8 Code Inspection

Software inspections are manual, collaborative reviews that can be applied to any software artifact from requirements documents to source code to test plans. Inspection complements testing by helping check many properties that are hard or impossible to verify dynamically. Their flexibility makes inspection particularly valuable when other, more automated analyses are not applicable.

Software Inspection

- Examine representation of a software system with the aim of discovering anomalies and defects
  - "Check software artifacts for constructs that are known to be problematic from past experience"
- Systematic & detailed review technique
  - Peer review (not author or his boss but inspection team)
- Applicable to all kinds of software artifacts
  - Requirement specification, design documents, source code, ...
- Defined in the 70's by Fagan (IBM)
  - Several alternatives & extensions proposed that vary in rigorousness of the review and focus on particular goals
  - Pair programming can be seen as a light & informal instance

Software Inspection

- Can be applied to essentially any document
  - requirements statements
  - architectural and design documents
  - test plans and test cases
  - source code
- May also have secondary benefits
  - spreading good practices, shared standards of quality
  - pair-programming

- Limitations
  - takes a considerable amount of time
  - re-inspecting a changed component can be expensive
- Used primarily in areas
  - where other techniques are inapplicable or ineffective
  - where other techniques do not provide sufficient coverage

Formal code reviews

- A formal code review is the process under which **inspection** is performed.
- Can be a simple one-on-one meeting or a detailed rigorous code inspection.
- May be organized by the programming or the testing team.

Why have Code Inspections

- Several eyes are better than one pair
- Not all problems are detected by automated tools
- Tools cannot decide whether the problem is real.
- Tools cannot decide whether the problem is serious. [Worth fixing.]

Code Inspection

- Code Inspection is the most formal type of review, which is a kind of static abalysis to avoid the defect multiplication at a later stage.
- The main purpose of code inspection is to find defects and it can also spot any process improvement if any.
- An inspection report lists the findings, which include metrics that can be used to aid improvements to the process as well as correcting defects in the document under review.
- Preparation before the meeting is essential, which includes reading of any source documents to ensure consistency.
- Inspections are often led by a trained moderator, who is not the author of the code.

Code Inspection

- The inspection process is the most formal type of review based on rules and **checklists** and makes use of entry and exit criteria.
- It usually involves peer examination of the code and each one has a defined set of roles.
- After the meeting, a formal follow-up process is used to ensure that corrective action is completed in a timely manner.

Code review checklist

- Design and Architecture errors
- Computation errors
- Comparison errors
- Control flow errors
- Subroutine parameter errors
- Input/Output errors
- Memory allocation errors
- Error discovered from previous code reviews
- Other checks
  - Does your code pass the lint test? E.g., How about gcc compiler warnings?
  - Is your code portable to other OS platforms?
  - Does the code handle ASCII and Unicode?

Inspections vs. Testing

- What attributes are well-handled by inspections but not testing?
- "Fuzzy" non-functional properties
  - Maintainability, evolvability, reusability
- Other properties tough to test
  - Scalability, efficiency Security, integrity Robustness, reliability, exception handling Time sensitive, real-time actions
- Requirements, architecture, design documents
  - Cannot "execute" these as a test

Inspection team

Inspections are characterized by roles, process, and reading techniques, i.e., who the inspectors are, how they organize their work and synchronize their activities, and how they examine the inspected artifacts. Inspection is not a full-time job: Many studies indicate that inspectors' productivity drops dramatically after two hours of work, and suggests no more than two inspection sessions per day. Thus, inspectors are usually borrowed from other roles: junior and senior software and test engineers, project and quality managers, software analysts, software architects, and technical writers. The same studies highlight the delicate relation between inspectors and developers: The efficacy of inspection can vanish if developers feel they are being evaluated. In classic approaches to inspection, managers and senior engineers who participate in inspection sessions are often borrowed from other projects to avoid misinterpreting the goals of inspection.

Inspectors must be selected in a way that balances perspectives, background knowledge, and cost. A developer is most knowledgeable about his own work, and is an invaluable resource in inspection, but he cannot forget days or weeks of hard development work to see clearly all the details that are apparent to someone reading an artifact for the first time. Inspection can benefit from discussion among many inspectors with differing perspectives and expertise, but the cost of inspection grows with the size of the inspection team.

Classic inspection postulates groups from four to six inspectors, but recent studies question the efficacy advantages of large groups of inspectors over groups of two. Modern approaches prescribe different levels of inspection: simple checks performed by single inspectors and complex check performed by groups of two inspectors, reserving larger groups for inspections requiring special combinations of expertise.

Single inspectors are usually junior engineers not involved in development of the artifact under inspection. They combine inspection with training, learning basic standards for specification and programming by checking compliance of artifacts with those standards. Junior engineers are usually paired with senior engineers for checking complex properties. The senior engineer acts as moderator; he or she is in charge of organizing the inspection process and is responsible for the inspection results, while the junior engineer participates in the inspection and the discussion.

Large groups of inspectors (from four to six) balance junior and senior engineers, and may include the developer of the artifact under inspection. A senior engineer, usually a manager borrowed from a different project, plays the role of the moderator, organizing the process and being responsible for the results. Other software and test engineers, both senior and junior, are in charge of reading the inspected artifact, and of discussing the possible problems connected to the relevant elements. The developer is present when the inspection requires detailed knowledge that cannot be easily acquired without being involved in the development. This happens for example, when inspecting complex modules looking for semantics or integration problems.

Developers must be motivated to collaborate constructively in inspection, rather than hiding problems and sabotaging the process. Reward mechanisms can influence the developers' attitude and must be carefully designed to avoid perverse effects. For example, fault density is sometimes used as a metric of developer performance. An assessment of

fault density that includes faults revealed by inspection may discourage developers from constructive engagement in the inspection process and encourage them to hide faults during inspection instead of highlighting them. At the very least, faults that escape inspection must carry a higher weight than those found during inspection. Naive incentives that reward developers for finding faults during inspection are apt to be counterproductive because they punish the careful developer for bringing a highquality code to the inspection.

Checklists

Checklists are a core element of classic inspection. They summarize the experience accumulated in previous projects, and drive the review sessions. A checklist contains a set of questions that help identify defects in the inspected artifact, and verify that the artifact complies with company standards. A good checklist should be updated regularly to remove obsolete elements and to add new checks suggested by the experience accumulated in new projects. We can, for example, remove some simple checks about coding standards after introducing automatic analyzers that enforce the standards, or we can add specific semantic checks to avoid faults that caused problems in recent projects.

Checklists may be used to inspect a large variety of artifacts, including requirements and design specifications, source code, test suites, reports, and manuals. The contents of checklists may vary greatly to reflect the different properties of the various artifacts, but all checklists share a common structure that facilitates their use in review sessions. Review sessions must be completed within a relatively short time (no longer than two hours) and may require teams of different size and expertise (from a single junior programmer to teams of senior analysts). Length and complexity of checklists must reflect their expected use. We may have fairly long checklists with simple questions for simple syntactic reviews, and short checklists with complex questions for semantic reviews.

Modern checklists are structured hierarchically and are used incrementally. Checklists with simple checks are used by individual inspectors in the early stages of inspection, while checklists with complex checks are used in group reviews in later inspection phases. The preface of a checklist should indicate the type of artifact and inspection that can be done with that checklist and the level of expertise required for the inspection.

The sidebar on page 346 shows an excerpt of a checklist for a simple Java code inspection and the sidebar on page 347 shows an excerpt of a checklist for a more complex review of Java programs.

A common checklist organization, used in the examples in this chapter, consists of a set of features to be inspected and a set of items to be checked for each feature. Organizing the list by features helps direct the reviewers' attention to the appropriate set of checks during review. For example, the simple checklist on page 346 contains checks for file headers, file footers, import sections, class declarations, classes, and idiomatic methods. Inspectors will scan the Java file and select the appropriate checks for each feature.

The items to be checked ask whether certain properties hold. For example, the file header should indicate the identity of the author and the current maintainer, a cross reference to the design entity corresponding to the code in the file, and an overview

of the structure of the package. All checks are expressed so that a positive answer indicates compliance. This helps the quality manager spot possible problems, which will correspond to "no" answers in the inspection reports.

FEATURES (where to look and how to check):

Item (what to check)

FILE HEADER: Are the following items included and consistent?	yes	no	comments
Author and current maintainer identity			
Cross-reference to design entity			
Overview of package structure, if the class is the principal entry point of a package			
FILE FOOTER: Does it include the following items?	yes	no	comments
Revision log to minimum of 1 year or at least to			
most recent point release, whichever is longer			
IMPORT SECTION: Are the following requirements satisfied?	yes	no	comments
Brief comment on each import with the exception of standard set: java.io.*, java.util.*			
Each imported package corresponds to a dependence in the design documentation			
CLASS DECLARATION: Are the following requirements satisfied?	yes	no	comments
The visibility marker matches the design document			
The constructor is explicit (if the class is not *static*)			
The visibility of the class is consistent with the design document			
CLASS DECLARATION JAVADOC: Does the Javadoc header include:	yes	no	comments
One sentence summary of class functionality			
Guaranteed invariants (for data structure classes)			
Usage instructions			
CLASS: Are names compliant with the following rules?	yes	no	comments
Class or interface: CapitalizedWithEachInternal- WordCapitalized			
Special case: If class and interface have same base name, distinguish as ClassNameIfc and Class- NameImpl			
Exception: ClassNameEndsWithException			
Constants (final):			
ALL CAPS WITH UNDERSCORES			
Field name: capsAfterFirstWord. name must be meaningful outside of context			
IDIOMATIC METHODS: Are names compliant with the following rules?	yes	no	comments
Method name: capsAfterFirstWord			
Local variables: capsAfterFirstWord.			
Name may be short (e.g., i for an integer) if scope of declaration and use is less than 30 lines.			
Factory method for X: newX			
Converter to X: toX			