Tick-the-Code Inspection: Empirical Evidence (on Effectiveness)

Miska Hiltunen
Qualiteers, Bochum, Germany
miska.hiltunen@qualiteers.com

Prepared for, and presented first at the **Annual Pacific Northwest Software Quality Conference 2007 (www.pnsqc.org)**

Miska Hiltunen teaches **Tick-the-Code** to software developers. After developing the method and the training course for it, he founded Qualiteers (www.qualiteers.com). Mr. Hiltunen is on a mission to raise the average software quality in the industry. He has been in the software industry since 1993, mostly working with embedded software. Before starting his freelance career, he worked for eight years in R&D at Nokia. He graduated from Tampere University of Technology in Finland with a Master of Science in Computer Science in 1996. Mr. Hiltunen lives and works with his wife in Bochum, Germany.

Abstract

This paper demonstrates that as software developers we introduce a lot of inadvertent complexity into the software we produce. It presents a method for removing inadvertent complexity and shows how any software developer can easily learn to identify it in source code.

The paper starts with a hypothetical scenario of software development showing how bugs can come into being essentially from nothing. The paper also claims that the current ways of producing software leave much to be desired. The main argument is that there is a lot of inadvertent complexity in the software produced by the industry and that it is possible and feasible to get rid of.

The paper presents four experiments and their results as evidence. All experiments use the **Tick-the-Code** method to check source code on paper. The experiments show that both the developers and the source code they produce can be significantly improved. The results indicate that, almost regardless of the target source code, the developers can easily find and suggest numerous improvements. It becomes clear from the results that it is feasible to use **Tick-the-Code** often and on a regular basis. In one of the experiments, the software engineers created almost 140 improvement suggestions in just an hour (of effort). Even in the least effective experiment, the participants created on average one suggestion per minute (70/h).

The last part of the paper demonstrates the effects of ticking code often and on a regular basis. For a software organization, nothing makes more sense than to improve the coding phase and make sure it is up to par. Once inadvertent complexity is kept in check on a regular basis, other concerns, like requirements analysis, can be more readily taken into consideration. As long as the organization has to waste time on reworking requirements and careless coding, maturity of operation is unachievable.

1. The Claim: Complexity Hole Causes Bugs

The software industry still shows signs of immaturity. Errors are part of usual practice, project failures are common and budget overspends seem to be more the rule than the exception. The industry is still young compared to almost any other branch of engineering. Tools and methods are changing rapidly, programming languages keep developing and ever more people are involved in software projects. The industry is in constant turmoil.

1.1. A Development Episode

Let's dive deep into a developer's working life. John, our example software developer, is about to create a new class in C++. The requirements tell him what the class needs to do. The architectural design shows how the class fits with the rest of the application. John starts writing a new method. The method starts simply, but soon the code needs to branch according to a condition. John inserts an `if` statement with a carefully considered block of code for the normal case of operation. In his haste to finish the class in the same day, John forgets to consider the case when the conditional expression isn't true. Granted, it is unusual and won't happen very often.

```
if(FLAG1 & 0x02 || !ABNORMAL_OP)
{
    header(param, 0, 16);
    cnt++;
}
```

One of the class member variables is an array of integers for which John reserves space with a plain number

```
int array[24];
```

In the vaguely named method `ProcessStuff()`, John needs among other things to go through the whole array. This he accomplishes with a loop structure, like so

```
for(int i=0; i <= 23; i++)
```

In order to send the whole array to another application, the method `Message()` packages it along with some header data in a dynamically reserved array

```
Msg *m = new Msg(28);
```

As it doesn't even cross John's mind that the code could run out of memory for such a small message he doesn't check for the return value from the statement.

By the end of the day, John has tested his class and is happy with the way it seems to satisfy all functional requirements. In two weeks' time, he will change his mind. The system tests show several anomalies in situations John would consider exotic or even impossible in practice. The tests need to pass though and John has no alternative but to

try and modify his class. This proves harder than expected and even seemingly simple changes seem to break the code in unexpected ways. For instance, John needs to send more integers over to the other application, so he changes the `for` loop to look like

```
for(int i=0; i <= 42; i++)
```

That change breaks his class in two places. It takes several rounds of unit testing for John to find all the places related to the size of the `array` in the class. The hard-coded message packaging routine `Message()` stays hidden for a long time causing mysterious problems in the interoperability of the two applications. The code works erratically, sometimes crashing at strange times and other times working completely smoothly.

This development example shows how seemingly innocent design decisions lead to complex and strange behavior. John's slightly hurried decisions are often mistakes and oversights, which cause real failures in the application. The failures affect the test team, frustrate John, anger his manager and in the worst case, cause the customer to lose faith in the development capability of the company employing John. There is a lot of seemingly innocent complexity in source code produced today all over the world. Taking it out is possible and even feasible, but because of a general lack of skill, ignorance of root causes of failures and a defeatist attitude towards faults, not much is being achieved. Constant lack of time in projects is probably the biggest obstacle for more maintainable and understandable code.

Although complexity in software is difficult to quantify exactly, we can divide it into two categories for clarification:

a) essential complexity (application area difficulties, modeling problems, functional issues)
b) inadvertent complexity (generic programming concerns, carelessness)

This paper deals with b) inadvertent complexity.

The current working practices in the software industry leave a gaping hole for inadvertent complexity to creep into the source code. In large enough doses, it adversely affects all activities in software projects, not just coding. Inadvertent complexity is rampant in source code today, yet currently there are no incentives or effective methods in use to get rid of it. Inadvertent complexity is holding the software industry back. Inadvertent complexity is an unnecessary evil, **it does not have to be this way**.

2. The Evidence: Complexity NOW

Next four real-life experiments. A few terms related to the **Tick-the-Code** method used in the experiments need to be explained. 'A tick' in code marks a violation of a rule. A tick doesn't necessarily mean a bug, in most cases it denotes an environment where bugs are more difficult to identify. Some ticks precede bugs. The best way to think about ticks is to see some of them (perhaps every third) as realistic improvement suggestions. 'A rule' is a black-and-white statement. Code either follows a rule or it doesn't. "Do not divide by zero" is a rule. Each rule has a name, which in this paper is written in capital letters (RULENAME). 'Ticking' refers to a checker going through source code marking rule violations one rule at a time. In other words, ticking produces ticks. 'An author' is the person responsible for maintaining the source code, not necessarily the originator. 'A checker' ticks the source code. 'A checker' needs to be familiar with the programming language (although not necessarily with the source code.) For more information, see Chapter 3. or [2].

2.1. Experiment: Free Ticking for an Hour

In this experiment, the participants were taught **Tick-the-Code** as a theoretical exercise. They were then given an hour to tick their source code using any of the predefined rules. The participants had been instructed to bring approximately one thousand physical lines of code with them. Table 1. summarizes the results.

Participants	Total time	Total lines (physical)	Total ticks	Average rate of finding
85	81h45min	93393	5900	72 ticks/h

***Table 1. Experiment: Free Ticking for an Hour**. With proper guidance, it takes on average less than a minute for a software developer to generate a tick (= source code improvement suggestion). The experiment took a little more than a week to conduct between Feb 6 and Feb 15, 2006.*

2.2. Experiment: Before and After

Almost eighty software developers from six software companies took part in **Experiment: Before and After**. The participants were first asked to "perform a code review" on their own code. They were given 15 minutes to complete the task and no further explanation as to what "performing a code review" means. After fifteen minutes, the number of markings they made was collected. The severity of the markings was not judged. The collected results were extrapolated to a full hour. The participants were then taught a new method of checking called **Tick-the-Code**. This took about two hours. They were given an hour to tick the same code as before. The number of ticks was collected.

The graph in Figure 1. shows the 'before' and 'after' performances of the 78 participants. In the graph they are ranked in ascending order by their 'before' results. In many cases, the number of findings for individual checkers were dramatically higher with **Tick-the-Code** than before. It indicates a clear skill gap. Most developers have had no idea what to look for in a code review, or have been overlooking smaller items whilst

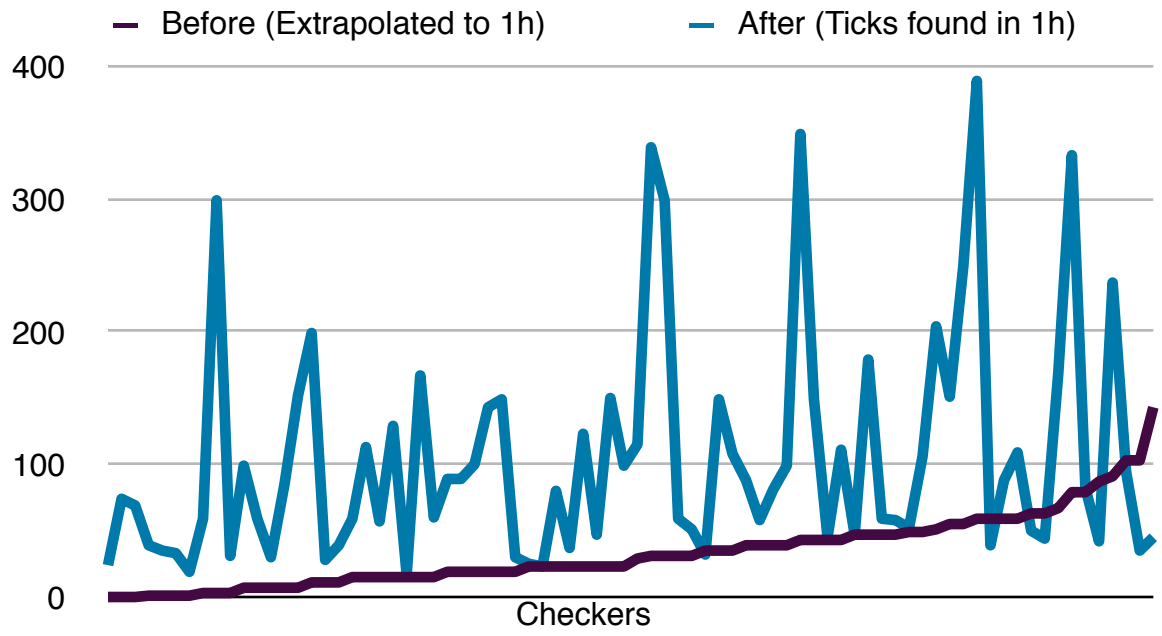


***Figure 1**. The number of findings before (the monotonically increasing line) and after (the spikes) learning **Tick-the-Code**. To be more descriptive, the checkers on the X-axis are sorted in ascending order of their extrapolated 'before' results. In most cases the extrapolated result is much lower than the one **Tick-the-Code** yields. Each pair of values ('before' and 'after') on the X-axis shows the performance of an individual checker.*

searching for those they believe to be more critical. Given better guidelines and stricter rules, they can find places to improve easily and quickly.

Tick-the-Code helps developers cover larger amounts of source code in a relatively short period of time. Table 2. summarizes the overall results of the ticking part of the experiment. In less than 80 person-hours, the novice checkers managed to tick over 80000 physical lines of code creating over 8200 ticks. Even if only 10% of the ticks were valid improvement suggestions, the developers had still generated over 800 ways of making their code better. This proves that developers can feasibly improve their code from its current state.

Participants	Total time	Total lines (physical)	Total ticks	Average rate of finding
78	78h	80218	8230	106 ticks/h

***Table 2. Experiment: Before and After**. The results of almost eighty software developers ticking real, production-level source code for an hour. Covering over 80000 lines of code systematically and target-oriented is possible and feasible with negligible time. The tick yield, i.e. the number of source code improvement suggestions is massive, considering that most source code had already gone through the existing quality control processes. The experiment was conducted between May 31 and Oct 23, 2006.*

2.3. Experiment: Five-minute Ticking, or Surely You Can't Tick Any Faster?

In this experiment there were 144 software developers from different organizations. Once again, all participants had about one thousand lines of production-level code. This time the participants were given much clearer instruction on the ticking method than in the previous two experiments. Each time a rule was presented, the participants had exactly five minutes (!) to search for its violations. Then the number of ticks was recorded. Sometimes the participants didn't get through all of their code, sometimes they managed to have spare time before the bell rang.

The participants checked eight, nine or ten rules in a training session. With the five-minute limit that meant that everybody spent effectively 40-50 minutes ticking. On average, each and every participant found 103 ticks in under an hour! In Table 3, you see that the average (extrapolated) rate of finding ticks is approximately 136 ticks/h.

Averaging all collected ticks over the effective time used for checking (108h, over all participants) gives us some interesting insights. In this calculation we treat the workforce as a mass of work, nothing more. The ticks the participants found have been given to this mass of work and the average has been calculated. See Table 4. for the results. The table shows that spending an hour on proactive quality could result in 46 suggestions of improving modularity by refactoring blocks of code inside bigger routines into their own subroutines (CALL). Alternatively, a software developer could suggest 53 unnecessary comments to be removed from clouding the code (DRY). We can also see that in one hour, given enough source code, anybody could find over five hundred hard-coded numbers, character constants and strings (MAGIC). Some of them are missing information linking them to their origin and relating them to other values in the code. That missing information is certain at least to slow down maintenance but it is also true that some of the plain numbers will just plain cause bugs.

Participants	Total time	Total lines (physical)	Total ticks	Average rate of finding
144	4d 12h 35min	> 100 000	14906	136 ticks/h

Table 3. Experiment: Five-minute Ticking*. The training courses took place between 26-Oct-06 and 12-Jun-07. Unfortunately the experimenter sometimes forgot to collect the numbers of lines from the participants, but everybody was always instructed to bring about one thousand lines of code.*

Isn't it illuminating to see that any software developer could complain about 92 bad variable, function and method names and he'd just need an hour to point them out precisely (NAME)? It would take longer to come up with better names but not prohibitively so. How much does an error cost? How much effort does it take to test, write an error report, find a responsible person, debug, fix, retest and whatever else is part of repairing an error? One hour, ten hours, a hundred hours, a thousand hours? How many improvement suggestions could be found and taken into consideration before the chaotic code results in errors? Table 4. is saying: "Many". That is the choice for each software developer to make. There is the option of continuing to fix errors hoping for the best and

then there is the option of proactively preventing errors using methods like **Tick-the-Code** to clarify code now.

Rule	CALL	CHECK-IN	DEAD	DEEP	DEFAULT	DRY	ELSE
Ticks/h	46	82	45	76	11	53	322
Rule	HIDE	MAGIC	NAME	NEVERNULL	TAG	UNIQUE	
Ticks/h	186	516	93	90	18	20	

***Table 4**. The average number of ticks found for each checking hour per rule. Software developers could find this many violations in one hour in the code they produce, if they chose to. 144 developers checked for 108h to create the data.*

2.4. Experiment: Open Source Wide Open

Ten source code files from four different Open Source projects were randomly selected to be ticked. The author ticked all of the modules with 24 **Tick-the-Code** rules. Once again, each file was approximately one thousand physical lines long. Physical lines are all the lines in a source code file. Physical lines are the simplest unit of measurement. Using them avoids all problems of how to interpret empty lines, or lines containing just a comment, or lines with multiple statements. As physical lines, they all count as one line each, respectively. In other words, the line numbering of your favorite editor matches the definition of physical lines exactly.

For some of the files, several versions from different phases of development were selected. File address-book-ldap.c (later a.c) is a C-language module and there were four versions of it: the first version from the year 2000, another version several evolutionary steps later from year 2002, and two more dated in 2004 and 2007, respectively. The format a.c(04) is used to refer to the third snapshot of file address-book-ldap.c. The file lap_init.c (l.c) only had one version to tick. There are two versions from ipc_mqueue.c (i.c) and OSMetaClass.cpp (O.cpp). The letters A and B are used to denote the older and newer version, respectively. For example, O.cpp(B) refers to the newer version of OSMetaClass.cpp. PeerConnection.java (later P.java) was the only Java file included in the experiment. See Figure 2. for the sizes of the sampled source code files. The file a.c, for example, has grown from a mere 400 lines to 1000 lines over the seven years of its development.

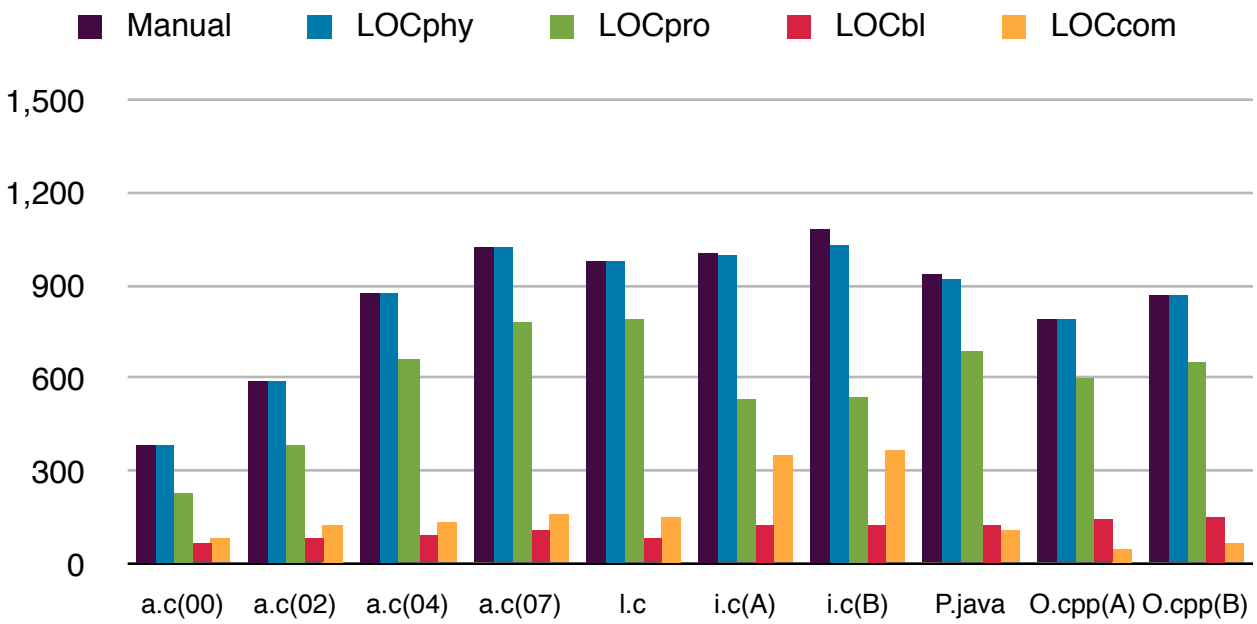


***Figure 2**. The sizes of the sampled files. 'Manual' is the figure the author counted by hand. LOCphy are physical lines (should be equal to 'Manual', slight errors are visible), LOCpro program code lines, LOCbl are blank lines and LOCcom commentary lines. All the LOC counts were generated by CMT++ and CMTjava.*

In addition to ticking all the modules with all the rules, Testwell Oy, a company in Finland (www.testwell.fi) helped with some additional measurements. Their tools CMT++ and CMTjava were used to measure C/C++ files and the Java module, respectively. The ticking took almost 29 hours to complete, while the tools took only seconds to complete their tasks. The ticking revealed over 3700 ticks, i.e. possible improvements. The total results are summarized in Table 5.

Checkers	Total time	Total lines (physical)	Total ticks	Average rate of finding
1	28h 52min	8517	3723	129 ticks/h

***Table 5. Experiment: Open Source Wide Open**. The experiment took a little more than a month to conduct between 5-May and 11-Jun-07.*

One of the best known complexity measurements for source code is the cyclomatic number [1]. It points out "unstructured" code and uses graph theory to compute the cyclomatic number. CMT++ and CMTjava calculate the cyclomatic number for their input files.

An interesting observation: the correlation between the cyclomatic numbers for the ten files and the normalized total tick count series is as large as 0.7. Although the sample is small, the files were selected in random, so the result isn't just a coincidence. It seems that the ticks and cyclomatic numbers are related and can both be used as indicators for

complex source code. See Figure 3. for a graphical view of the cyclomatic number and tick counts. Additional experiments are needed to confirm the link. One reasonable method would be to remove the ticks already found by refactoring the modules and then measuring them again. If there is a relationship between the two, both the tick count and the cyclomatic number should go down.

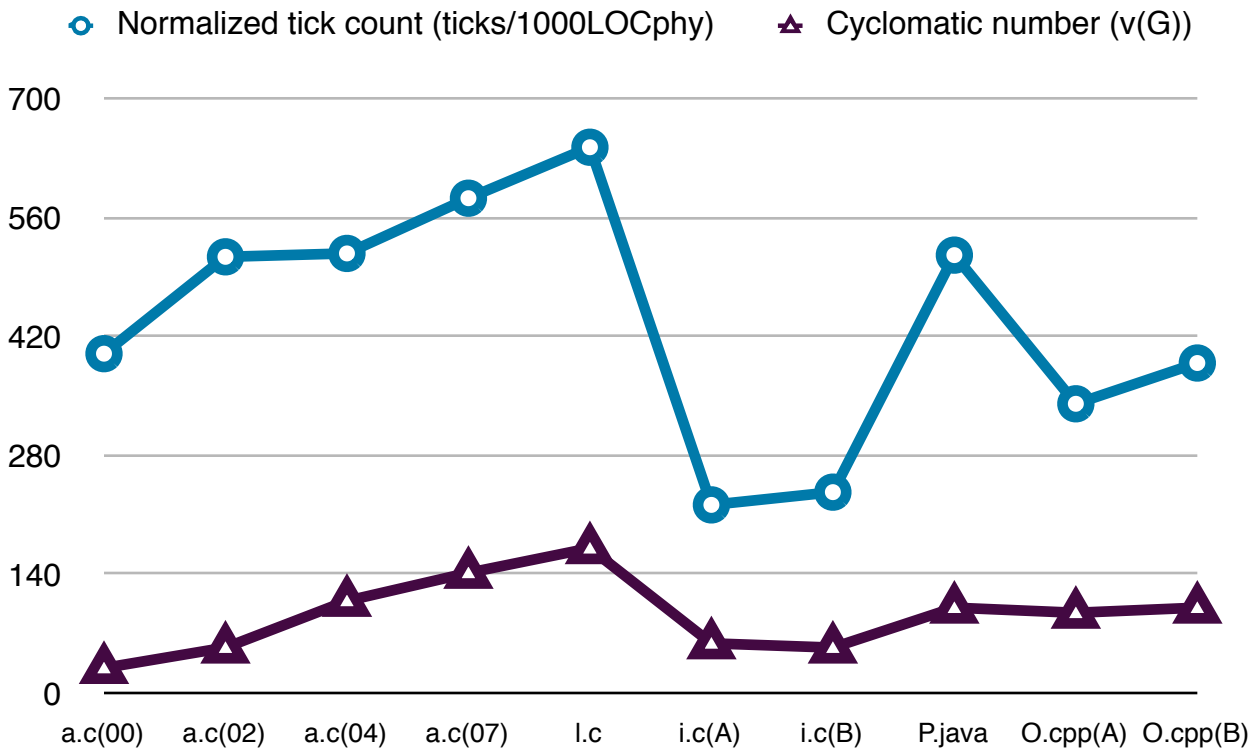


***Figure 3**. Normalized tick counts and the cyclomatic numbers. The correlation between the series is large, 0.7. Please, note that the Y-axis values of the two series have different units.*

The **Tick-the-Code** rules are based on certain principles of coding. Breaking the principles for whatever reason will usually cause trouble sooner or later. Examining the rules more carefully, we can divide them into four categories:

1. Extra baggage
2. Missing info
3. Chaos-inducers
4. Risky assumptions

1. Some of the rules point out unnecessary items in a module file. For example, a comment that repeats what the code already says is just a waste of disk space. A prime example is the line

```
count++; // increment counter
```

Not only are they a waste of disk space, any extra comment increases chances of outdated comments. Such comments are distracting, and impair maintenance. One rule points out such redundant comments (DRY). Another rule points out duplicated blocks of code (UNIQUE), which are also usually unnecessary and harmful. A few other rules make up the “Extra baggage” category.

2. A few rules aim at noticing when something is missing from the code. One rule asks for an `else` branch to be found at each `if` statement (ELSE). Whenever a literal number is used in code (hard-coded), another rule is violated (MAGIC). Literal numbers don’t reveal where they come from or if they are related to any other numbers in the code. The missing information causes bugs very easily, or at least slows down the maintenance work unnecessarily. The category “Missing info” consists of similar rules.

3. There are elements in code that add to the confusion. Usually software is complex to begin with, but certain constructs make it even harder to understand. Using the keyword `return` too readily can lead to functions having too many exit points, which can hinder understanding of the flow of code and impair its testability (RETURN). Blocks of code that could be their own subroutine but are inside a bigger routine instead are unnecessarily complicating the source code (CALL). Bad names give code a bad name (NAME). This category is called “Chaos-inducers”.

4. Blindly assuming that everything is fine is dangerous in software. Anything could go wrong, especially with pointers. A good programmer defends his code by generously sprinkling checks for NULL pointers in it. Some rules spot possible references through NULL pointers (NEVERNULL), variables used without any checking (CHECK-IN) and freed, but not invalidated pointers (NULL). These rules reveal locations where a more defensive approach is possible. The category is called “Risky assumptions”. Table 6. shows the categories and their rules.

Extra baggage	Missing info	Chaos-inducers	Risky assumptions
DEAD	DEFAULT	CALL	CHECK-IN
DRY	ELSE	NAME	NEVERNULL
INTENT	MAGIC	RETURN	NULL
ONE	PTHESES	SIMPLE	CONST 1ST
UNIQUE	TAG	FAR	ZERO
	ACCESS	DEEP	PLOCAL
	HIDE	FOCUS	ARRAY
			VERIFY

***Table 6**. The four categories of rules. Twenty-four of the rules form the ‘active’ rule-set in **Tick-the-Code**.*

The normalized tick counts for all categories are shown in Figure 4. Apart from the file i.c, all files leave out valuable information, which needs to be dug up or guessed in maintenance. Every tenth line of file a.c contains some confusing constructs. On the other hand, there doesn't seem to be too much extra baggage in any of the example files. This is hardly surprising considering the tendency in human nature to be too lazy rather than overly explicit or verbose. All four categories correlate strongly with the cyclomatic number. The correlation for "Extra baggage" is 0.67, for "Missing info" as high as 0.96, for "Chaos-inducers" 0.73 and for "Risky assumptions" 0.68.

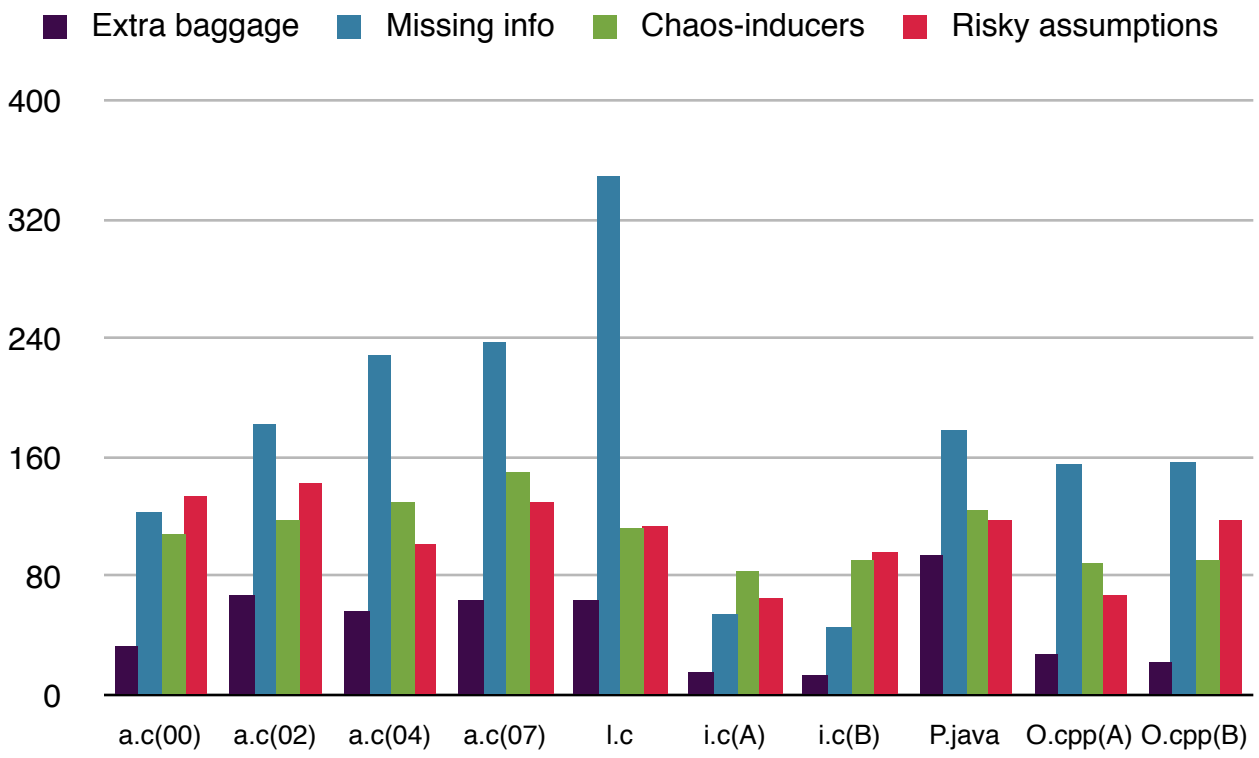


***Figure 4**. The normalized tick counts for the rule categories. Files a.c and l.c will be most likely to suffer problems in their maintenance because of missing information. On average, more than every third line (350/1000) in l.c is missing some information. In this group, the file i.c seems much better to maintain.*

2.5. Conclusions Drawn from the Experiments

A. Software developers could perform much better code reviews than they currently do.
B. Currently produced source code leaves a lot to be desired in terms of understandability and maintainability.
C. Tick count measures (at least some aspects of) software complexity, just like cyclomatic number does.
D. Software developers could prevent many errors if they chose to do so.
E. The effort to do so would be negligible.
F. Systematic ticking of the code leads to code that is easier to maintain.
G. Easier-to-maintain source code benefits all stake-holders in the software industry.

3. A Solution: Tick-the-Code

[2] describes the details of **Tick-the-Code**. Suffice to say that it is a rule-based approach for reviewing code on paper. Checkers mark each and every rule violation on the printed paper and deliver the ticks to the author of the code. The author considers each and every tick, but maintains the right to silently ignore any one of them. Whenever possible the author replaces the ticked code with a simpler, more maintainable version. Sometimes the change is simple, sometimes it is more than a trivial refactoring.

The rules are extremely clear and explicit. They designate an axis and show which end of the axis is the absolute evil. For example, there is a rule called ELSE, which says that "each `if` must have an `else`". On the one end of the axis there is an `if` with an `else` branch and on the other extreme there is the `if` without an `else` branch. The rule says that the former case is right and that the latter is absolutely wrong. The reasoning behind the rule is that if the code could possibly be missing something important (the code for the exceptional case in the `else` branch), it could lead to problems. And vice versa, an `if` with a carefully considered `else` branch is less likely to be problematic. Missing information, or in this case missing code, is a hole in the logic of the code.

Every ticked `if` provides the author with a chance to double-check his thinking and fix it before a functional bug occurs. The same principle applies to all the rules, most of which are not quite as simple to find as rule ELSE. Each rule is based on a solid programming principle and divides a dimension into two extremes, one right, one wrong. Making the world black-and-white is an excellent way to provide clarity, even though it only catches certain errors. The good thing is that it normally catches all of them. Sometimes the `else` branch contains the wrong kind of code, sometimes it is the `if`. Combined with a sensible author, this kind of merciless ticking starts to make sense in practice.

The search is very often the most time consuming part of an error hunt. Not the fixing, nor the testing, nor the verification of the fix, but the search. In **Tick-the-Code** the search has been tweaked to be as fast as possible. Even though it is manual, it is fast in comparison to most other methods. It might seem that **Tick-the-Code** is aimed at making the source code better. That is the truth, but not the whole truth. The main purpose is to make the checkers better. In other words, in ticking the checkers get to see a lot of source code, some of it written by themselves, most of it written by somebody else, and they look at code from a different stance than during the source code construction phase. Exposure to others' source code gives them examples of how problems can be solved and have been solved.

With the help of the rules and principles the checkers learn to prevent failures. By seeing the code their colleagues are writing they also learn at least what the colleagues are working on. Ticking code helps spread information about the developers and their tasks among the checkers (i.e. developers). It helps make the current product better and at the same time teaches the checkers new ideas and thus helps make future products better.

4. The Effect: Complexity in the Future - a Thought Experiment

Regular use of **Tick-the-Code** makes the produced source code clearer and easier to maintain. Clearer source code contains less careless programming errors. There are still errors, though. Even a perfect implementation phase cannot produce perfect source code if the requirements are poor. The developers will notice that meeting bad requirements doesn't make the customer happy. They can apply the lessons learned from the rule-driven and systematic ticking of code. Their analysis can help in gathering requirements. They notice missing requirements, vaguely described requirements and several unnecessary, or minor requirements or redundant ones. Some of the requirements just add to the overall confusion. The developers can considerably improve the quality of the requirements when they add the missing ones, reformulate the fuzzy ones, uncover the hidden assumptions and remove the unnecessary or dangerous ones and separate the minor ones from the important ones. When both the implementation and requirement gathering phases are under control, reworking becomes almost unnecessary. Requirements meet customer expectations and implementation fulfills the requirements.

Similarly, as the organization saves time with less rework, they can turn their attention to other problematic areas. With problems in the coding process, the organization isn't able to build a real software culture. Their whole development process is immature; there are design and architecture problems, and both error handling and testing have been overloaded. This all can change now systematically. With the basics under control, a software development culture can emerge and higher-level concerns like usability can come to the fore.

The most natural place to start making a software organization more mature is coding. As the experiments show, many software organizations try to build on the quicksand of unnecessarily complex code.

5. References

[1] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, No.4, December 1976

[2] M. Hiltunen, "Tick-the-Code Inspection: Theory and Practice," *Software Quality Professional*, Vol. 9, Issue 3, June 2007

6. Appendix

A1. Rules

The table provides a list of all the rules mentioned in this paper. For various reasons, exactly 24 rules form the active rule set of **Tick-the-Code**.

Name	Rule
DEAD	Avoid unreachable code.
DRY	A comment must not repeat code.
INTENT	A comment must either describe the intent of the code or summarize it.
ONE	Each line shall contain at most one statement.
UNIQUE	Code fragments must be unique.

Table A1. Extra baggage rules.

Name	Rule
DEFAULT	A ‘switch’ must always have a ‘default’ clause.
ELSE	An ‘if’ always has an ‘else’.
MAGIC	Do not hardcode values.
PTHESES	Parenthesize amply.
TAG	Forbidden: marker comments.
ACCESS	Variables must have access routines.
HIDE	Direct access to global and member variables is forbidden.

Table A2. Missing info rules.

Name	Rule
CALL	Call subroutines where feasible.
NAME	Bad names make code bad.
RETURN	Each routine shall contain exactly one ‘return’.
SIMPLE	Code must be simple.
FAR	Keep related actions together.
DEEP	Avoid deep nesting.
FOCUS	A routine shall do one and only one thing.

Table A3. Chaos-inducers.

Name	Rule
CHECK-IN	Each routine shall check its input data.
NEVERNULL	Never access a ‘NULL’ pointer or reference.
NULL	Set freed or invalid pointers to ‘NULL’.
CONST 1ST	Put constants on the left side in comparisons.
ZERO	Never divide by zero.
PLOCAL	Never return a reference or pointer to local data.
ARRAY	Array accesses shall be within the array.
VERIFY	Setter must check the value for validity.

Table A4. Risky assumptions rule category.

A2. Miscellaneous questions and answers

1. How is **Tick-the-Code** different from implementing coding standards?

The answer depends on what “implementing coding standards” means. To me, **Tick-the-Code** “implements coding standards”. What I find is that often people in software organizations think they have implemented coding standards when they have written a document titled “Coding standards”. The other half is normally missing. Coding standards need reenforcing. Without constant and active checking, the coding standards are not followed. At least you can’t know for sure.

Tick-the-Code provides a limited set of clear rules and an extremely effective way of reenforcing them. The rules don’t deal with stylistic issues and they don’t set naive numerical limitations on the creativity of the developers. On the other hand, the veto rights of the author (to ignore any tick) keep the activity sensible. It makes it possible to break the rules when necessary without losing effectiveness. Too strict checking can easily turn in on itself, and becomes counterproductive.

2. How long does it take to learn **Tick-the-Code**? Does it depend on the number of rules?

The technique of checking one rule at a time is fairly quickly explained and tried out. In three hours, I’ve managed to teach the method and hopefully instill a sense of motivation and positive attitude towards striving for quality to help make **Tick-the-Code** a habit for the participants. It does depend on the number of rules, yes. Some of the rules are simple searches for keywords, while others demand a deep understanding of software architecture. You learn the rules best when you try them out and when you receive feedback on your own code so that you see when you’ve violated certain rules.

3. How many rules did you use in your tests? What is a reasonable number? Can you have too many rules?

Good questions. In most of the experiments, the participants checked for up to an hour. Normally they checked with up to nine different rules in that time. I have a rule of thumb that says that you can check one thousand lines of code with one card in one hour. One card contains six rules and an hour of checking is a reasonable session with complete concentration. In the experiments the time for each rule is limited so that the total checking time won’t exceed one hour.

You can definitely have too many rules, yes. Some coding standards contain so many rules and guidelines that if they weren’t written down, nobody would remember all of them. The worst example so far has been a coding standard with exactly 200 rules and guidelines. How are you supposed to reenforce all those rules?

4. Does awareness of the rules improve the code a developer produces?

Yes, that is the main idea of **Tick-the-Code**.

5. Can different people look at the same code using different rules?

The fastest way to cover all 24 rules is to divide them among four checkers and have them check the same code in parallel. Yes, that is possible and recommended.

6. Can it be automated?

Some of the rules are available in static analysis tools, so yes, they can be automated. The fact that those rules are most often violated, leads me to believe that such automation is not working. Whatever the reason, code always contains violations of simple rules. Simple means simple to find, not necessarily error-free. Breaking a simple rule can have fatal consequences in the code.

My opinion is that automating the checking of some of the rules is taking an opportunity away from the checkers. They need to expose themselves to code as much as possible in order to learn effectively. What better way to start exposing than looking for rule violations, which you are sure to find? Getting a few successful finds improves the motivation to continue and makes it possible to check for more difficult rules.

7. Where do you plan to go from here?

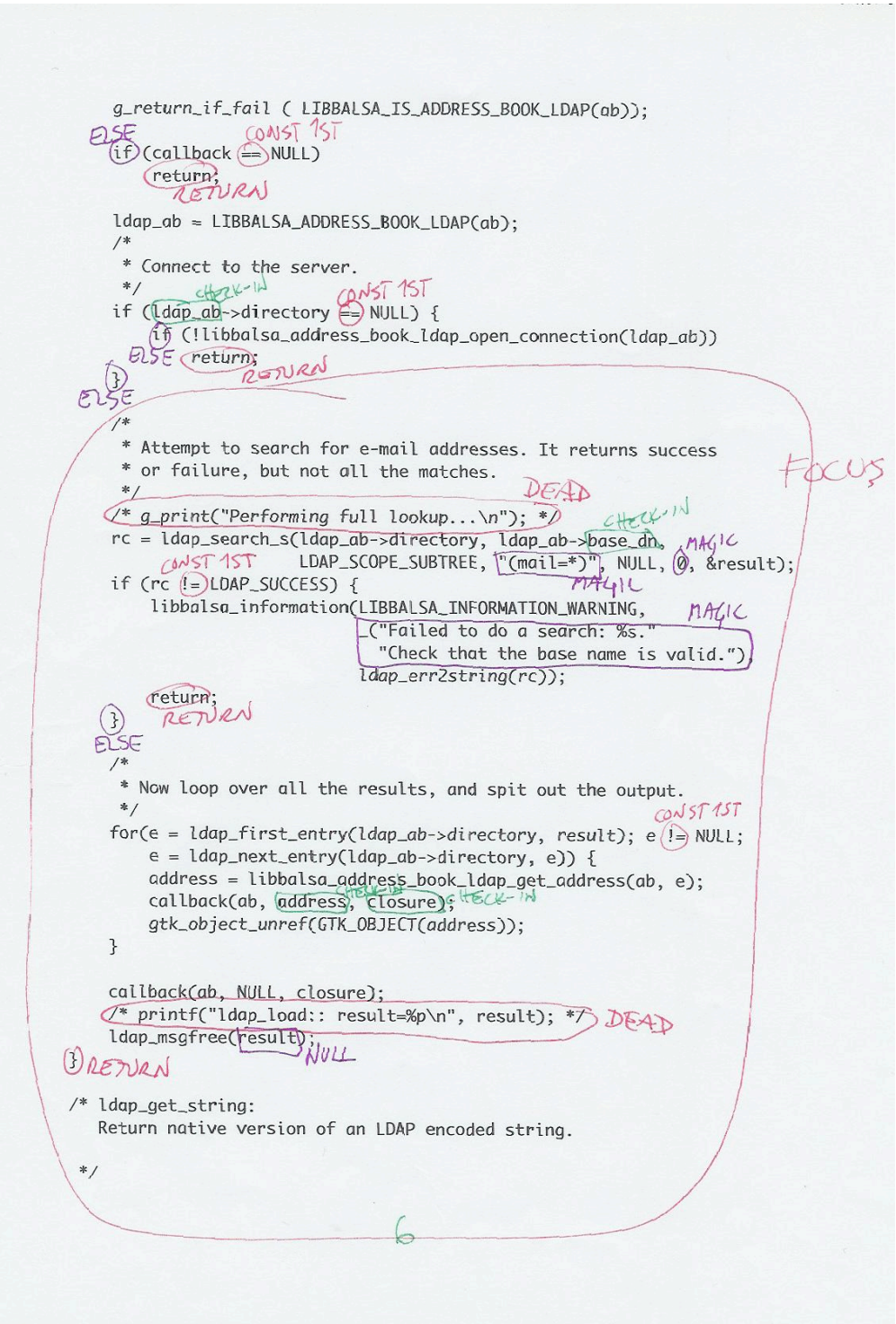
There will be a few articles more in this area, maybe a paper or two, too. I have a book in the works about **Tick-the-Code**. In my plans **Tick-the-Code** is just one part of a quality-aware software development philosophy. I call it 'qualiteering'. That's why my website is called www.qualiteers.com.

I will continue training people in willing companies and spread the word. It is my hope that **Tick-the-Code** will reach a critical mass in the development community so that a vibrant discussion can start in **Tick-the-Code** Forum (www.Tick-the-Code.com/forum/).

One possibility is to follow a few open source projects up close and a little bit longer to see the effects of **Tick-the-Code** in the long-term. If you're interested in volunteering your project or company, let me know (miska.hiltunen@qualiteers.com).

A3. An example file

In Figure A1, there's a page of code from one of the open source modules of chapter 2.4 Experiment: Open Source Wide Open.



```
g_return_if_fail ( LIBBALSA_IS_ADDRESS_BOOK_LDAP(ab));
if (callback == NULL)
    return;

ldap_ab = LIBBALSA_ADDRESS_BOOK_LDAP(ab);
/*
 * Connect to the server.
 */
if (ldap_ab->directory == NULL) {
    if (!libbalsa_address_book_ldap_open_connection(ldap_ab))
        return;
}
/*
 * Attempt to search for e-mail addresses. It returns success
 * or failure, but not all the matches.
 */
/* g_print("Performing full lookup...\n"); */
rc = ldap_search_s(ldap_ab->directory, ldap_ab->base_dn,
     LDAP_SCOPE_SUBTREE, "(mail=*)", NULL, 0, &result);
if (rc != LDAP_SUCCESS) {
    libbalsa_information(LIBBALSA_INFORMATION_WARNING,
                         _("Failed to do a search: %s."
                           "Check that the base name is valid."),
                         ldap_err2string(rc));
    return;
}
/*
 * Now loop over all the results, and spit out the output.
 */
for(e = ldap_first_entry(ldap_ab->directory, result); e != NULL;
    e = ldap_next_entry(ldap_ab->directory, e)) {
    address = libbalsa_address_book_ldap_get_address(ab, e);
    callback(ab, address, closure);
    gtk_object_unref(GTK_OBJECT(address));
}

callback(ab, NULL, closure);
/* printf("ldap_load:: result=%p\n", result); */
ldap_msgfree(result);
}

/* ldap_get_string:
   Return native version of an LDAP encoded string.

*/
```

Figure A1. Ticks on this page of code are the author's handwriting.