PhUSE 2011

Paper RG01

Modular Programming - Some Lessons Learned and Benefits Gained

Ross Farrugia, Roche Products Ltd, Welwyn, UK

ABSTRACT

By encouraging programming teams to forward think and plan ahead time savings can be found in the long run. Modular Programming is a programming technique that enables this; beginning from thorough requirement analysis, and the simple idea of breaking programs down into clear separate modules of code. We will then further look into the greater benefits that can be available of re-usability of code at a higher level, as well as the increased ease of validation this can provide. Tips, benefits and considerations will be given on this technique already used widely in software design.

INTRODUCTION

This paper will cover what modular programming means in programming, and some best practices I've found of this method. We'll look into benefits and considerations whilst adopting this approach. Real-life pharmaceutical statistical programming examples will be given to help understanding. Also advice on how this can influence and potentially improve the way we validate our programs.

Most experienced programmers will be familiar with the technique, so a lot of this paper may be more applicable to more junior programmers, but there still may be some useful ideas included that anyone can appreciate.

WHAT IS MODULAR PROGRAMMING?

Modular programming is a software design technique that increases the extent to which software is composed of separate, interchangeable components, called *modules*. What does this mean to us as statistical programmers? It's the idea of splitting our programs up where possible into clearly separated modules of code which can be easily maintained and re-used across different reporting events, studies and even projects. The easiest way of doing this being the use of macro modules – depending on your requirements you may choose different granularity of code/modules.

The use of macros is not the only way to modular program. Even just structuring your program according to these ideals will make for greater maintainability, reusability and understandability, all important qualities of a good program. For the purpose of this paper I will focus on achieving modular programming through the usage of macro modules.

AREAS OF USAGE

Any program could have modular programming applied, be it an analysis dataset, an output or even a test program sitting in a personal directory. For a start, it will make the programs a lot easier to come back to and read a year later when our memory may have abandoned us. But more importantly, if we are performing any action that we can foresee will need to be repeated again in future, then why not take this out into a separate module that can be referenced by any other program. The module of code could perform a number of data steps or even just one function, either way it saves us repeating the task later and changes to code can be performed efficiently in one place.

HOW DOES THIS WORK IN PRACTICE?

It is important to start any programming task by giving enough time for thorough 'Requirement Analysis' – where you will consider your requirements (the outputs) and how best to program them. It is during this time when you can start to plan how you will break up your program into the set derivations (or coding tasks) required, and aim to keep all derivations in the analysis dataset. These will form your individual modules of code.

In an analysis dataset program for example, you may be able to group multiple variables with similar derivation rules into one module. A simple example would be variables for:

"Worst laboratory value per parameter for each patient"

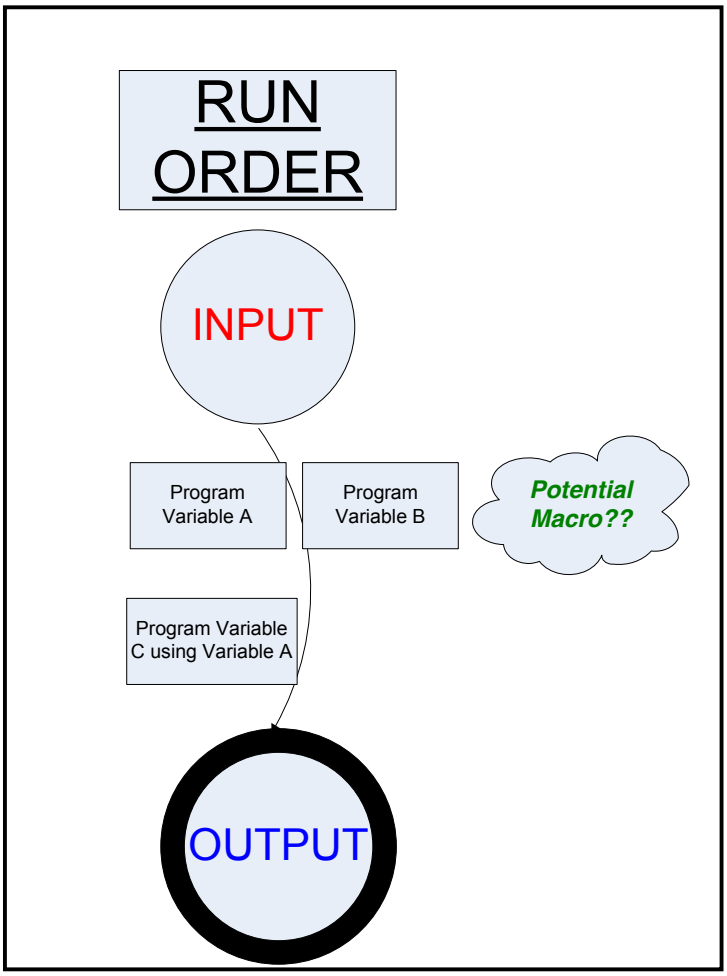
"Worst laboratory value per parameter *per treatment cycle* for each patient"

"Worst laboratory value per parameter for each patient *during a certain phase of study*"

The basic underlying code for all these would be the same with one small different clause for each.
Alternatively, you may find that one module may only be needed for one variable in this program, but may be able to be re-used in another program. A simple example would be variables for:

"Baseline weight"
"Baseline pregnancy status"
"Baseline white blood cell count"

If the baseline identification rule was consistent then a single macro module with carefully selected flexibilities (macro parameters) could be sufficient here.
Once you have your modules planned out, you'll need to consider the structure of your code – think about the necessary run order of these modules, the purpose, any dependencies, complexity of each, and the overall efficiency of the program – all of these factors will impact your modular design.
Some people find it easy to jot this design down as a visual during the planning stage. A very simplified example is shown below:



Next, it is important to examine each module. Ask yourself would each module benefit from being a separate macro, or a macro (or just a clear piece of code) within your program. To help make this decision, consider the following:

- Is this piece of code likely to need to be repeated in this program or in another program on this study?
- Could this be re-used on future studies?
- Is this code complex/time-consuming enough to justify the extra overhead of creating a separate macro?

If you do decide on a separate macro, then consider the following:

- Is the requirement applicable at the project level or just specific to a study?
- Does the derivation rule apply only to a single domain of data or more than one?

If you decide to make the macro robust enough for project or across-project usage, then consider the following:

- What rules have you seen here in previous study requirements?
- What are you aware is requested or could possibly be for future studies?

Then plan the macro flexibilities accordingly. If it's a new requirement or first time of introducing modular programming then discuss within your project team, and ensure to make sensible judgements.
You'll end with clear, modular programs that may look something like the below:

```
**  Highest CTC 345 day of/after or during inf AE in Cycle  **;

%g_minmax(dsin       = ae_ctc5,
          invalobs   = ,
          incmiss    = Y,
          inwhere    = %str(upcase(aecpe) like ('CYCLE__%') and (aeinfu eq 'YES' or aeinfnfg eq 'Y
          dsinsub    = aecpe aept,
          dsinseq    = aeintc aebegdt aepage,
          dsinval    = aeintc,
          style      = MAX MIN MIN,
          outfg      = aeina3fg,
          dsout      = ae_ctc6,
          debug      = &debug);


*******************************************************************************;
**  Add details of concomitant treatments                                    **;
*******************************************************************************;

%ae_conmd(dsin      = ae_ctc6,
          dsin_medo = &medo,
          dsout     = ae_conm1,
          debug     = &debug);


*******************************************************************************;
**  Add details of last study treatment before AE                            **;
**  (Treatment name, start date, study day)                                  **;
*******************************************************************************;

%ae_lstmd(dsin     = ae_conm1,
          dsin_medt = &medt,
          mttxt     = PERTUZUMAB PLACEBO TRASTUZUMAB DOCETAXEL,
          dsout     = ae_lstm1,
          debug     = &debug);
```

It is important to point out that macro modules are only used where sensible though. There may be steps of code in between where re-formatting of data or some manipulation is needed, or it was just deemed a macro was not justified, such as below example:

```
**  Highest CTC 345 day of/after or during inf AE in Cycle  **;

%g_minmax(dsin       = ae_ctc5,
          invalobs   = ,
          incmiss    = Y,
          inwhere    = %str(upcase(aecpe) like ('CYCLE__%') and (aeinfu eq 'YES' or aeinfnfg eq 'Y
          dsinsub    = aecpe aept,
          dsinseq    = aeintc aebegdt aepage,
          dsinval    = aeintc,
          style      = MAX MIN MIN,
          outfg      = aeina3fg,
          dsout      = ae_ctc6,
          debug      = &debug);


*******************************************************************************;
**  Add details of concomitant treatments                                    **;
*******************************************************************************;

%ae_conmd(dsin      = ae_ctc6,
          dsin_medo = &medo,
          dsout     = ae_conm1,
          debug     = &debug);


*******************************************************************************;
**  Add details of last study treatment before AE                            **;
**  (Treatment name, start date, study day)                                  **;
*******************************************************************************;

%ae_lstmd(dsin     = ae_conm1,
          dsin_medt = &medt,
          mttxt     = [redacted] PLACEBO [redacted],
          dsout     = ae_lstm1,
          debug     = &debug);
```

BEST PRACTICES FOUND

When implementing modular programming at a project level, we found it was really useful to have some conventions agreed from the start. This is especially useful for new people joining your team.

For all macros we had a set naming convention:

- Generic macros (a derivation able to use across-domains) - %g_*xxx*
- Domain-specific macros (a derivation only usable on one domain - use first 2 letters to indicate domain) - %ae_*xxx*, %lb_*xxx*, %dm_*xxx* etc
- Utility macros (could be used in any program to perform a set function, e.g. clearing work libraries at the end of a program) - %u_*xxx*

We then set up a generic macro specification which could be easily adapted for new macros, and would ensure that the knowledge of any macro is not confined to the macro creator. Some important points you may want to detail in the macro specification are:

1. Scope and Purpose of macro.....
2. Outputs produced by macro .....
3. Macro Assumptions .....
4. Description of Derivations.....
5. Macro Flexibility .....
6. Validation Checks .....
7. Macro Parameters.....
8. Example .....

Another useful idea is to set up a macro template program, with agreed conventions included, so programmers can just pick up this as a starting point.

One thing we observed was that it was very easy for those working on the project a long time to become familiar with the macros we developed and use them as second nature, but it was not so easy for people new to the project to pick these up, especially as there was such a large quantity created. This led to the introduction of a macro index document, which became a constant source of resource to both new and old team members, and saw real increased efficiency of the team. An extract of this document is shown below:

Name	Description / Notes	Contact for advic	Validation Method
g_centfile	Contains variable/dataset attributes at project level	Littish / John	Code review
g_attrib	Add final dataset attributes according to project standard	Littish	Code review
g_baselinech	Calculate baseline change and change from baseline change	Littish	Code review
u_abort	Used everywhere - aborts processing, used as part of check code.	Littish	Code review
u_chkattrchg	Check attributes against reference library. Useful in particular for GDMs	John	Code review
dm_popn	Merge population data from EXCL onto DEMO	Waseem	Code review
ev_bor	Derive EVENT VAD variables such as best overall response, Clinical Benefit (CBR)	Littish	Unit Testing
tm_cnfm	Derive tumor response confirmation as per RECIST	Littish	Unit Testing

The links shown in the left column then would take the user to the macro specification document.

Backward compatibility should be considered from the start, as often you'll later spot additions that you'd like to make to macros, at a time when they've already been used for a delivery. The easiest way we found of doing this though was to just add a new macro parameter that has no impact on the old ones. But a more efficient solution for programming run times could be to update existing macro parameters. In this case, it's best to set up an area where you can run benchmark programs up front using your macro, and then you can re-run these later to ensure any change to the macro does not have any impact on the previous macrocalls. This is a form of regression testing.

A REAL-LIFE EXAMPLE

We had an analysis dataset for adverse events where we required flag variables for the following 3 cases:

"Most severe adverse event by treatment cycle"

"Most severe *treatment-related* adverse event by treatment cycle"

"Most severe *during infusion* adverse event by treatment cycle"

In the planning stage, the programmer identified that the fundamental code would be the same across the 3 derivations. They then looked back to a past requirement on an old study where this was also needed for a different domain, cardiac symptoms. So it was decided the macro could be designed to use across domains. By consulting the team it was decided that this request could be needed in future for any subset of adverse events, not just treatment-related or during infusion, so this would need to be made flexible.

The result was the macro %g_cycfg, a macro to flag events according to some user-defined "worst" criteria by treatment cycle (or any other timepoint). The flexibilities to allow this are shown below:

OPTION NAME	MANDATORY/ OPTIONAL	DEFAULT	OPTION DESCRIPTION
dsin	Mandatory	<none>	Input dataset
worstvar	Mandatory	<none>	Variable containing "worst" values
hilo	Mandatory	<none>	'H' or 'L'; Is high value or low value worst?
where	Optional	<none>	Where clause for input data
byvar	Mandatory	<none>	By cycle/timepoint variable
outvar	Mandatory	<none>	Output flag variable name
dsout	Mandatory	<none>	Output dataset

Now if we take the "Most severe *treatment-related* adverse event by treatment cycle" variable, this would have been programmed just as:

```
%g_cycfg (      dsin        = ae1,
                worstvar    = aesever,
                hilo        = H,
                where       = %str(aerel='YES'),
                byvar       = aecycle,
                outvar      = relsevcy,
                dsout       = ae2           )
```

INFLUENCE ON VALIDATION

For this section I'm going to encourage reading my paper at last year's Phuse conference – see paper RG01 "A Case Study in Using Unit Testing as a Method of Primary Validation". I believe modular programming and unit testing can go hand-in-hand to create clear and robust programs that can assure us of programming accuracy regardless of the input data. If you decide to adopt any of the approaches shown above then I highly recommend trying out unit testing too.

BENEFITS AND CONSIDERATIONS

BENEFITS

Here are some of the main advantages we've found of this method, and reasons why we continue to use it across all of our studies on our project:

- Many software quality factors can be improved by modular programming, such as maintainability, re-usability and understandability
- Re-usability is the key driver of these – saves different people repeating the same work time and time again
- Programs can easily be picked up, understood and adapted across all studies by any of the study programmers
- Future study programs take significantly less time to produce as most of the code is already available
- Consistency of derivations across studies within a project is increased
- In many cases, we noticed it took limited SAS experience to create the programs on later studies
- In combination with unit testing we found modular programs became easier to validate, and less issues were spotted during later more extensive data transfers where more new data scenarios are seen. This gave the team a high level of confidence in the quality of our programs

CONSIDERATIONS

If adopting this method at a project level here's some things that are definitely worth weighing up:

- Project macros do need to be well managed (e.g. a macro index as mentioned above) else you'll end up with macros repeating functions that you already have a macro for, or people just simply won't use them
- Can take extra time for new starters on your project to pick up the ideals, conventions and to feel comfortable using the macros already available
- Macros do take more initial time investment, but the macro program template can help this, and I believe you'll more than see this time saving back in the long run
- Be wary that macros can easily be over-complicated with many different parameters, when really separate macros may have been the ideal. Modular programming is all about simplifying and breaking up programs into easily digestible chunks
- Be careful designing your macros to ensure efficiency of the program, as poorly designed macros, which you need to run over and over may lead to increased run times

- Backward compatibility/regression testing as mentioned above
- The main point is planning up front - this is key! Bouncing ideas with other programmers is always a good idea. Strategize how best to create your program before you write your first 'data xxx;'

CONCLUSION

For established projects not already using modular programming then this is always going to be a difficult thing to implement, but for new early projects especially if you have this approach in the team's minds from the start then this definitely can give substantial future time saving for just a little extra initial time spent. Then when you're thinking project-level remember it is highly beneficial to have project rules and requirements defined and considered to gain real benefits. Ensuring true re-usability and avoiding re-inventing rules means that we can re-produce and deliver more timely and with greater consistency of the rest of the project deliverables.

ACKNOWLEDGMENTS

I'd like to give thanks to Hinal Patel and Christine Leigh for their review of this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Ross Farrugia
Roche Products Ltd
6 Falcon Way, Shire Park
Welwyn Garden City, UK / AL7 1TW
Work Phone: 01707 365918
Email: ross.farrugia@roche.com

Brand and product names are trademarks of their respective companies.