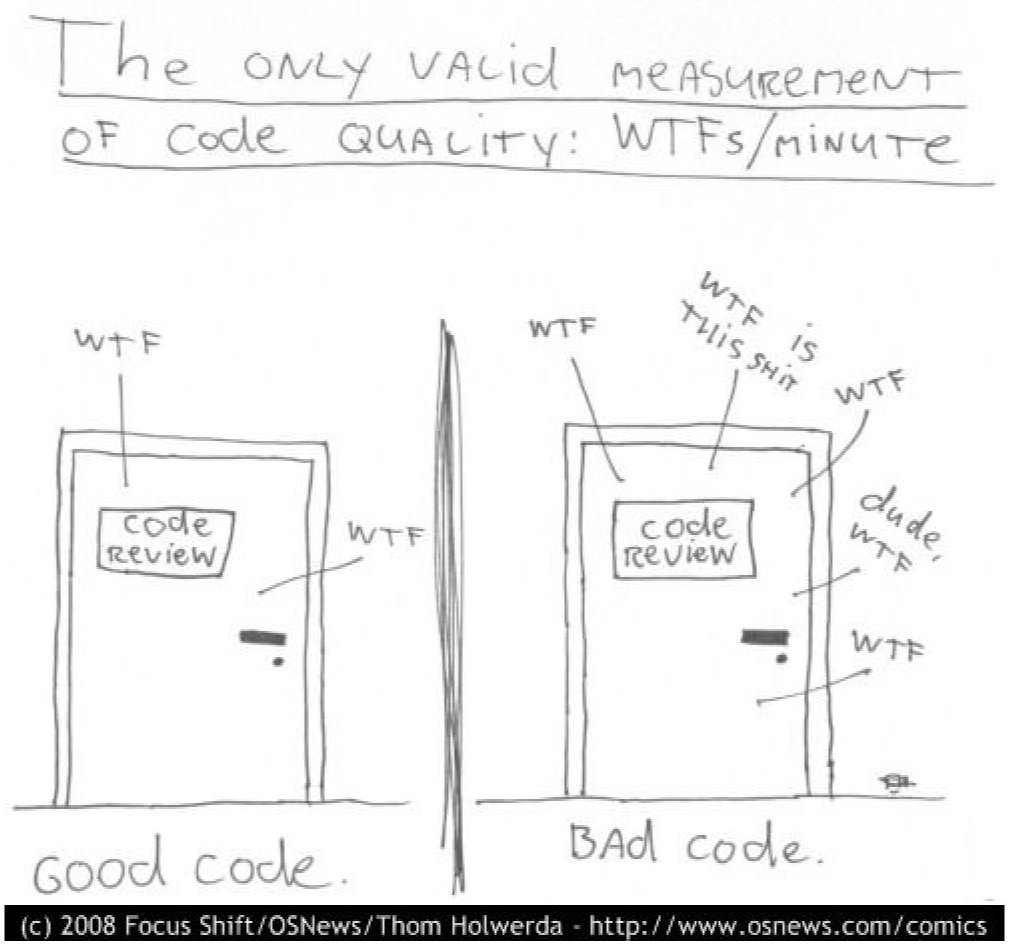
Reviews, inspections, and walkthroughs


The only valid measurement of code quality: WTFs/minute
WTF
code review
WTF
WTF
WTF is this shit
WTF
dude, WTF
WTF
Good code.
BAd code.
(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Human inspection

In earlier weeks, we identified three broad ways in which quality concerns in a software engineering project can be identified:

- Formal methods.
- Execution-based testing.
- Human inspection.

Of these, CS & SE academics have historically been extremely excited about formal methods, moderately excited about execution-based testing (particularly automated execution-based testing; execution-based testing is referred to as just "testing" herein)), and the least excited about human inspection.

In practice, while all three techniques have their place, the single most useful family of techniques for improving software quality has proven to be human inspection, with testing second and formal methods a distant third[1].

Inspection is the MOST important?

Steve McConnell's excellent book *Code Complete* has a fairly remarkable table in it (reproduced below) that lists some estimates of the proportion of "defects" that exist at a given point in a project, that are typically found by applying different quality quality assurance techniques:

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal Design Reviews	25%	35%	40%
Formal Design Inspections	45%	55%	65%
Informal Code Reviews	20%	25%	35%
Formal Code Inspections	35%	65%	80%
Personal Desk-Checking Code	20%	40%	60%
Unit Test	15%	30%	50%
New Function (component) Test	20%	30%	35%
Integration Test	25%	35%	40%
Regression Test	15%	25%	30%
System Test	25%	40%	55%
Low volume beta test (<10 sites)	25%	35%	40%

[1] I am being a little unfair to formal methods here if you define them broadly enough. Modern programming languages, and the compilers that check code written in them, make an order-of-magnitude difference to programmer productivity compared to assembly language!

High Volume Beta Test (>1000 sites)	60%	75%	85%

Table 1: Defect detection rate versus project phase, as published in Code Complete.

While the particular figures are rubbery (for instance, the analysis assumes that all defects are of equal importance) it does seem that a) no one technique is going to eliminate all faults, and b) inspections are pretty good at finding faults when compared to execution-based testing (with the exception of "high volume beta tests", which back in the day were very expensive. This may not be quite as true today in some circumstances).

Furthermore, as Myers found way back in 1978[2], it seems that inspections find *different* faults to execution-based testing. So combining multiple quality assurance techniques is more effective than using any one technique alone.

Just on that basis, it's reasonable to conclude that inspection is at least as useful as testing as a quality assurance technique. However, that's not the whole picture.

As we explained in chapter 1, quality is not just about the immediate functionality of the software. In many applications, maintainability and portability are key quality characteristics, and they can only be assessed to a limited extent through testing. Static analysis can provide some metrics that correlate with these properties, but the primary way which these aspects can be assessed is through human review.

But most important of all, executable code is only one of a number of types of artifacts produced in the software engineering process, and as noted earlier, defects can occur in any of them. Furthermore, defects in non-code artifacts are extremely costly to fix if they remain undetected until code testing, or even worse, deployment.

Early software life cycle models

"Traditional" software engineering processes, developed in the late 1960s, envisaged a step-by-step approach to software development, where requirements were fixed, then a design was completed, then a system was implemented, tested, deployed, and maintained, all in strict sequence. This cascade of distinct phases, shown in the figure below, became known as the "waterfall" model:

[2] http://dl.acm.org/citation.cfm?id=359602

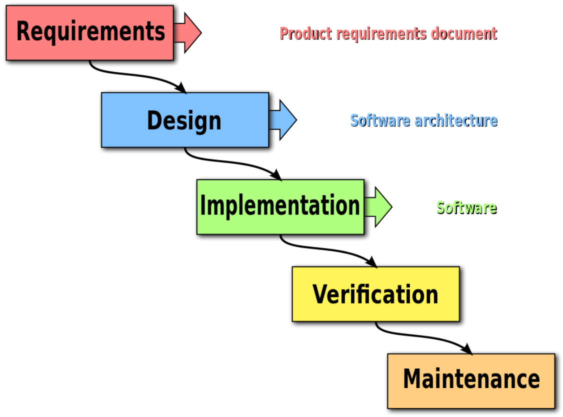


Figure 1: Depiction of the waterfall model[3].

Most modern software development projects do not use the waterfall model. Instead, they use some kind of *iterative* process, multiple delivery points, and many opportunities for feedback between requirements, design, implementation, and testing. *Agile* processes[4] have become increasingly popular.

Digression - the waterfall model and its weaknesses

It's not widely known, but when Winston Royce drew up the original version of the waterfall model[5], he wasn't promoting it as a good idea. He was trying to describe current industry practice and critique it. In practice, it is neither feasible nor desirable to set the artifacts from one phase in stone before getting started on the next. Inevitably -- and even with extensive QA before proceeding to the next phase -- design throws up questions about requirements, implementation reveals issues with the design, and so on.

On top of that, the context in which the system is to be built, the customer's understanding of what the system might do, and the context in which it operates change over time.

However, while the order *in which activities are conducted in the waterfall model hasn't stood the test of time, the activities themselves remain the core of any software engineering project.*

[3] https://commons.wikimedia.org/wiki/File:Waterfall_model.svg
[4] For an introduction to Agile processes, you could do worse than starting with the Scrum Primer, which describes one of the most popular Agile processes, Scrum. (http://scrumprimer.org/) For a description of what makes Agile processes, Agile, try the Agile Manifesto (http://agilemanifesto.org/) Agile processes are discussed at length in FIT2101.
[5] Winston W. Royce (1970). "Managing the Development of Large Software Systems" in: In: Technical Papers of Western Electronic Show and Convention (WesCon) August 25–28, 1970, Los Angeles, USA., https://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf

In the 1970s people noticed that, that while mistakes could happen in any phase of the waterfall process, mistakes in some phases were much more costly than others. Barry Boehm, in his 1981 book *Software Engineering Economics*[6], presented data about the relative costs of fixing faults in the different phases, from several projects using a waterfall model. Fixing mistakes in design was more costly than fixing mistakes in the requirements phase. Fixing mistakes in implementation was more costly again. In general, Boehm found the later a mistake was found, the more is costly it was to fix. The cost of fixing mistakes after the software was deployed was hundreds of times more costly to fix than mistakes found in the requirements phase.

FIGURE 1.5 The relative cost of fixing a fault at each phase of the classical software life cycle. The solid line is the best fit for the data relating to the larger software projects, and the dashed line is the best fit for the smaller software projects. (Barry Boehm, *Software Engineering Economics*, © 1981, p. 40. Adapted by permission of Prentice Hall, Inc., Englewood Cliffs, NJ.)

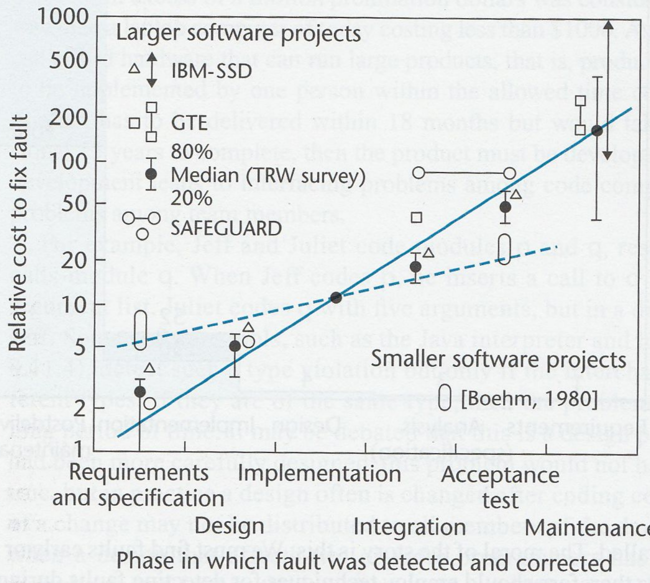


Figure 2: Boehm's findings on project phase versus cost of fixing faults.

Boehm's results, shown in the figure above oversimplify things a little. There's another factor to keep in mind; not only where the mistake was found, but where the mistake was made. Typically, a single mistake in any one artifact has a relatively small impact on the artifacts that directly depend on it, but a bigger impact the further down the line you go. A small mistake in requirements can potentially lead to a very big reworking if it's detected only in acceptance testing. On the other hand, a bug introduced in coding is typically much easier to fix.

The takeaway from all this that mistakes in "high level" artifacts, particularly requirements and architectural design, are really, *really* expensive if they're not caught ASAP. While agile

[6] Barry W. Boehm. 1981. Software Engineering Economics (1st ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

methods can reduce the slope of Boehm’s cost function, it's still going to be very expensive to fix mistakes like misunderstood requirements that make it into code. The way to catch mistakes in these artifacts early is through human review. And that's why I say that of all the quality assurance tasks you will perform, *human review is by far the most important*.

Fagan Inspections

Hopefully I've convinced you of the importance of manual inspection as a quality assurance technique. But just saying "do manual inspections" isn't enough. Should you send your code off to your grandparents and ask them to have a look? If you've got dedicated SQA staff on your project, how should they be involved? Should you just email the code to the reviewer(s)? Or should you schedule a meeting? Is the purpose of the review to identify problems, or propose solutions?

While getting your grandparents to review your code is unlikely to be useful, there are a variety of views on how manual artifact reviews should be conducted. We'll start with one of the oldest and best-known techniques, known as the "Fagan Inspection" after the person who first described the methodology, Michael Fagan of IBM.

Fagan first described his inspection technique in an *IBM Systems Journal* article[7] in 1976, after its use at IBM for some time before that.

The technique makes certain assumptions about software development which are very much of their time: the code he describes was implemented in assembly language, a waterfall life cycle was used, and the "design" presented was at such a level of detail as to correspond to a line of code in a language like Java or Python. However, we will present the technique as he proposed it.

Fagan advocates inspections for most artifacts in a software process - including (detailed) design documents, code, *and test cases*, but omits the detailed description of test case inspections. We do the same here - the extensions and modifications are pretty obvious.

Who?

According to Fagan, typically involves four participants:

- The *moderator*, a developer, preferably with special training in running Fagan inspections, but not necessarily directly involved with the feature being inspected.
- The *designer* - the person who designed the feature which the code under inspection implements.

[7] M. E. Fagan, "Design and code inspections to reduce errors in program development," in *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976. http://www.mfagan.com/pdfs/ibmfagan.pdf

- The *coder* - the person who wrote the code.
- The *tester* - the person who is writing the test cases for the code.

If the same person designed and wrote the code, that person should play the *designer* role and another programmer working in a closely related area should be brought in to play the role of *coder*. If they also wrote the tests, drag in another programmer (preferably one who has testing experience) to play the *tester*.

Fagan states that four is generally enough and the number shouldn't be "artificially" inflated, but if the code has significant external APIs, it may pay to include programmers who are involved in the development of those APIs.

Fagan proposes that the *same* people be involved in the inspection of a design and the inspection of the related code.

What?

The figure below shows a flowchart for a Fagan inspection of a design. In Fagan's original model for inspections, the follow-up code inspection omits the overview, as the same people who inspected the design inspect the code.

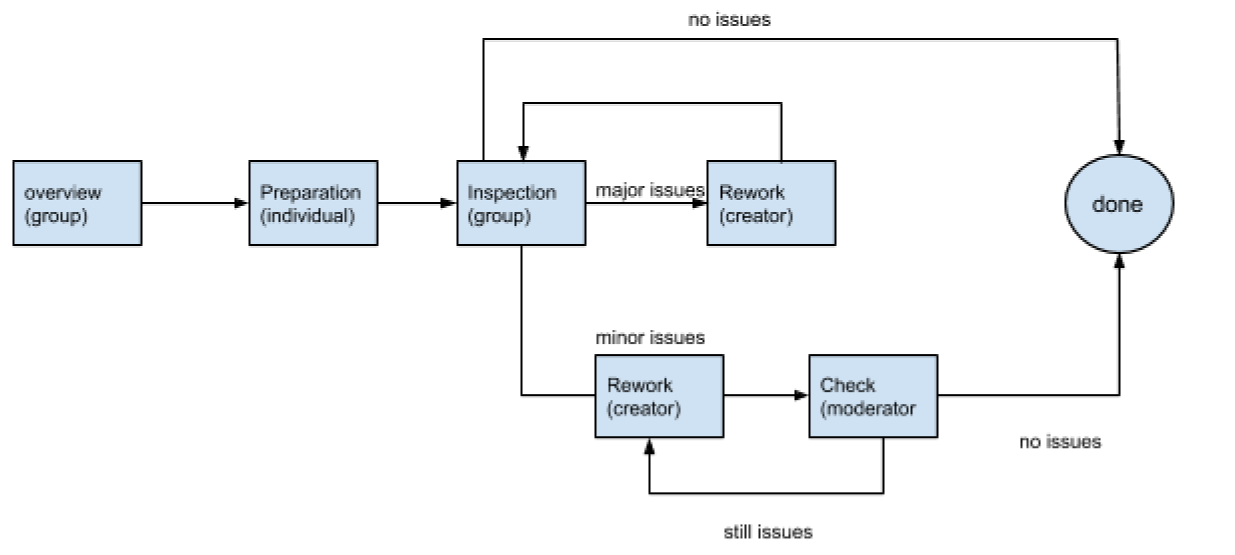


Figure 3: Fagan inspection flowchart.

Overview

In this first stage, the designer explains the design of the "component" under review to the participants, and provides the artifact to the inspection team.

Preparation

The review participants then go off and study the provided artifact, to understand the artifact and look for problems. As well as the design artifact, they are provided with:

- A list of the most common problems found by recent inspections.
- A checklist of clues that provide hints on how to find these common problems.

Participants should be provided with sufficient time to examine design or code properly. Studies have suggested that experienced reviewers perform best when they review no more than 150-200 lines of code per hour, though that will vary according to the complexity of the code.

Inspection

Participants then physically meet. First, a participant chosen by the moderator (usually the coder) explains the artifact in detail. During this process, the other participants ask questions and through this process identify any problems with the artifact. The point is not to find or debate the merits of various solutions, but just to identify the problems. The moderator takes note of the problems found and writes a report which is circulated promptly to all participants after the meeting. Fagan suggests this report should be produced within one day of the inspection meeting.

According to Fagan, an inspection should take no longer than two hours; how much code can be reviewed in that time depends on the complexity of the code. As experience is gained, the appropriate amount of code to be reviewed in one inspection session will become clearer.

Rework

After the inspection, the person responsible for the artifact fixes the errors.

Follow-up

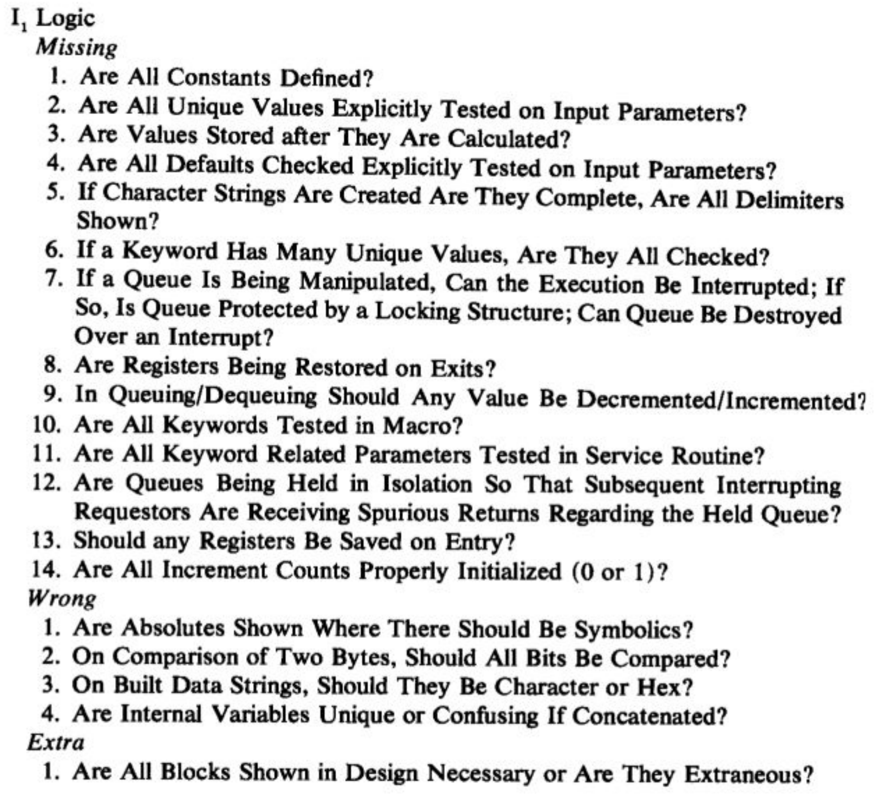
If the problems are minor (according to Fagan, requiring less than five percent of the artifact to be reworked) the moderator can either choose to check the fixes themselves, or reconvene the inspection panel. If the problems require more extensive rework, the panel must be reconvened and the fixes agreed to before the inspection process is considered complete.

What should be on the checklists?

As Fagan says, the preparation and inspection should be aided by a checklist that gives hints on how to detect the most common errors found in previous inspections of this type. These should be based on previous inspections in your project, and previous inspections conducted on similar projects within your organization.

As an undergraduate student, however, you don't have a history of past inspections to base your checklists off. So what's a good starting point for a checklist?

To start at the very beginning, let's have a look at what Fagan was using for "design reviews" (remember, "design" in this context was very detailed design at a level roughly equivalent to high-level language source code). It neatly illustrates both the need for specifics of the environment you're programming in, and some general principles:



I_1 Logic

Missing

1. Are All Constants Defined?
2. Are All Unique Values Explicitly Tested on Input Parameters?
3. Are Values Stored after They Are Calculated?
4. Are All Defaults Checked Explicitly Tested on Input Parameters?
5. If Character Strings Are Created Are They Complete, Are All Delimiters Shown?
6. If a Keyword Has Many Unique Values, Are They All Checked?
7. If a Queue Is Being Manipulated, Can the Execution Be Interrupted; If So, Is Queue Protected by a Locking Structure; Can Queue Be Destroyed Over an Interrupt?
8. Are Registers Being Restored on Exits?
9. In Queuing/Dequeuing Should Any Value Be Decremented/Incremented?
10. Are All Keywords Tested in Macro?
11. Are All Keyword Related Parameters Tested in Service Routine?
12. Are Queues Being Held in Isolation So That Subsequent Interrupting Requestors Are Receiving Spurious Returns Regarding the Held Queue?
13. Should any Registers Be Saved on Entry?
14. Are All Increment Counts Properly Initialized (0 or 1)?

Wrong

1. Are Absolutes Shown Where There Should Be Symbolics?
2. On Comparison of Two Bytes, Should All Bits Be Compared?
3. On Built Data Strings, Should They Be Character or Hex?
4. Are Internal Variables Unique or Confusing If Concatenated?

Extra

1. Are All Blocks Shown in Design Necessary or Are They Extraneous?

Figure 4: "Design review" checklist from Fagan's original paper, 1976.

Clearly, most of these items are specific to writing assembly language systems programs for IBM mainframes in the 1970s. If you're writing in a high-level language, thankfully you don't have to explicitly save and restore registers for instance!

However, there are a couple of items which very much relate to contemporary programming. Items *#2* and *#6* under the *Missing* category, for instance, say something like *"does the program* *deal appropriately with all the different possible values a variable might take at this point?"* Failing to do so remains a primary source of coding bugs today. Similarly, *Missing #14* asks whether loops are initialized correctly, which is still a common source of errors. And *Extra #1* is a sound point that is particularly applicable to the higher-level design that we would review today - in essence *"are there extra elements in this design that aren't necessary, thus making the design more complex than it needs to be?"*.

Based on my own experience (rather than the research literature), I've attempted to come up with some ideas that can help to build useful initial checklists. You should adapt these broad ideas to the particular artifact that you're reviewing.

Notation: Does the artifact follow the syntax rules for this type of object? For instance, if it's a UML class diagram, is it legal UML? This can be expanded on considerably. For instance, we have found that student class diagrams often use incorrect syntax for multiplicities, so we'd put that as a specific item on a review checklist for UML class diagrams. Generally, compilers will catch syntax errors, so *notation* is more important for non-executable artifacts.

Completeness: In general, is everything that needs to be there, there? To take some examples for different types of artifacts:

- In an if statement, does the condition cover all the cases?
- In a UML sequence diagram, are the messages sufficient to realize the use case scenario?
- Does a test case unambiguously specify the expected behaviour?

Testability: As Barry Boehm put it[8]: "A specification is testable to the extent that one can identify an economically feasible technique for determining whether or not the developed software will satisfy the specification. In order to be testable, specifications must be specific, unambiguous, and quantitative wherever possible." I'd put it a little more simply: Can you think of a way to test it? If you can't, it's not good enough. This applies to requirements, design, *and* code. Code that is hard to test is often badly designed, and will be hard to debug or modify, an additional reason to be suspicious.

Consistency: There are two main forms of consistency to look for: consistency *within* an artifact, and consistency *between* artifacts. A common form of inconsistency is when a lower-level artifact (such as code) contradicts behaviour specified in a higher-level one (such as a class diagram).

Error handling: In a lot of software, there is only one way things can go right, but multiple ways in which things can go wrong. This often means that error handling ends up taking more programming effort than normal behaviour. Furthermore, it means that omissions in handling

[8] Boehm, Software Engineering Economics

anomalies are a major source of bugs. So spend time thinking about the ways that error handling is specified in the artifact, and consider whether there are cases that are omitted, or poorly handled. For example, consider:

- Can code handle empty lists or null references?
- What happens if specified preconditions are violated?

Clearly identified purpose: A strong rule of thumb when evaluating any artifact is that you should be able to examine the different parts of it and clearly and succinctly answer the question *"what is this part for?"*. For example:

- When considering a requirement, what is the business/organizational rationale for it?
- When examining a design, what does a class or method do?

A related point is that entities within artifacts should generally have names that clearly reflect their purpose.

Writing quality: Spelling and grammatical errors in deliverable artifacts are a serious flaw, and should be ruthlessly eliminated. In internal artifacts, they are slightly less important, but should still be corrected where they impede comprehension.

Fagan inspections - costs and benefits

As noted earlier, there is strong evidence that Fagan inspections are effective at detecting defects, including on artifacts where both testing and inspections can be conducted.

However, it's not quite that simple. As Myers found, the faults found by inspection and the faults found in testing tend to be somewhat different in nature. Mantyla and Lissenius[9] examined the nature of defects found in inspections in more detail, and observed something very interesting: about 70% of defects found in inspections actually related to what they termed "evolvability" rather than functional defects. That is, they were defects that related to the maintainability, portability and extendability of the software, rather than defects that would cause the software to not deliver the initial functionality required. Perhaps it's one of the reasons why reason an approach known as "cleanroom software engineering" that gained a bit of attention during the 1990s, never took off. In CSE, unit-level functional testing was ditched entirely in favour of inspections and system-level testing. Maintainability, portability and extensibility are important, but so is functionality!

Fagan inspections are a relatively costly process. They require four people to be involved, and they require those four people to be in the same place at the same time in a dedicated meeting

[9] M. V. Mäntylä and C. Lassenius, "What Types of Defects Are Really Discovered in Code Reviews?," in *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430-448, May-June 2009. doi: 10.1109/TSE.2008.71

space. Those costs add up. So it's worth asking; a) how much benefit we get from having three inspectors, rather than one or two (or, conversely, would we do better with more?), and b) whether it's really necessary to have them in the same room at the same time.

There have been a couple of studies suggesting that, in fact, the vast majority of defects found in inspections are found during the preparation phase, *not* during the inspection meeting. Eick et al[10] conducted an industrial study which used statistical techniques to estimate the proportion of faults that remain undiscovered after inspections. Almost as a byproduct of this study, they found that 90% of faults were found in preparation. They also found, interestingly, that despite having quite large inspection teams (from 5 to 12 people) the majority of faults were only found by a single reviewer.

Johnson and Tjahjono[11] conducted a study with student projects specifically comparing the fault-identification effectiveness of "real" and "nominal" inspection teams. Their conclusions were that "nominal" inspection teams, which used a tool to conduct their reviews, without a schedule, and without physically meeting, found as many faults as "real" inspection teams, without the meeting overhead. The one advantage of physical meetings was that they resulted in fewer false positives. While hardly the last word on the issue, it does suggest that actually being in the same place may not always be worth the cost.

Requirements - a special case

While I can't point to anything in the academic literature specifically on this topic, I would argue that requirements review is a special case in which face to face meetings are highly beneficial. Agile development philosophies, the stories from project failures such as Queensland Health, and my own personal experiences (including project failures) point in this direction.

By definition requirements have to be verified, in part, by clients and users, or their proxies. Most of the time, these people will not be IT professionals. However, the documents have to be written with sufficient precision for IT professionals to base their designs off. Requirements documentation needs to be written in a language that is the intersection of what clients and users understand, and what IT professionals understand. This is difficult to achieve, particularly in the first attempt. Furthermore, clients don't always appreciate the consequences of not getting requirements right, so simply sending them a big fat requirements document and asking them to check it doesn't tend to work well.

---

[10] Stephen G. Eick, Clive R. Loader, M. David Long, Lawrence G. Votta, and Scott Vander Wiel. 1992. Estimating software fault content before coding. In Proceedings of the 14th international conference on Software engineering (ICSE '92). ACM, New York, NY, USA, 59-65.

[11] Philip M. Johnson and Danu Tjahjono. 1997. Assessing software review meetings: a controlled experimental study using CSRS. In Proceedings of the 19th international conference on Software engineering (ICSE '97). ACM, New York, NY, USA, 118-127.

The best way to overcome the different languages spoken by IT professionals and clients/users is to get them in the same room, so that clarification can happen. This needs to happen for gathering requirements in the first place; and in my view, it also needs to happen when requirements are validated.

Tool-based support for inspections/reviews

There are a number of different tools that assist in code reviews, such as the (proprietary) *Atlassian Crucible* and the open-source *Phabricator* tool suite. These are particularly useful for "virtual" (not in-person) reviews.

These tools integrate with your version control system[12]. Typically it works like this: when an change is made to the project repository, an email is sent to a list of nominated reviewers. The reviewers can either approve the change, request clarification on some points, request modifications, or reject the change entirely.

If you're trying to manage a virtual review process, adopting some kind of tool like this is a no-brainer.

Instantaneous reviewing - pair programming

For completion, it's worth noting the ultimate in fast turnaround review processes, as particularly advocated by Kent Beck as part of eXtreme Programming (XP) (it's so eXtreme, it has capital letters in the middle of words. Give Beck a break - it was the 90s).

Beck describes pair programming as involving "two people looking at one machine, with one keyboard and one mouse". The person at the keyboard concentrates on the here and now, while their partner is reviewing both the code and thinking strategically about how the code fits in with the larger system. According to Beck, pairing should happen dynamically within the team, rather than being a long-term partnership between two programmers.

Pair programming was controversial when Beck advocated it 15 years ago, and remains controversial now. While it is widely (but universally) accepted accepted pair programming improves code quality, whether it is cost-effective is the subject of heavy debate.

Summary

- Human review of artifacts is the single most important quality assurance method we have.

[12] A version control system is something that keeps track of *all* the revisions to your code and other artifacts as they evolve. It keeps track of who changed what, and allows you to retrieve *any* previous version of the file. They also have tools to manage conflicting edits to the same artifact (think file).

- All artifacts can benefit from human review, including code.
- It is comparable in effectiveness to testing of code, and finds different problems.
- Fagan inspections are a standard, well-regarded methodology for in-person review.
- Virtual inspections seem to be comparably effective and less costly.
- Checklists of things to look for are a vital part of the inspection process.
- Tool support is important for virtual inspections, and are readily available.
- Pair programming is an agile technique which (among other things) gives instantaneous code review.