The Case for Readable Code

Trygve Reenskaug
Dept. of Informatics, University of Oslo

Abstract

Readable code is the key to correct and maintainable programs. Pure class oriented programming does not scale and tends to lead to code that is hard to read. Extensive subclassing is an effective obfuscator and should often be replaced with delegation. A strategy of divide and conquer can be achieved with suitably structured components. This opens a path to readable, object oriented programs. Pair programming and, even better, peer review are work processes that help getting it right the first time.

Introduction

There is no end to the number of different programs that can be executed in a computer. A program may crash or it may go into an infinite loop. It is all the same; the machine executes the instructions given to it.

There is almost no end to the number of programs that will satisfy a given specification and pass the acceptance tests. But tests only cover a minuscule number of the potential executions. Untold glitches and errors may lurk within the untested parts of the code only to be found by undisciplined users who run the program in ways not anticipated by the testers.

There are relatively few programs that will satisfy a given specification, pass the acceptance tests with flying colors, and have been read, understood and accepted by a human reader. These are the "no surprises" programs that blend in with the users' work and that can be adapted to the inevitable changes in the users' needs.

The remainder of this comment is about how to create one of these very desirable programs. This gives me an opportunity to ride a dear hobbyhorse of mine: *The key to quality programs is that the code must be readable, and in addition, that it must actually be read.*

I will first discuss why class oriented programming makes it difficult to write readable code and look at ways to overcome these problems. I end this commentary with discussing some work processes that facilitate the writing of readable code.

Subclassing is evil

The *procedure* in procedure-oriented programming is an ideal unit for independent reading. There is one entry point, one exit point, and well-defined calls upon required services. Compare with the code defining a class with its methods. There are a potentially large number of entry points, each having its own exit point. Required services are invoked indirectly through expressions that give the links to the service providers. This can get very complex, and we will have to restrict ourselves to the subset of all possible classes where the code is readable and checkable.

Subclassing is a very powerful feature of object oriented programming. Common code can be factored out into a superclass; common changes can be done in the superclass and apply equally to all its subclasses. In theory, this practice should be straight forward, but there is a serious snag. Subclassing adds another dimension to the already complex class. It is often necessary to read the superclass code to understand the behavior specification. The services provided by the superclass can be less than obvious and they may have changed since a previous reading. A

reviewer can only trust the superclasses if they are part of a trusted and well-known library.

One or more levels of superclasses may be evolving as parts of the current project. Any previous check of a class is invalidated when one of its superclasses is changed. Do we recheck all the subclasses? I hear the word "refactoring", but semiautomatic refactoring cannot replace code reading in a world of careful code review.

My conclusion is that subclassing should be severely restricted because the inevitable superclass evolution will make us loose control. I suggest that subclassing can often be replaced by *delegation*, thus keeping the number of inheritance levels within reasonable bounds.

I work at finding a discipline of object oriented programming that ameliorates the obfuscation associated with class oriented programming. I believe we need to

- replace most of the subclassing with delegation,
- enforce an object structure that gives us readable code through a strategy of *divide and conquer*.

Delegation with the Andersen Representation objects

In his cand.scient. Thesis, Jørn Andersen discussed how the query operations from relational algebra can be translated into an object oriented context [1]. His premise was that there was a set of encapsulated, black box objects and that the results of the queries should likewise be sets of encapsulated, black box objects. In relational terms, a *relation* became a set of objects, a *tuple* became an object, and an *attribute* became a message.

The SELECT operation is simple; it just returns a subset of the original objects. A JOIN is harder, the result should appear as instances of a new class with attributes from both the argument classes. Andersen's solution was to introduce a *Representation* class. An instance of this class associates the message names with the objects that shall handle them. His solution is similar to the *Facade* pattern [2], but he utilized the stored program facility of Smalltalk to make all facade objects instances of the same, *Representation* class. The result was a very powerful delegation mechanism. A Representation object will dynamically and automatically extend its interface when a new handling object is added. A Representation object will dynamically and automatically shrink its interface when an object is removed.

The Andersen Representation object appears to give dynamic, multiple inheritance in a simple way. It should be explored as a readable replacement of uncontrolled subclassing.

Divide and conquer with components

The Facade pattern and the Andersen Representation objects are both open constructs; their handling objects may simultaneously participate in other constructs. This may lead to unduly complex structures that make the code hard to read. I close the constructs by defining a *component* as an object that encapsulates a number of *member objects*. Being encapsulated, a component is characterized by its *provided and required interfaces*. Components are ideal building blocks in a strategy of divide and conquer since they divide the object space into manageable parts. A reader can check the code for each provided operation in turn. The code is local to the component; any required services are captured in the required interfaces. The implementations of the required interfaces are in other components that can be checked separately.

An Andersen Representation can serve as the port into a component. The member objects can be structured according to a suitable paradigm in order to make the code more readable (and thus reliable). An example is the DCA (Data-Collaboration-Algorithm) paradigm discussed in [3].

Testing cannot inject quality into an inferior product

Testing has never been in the forefront of my attention because no industry has ever been able to test quality into an inferior product. In programming, this means that it is too late to save an inferior program when it gets to the testing stage. The best is to focus on getting it right the first time and use testing to check that we have made no serious blunders.

A minimum requirement for a reasonable test is that all statements shall have been executed at least once. Otherwise, a program may pass the tests while glaring deficiencies remain unnoticed. This requirement is hard to satisfy in procedural programming, and I have never been able to satisfy it with objects. I still work at finding a discipline of object oriented programming that facilitates this reasonable level of testing.

Literate programming

Donald Knuth had proposed the notion of *literate programming*. The main idea was to treat a program as a piece of literature in the form of a textbook. The text could be formatted and printed like a book, and a filter could extract the code and compile it. This idea fitted very well with our idea that a program should primarily be written for human reader and it looked like a good idea to develop code and textbook together. So we extended our multimedia authoring tool with two new media: Smalltalk class definitions and Smalltalk method definitions. Our experience with literate programming was reported at OOPSLA-89 [4]

A colleague and I worked together on a major project where we wrote the code directly into a literate programming document. This combination of authoring and coding was very inspiring and great fun. We were highly motivated to write about a good idea whenever we hit upon it. We once saw an obvious way to optimize a certain method. We worked on it for some time before discovering a catch; the optimization could not work. There and then we were very motivated to write a warning to future maintainers directly below the relevant code in the document. The next day would have been too late; we were on to other problems and other solutions.

Literate programming worked beautifully until we got to a stage where we wanted to refactor the program. The program structure was easy to change, but it implied a radical change to the structure of the book. There was no way we could spend a great deal of time on restructuring the book so we ended up with writing appendices and appendices to appendices that explained what we had done. The final book became unreadable and only fit for the dustbin.

The lesson was that the textbook metaphor is not applicable to program development. A textbook is written on a stable and well known subject while a program is under constant evolution. We abandoned literate programming as being too rigid for practical programming. Even if we got it right the first time, it would have failed in the subsequent maintenance phases of the program's life cycle.

Pair programming

Dijkstra is the source of many pregnant maxims such as *program testing can be used to show the presence of bugs, but never show their absence* and *nothing is cheaper than not introducing the bugs in the first place.* [5] This is all well and good, but easier said than done.

One of the keys to success is to admit the fallibility of humans and make sure that at least two people are involved in the creation of a program. One solution is to let two programmers work together in front of a common computer. I have one very successful experience with this mode of working when a colleague and I developed the base architecture and core program of a family of software products. We spent almost a year on the architecture before getting down to the concrete programming. We were a very good team; one being creative and jumping to conclusions, the other insisting on stopping to think whenever the code was less than obvious.

(This effort was also an exercise in literate programming as described previously). Other attempts at pair programming have failed because conflicting personalities made the effort more competitive than cooperative.

I believe that pair programming can be very effective under the right conditions. If a fairly stable team is working on a common body of programs, the pairing can be varied. All team members get to know the programs intimately well and any of them can confidently work on the inevitable changes and extensions.

Pair programming is still not the ideal solution because two programmers working closely together can easily fall into the same trap. Further, they get intimately acquainted with the intricacies of the programs so that the code may not be readable for a future, uninitiated maintainer.

Peer review

I believe it was an article in the *Datamation* magazine some time in the sixties that first brought peer review to my notice. It sounded great. I had just written a FORTRAN subroutine and ran around to find somebody who was willing to read and comment it. I finally persuaded a colleague to do so.

All the benefits mentioned in the Datamation article were attained in this first exercise. First, my colleague pointed out that a certain statement could be improved by using a FORTRAN feature I was not aware of. Second, my colleague asked me to explain the exact meaning of another statement that he was unfamiliar with. And finally, my colleague found a bug I would never have found by blind testing. The program worked beautifully for N<1000. It failed gracefully for N>1000. But it crashed for N=1000. Careful reading of the code might have highlighted the number 1000 as critical so that it should have a special test. But such reading would have revealed the bug and the test would have been superfluous.

The outcome of this first attempt at peer review was that both my colleague and I learnt something new in addition to the main result of a bug free subroutine. All this achieved at the cost of 15 minutes proof reading.

We used peer review in all our work from that day on. Every subroutine had two comments: One identified the original programmer and another identified the reader. The important feature was that it was the reader who was responsible for the correctness of the code. In the rare case of a bug, the original programmer could point at the reader and say: “your fault!”

When I read my own code, I know what it is supposed to say and naturally assume that I have written what I intended. My colleague has no such pre-conception and he knows perfectly well that I am fallible. His chance of finding a deficiency is far better than mine when we read the same code.

I said above that code should primarily be written for human reader. With peer review, there is an immediate check the code is indeed readable. We can be reasonably certain that future maintainers can read and understand the code since the reviewer has already done so.

Conclusion

Program testing can never show the absence of bugs. Indeed, *the more bugs we find during testing, the more bugs remain in the shipped product.* (Because a given test regimen can only find a certain percentage of all bugs). Contrast with a competent reviewer who reads all the code and can reveal bugs, glitches, clumsy code, and potential traps for a future maintainer.

Effective code reading is only feasible if the code is partitioned into reasonably independent chunks and if the remaining dependencies are well defined. I have tried using peer review in object oriented programming, but have as yet not succeeded because I have not been able to partition the system into reasonable chunks. I expect that some changes to my programming method will help:

- Subclassing must be kept to a minimum both to reduce system complexity and to

make it less vulnerable to the inevitable program evolution.

- Subclassing can often be replaced by delegation.
- Chunking objects into components with a corresponding chunking of the code is essential.
- Literate programming is tempting for an example educational program, but is too rigid for general programming.
- Pair programming is powerful but may not lead to chunked and readable code.

My pet idea is peer review. May it become an essential element in the creation of quality programs.

References

1 Andersen, J. *Queries on sets in an Object-Oriented Database*; Cand. Scient. Thesis; Department of informatics, University of Oslo; 1991.

2 Gamma, E,; Helm, R,; Johnson, R; Vlissides, J. *Design Patterns*; ISBN 0-201-63361-2; Addison-Wesley, Reading, MA. 1995.

3 See my chapter on *Programming with roles and classes* in this book.

4 Reenskaug, T.; Skaar, A. L. *An environment for literate Smalltalk programming*; OOPSLA 1989; ISBN: 0-89791-333-7; ACM Press; New York, NY 1989; pp. 337–345.

5 For more Dijkstra contributions to computing, see *E. W. Dijkstra Archive;* University of Texas; [web page] http://www.cs.utexas.edu/users/EWD/