caused too much chaos and destroyed too much evidence). It can also be used to produce helpful diagnostics on error – i.e. ones that provide the information that you need to locate the bug.

The same approach can also help with (inherently tedious) manual debugging. It can be used for targetted, comprehensible tracing, and by providing checking and diagnostic functions to call when you need to track down a bug. Much of the course will target these aspects, and its aim in that is to improve the effectiveness of your debugging.

- Always try to debug with the target optimisation that you are planning to use.

It seems strange to recommend this and not debugging options, but some checks are done as part of optimisation and many bugs show up only in optimised code. This is particularly true for C and C++, because most 'optimiser bugs' (i.e. bugs that show up only when compiling with optimisation on) are actually breaches of the language standard, which get exposed only by optimisation.

All rules have exceptions, and you sometimes have to drop optimisation for debugging; for example, some compilers do not support it with `-g` at all, so you have a straight choice between optimisation and generating symbols for a debugger or tracebacks. But you should avoid running unoptimised more than necessary.

1.1.5 The Development Process

Development in academia typically neglects the design phase, and coding begins at the keyboard, but that misses the point that debugging takes longer than either. In fact, most debugging occurs in actual use (i.e. when attempting to analyse some real data has failed).

It has been measured at 10–100 times as much effort as the design – the following picture is **not** an exaggeration, and the effort is proportional to the area.

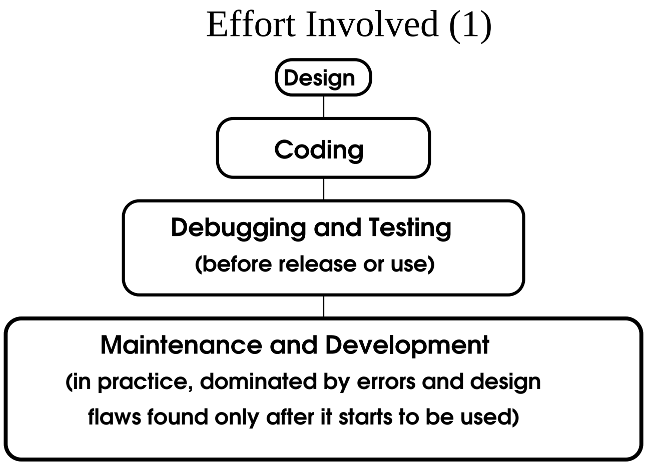
Effort Involved (1)
Design
Coding
Debugging and Testing
(before release or use)
Maintenance and Development
(in practice, dominated by errors and design
flaws found only after it starts to be used)

Figure 1.1

Managed development is the best approach to this - this does not mean development toolkits, nor even `make` and version control systems, which are mentioned later. More effort is spent in the design phase (typically 3–10 times as much), the code includes internal checks and diagnostics, and it takes perhaps 50% longer to write than before. The initial debugging is often much slower, because you have to debug the internal checks! But the overall effort can be 2–5 times less.

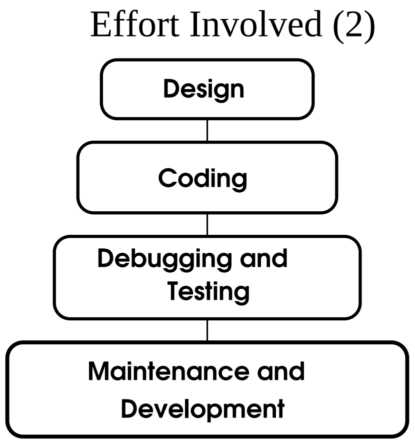

Figure 1.2

This can be taken to extremes and the process can prove the correctness of the design (mathematically), and almost prove that the code implements the design. IBM Hursley used the language Z to do that for *CICS*, and the following figures are (from 25 year old memory):

- The design took 3 times longer than average for a project of that size
- The project completed its development (including debugging) **ahead** of schedule
- The bug reports were 5 times below average
- The total project cost was 30% below target

Now, in academia, you and I very often code at the keyboard, we fix the syntax errors, and . . . Oops! We then fix the simple errors, tediously, and our problems really start when we hit the first difficult error. It is often worthwhile to go back a step, code and insert proper checking features, and the difficult error often becomes easy.

- Including checking code may double the time to the first complete run (i.e. demonstration) **and** halve the total time until it mostly works (i.e. you are getting useful results from it)!

1.2 Overall Design

1.2.1 Software Reuse

This is commonly referred to as *Don't reinvent the wheel*, and means using existing libraries and so on., rather than writing your own from scratch. It is currently almost a mantra, especially in the C++ area.

- It is a very good principle, but a very bad dogma.

You are adding a dependency on what you use, both on the code itself and on its specification. However, in general, start by reusing where possible and replace reused code if you need to; this makes program development quicker and easier. But you should think about it more carefully for production code. 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 are most important depend mainly on your requirements, and your skill is a secondary consideration. That sounds strange, but is so.

In the following cases, you should almost always reuse, but do not include the source in your program; use the latest, most improved version when building it.

- When there is a standard and stable interface.

There is usually choice of software that implements it, and no changes are needed to use a different version. For example, the BLAS, LAPACK, simple use of the C++ library and so on.

- When there is reliable, portable and stable software.

This includes the NAG library, FFTW, PCRE and so on.

In the following cases, you should usually reuse, but watch out for maintenance and reliability issues.

- When your system has a library that does the job.

Examples include Intel's MKL, AMD's ACML, but it also includes reasonably well-managed software projects, such as Boost. Because of the imprecision of the C++ standard, advanced use of the C++ library also comes here.

- When there is suitable open source to include in your program.

Naturally, you must check that the copyright conditions are acceptable. This includes most of Netlib, some of the above software and so on.

There are some cases when you should **not** reuse but, even here, you should start by trying to reuse. It is a good way to get a first version running, even if you replace the code thereafter.

- When the software doesn't do what you need it to do **and** extending it is more complicated than coding it from scratch.
- When you need a high level of portability, **and** the software is too system specific.
- When it simply does not work on your data **and** you are quite sure it is not a bug in your code.

1.2.2 Consistency

- A consistent style is a very important tool.

Creating and maintaining that was one purpose of the Fortran tools `Toolpack` and `NAGWare`, the C tool GNU `indent`, most 'pretty-printers' and many other tools. It means that you can tell what code does at first glance – and, more importantly, what it will **not** do – and you can trust that.

Almost-consistency can be worse than none, by encouraging mistakes and making it hard to spot them. However, you can use more than one style in a program, provided that the boundaries are clear, such as between modules. There is no need to be dogmatic.

Consistency of style also helps instrumentation – for example, you can add tracing code semi-automatically or or can put wrappers around library calls, without having to make every change manually, with the consequent risk of errors. Roughly parsing Fortran is almost trivial, but a minimal C or C++ parser is a compiler's front-end (most people use `gcc`'s for this). However, parsing C and C++ well enough for such purposes can be very simple on consistent code.

The best tools for instrumenting code are Python and (if you know it) Perl, but the latter is not an easy language to learn. For simple tasks, you can use the basic Unix utilities `awk` and even `grep` and `sed`.

- However, by far the biggest gain comes from consistency of semantics.

This means that the same construct or property means the same everywhere. For example, what does a positive definite matrix mean? Does it include ones that are very close to semi-definite? Or does it mean that all eigenvalues are bounded away from zero by more than the precision – for example, a condition like `min(eigenval) > eps*max(eigenval)`? Those two are incompatible meanings of the same term.

- If components `A` and `B` interact, they had better assume the same meaning, and a failure of this is a **major** cause of hard problems.

1.2.3 Descriptions, Specifications and Comments

- Do not underestimate their importance of these.

They rarely help when shaking the initial bugs out, but their benefits come from then onwards. You may think that your program will be used for a short time, and then thrown away, but it is surprising how often such programs are found to be useful, get modified and end up lasting many years. Also, projects that succeed generally attract co-workers, so other people may need to work on the code.

Of most immediate importance is the fact that examiners do not like analysing code. If you have got something right in principle but made an error in the detail, and the examiner notices a problem, you will not get any credit unless it is very clear that you got the principle right. Make it clear **what** you are doing and **why**.

- The sole criteria for this sort of documentation are that it is complete enough for its purpose and correct in context.

It need not be as precise as a published paper, but should be as good as the notes that you use for writing such a paper. So, when you update the code, also fix the documentation that corresponds to it. Sometimes, you do not have time, in which case you should always make time to say so, but please try to avoid doing that – e.g.:

```
/* WARNING: comments are for release 1.3 */
```

It is your choice whether you use separate documentation or block comments; the latter are a little easier to keep in step, but this is more a religious matter than a technical one. Either works, and many people use whichever seems more appropriate (I do).

There are also methodologies to integrate source and documentation, and their proponents get very evangelical, but I do not like them myself. The technique dates from 1960s, and took many forms. If you are interested, look at doxygen (`http://www.doxygen.org`) or CWEB by Knuth and Levy or Wikipedia on “literate programming”. If you find one that suits you, why not use it? If you do not find them useful, why add to your difficulties?

Reverse engineering is working out what a program is supposed to do from what it actually does (or its code). Without documentation, you have little option, even on your own code, if it is long enough after you write it. It is normally needed when looking at other people's code, can be incredibly time-consuming, and often increases debugging time tenfold. You should avoid it being necessary.

- Obviously, writing good documentation takes time but, generally, the best balance is more of it.

In many cases, such as where the problem is scientifically and numerically very complicated, the documentation should be longer than the code! That is not an exaggeration, and you will see it in the commenting of some high-quality numerical codes.

1.2.4 Specifications

As mentioned, you can use block comments or a separate file. Such documentation should cover the program's main purposes:

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

It should also cover the essentials of its use:

- Its input format and the constraints on the data
- Its output and any guarantees on that
- Roughly the errors that it intends to diagnose
- What it assumes but does not check

You can include anything else of that nature that seems relevant. This will usually include your name and the date, and possibly copyright and other legal restrictions, but this course does not cover such aspects.

The above topics are critical because **when** (not **if**) a program fails, obscurely, you have a reminder of which assumptions to check; a good half of 'difficult' failures are false assumptions. You can also decide between a simple bug and a data error. It also helps to know where and how to fix the problem; fixing a bug wrongly is a common cause of wasted time. For example, is a performance problem a bug or an unavoidable feature? In the former case, it is fixable; in the latter, only a redesign will help.

1.2.5 Commenting

- The main purpose of detailed commenting is that it helps to keep your own mind clear.

This is absolutely critical a decade later or when someone else modifies the code. For example, component `A` creates a symmetric real matrix, and assumes that it is a non-degenerate variance matrix, so it is always positive definite. Component `B` then divides by its determinant, which then fails on an indefinite matrix. Which of `A` or `B` needs fixing?

- You can use it to find your way around your own code.

For this use, you use it to introduce significant segments of code – what counts as significant is a matter of judgement. Call tree information can be very useful, too – this is where procedures are called from and go to - it is a major headache to maintain, so is rarely done, but there are some tools for adding it automatically. The details are entirely a matter of taste, so do whatever speeds up your debugging.

Simple code needs very little low-level commenting. As an aside, an old assembler rule was to comment every line, which was an exercise in futility set by born bureaucrats:

```
ADD R1,=1     Add one to register one
```

There is also a dogma that some languages (like Pascal) are *self-commenting*, and that is complete and utter nonsense, too, because the language properties can convey only certain sorts of information. The main rule is to think of what you want comments for.

- Describing pitfalls is **by far** the most important use of low-level comments.

They can remind you not to make same mistake twice, document assumptions that may break later, or document horrible and unobvious hacks. The following are typical (and realistic) examples:

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

and:

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

Something that is worth noting is that using appropriate identifier names is also a form of commenting, and is especially useful for ones used by components other than the one that defines them. Longer names help to avoid name clashes, which is a common cause of obscure errors, but they also make your code **much** easier to read. You can use the same names as in the paper you got the algorithm from is a good convention. And `velocity` is usually clearer than `V`, and `Cholesky_solver()` instead of `solution()`. Generally, keep simple names (e.g. `A`) for local scratch variables.

In my experience, good commenting can slow your coding by 25%, and rarely speeds up the initial debugging much. However, even when I was 30, it helped a month later, and often speeded up debugging by a factor of two a year later. It helped even more on other people's code, and was often the difference between success and failure in locating a problem.

- Overall, in research, effort spent in writing good comments repays itself about 3:1, and can repay 10:1 for production code.

1.3 Program Structure

1.3.1 Procedures

- All large programs should be subdivided, even if the language has no formal module mechanism.

Less obviously, you should document components, almost as if they were programs. You get all of the advantages described above and more, and it is critical when 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!

- You should break programs up into components that are simple and small enough to understand, but mincing into hundreds of tiny pieces is also bad.

Again, this is a matter of judgement. You should use modules if the language supports them, and some sort of equivalent if not. You should also do the same to data structures and types; if they are independent, then separate them, but keep closely related things together.

- Remember that procedures are low-level components, too.

At least for the major procedures in your program, document their purpose and interface, precisely. It is sometimes obvious from the context, but usually is not. Typical information needed is:

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

A trivial (and close to minimal) example is:

```
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
```

- You should treat object types as components and, generally, do the same for any set of data that is handled together.

As for procedures, you should document what their function is, precisely; it is sometimes obvious from the context, but usually is not. Typical information needed is:

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

This is exactly the same as for programs, but at a lower level – indeed, Fortran calls them subprograms! A trivial (and close to minimal) example is:

```
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
*/
```

1.3.2 Interfaces

An interface is **any** way of passing data or control, and includes network interfaces, procedure calls, often files, and the specification of data structures or objects. Essentially, anywhere a component `A` meets a component `B`. The guidelines described here apply at all levels. These commonly include objects, modules and procedures, but also include suites of programs operating on files, and other levels, both higher and lower.

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. You should think in terms of interfaces to data and code, and write and use conversion and access functions, as appropriate. There is one very important guideline:

- Avoid exporting internal details for other uses (i.e. access not through documented interfaces), because it almost always causes trouble.

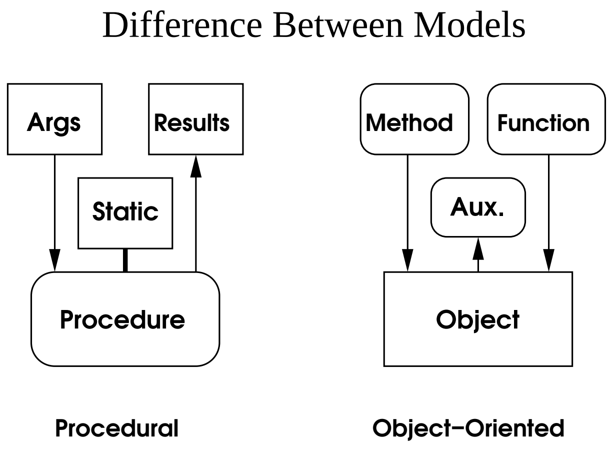

Figure 3.1

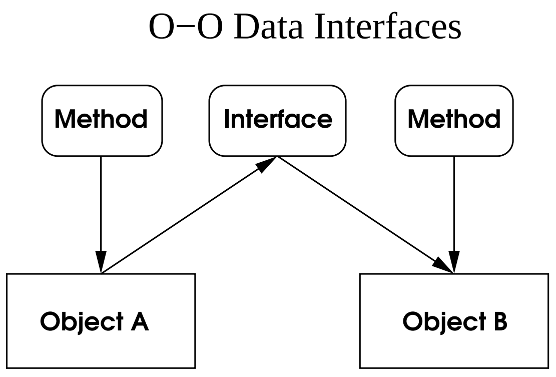

Figure 3.2

- To a first approximation, debugging is all about interfaces.

Most serious bugs occur at interfaces, because people forget what they were assuming earlier, and this is the most common error made by experienced coders. Also, languages may not allow interface checking, and compilers rarely do it even if they could. Even when it is done, it will pick up only some of the interface errors, but it could include most of those that cause crashes. The best common compilers (in this respect) are NAG Fortran and Python.

KISS is more relevant to this area than anywhere else. You should keep the interface concepts simple, because 'clever' designs often hide '*gotchas*'. When you design an interface, ask yourself whether you are **sure** that there are no inconsistencies lurking. To help with this, you should minimise complicated interactions, especially between many components, and long-term ones (i.e. where an action causes an interaction at a much later time).

1.3.3 Design Principles

You should avoid updatable data objects and arguments (where 'argument' means any data used in an interface), whenever possible – ideally keep the data simplex (i.e. input or output but not both). It makes it much easier to answer questions like "what was the value before the action failed?"

- Updatable data objectss can be a real problem for debugging.

For example, a procedure fails during the millionth use, and you need to know the value that triggered the failure. The same applies to updatable files when used as interfaces; it is better to separate input and output files. But sometimes you have no alternative, either because the algorithm is based around incremental updates or the need to save space is critical. Many matrix algorithms are like that.

- You should avoid context-dependence, and keep units of data conceptually simple.

Do not make the interpretation of the data context-dependent if you can avoid it; for example, using different units or unit systems in different contexts is not clever, or using the same variable for a variance matrix in one place and a transformation matrix in another.

- You should also avoid unobvious or unspecified side-effects.

The most common form is the updating of global data, but it includes the updating of environment files. The general rule is that the more complicated it is, the harder it will be to debug.

State changing is rarer, but can be truly **evil** – POSIX signal handling is one example and IEEE 754 floating-point handling is another. If you need to use anything like these (and I do not recommend it), be sure that you reset properly before leaving any procedure in a module that uses it; this does not mean just returning, but when calling code in other procedures. One if the reasons that it is so nasty is that languages like C++ introduce a lot of implicit procedure calls.

The same applies to suites of programs, which often keep their state in files: for example, CVS, Web browsers, GUIs etc. If one component changes the global state, it can cause chaos to another.

- Generally, you should avoid global state changes if you can do so, because debugging them is usually a foul task.

If you have to, you must remember to handle problems caused by failure, which often will not have cleaned-up correctly before exiting. You need a primitive to restore a known, clean state; most such primitives are too half-hearted and do not clean up thoroughly enough, but a few always destroy too much data and so cannot be used. By far the easiest approach is to change global state only during initialisation or termination.

1.3.4 Encapsulation

- Encapsulation is the most useful technique of all, and can speed up debugging by a large factor.

This is where all accesses to some data are through a defined set of interfaces, and no other access is allowed. This is usually via procedures kept in a module, which may provide extraction and insertion primitives, or only ones that operate on the data. You can also encapsulate only updates to the data, and export the data read-only, so that it can be accessed directly.

With this, you know where to start when data goes bad, it provides a place to add checking or tracing, and it allows changing the internal details easily. It can be applied to a data type (e.g. a class in C++ terms) and is a basic principle of object-oriented design. The object's internals are known to only a few components (often called 'friends') and all other code uses only the exported interfaces.

- Encapsulation can apply to any data or interface.

This applies to objects, global or static data, system state, file I/O, the user interface, memory management, application-specific components or state, device control, networking, GUI use and more. It is a very general technique but, as usual, nothing comes for free and it is extra effort. It is often confused with object-orientation, but is far more general.

The approach helps with most hard problems, but it will not solve all of them. Unset indices and pointers can corrupt arbitrary (often unrelated) data, and so can using a subtly wrong command on a file!

1.3.5 Interface Design

Whether the interfaces are to procedures or modules, multiple simple ones are better than a very complex one. But there is a very relevent acronym: *TANSTAAFL – There Ain't No Such Thing As A Free Lunch*, which was probably invented by Larry Niven. Interactions between units are part of the interface, too, and need at least as much design and specification. The POSIX standards (all of them) get this very badly wrong.

- Do not think just in terms of arguments and global data, but in terms of **any** interacting constraints and assumptions.

This includes abstract properties such as whether component `A` guarantees what component `B` assumes. And remember the application's own state changes – state changes are part of the interface, too.

Most languages use a very poor data model for procedure arguments and imported globals: the properties of an entity (e.g. a structure) apply at one level, usually the lexically enclosing scope. They can change when that calls another procedure or module, which is not helpful for either debugging or parallelism.

- You should attempt to ensure that properties apply recursively.

For example, read-only arguments should refer only to data that is read-only during the entire time of that procedure call, including when it is in called procedures. If that is the case, debugging is much simpler. Unfortunately, that is not always possible, but you should minimise places where it is not so, make them explicit in the code, and document them. A C++ copy or move constructor that has the side-effect of modifying global data that may be used as its own argument is the material of nightmares!

Global and static data is not as unclean as traditional dogma claims. Its worst problems are pointer aliasing and scoping errors, which are very common causes of hard-to-locate problems.

- It is never safe to cache **any** argument pointer in a global or local pointer variable.

This applies even in languages like Python, and even more in C++ and Fortran. Exceptions to this do exist, but be **very** careful if you need to do anything like that. Remember that includes everything referred to by an argument, directly, or indirectly – if in doubt, copy the data (if possible), and watch out for shallow versus deep copying problems.

If you do not understand the previous paragraph, you should avoid using multi-level structures that are connected using pointers. Unfortunately, there is no time in this course to cover the topic of data design, which is better suited to a programming course, anyway.

1.3.6 Procedure Interfaces

It is generally worth designing these for reliability and ease of debugging, because that can save a great deal of time.

- You should use a pure functional methodology when possible.

This means that procedures have no side-effects and do not update their arguments – this includes **all** data pointed to by any of the arguments. Such procedures can be used safely anywhere, including as Fortran `FUNCTION`s and as all C++ STL member functions.

- Often that is not possible, so the next level is to also use pure output arguments.

You can write results into them, or even alias pure input arguments into them (using pointers), fairly safely. You can also use encapsulated static or global data if you need to, because you can be sure that it doesn't alias any arguments. Up to this point, debugging is fairly simple.

- However, you sometimes need to go beyond that, and then it is best to design argument properties for a single purpose.

For example, data arguments should usually be one of: read-only input, not updated during the life of the procedure; pure output, written only just before the procedure returns; or workspace, which has undefined contents at entry and exit.

- It is very important to document which component allocates space, and similarly for deallocation, and extension (if relevant).

All that may be semi-automatic, if space is allocated as needed and deallocated when no longer used, but that should be documented. Remember that copying can be shallow or deep, and there are are more subtle variations, too. The details are very language-specific and outside the scope of this course.

- The critical requirement is to make argument use and memory control **very** clean and clear, because mistakes with them is one of the most common causes of hard problems.

1.3.7 Object Orientation

Like code, data should be structured, and object orientation is one method of doing that. The approach is more general than formal object orientation, and can be applied to any related group of data, whether structures, arrays or collections of them. They may be defined as an object type, or may not be, but you should think in terms of 'objects' and 'object types'. As a reminder:

- Use modules if the language supports them
- If data are independent, then separate them
- Keep closely related things together

A trivial Fortran example might be:

```
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 data in a COMMON block, though that is old Fortran and not generally recommended. A trivial C++ example might be:

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

It is not essential to use a structure or class, though it is cleaner; just including the declarations in a single header is better than nothing. Most of the benefits of object orientation come from the extra discipline in the use of data.

1.3.8 Basic Object Methods

All object types need the some basic primitives, generally one of each, but sometimes you may want alternative versions. There are more details on these in the next lecture. Note that the list given here is **not** what you will see in most books and Web pages on object orientation, but the next lecture will explain why.

- A constructor to initialise them.

You should always use some procedure to create and initialise objects, though it does not need to be a formal constructor. Obviously, you can take short cuts with simple arrays, and pure workspace does not need initialising, but the principle stands. The use of uninitialised data causes foul bugs, which can hide for decades, especially when zero is is a valid value for most fields. Some completely unrelated factor alters that, and your program crashes or gives wrong answers for no apparent reason.

- A destructor to destroy them.

You should do the same when you have finished with them, and call some procedure to tidy up. Using 'dead' values is almost as bad as using uninitialised data, and can cause exactly the same problems, but they are rarer. This is primarily a problem when you use pointers, as all C and C++ programs do.

- A display method to show their contents.

This procedure is for diagnosis, and not printing the results in your normal output – it should provide you with enough information to see what the object really contains.

- A checker to check their validity.

This is the principal debugging tool and is described at length in the next lecture. It is also the one most often omitted by books and Web pages on object orientation.

There are some others that are very often needed, but are not covered further in this course.

- One to copy or move an object.

Direct copying (including `memcpy` etc.) can work, but are dangerous; you only have to add one field that is a pointer, and you get shallow copying, which is usually **not** what you want. More generally, they are only supported for *POD* types (C++) and `SEQUENCE` types (Fortran).

- 'Binary' dump and restore methods.

These can do this to and from either a block of memory or a file, and should preserve the value exactly. They are used for checkpointing, and similar purposes.

And, of course, most objects need some some actual computation actions! You also often need some external I/O, usually to and from a human-readable format.

- One to print the object in a suitable format.

This displays the value for use in your output, and is obviously needed only for objects whose values are results. It is often very different from the diagnostic method, because its purpose is very different.

- One to read in a suitable format.

This is needed when the object values need to be read as primary input, and can be quite complicated if it is intended to be a user-friendly free format. A **very** important warning is to remember to include thorough checking, whether or not that is the case, as many errors occur at this stage.

You may also need to import from other programs, either in a text format or their binary format (e.g. Matlab MAT files), but that is beyond the scope of this course.