DOUBLE PROGRAM ENTRY AND COMPUTER-AIDED CODE REVIEW

Herwig Egghart, Scientific Games International

October 20, 2009

Abstract:

Four decades after the first NATO conference on software engineering, the quality of software is still suffering severely from programmers' human fallibility. Although many of the resultant defects are just typing errors or simple oversights, eliminating them by conventional code review is a tedious and error-prone endeavor. This paper shows how to mitigate the impact of human failure by duplicating the implementation process and feeding both outputs into a computer-aided review process. The effect is a character-by-character code review of unprecedented efficiency and reliability.

1. DOUBLE DATA ENTRY

In the early days of computing around 1970, my father helped establish a state-of-the-art punchcard system at the oil company he was working for. As a result, one of the tales I grew up with was how raw-data sheets from bookkeeping were processed into punchcards by a team of two keypunch operators. The first operator started with an unpunched card in a keypunch machine and struck a sequence of keys based on what she saw on the raw-data sheet. Then she passed the punched card and the raw-data sheet to the second operator, who would put the card in a verifier machine and strike the appropriate keys as well. A red error light appeared whenever holes and keystrokes didn’t match, in which case both operators would take a closer look to find out whose fault it was.

Good keypunch operators were able to punch (or verify) several thousand holes without error; and when an occasional mistake happened to one operator, chances were excellent that the other operator got it right, so the error was quickly found and fixed. This in turn meant that the quality of each operator’s work could be measured (in terms of keystrokes per error), and in fact this statistical feedback was a major source of pride and motivation in an otherwise rather boring job.

The era of punchcard computing has gone by, but the method of double data entry continues to be used in various contexts. Whenever you set a new computer password, you have to enter it twice to catch possible typing errors. Commercial software packages offer double data entry for reliable business-data input, and even scholars who digitalize ancient books appreciate this method as a cost-effective way to obtain error-free transcriptions. Yet, in spite of ubiquitous complaints about software quality, a similar approach has not been proposed for programming so far.

2. DOUBLE PROGRAM ENTRY

In 1994, I witnessed a keynote address by Watts Humphrey, in which he talked about the power of code reviews in the Personal Software Process (PSP). In particular, the "yield" of defects found before testing was presented as a vital quality factor, whereas the role of testing shrank to a filter only able to scale down the remaining number of defects by some percentage.

In his PSP book "A Discipline for Software Engineering" [1], Humphrey recommends to review new code before even compiling it. He argues that a search is more motivating if there's more to find, and that the number of compile-time errors tends to correlate with the overall number of defects in the code. Thus, a "clean first compile" is a good sign of quality and a rewarding experience for the person who thoroughly wrote and reviewed the code.

From a psychological perspective, the challenge is to resist the temptation of premature compiling and testing. Ideally, programmers should dwell on the program text till they feel confident with every single character, similar to keypunch operators' confidence in the holes they had punched and verified. So in order to approximate this level of confidence, we shall now translate the keypunch workflow of the seventies into a programming method for the twenty-first century.

2.1. The Raw-Data Sheet

Every serious programming task has to start with a written description of the work to be done. This could be a (more or less formal) specification or design document, or a particular section thereof. It could be the description of a feature to be added, or of a defect to be fixed. Maybe it's just a functional description with the program logic yet to be designed; maybe it's some algorithmic pseudocode ready for translation into the final programming language. Whatever its actual form may be, the point is that it serves as a stable basis for the programmer's keystrokes and thus corresponds to the raw-data sheet processed by keypunch operators. Hence, in the sequel the term "raw-data sheet" will be used in both cases.

Of course, there is one important difference between raw-data sheets in the keypunch and the programming world. Keypunch operators had no degree of freedom when they punched a card for a given raw-data sheet. By contrast, programming requires many little decisions all the time, and the resultant keystrokes depend on the raw-data sheet plus these additional implementation decisions. In fact, making good decisions based on existing code and incoming raw-data sheets is the intellectual challenge of programming in the first place; but then there is always the second challenge to avoid "silly" typing errors and oversights, and before that latter challenge all human beings (whether keypunch operator or software engineer) are equal.

2.2. The Two Operators

The next question is how to reflect the separation between the operator at the keypunch machine and the operator at the verifier machine. Clearly, if we simply gave the raw-data sheet to different programmers for independent implementation, we would essentially end up with N-version programming, where all the versions have to coexist at run time and pass their results to some kind of voting mechanism (see e.g. "Designing for high integrity: The software fault tolerance approach" by M. R. Moulding [2]).

However, since our goal is to obtain a single version free of typing errors (rather than N versions without such assurance), we have to introduce a certain amount of dependency between the two implementation passes. Ideally, every implementation decision of the first pass should be available during the second pass, but not in written form in order to minimize single points of failure. (Note that without this restriction, the second operator might as well use the code from the first operator rather than the raw-data sheet, which is of course strictly forbidden.)

So either the same programmer has to fulfill the role of both operators, or a colleague who witnessed the entire "keypunch" pass becomes the new operator for the "verifier" pass (which would be a combination of pair programming and double program entry). In both scenarios, the point is that all implementation decisions have to be reconstructed from memory while operating the "verifier machine". That way, a high degree of independence is achieved while at the same time avoiding the N-version pitfall mentioned above.

This is not to say that N-version programming and double program entry couldn't be combined in a meaningful way. If each of the N implementors performed double program entry, the quality gain in each of the N versions would certainly be no disadvantage to the system's overall reliability.

2.3. The Punchcard

The fact that every implementation detail has to be reproduced puts a natural limitation on the amount of code that can be written during one cycle of double program entry. Unlike keypunch operators, who simply punched and verified one card after the other, programmers are faced with work packages of variable size and complexity, some taking several days or weeks to complete. So again, a literal translation of the keypunch workflow won't work, because the information to be memorized would be prohibitive for many practical programming tasks.

On the other hand, experience shows that large and complex raw-data sheets can always be decomposed into meaningful parts or increments, which can then be implemented as a sequence of multiple "punchcards" with double program entry used for each of them. If the raw-data sheet describes a sequence of processing steps, for example, the most fundamental steps could be tackled first and additional steps saved for later punchcards. Or if the raw-data sheet contains a list of required changes, a natural decomposition would be one punchcard for every item on the list.

The ideal punchcard is a logically coherent sub-task of manageable size, whose implementation should take no longer than a couple of hours. If it's too small to make stable implementation decisions, many decisions will have to be overthrown by subsequent punchcards, which is rather inefficient and error-prone. However, if it's too big to be kept in mind at one time, the results of two implementation passes won't have enough in common to be useful for typing-error detection.

In my experience, it has seldom been a problem to find good punchcards and decide in which order to implement them. I usually make a hardcopy of the raw-data sheet and tag suitable punchcards by handwritten symbols, which I cross out during implementation to keep track of what's left to do. A punchcard may span several software modules, but it should be coupled with other punchcards as little as possible. Of course, certain unidirectional dependencies are inevitable, but they can be easily addressed by implementing dependent punchcards after the punchcards they depend on.

2.4. The Keypunch Machine

When the raw-data sheet has been analyzed and structured into one or more punchcards, it's time to sit down at the keypunch machine and start with the first punchcard. The "keypunch machine" is just the normal programming environment, consisting at least of a version-control system and a source-code editor, either working directly on operating-system files or embedded in some integrated development platform.

My own experience is based on operating-system files which I reserve from version control and manipulate with a traditional text editor, so my description of double program entry will be based on this assumption. However, I see no principal problem with the generalization to more sophisticated environments, and I encourage all suppliers of such environments to remove possible obstacles and provide user-friendly support of double program entry in the future.

The most important rule with regard to the keypunch machine is to work exclusively on one punchcard at a time. For each module affected by the punchcard, a well-defined source code has to exist in version control before and after the punchcard has been implemented, and the difference between those two versions should reflect just that punchcard and nothing else. (Any ideas pertaining to future punchcards may be collected as written notes, but they are not allowed to enter the actual source code yet.)

Another rule is to refrain from compiling and testing at this early stage. Even if there are uncertainties about programming-language features or library routines, it's still better to stop and do some "basic research", e.g. by reading documentation or performing little experiments with toy programs. The resultant insights can then be used to properly finish the implementation of the punchcard in the program text. (The idea of separate experimentation is an input from the Cleanroom community; see e.g. Michael Deck's paper in the 1994 proceedings of the Pacific Northwest Software Quality Conference [3].)

To punch a card isn't always straight-forward in the world of programming. Sometimes it's necessary to explore different choices to see how well they fit into the structure of existing code. So whenever you hit a dead-end or discover a simpler and cleaner alternative, don't hesitate to undo some of your changes and improve the implementation till you honestly feel, "If I ever had to do this again, that's how I'd do it."

2.5. The Verifier Machine

We have now reached the point where double program entry departs radically from today's software-engineering mainstream. So before further pursuing this novel path, let us analyze the situation immediately after leaving the keypunch machine. At this point, the programmer has a fresh memory of the punchcard just implemented and a clear mental picture what the new code should look like. The question is how to best exploit this knowledge to make sure every character indeed matches the programmer's intention.

The traditional answer has been meticulous proof-reading. But the human mind is not a digital computer, and the intended code is not available as a low-level character stream which can be easily compared with the actual code to be reviewed. On the contrary: The mental effort to reconstruct intended characters is constantly undermined by the actual characters before one's eyes; and the more attention shifts down to the character level, the harder it gets to maintain the necessary high-level overview. Hence my proposal to first make the intended code explicit (by actively re-typing rather than just reading the new characters) and then simply check if both character streams are the same.

The "verifier machine" is similar to the keypunch machine from the previous section, except that it operates on a second copy of the original code reserved from version control. I personally prefer short file names like "X" and "Y" (plus language suffix) for this second copy, which has the advantage of a clear optical separation between files belonging to the keypunch machine and files belonging to the verifier machine. However, since these are just temporary names not intended to be shared with other people, any naming convention is fine as long as it's convenient for the programmer to work with.

Before the second implementation pass begins, all artifacts from the first implementation pass have to disappear from the programmer's desk as well as from the computer screen. Only the underlying raw-data sheet is supposed to be visible, and the first step of the second pass is to read the relevant parts of the raw-data sheet once again. If it turns out that some detail has been missed during the first implementation pass, this is still a good time to go back to the keypunch machine and fix the code before finally switching to the verifier machine.

To take full advantage of double program entry, I recommend a pilot's discipline and concentration for the second implementation pass. There should be no need for trial and error any more, as in the first implementation pass. That was just the flight simulator, so to speak, which allowed the pilot to get acquainted with the mission and work out all the details. Now it's time for a clean and flawless flight, and the computer keyboard should be touched as carefully as control panels in a cockpit. Again, the programmer may become aware of certain issues missed before, but the flow of the second implementation should continue, and the first implementation should be fixed afterwards (without looking at the second implementation).

Of course, it can also happen that the second pass reveals a fundamental problem not yet realized during the first pass. When getting stuck with such an unexpected problem, it's often necessary to fall back into try-and-error mode again. This is still better than not finding the problem at all, but it means that the first implementation is obsolete, the second implementation assumes the role of the first implementation, and a third implementation is required to secure the second implementation. If necessary, this has to be repeated until there are two final implementations of the punchcard, at least one written in a rather straight-forward way.

2.6. The Red Error Light

After the arduous part of double program entry comes the more relaxing part of computer-aided code review. The unreliable connection between human mind and computer keyboard has been bridged with two-fold redundancy, so all characters which came out twice identically can be assumed to reflect truly what the programmer had in mind. What's left is to take care of non-identical characters, which first need to be identified before they can be resolved. And since this search requires low-level precision but no high-level understanding, it's a perfect job for a digital computer.

The "red error light" of the verifier machine can be simulated by whatever file-comparison tool is available on the development platform. Ideally, the tool is smart enough to show all the differences between two files at once, but it's also usable if it only shows the first differing line of both files. (In the unlikely case that no appropriate tool exists, it might be worthwhile to develop one.) For each module affected by the punchcard, the two implementations have to be compared; and as long as one or more differences pop up, the red error light is considered to be burning.

In my experience, the most efficient procedure is to redirect the file-comparison output to another file and then load all the three files into the editor. That way, I can scroll from one difference to the next (in the upper window) and conveniently switch between keypunch and verifier machine (in the lower window). Not every difference is necessarily a defect, but it's always an opportunity for the programmer to think about the code fragment once again, make a conscious decision, and manually enforce that decision in both copies of the code.

While resolving differences, it's never wrong to launch another file comparison in between. It's nice to watch the number of differences go down, and occasionally a difference might "survive" within a line already touched to resolve another difference. That's also the reason why no shortcuts are allowed during difference resolution: If you simply copied the "right" code over the "wrong" code, you would circumvent double program entry and thus risk a single point of failure.

Eventually the red error light goes out, and the two outputs of double program entry have converged into one consolidated new version of each module. Whatever single mistake happened during the keypunch pass, the verifier pass, or while resolving differences, the rules of computer-aided code review make sure that only the "fittest" characters of both implementations survive. Hence, although the resultant code has never been compiled yet, it's typically cleaner and more reliable than conventional code is after eliminating compile-time errors.

2.7. Pride and Motivation

Having said all this, does it come as a surprise that after computer-aided code review there are often no compile-time errors left at all? In fact, Watts Humphrey's "clean first compile" has become an everyday experience since I incorporated double program entry into my personal software process in 2003. Yet, I still feel as thrilled as the students in Humphrey's PSP book whenever the work of several hours compiles instantaneously without a single error message.

In the meantime, I have spent thousands of hours using double program entry, and I'm not sure this would have been possible without the recurring challenge to "beat the compiler" in terms of defect removal. But with this additional incentive, it definitely works in the long run for disciplined software engineers; and the hardship of double program entry is rewarded by a great feeling of professionalism, similar to the pride and motivation that helped keypunch operators in their daily struggle for error-free punchcards.

Nevertheless, occasional compile-time errors can occur, and now it's important to keep up the spirit of double program entry lest a single point of failure sneak in at the very last moment. Every correction of compile-time errors has to take place in both copies of the current code, and afterwards another pass of computer-aided code review is due. When the two copies (which still originate from independent keystrokes) are equal, the modified code may run through the compiler once again. As long as yet another compile-time error appears, the whole cycle has to be repeated.

When neither computer-aided code review nor the compiler finds anything wrong with the finished punchcard, it's time to replace the new module versions into version control. If the raw-data sheet contains additional punchcards, they are subsequently implemented one after the other. And if further changes become necessary due to run-time problems or new requirements, they should also be prepared as raw-data sheets, structured into punchcards, and implemented in the way just described.

3. POSSIBLE OBJECTIONS

I don't know to what extent our programming culture is ready for this new method yet. Although fairly simple and immediately applicable, the approach might sound so strange to many people that it could take a long time before it has a noticeable impact on the way industrial software is developed and maintained. This section is an attempt to remove initial barriers, dissolve certain doubts and objections, and shed some light on aspects of double program entry not discussed so far.

3.1. Doesn't it take twice as long to enter everything twice?

Not at all. Any programmer who has ever lost a half day's work to a system crash or handling error knows how surprisingly fast such a loss can be recovered based on memories from the first implementation pass. The point is that all the thinking and problem-solving has been done already, so the second pass is mainly concerned with typing a known solution into the keyboard once again. Now suppose for the sake of argument that the lost implementation were somehow found again: Wouldn't that be a great opportunity to compare the two versions and merge the best of both into one version? And why not create such a situation on purpose if it leads to higher code quality?

I think the main question is not how many minutes are consumed by the second implementation pass, but whether it's time well spent if the goal is a proper character-by-character code review. And since the second pass is typically much faster than the first pass, it has the additional advantage that the programmer can actively relive everything in a time-lapse fashion, which fosters a coherent global overview and reduces the likelihood of omissions and inconsistencies.

3.2. Is repetitive labor appropriate for highly qualified engineers?

Well, consider highly qualified specialists like surgeons or astronauts, who repeatedly practice every step of an operation before they carry it out in a living body or in outer space. Nobody would expect them to do everything just once and get it right immediately. Why should programming be different in this regard? And if we agree, after decades of supporting evidence, that high-quality software is only achievable through discipline and hard work, then why not tolerate a certain amount of repetition as well? From an engineering viewpoint, I find that much more appropriate than an arbitrary number of review iterations aimed at finding wrong characters with the naked eye.

There are certainly greater pleasures than a second implementation pass, although it can be sometimes quite refreshing to calm down one's creativity and focus just on typing the right characters. Moreover, the monotony of the second pass can be mitigated by watching out for error-prone maneuvers and shielding them with appropriate countermeasures. A standard example are copy-and-paste maneuvers, which always bear the risk that some of the necessary adjustments are forgotten. However, abolishing copy-and-paste is no solution either, because then the risk would be to lose some of the desired similarities. With double program entry, both risks can be easily avoided by using copy-and-paste in one but not in the other implementation pass. (Another technique is to copy and paste from two distinct sources that are known to share just the desired similarities.)

3.3. What if the raw-data sheet is wrong to begin with?

In other words: Isn't it more important to avoid specification and design errors than to indulge in source-code character-level perfectionism? My plain answer is that programming deserves as much attention and respect as any other phase of the software process. Even if the raw-data sheet is a detailed design leaving almost no decisions to the programmer, there is still enough that can go wrong during a sloppy implementation phase. So yes, a high-quality raw-data sheet is important, but it can never guarantee a high-quality end product by itself. On the other hand, a clean and careful implementation process is often able to reveal weaknesses in the input raw-data sheet (or even beyond), thus acting as additional quality control for previous phases.

Speaking of previous phases, I should also mention that double program entry can be generalized from programming languages to more abstract (semi-)formal languages used in software engineering. I have tried this idea to secure some critical parts of semi-formal specifications, and the computer-aided review process turned out as effective and efficient as in case of computer-aided code review. So if applied early enough in the development process, the double-entry approach can already raise the quality of raw-data sheets before the implementation phase begins.

3.4. What if a mental error enters both implementations?

Then computer-aided code review won't detect it, which simply reflects the fact that there are different kinds of defects, and double program entry specializes in oversights and typing errors. However, the method has a positive effect on the mental side of programming as well: The incremental punchcard structure is a good basis for intellectual control; the creative first pass encourages an iterative search for good solutions; and the clean second pass consolidates the result with good chances to discover mental errors made before. In short, double program entry helps programmers develop their full potential by drastically reducing "unnecessary" defects. And on top of this solid foundation, more advanced methods can work effectively to cut down mental errors even further.

3.5. What about code-layout and comment variations?

This may sound like a minor issue, but it can hamper the flow of computer-aided code review significantly if not handled properly. For programming-language text, the key to a smooth review process is a good coding standard which minimizes the need for ad-hoc formatting decisions (or alternatively the use of an automatic code formatter).

As for comment text, my general advice is to keep it as small and concise as possible. Not the quantity but the quality of comments makes them valuable for program comprehension, and the foremost goal should always be to maximize the clarity of the non-comment text actually compiled. If a block of code has a lucid structure supported by meaningful identifiers, it's often sufficient to put a one-line block comment in front (and perhaps embed a few well-placed line comments for additional hints).

Of course, there can be situations where a more elaborate paragraph is necessary to explain some relevant design rationale or other important background information. But if the whole code teems with verbose comments telling only what the program text says anyway, it gets unwieldy not just for double program entry but also from a general readability/maintainability perspective. So on a positive note, double program entry fosters a succinct and disciplined commenting style as well as utmost layout consistency; and whenever a deviation happens in these areas, it should be seen as a chance to pick the most appropriate variant in computer-aided code review.

4. SUMMARY AND OUTLOOK

This concludes my detailed presentation of double program entry. I hope that it contains some useful thoughts for software engineers who seek supreme code quality, and that my positive experience will encourage them to try the method out in their daily work. In order to facilitate the practical application and evaluation of double program entry, I have summarized its essence in fourteen rules to be found in the appendix of this paper.

I also hope that academic researchers will get interested in the approach, conduct empirical studies to quantify its benefits, and make a larger public aware of its potential. Their findings may convince industry leaders, notably in the realm of high-integrity and safety-critical systems, that double program entry is an effective weapon against many single points of human failure. This in turn could stimulate improved tool support for various platforms, which should further increase the method's visibility and attractiveness for a growing number of organizations.

My ultimate vision is that teachers will employ double program entry as a framework to convey sound software-engineering principles to young people from the very beginning. By educating a new generation of conscientious engineers who have learned to adjust to their human imperfection rather than persistently ignoring it, the way to a more mature programming culture could be paved: A culture in the spirit of Edsger W. Dijkstra's "Humble Programmer" [4], that values clean programming in the small as the soil on which superior programming in the large is able to thrive.

References:

[1] W. S. Humphrey. *A Discipline for Software Engineering.* Addison-Wesley, 1996.

[2] M. R. Moulding. *Designing for high integrity: The software fault tolerance approach.* In *High-Integrity Software,* Plenum Press, 1989.

[3] M. Deck. *Cleanroom Software Engineering: Quality Improvement and Cost Reduction.* In *Twelfth Annual Pacific Northwest Software Quality Conference,* Portland, OR, 1994.

[4] E. W. Dijkstra. *The Humble Programmer.* ACM Turing Lecture 1972, Commun. ACM 15 (1972), 10: 859-866.

APPENDIX: The Fourteen Rules of Double Program Entry

1. Don't trust any keystroke which has been performed only once.
2. Structure all coding activity into increments you can keep in mind at one time.
3. Before starting an increment, make sure the old code is saved in version control.
4. Implement and polish one increment till it's as simple and clean as possible, BUT DON'T COMPILE YET.
5. Get a second copy of the old code from version control and enter the whole increment once again (without looking at the first version or any notes you took for it).
6. Take advantage of diverse editing strategies, e.g. copy/paste/adjust a code fragment in the first version but enter it from scratch in the second version.
7. If you realize a weakness and improve it in the second version, go back and manually adjust the first version afterwards.
8. If the improved second version deviates significantly from the first version, discard the first version and start a third version instead.
9. When you have two stable versions stemming from independent keystrokes, let the computer show the differences between them.
10. Analyze each difference as a potential defect and eliminate it by manually changing one of the versions (or both if necessary).
11. When the computer shows no more differences between the two versions, compile the new code for the first time.
12. In case of any compile-time errors, fix them independently in both versions, compute and eliminate differences, and then compile again.
13. When there are no more compile-time errors, save the new code in version control and implement the next increment in the same way.
14. In case of new incoming change requests or defects found during code execution, schedule them as separate increments and continue to secure every keystroke by double program entry and computer-aided code review.