PhUSE 2010

Paper AD07

What makes a 'good' program?

Dean Grundy, Roche Products Limited, Welwyn, UK

ABSTRACT

As SAS® programmers, we create hundreds and thousands of programs; generating tables, listings, figures, datasets etc. But are these programs actually any 'good'? What exactly makes a program 'good'? If my program produces what is required, and is correct in its output, then is this a 'good' program? For example, if the runtime is inefficient, is this a 'good' program? Efficiency is a factor often used, and is easily measurable. However, is there more to a 'good' program than just correctness and efficiency? Perhaps the programs ability to cope with unexpected cases, or the ease of which the program can be updated and maintained? The aim of this paper is to discuss a variety of such software quality factors; how and to what extent they can be applied to the world of statistical programming in the pharmaceutical industry.

INTRODUCTION

What makes a 'good' program? How can a program be assessed in terms of quality? This paper does not intend to convey 'how to program', but how to assess our programs and consider some of these findings when designing our programs. This paper aims to tackle the subject, looking at both quantitative and qualitative methods to establish how we as programmers can determine whether we are developing a 'good' program.

QUANTITATIVE METHODS

In terms of overall software quality, metrics on how well the team has performed by measuring the quality is popular. It can be done in a variety of ways, but is usually focused on whether the programs we write produce the correct result. A couple of examples can include amount of iterations the program enters quality checks before is passed by the tester(s), or alternatively a count of the problems found in the programs.

If we go down the route of collecting metrics on how many corrections are required or 'bugs' are found in the programs during testing, we can find ourselves with information that may not have much meaning. This information can be contested in many ways. Examples of this could be the severity of the fault not being taken into account within the metrics, as well as the impact of how much effort is required to make the correction. Factoring these in can increase the value of the metrics, however it can be rather complicated.

For such quantitative methods to be applied, many factors have to be taken into account for the result to be truly meaningful, however the outcome also only covers a small part of the overall quality – as we are only concerning ourselves with error counts.

Often as programmers, we can write a program that is 100% correct, but we may not think it is an entirely 'good' program, especially if it takes an excessive time to run. Runtime efficiency is a common measurement used to determine a 'good' program. But how can we produce quantitative measures of efficiency? To do this, we would need some sort of baseline to gauge this against, but determining a baseline for runtime can be rather arbitrary. If a program runs in half the time of another, how can it be known whether there is potential for further dramatic improvement or not? Without metrics on the original, we are unable to determine quantitative measures on the current.

Metrics on runtime efficiency may be possible, however it could prove unfeasible, potentially having so many parameters, including number of data points, amount of datasets used, and relies heavily on hardware specifications.

Consider these two methods were both implemented, we are geared towards 100% correct, super-fast programs. What would the code look like? Writing well structured and understandable code most often impacts on efficiency (or vice-versa). An incomprehensible program surely cant be a 'good' program, especially for anyone who has to pick it up and edit it. Here identified is our third measure of quality – understandability. With understandability being very

much a qualitative measure, and with no clear way on how we can collect metrics on this, we conclude our discussion on quantitative methods.

SOFTWARE QUALITY FACTORS

Software quality factors are qualitative methods of software quality, and can cover a wide range of areas.
There are many schools of thought on what constitutes a complete set of factors, however this paper will cover the three we have already touched upon; Correctness, Efficiency and Understandability, along with Maintainability, Timeliness, Robustness, Reusability and Portability.

CORRECTNESS

Correctness is probably the most important factor within pharmaceutical programming. What we produce should be 100% correct, and we have measures in place to ensure this. Correctness should not only be measured on whether the deliverable matches the specifications, but does the program meet its requirements, and does it fulfill its purpose? A program that is 100% correct in producing something that a customer did not want or need may be a 'good' program is many ways, but it cannot be 'good' at its job.

We are all familiar with testing for correctness in some shape or form; double-programming, unit-testing and independent code review to name a few, however these are ordinarily done against the programming specifications, by other programmers who have as much knowledge of the customer needs as the developer. Other methods we can use to ensure we reach further, reaching to our customers, can include prototyping. This enables the customer to view the pre-final product and perform a comparison of the product against their expectations and requirements.

EFFICIENCY

Efficiency is often perceived as 'how long does a program take to run', and although runtime is a major part of this, it is rather a byproduct of the wider scope. This wider scope of efficiency being a scale of how much system resources the program uses.

An example of a typical efficiency boost that can be gained is by reducing the amount of data that is processed. Reducing data to the minimum required as early as possible – both vertically (subsetting the data) and horizontally (keeping only the required variables) can minimize the strain on the system, and directly impacts the runtime.

The other side of efficiency, where we may not see direct results (unless things take a turn for the worse), is often the side that gets neglected. Datasets that are kept in work library are stored on the system drive while the program is running. Formats and macro variables (often to a much lesser extent) also take up space as they are stored. This may not impact runtime enough to be noticeable, however problems can occur if the program requires more system resources than available.

A small investment in efficiency often can yield good returns in runtime, however taking this too far can result in a lot of hours invested for little additional improvement.

As mentioned, determining how efficient a program is can be difficult. In terms of runtime, instead of measuring, often efficiency can be gauged by a 'feel' for how long it should take. Investigating the program log and identifying specific sections or steps that take a long time to run as being areas for updating can prove to be worthwhile. Also, analyzing the program thinking about the amount of system resources in use can reduce the load on the system.

UNDERSTANDABILITY AND MAINTAINABILITY

Understandability and maintainability are often treated as separate factors, however they often compliment each other, so we will discuss them together. Understandability deals with the ability that another programmer may be able to pick up, understand, and use the program. Maintainability is more concerned with any changes, expansions and extensions to the program that may be required.

A program that is easy to understand will naturally compliment maintainability to a high extent, however a program that is easy to understand may not always be easy to update. For instance, this can occur if the update required carries through the length of the program, rather than being limited to one section.

The assessment of understandability and maintainability can be rather subjective, as this will invariably differ from programmer to programmer. However, with this in mind, the assessor can take a mindful view of whether their code is clear.

Aside from programming structure and keeping the code simple, the use of comments can aid enormously in terms of being able to understand the program and locating code that requires updates.

TIMELINESS

This factor assesses whether the program is delivered on schedule. If a program cannot be used due to it being too late, it is not meeting its purpose.

Within our industry, we are seemingly constantly working to tight timelines that cannot be missed, so surely timeliness must always be a major factor in the quality of our programs? Maybe, but other software industries manage to incorporate this, and we all work to timelines. For example, we can look at operating system software. Two different types of implementation are planned, both have deadlines. One is an enhancement providing more functionality, and the other is an urgent bug fix threatening the security of the system. Missing the deadline for the enhancement may be viable to increase other such quality factors as maintainability, however with the bug fix time is crucial.

Delivering ahead of schedule can also be a viable means of evaluating timeliness. In the previous example, delivering the bug fix ahead of the deadline could have a major impact, however for the enhancement request this may be of minor importance.

The importance of this factor can vary within pharmaceutical programming also, depending on the purpose of the program. For example, if an urgent health authority request with tight turnaround timelines, then this factor becomes of high weighting. Also when the deliverable is to be used can be taken into account. If an output is produced a week ahead of schedule, but isn't to be picked up and looked at until the deadline day, then this is as good as being delivered on that day.

ROBUSTNESS

When evaluating robustness, we are evaluating the program's ability to handle unexpected conditions. We often deal with unexpected conditions in the form of dirty data. Robustness can also relate to the handle possible future conditions, such as a larger set of data at a later date.

Dirty data which goes against assumptions can be highlighted by defensive programming; ensuring that all assumptions made within the programming are upheld within the data.

Robust programming can also be extended to any user-defined input, such as macro parameters. Checks can be put in place to detect any unexpected entries, or the validity of those entries. Validity checks in this context ensuring that any expected existence, such as a dataset or format, specified as an input by the user is available.

We also tend to create many of our programs on pre-final data, and we often find ourselves in the position of rerunning our programs as final on data not encountered by the program. Here the programs need to be able to handle the previously mentioned 'possible future conditions'. Generic dynamic coding can raise this level, for example rather than limiting to a certain number of visits, inputting via a centrally-defined macro variable, or querying to determine the maximum visit within the program to dynamically set the limit.

The use of exception handling also comes into play, but appropriate for the case. Dirty data might just require writing the data to the log as information to warn the user, however an invalid parameter when calling a macro may call for a different, more suitable course of action, such as exiting the program.

In all cases, meaningful error messages to inform the user is helpful.

Testing the level of robustness is tricky, and follows a course much like the maintainability; we need to review our programming code from a different view. Results will undoubtedly be subjective, however reviewing and thinking about how robust our programs are can shed light on problem areas sooner rather than later.

REUSABILITY

Reusability is probably the most difficult of the software quality factors to test, as it assumes knowledge of the future. Rather than being 'reusability', this is often 'reusability potential'. The potential for reuse can be evaluated, as with experience of programming to requirements, common themes can become apparent.

Reusability does not necessarily mean that the program in its entirety need be reused, however utilizing certain parts can also be viable. More often than not, we can identify areas within our programs that may be able to be reused. We can identify these and create or modify them in a way in which reuse is made easier. Modularizing can enable the ease of reuse within programs, and is a common technique used with the utilization of macros.

PORTABILITY

Is the code system or platform dependant? Also, in the case of SAS, is it version dependant?

In our work, we may be required to send programs to health authorities, or a CRO may be required to write programs on their own systems for a sponsor to execute on theirs. In these cases, its quite possible that the program may be required to be self-contained.

Limiting such dependant code to a single section of the code, encapsulating, or even just highlighting the affected code can promote portability. Even if the program isn't completely standalone, the programmer can look to ease the process regarding portability; for example documenting any dependencies that the program has, and where they can be found.

THE BALANCING ACT

As previously alluded to, over-compensating for one factor can have detrimental effect on other factors. Earlier mentioned that over-investment in efficiency could cause the understandability to reduce, also having a knock-on effect on maintainability. On top of this, there are more factors competing with each other.

Efficiency competes with almost everything, with the exception of correctness. Reusability can thrive in programs that are structured where anything that can be reused is kept completely separate, however efficiency is often found in cutting to the bare minimum. Likewise, portability, if taken to the extent of being standalone, and encapsulating all non-portable code into one place, may have impacts on efficiency. Robustness can have a clear negative effect on efficiency, as almost by definition it requires more lines of code to run the additional testing.

Similarly, timeliness also competes with everything. If analyzed in a certain way, one could even come to the conclusion that it competes with correctness, although producing incorrect analyses to hit the timelines is usually not an effective trade-off. If a programmer invest too many hours in improving efficiency, or tidying up the code to improve other factors, then timeliness suffers; especially in a scenario such as our example of the urgent bug fix.

Depending on the implementation of these factors, and to what degree they are implemented, other conflicts are possible. It may be too generic to state that portability affects maintainability and understandability, however implemented in such a way that removes all post-SAS version 8 statements into a separate section may over-complicate the code, making it difficult to follow, understand, and update.

CONCLUSION

What makes a 'good' program? There is no one-size-fits-all answer. It very much depends on the situation and purpose of the program. The 'good' balance of the right factors for the right job should result in a 'good' program.

The quality can be assessed once the program has been completed, during development, and even before commencing development.

An assessment before starting development can provide the programmer with a useful implementation plan. An assessment resulting in what balance of which factors will constitute the optimum program quality for the situation can create a completely different type of specification – a 'Program Quality Specification'. A specification not for what is to be created, but how it is to be created.

With keeping more software quality factors in mind, other than just correctness and a little efficiency thrown in, throughout the development process, perhaps even incorporating into our validation activities too, we can create better, faster, stronger programs.

REFERENCES

Douglas Hoffman http://www.softwarequalitymethods.com/Papers/DarkMets%20Paper.pdf
Marc-Alexis Côté http://profs.etsmtl.ca/wsuryn/research/SQE-Publ/Quality%20model_requirements.%20SQM2006.pdf

ACKNOWLEDGMENTS

Thank you to the following for reviewing this paper:
Timothy Barnett, Roche Products Ltd

Ravinder Bahia, Roche Products Ltd

CONTACT INFORMATION

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

Dean Grundy
Roche Products Ltd
6 Falcon Way, Shire Park
Welwyn Garden City
AL7 1TW
Work Phone: +44 (0) 1707 36 6463
Fax: +44 (0) 1707 38 3145
Email: dean.grundy@roche.com
Web: http://www.rocheuk.com