Article By: Jason Cohen,
www.SmartBear.com

Lightweight Code Review Episode 6: Checklists - You build me up just to knock me down

This article describes how to build a checklist people will actually use.

The code review checklist is the bane of developers. Thirty-odd check-boxes await, each requiring thoughtful consideration before the liberating tick mark can be applied. Twenty source files, freshly altered, are awaiting verification. The math is simple: 20 x 30 = 600 decisions, no matter how you tackle the problem.

This is going to suck.

I've seen dozens of checklists, both in the field and on web sites, and almost none are practical. You're not going to make 600 thoughtful decisions during a code review. And before you reach for that sample checklist from Mozilla.org, you might realize that what works for a C++ open-source GUI project might not be appropriate for your 3-tier Java web application.

In this article you will learn how to build a truly useful checklist and how to tune it to your development group and keep it fresh as time passes.

Quantity: The silent killer

28. That's the average number of checklist items I found in the top seven sample checklists from the Internet (according to Google, the Knower Of All). My experience from the field is similar — it's not unusual to find two pages of items, each just as important as the others, each requiring due consideration before its own white square can be checked.

This is my biggest peeve with checklists. Marathon checklists are built with the idea of completeness. The long list is usually broken up into categories - Correctness, Maintainability, Style, Error Handling, Performance, Mind-Numbing, and so on. Items range in scope from generic concepts like "Method does what it is documented to do" to specific checks like "All exception classes have a constructor that takes a string."

The fact is that normal people do not have the patience to go through lists like this. If I stack a 500-line code change on your desk and insist that a 30-item checklist be applied to every file, how would you even go about that? You could pick one checklist item at a time, scanning through all the code. That's efficient for simple things such as style checks, but for conceptual items like "Method does what is documented to do" you'll need to study the code to get the answer. So applying the list to every file sounds more reasonable, but on the twelfth file are you really going to be diligent on every item? Really?

What actually happens is that people end up using the list as a general guide. They do not consider every item in every file. Which items will they remember as they look at the code? Will it be the most important ones? Who knows?

Magic Numbers

What if we had only one checklist item? Clearly the reviewer would be able to keep that item in mind as the code is scanned. How many more items could the reviewer keep in mind as the code is read, without ever having to refer back to the checklist document? If we cap the

checklist at this limit, we should expect reviewers to actually apply the entire checklist, every time.

In 1956 George Miller published a now-famous paper called "The Magic Number Seven, Plus or Minus Two." He measured human short-term memory capacity for "chunks of information." For example, most people can remember an American 7-digit local number long enough to dial it, whereas when dialing a 10-digit long-distance number most people have to refer back to the number at least twice.

Miller's results have been applied to various aspects of psychology and even computer science. For example, Miller's work is frequently cited as a reason for limiting a method's cyclomatic complexity to 9.

If we expect reviewers to think about the entire checklist when reviewing a file, we have to keep the checklist short. Under 10 items seems reasonable.

So that's the first rule: **No more than 10 items**.

Making them count

But my two pages of items are important! They keep code consistent, maintainable, and catch lots of common errors. Can we ensure those things are happening if we throw out most of the list? Yes, if we're stricter about which items are included on the list and use automated tools where helpful.

An item that appears on almost every checklist is: "Does the code accomplish what it is meant to do?" Is this question really necessary? It's a code review - what else would you expect the reviewer to do? Another common one is: "Can you understand the code as written" or, similarly, "Is the code documented properly?" Again, it's not possible to review the code at all without having to understand it.

Don't waste precious checklist items on directives that the reviewer will do anyway. The second rule is: **No obvious items.**

In my experience, the majority of items on long checklists pertain to simple programming rules or coding style. Style items include naming conventions, whitespace and block positions, and where comments are located. Simple programming rules identify basic programming errors and are often specific to a language. A Java rule might be "Always override hashCode() if you override equals()," whereas a C# rule could be "Classes marked with the Serializable attribute must have a default constructor."

These items can be found by static analysis tools. At least with mainstream languages, there are dozens of tools, both free and commercial, that check for style and basic programming errors. You cannot use the excuse that some cost money to buy and all cost time to set up - it cannot possibly be more expensive than having a human spend time digging up these mundane errors

during a code review - developers need to be doing things that only a human can do, like checking code correctness and maintainability.

The third rule is: **No items that can be automated**.

So which items should stay on the checklist?

Before we had a proper build system, we used to manually update the build number in a header file. Don't laugh - we've all been there! Of course we'd always forget to kick the number and, even with code review, the reviewers forgot to check for that too. After all, if that header file wasn't included in the review, there was nothing to jog the reviewer's memory. It's one thing to verify something that's put in front of you; it's a lot harder to review that which is missing.

If it's easy for the author to forget, the reviewer will forget too. Only a checklist can help this situation. So the fourth rule is: **Items should be things that you commonly forget**.

And now, an example of a Perfect Checklist Item: "Recovers properly from errors everywhere in the method." Incorrect error-handling often turns error dialogs into fatal crashes or memory leaks. In my experience, few reviewers remember to look at error paths, and authors forget too, especially when a change is made in a large method. You can't generally automate the check for correct unwinding, especially when the method is changing shared state like global variables, registering for events, or accessing external resources.

Building the List: The Week of Pain

The question I get asked most about checklists is: Do you have a sample checklist we can use? As a politician would say, this is the wrong question. The right question is: How do we build a checklist appropriate for our software development, and how do we adjust the list over time?

Let's make this personal. I've seen the following technique applied successfully on several occasions. It's a take-off of the "personal checklist" concept from the Personal Software Process from the Software Engineering Institute.

Try this sometime: For one week, every time you make a mistake, no matter how small, write it down. Everything. Misspell a word while writing an email? Write it down. Close an application when you meant to close just one window? Write it down. Have a lingering compiler error when you run the unit tests? Write it down. Do this for a whole week.

You'll discover a few things during this experience. First of all, it's infuriating. You make a lot of mistakes and they're tiresome to chronicle. Second, you can't bear to do this for a whole week. It's maybe a little too much introspection. The ego can take only so much self-examination.

Third, and most importantly, you'll find a pattern. You make the same kind of mistakes over and over - it's not random. In my case, for example, I always misspell the word "definitely," and I apparently use it all the time. And I close all of QuickBooks when I mean to close just one window - the non-standard design of the close-box was confusing me subconsciously.

Of course the next step is to correct the common problems. Many of them you'll correct just by virtue of being conscious of them. In the case of QuickBooks, I freeze just as I am about to close everything - just long enough to remember the right way and avoid the 90-second restart cycle. Others errors you can correct with automated methods. With my misspellings I just use the auto-correct feature in Outlook and now "defenatly" gets transformed automatically.

Software checklists should be built in exactly the same way. Take a look at the last one or two hundred defects found by code review and you'll find a pattern. Or look at the code fixes for defects found in QA and find the pattern in the root causes. This isn't as hard as it sounds - you don't need detailed analysis or 5 columns of data on each defect - just summarize the code fixes in one line or less and the patterns will jump out at you.

A checklist derived from these patterns will be tuned specifically to the errors you make most often. So the fifth rule is: Let empirical evidence determine the checklist.

This technique provides the initial checklist, but it turns out the patterns necessarily change over time. With a short checklist, everyone gets used to keeping an eye out for those certain things. Just as with the personal introspection project, everyone will get used to finding and fixing the same problems. Soon code authors will think about these same things while the code is being written - the problem will be fixed before it starts.

At this point, the number of defects associated with that checklist item will diminish, eventually becoming so infrequent as to make that item useless. This is a good thing - the developers have developed habits and techniques to avoid this type of error. Now you can replace that item with the next common pattern.

This implies you're actually tracking how many defects are attributed to each item. You're doing that, right? Don't forget to have a "none of the above" category as well - when you find an item is stale, the list of defects in that category will reveal the next pattern that needs fixing.

The sixth rule is: **Track defects by item so you can replace stale items**.

Checklist for Checklists

So to summarize, here's a checklist you can use when building your own checklist:

- No more than 10 items in the list.
- No obvious stuff.
- Nothing that can be automated.
- Things that are easy to forget.
- Use empirical evidence to build the list.
- Replace stale items.

As if a checklist for checklists wasn't recursive enough already, I'd like to point out that this list is short, contains items apparently not obvious to most checklist designers, and it was built empirically from patterns I've found in the field.

Now go roll your own!

> *Jason Cohen is the author of Best Kept Secrets of Peer Code Review and the founder of SmartBear Software. SmartBear sells CodeCollaborator, the only lightweight peer code review tool and is well known for conducting the largest case study of peer review ever published.*

SmartBear Software

+1 978.236.7900

www.smartbear.com