Practical Code Inspection Techniques for Object-Oriented Systems: An Experimental Comparison

Alastair Dunsmore, Formedix
Marc Roper and Murray Wood, University of Strathclyde

Software inspection is the process of reading software artifacts to find defects. Over the past 25 years, it has become an effective and efficient means of detecting industrial software defects. However, in spite of its broad application, we have found a significant lack of information indicating how to apply inspections to object-oriented code. Inspections were developed when the procedural programming paradigm was dominant, but the last 10 years have seen the OO paradigm growing in influence and use, particularly since the introduction of C++ and Java.

The aim of empirical software engineering research is to help determine the effectiveness of development processes and techniques. Most major conferences have sessions dedicated to empirical studies and journals such as the *Empirical Software Engineering Journal* are devoted to empirical research that covers the whole spectrum of software engineering.

Here, we highlight problems that could arise when inspecting OO code and discuss the results of a long-term empirical study investigating reading strategies specifically developed to address these difficulties.

### Problems inspecting object-oriented code

In an OO system, the information required to understand a piece of functionality is often widely distributed. To understand one piece of code, the trail of method invocations must be followed through many other methods and classes, possibly including the traversal of inheritance hierarchies. A study investigating the inspection of OO code and an industrial survey have noticed that many hard to find defects in OO code share this same characteristic—the
information required to understand the defect is distributed throughout the system. We have termed this characteristic delocalization, after the description by Elliott Soloway and his colleagues of delocalized plans in program comprehension.\textsuperscript{8} In a delocalized plan, the “code for one conceptualized plan is distributed non-contiguously in a program.” Soloway and his colleagues suggest that these plans are difficult to understand because, at any one time, only fragments can be seen, leaving the reader to guess based on what is locally apparent.

Upon reflection, delocalization appears to be a fundamental characteristic of the OO paradigm. Key features such as inheritance, polymorphism, small methods, and class libraries distribute closely related functionality throughout the code. To illustrate the nature of delocalization, consider the Java method in Figure 1.

To fully appreciate this code, we must examine a variety of sources of delocalized information—for example, other classes in the system, system documentation, and classes in the Java class library (see Figure 2).

Associated with the delocalization problem is the issue of a reading strategy.\textsuperscript{3} Two general strategies for reading and understanding program code are systematic and as-needed. Soloway and his colleagues describe these strategies in the context of comprehending a program for the purpose of maintenance:\textsuperscript{8}

- **Systematic strategy**: Programmers using this strategy started at the beginning of the program and documentation and traced the flow of the entire program, using various forms of simulation (for example, symbolic, actually plugging in values, and so forth).
- **As-needed strategy**: Programmers using this strategy chose to study portions of the code and documentation, which they believed will be useful for constructing their enhancement. They read those portions as they decided that they needed them.

Systematically reading and understanding all code and its dependencies is a possible approach to dealing with delocalization. However, this would be expensive, time-consuming, and, due to limitations on the amount of information we can usefully retain in short-term memory at one time, unrealistic. More practically, when inspecting OO code, an as-needed reading approach must be adopted. However, the danger is that an as-needed approach will force inspectors to make unverified assumptions.

A related problem is how to select the code to be inspected. There are restrictions on the time an inspection lasts and the amount of code that can be reviewed (approximately 200 lines of code in two hours).\textsuperscript{9} If the inspection lasts too long or if too much code is read, the inspection’s effectiveness is reduced. Owing to the large number of dependencies within OO code (more than procedural languages), it becomes difficult to isolate a reasonably sized

---

**Figure 1. A Java method illustrating the nature of delocalization.**

```java
private void purge() {
    GregorianCalendar today = new GregorianCalendar();
    today.roll(Calendar.DATE, false);
    for (int i = 0; i < reservations.size(); i++) {
        if (today.after(((Reservation) reservations.elementAt(i)).getDate())) {
            reservations.
                removeElementAt(i);
            date = 0;
        }
    }
}
```

**Figure 2. Examples of delocalization in the purge() method.**
section of code. We must address the problem of identifying suitable chunks of code to inspect (that fit within inspection limits) and determining how to break those chunks free of the system (minimizing dependencies).

Furthermore, current reading techniques are based on a static (compile-time) view of the code. However, the dynamic (run time) view of OO code is quite different. Erich Gamma and his colleagues state that

In fact, the two structures [run time and compile time] are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice-versa.10

Reading techniques

To address these issues, we developed and investigated abstraction-driven, use-case-driven, and checklist-based techniques. In developing them, we aimed to minimize any assumptions about the development process in which the inspections occurred. Therefore, we simply assumed the availability of requirements specifications (in any notation), class diagrams, use cases, and associated sequence diagrams.

Abstraction-driven technique

This approach, based on Stepwise Abstraction,11 requires inspectors to reverse engineer an abstract specification for each method in the code. The technique was specifically developed to address delocalization—that is, to find a way to resolve the references to nonlocal information by providing many of the benefits associated with systematic reading, but in a reasonably efficient manner. Creating abstractions forces a deeper understanding of the code and provides an abstract summary of the method for reference in future inspections. The basic technique

- Analyzes interdependencies (couplings) within the whole system, first inspecting those classes with the least dependencies, ideally in chunks of no more than 200 lines of code.
- Analyzes methods within classes, inspecting methods with the least dependencies first.
- Inspects classes and methods using the abstraction-driven reading strategy. This involves reverse engineering an abstract specification for each method.

- Traces and understands, during inspection, any references to external classes. This might involve reading other methods, documentation, or previously created abstractions. This understanding is necessary to correctly specify each method.
- Ensures that as the inspection of the overall system proceeds, more of the classes have abstract specifications. This should limit the need to spend time understanding other classes during future inspections (and maintenance tasks).

The abstract specification for each method should identify any changes of state and outputs in terms of inputs and prior state. The specification should be

- Brief (as short as possible while capturing all aspects of the method)
- Declarative (describe what the method does, not how it does it)
- Complete (cover all aspects of functionality including that derived from references to other classes)

Take, for example, the isRegistered method:

```java
public boolean
isRegistered(String e) {
    int i = 0;
    while (((i < theUsers.size()) &
    (((Person)theUsers.
        elementAt(i)).getEmail()).
        equals(e)))
    i++;
    return i < theUsers.size();
}
```

The following shows, for part of this method, how an inspector can gain an understanding using a stepwise reading approach:

- (Person)theUsers.elementAt(i) gets the ith element from the vector theUsers and casts it to a Person instance. A vigilant inspector would check whether all users can be cast to the Person type.
- (((Person)theUsers.elementAt(i)).
    getEmail()) gets the email (a String) of the ith element in the vector theUsers.
- (((Person)theUsers.elementAt(i)).
    getEmail()).equals(e) compares the
input String e with the email of the ith element in the vector using String equals(). String equals() returns true if the two String instances consist of identical characters.

Continuing this process, the inspector builds up a complete abstraction of the method:

Returns true if the input string e matches the email address of one of the Person elements in the user collection; otherwise, returns false.

This example highlights how the abstraction process could encourage the inspector to develop a greater understanding of the code’s delocalized aspects, making assumptions or misinterpretations less likely, producing reusable abstractions as side-product.

Use-case technique

The use-case technique reads OO code from a dynamic model viewpoint. Use cases form part of the Unified Modeling Language. The technique aims to ensure that each object responds correctly to all the possible ways in which it might be used. In other words, is it a good citizen of the system? More precisely, with respect to the use cases in which the object participates, the technique verifies that:

- The correct methods are being called by users of the class.
- The decisions and state changes made within each method are correct and consistent.

The basic approach devises several scenarios from the use case and examines how the class under inspection deals with them. A scenario is defined as “a specific sequence of actions that illustrate behavior.” Defects are discovered by noticing missing or incorrect methods, erroneous state changes, and so forth. The principle behind the technique is that it forces the inspector to consider the context in which an object is used. It might be that several inspections are necessary to completely check a class since some parts might not be involved in an individual use case.

Inspectors take each use case in turn (see Figure 3) and devise scenarios based on the preconditions, success and failure conditions, and the exceptions—for example, seat booking successfully cancelled, no such booking held in the system. The scenarios drive the code inspection. For each derived scenario, the inspector must:

- Make a note of the anticipated final outcome in relation to changes in state or outputs from classes under inspection.
- Trace the scenario through the sequence diagram (which is assumed to be available as a result of design) by following message calls between objects. During this time, inspectors must keep in mind the state of the system that would cause this scenario to occur. (See the example in Figure 4.)
- On encountering the class under inspection, verify that the expected methods are being called to support the scenario.
- When a method in the class under inspection is called, verify any decisions and state changes that are made to check that they are correct and consistent with respect to the scenario.
- Follow any further method calls while inspecting a method. If they are associated with the class under inspection, the method
is similarly verified; otherwise, the inspector returns to following the sequence diagram.

Compare the final and predicted state/outputs once the scenario has been completed. If a difference exists between the two, the inspector must find and highlight a candidate defect.

Checklist technique

Checklists are a straightforward and commonly used technique to help with individual code inspection. They are based on a series of specific questions intended to focus the inspector’s attention toward common sources of defects. Oliver Laitenberger and colleagues13 have questioned the effectiveness of checklists because of their general nature, focus on defect types driven by historical data, and because they don’t force the inspector to understand the artifact under inspection. We tried to address these concerns by designing a checklist that focused on OO characteristics (as well as general sources of error) and by providing specific guidance and a checklist structure that were intended to encourage a progressive understanding of the code.

Figure 5 shows part of the final checklist used. The checklist contains two components—one highlights possible features of the code in which to concentrate, and the other provides questions to help identify defects for

<table>
<thead>
<tr>
<th>Feature</th>
<th>Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each class:</td>
<td></td>
</tr>
<tr>
<td>1   Inheritance</td>
<td>Is all inheritance required by the design implemented in the class?</td>
</tr>
<tr>
<td>2   Class Constructor</td>
<td>Is the inheritance appropriate?</td>
</tr>
<tr>
<td>3   Are all instance variables initialised with meaningful values?</td>
<td></td>
</tr>
<tr>
<td>4   If a call to super is required in the constructor, is it present?</td>
<td></td>
</tr>
<tr>
<td>For each method:</td>
<td></td>
</tr>
<tr>
<td>5   Are all parameters used within a method?</td>
<td></td>
</tr>
<tr>
<td>14  Are all assignments and state changes made correctly?</td>
<td></td>
</tr>
<tr>
<td>15  For each return statement, is the value returned and its type correct?</td>
<td></td>
</tr>
<tr>
<td>16  Does the method match the specification?</td>
<td></td>
</tr>
<tr>
<td>For each class:</td>
<td></td>
</tr>
<tr>
<td>17  If inherited methods need to behave differently, are they overridden?</td>
<td></td>
</tr>
<tr>
<td>18  Are all uses of method overriding correct?</td>
<td></td>
</tr>
</tbody>
</table>
To encourage a gradual understanding, we grouped the questions into three sections that deal with class-level issues (inheritance, construction), method-level issues (data referencing, method behavior, and so forth), and finally, method-overriding issues. The questions are answered in order for each class under inspection.

**Inspection study**

Our study compared the three reading techniques’ defect detection capability. We organized it in a university environment using third-year honors computer science students. We split participants into three groups of 23, and each group contained the same mix of abilities. Each group focused on only one technique. Just over half of the defects seeded in the code (eight out of 14) were delocalized in nature. We based the inspection exercise on approximately 180 lines of code from two OO classes (from a system made up of 18 classes and approximately 2,800 lines of code) and limited it to a 90-minute session. Further details appear elsewhere.

Figure 6 shows the average defect-detection rates for each of the three reading techniques. Participants using the checklist technique appear to find more defects and at a quicker rate, although performance drops off sharply after the first 60 minutes. The defect-detection rates of the abstraction-driven and use-case participants are similar, perhaps because both techniques have a higher initial overhead. The abstraction technique requires creating method abstractions, while the use-case participants must generate scenarios from use cases and trace these through the sequence diagrams as a precursor to reading the actual code. While the abstraction-driven performance seems to be levelling off toward the end of the 90 minutes, the use-case performance is still rising. This suggests that the use-case participants might have discovered more defects had we given them a longer period of time.

Figure 7 summarizes the performance of each reading technique. Participants using the checklist technique produced the strongest performance, detecting an average of 52 percent of defects, while abstraction discovered 44 percent and use case 40 percent. The Kruskal-Wallis statistical test showed a significant result at the 10 percent level but not at the 5 percent level. Figure 8 shows the percentage of participants discovering each defect, split by reading technique.

The key lessons from this study follow.

**Checklist technique**

As well as having the strongest performance overall, anecdotal evidence from participant post study reports and the timing information that was gathered shows the checklist to be the most efficient technique to apply. About three-quarters of the checklist participants reported that the technique was straightforward to use. Delocalized defects were spread throughout the response range rather than being clustered (see Figure 8). This might suggest that checklist questions designed to focus on delocalization issues were having some effect.

One threat to the validity of the checklist...
results concerns how we constructed the checklist questions. We generated some of these questions from defect information gathered from two previous studies, and it is possible that the defects seeded into the code were too similar to those historical defects.

**Abstraction-driven technique**

Although this technique’s overall performance was not quite so strong so the checklist, it appears to be effective at detecting delocalized defects techniques but less effective for other defects (see Figure 8).

Participant reports show that the abstraction technique encouraged a deeper level of code understanding (perhaps leading to the better detection of delocalized defects). From grading based on previous programming classes, we found it helped the weaker participants’ defect-detection ability. However, the technique had the opposite effect on the stronger participants, prompting speculation that it might hinder the natural abilities of the more capable participants (a result similar to that found in the first application of the abstraction technique).

An outstanding question is the most appropriate format for the reverse-engineered abstractions—both in terms of ease of creation and usefulness. For more complex methods, it is challenging to write brief, clear but sufficient specifications (some specifications ran to about 12 lines of natural language). Further work is required to explore more formal or graphical specifications compared to the declarative, natural-language specifications generated in this work.

**Use-case technique**

Several participants noted in their post-study report that one of this technique’s main strengths was that it dealt with methods in the context of the executing system, giving them a better idea of whether the code was operating as expected. Overall, the defect results were the weakest of the three. Several issues that require further investigation:

- The technique was slow and time consuming. Participants had problems creating the scenarios from the use cases, had to record too much intermediate state information, and had difficulties following large sequence diagrams.

![Figure 8. Defect detection percentages split by technique: (a) checklist, (b) abstraction, (c) use case.](image)
ternal threats to validity that replication can help to address: use of students as participants, dependence on the particular system used, nature of seeded defects, quality of use cases, and checklists. Further studies should also investigate whether our strict limit on the inspection time had a significant effect on the use-case approach’s performance, which was still on an upward curve at the end of the exercise (see Figure 6).

This work addresses some of the issues facing effective OO code inspection. Validity concerns limit the findings, particularly using students as subjects rather than using skilled practitioners. The significant message is that those using inspections in practice should consider tailoring their methods to deal with delocalization and the difference between the static and dynamic views.

Checklists are the easiest way to do this. If checklists are tailored to the particular development environment using historical error data and they integrate questions that target OO features and encourage code understanding, then they can be an effective aid to OO code inspectors. We also advise using questions that force detailed consideration of the code, and its interrelationships in particular.

The abstraction technique that was developed provided encouraging results with respect to the detection of delocalized defects. This technique has a higher overhead than checklists, but offers a potential long-term advantage through the creation of abstractions. Further studies are required to determine the most appropriate form of the derived abstractions, both in terms of ease of creation and usefulness as documentation that reduces the need to read the associated code. The reading ordering of code for inspection using stepwise abstraction to help with delocalization are aspects of the technique that are recommended.

While the overall results for the use case approach that was developed were weaker, the idea of focusing some of an inspection team on the dynamic view is intuitively appealing. Arguably, such an approach focuses better on interclass relationships and, if based on use cases, provides a technique that is explicitly comparing code against requirements. Our study’s participants found this approach the most demanding; this problem might be reduced if used by those with more industrial experience. Further work is required to refine this approach into a practical reading technique.

While the types of defects found by each technique in this study are not significantly different (see Figure 8) we think that, where practical, OO inspections should be based on teams of inspectors using a combination of at least two techniques. The need for a combination of complementary views is advocated by the Perspective Based Reading technique, where different perspectives are used to represent different stakeholders of the software—for example, tester, designer, and analyst. Each perspective is expected to highlight different types of defects. Using a combination of reading techniques, such as those discussed in this article, seems to offer the potential to deal with recurring defect types, unusual defects that may require deeper insights and, particularly, defects that are associated with the features of object-orientation that can distribute functionality throughout a software system.

Finally, a concern with our results is that individual techniques were only discovering 40-50 percent of defects when the literature suggests that inspections typically discover 50–90 percent of defects. We might attribute these results to the study validity issues raised earlier (for example, nature of defects and inexperienced participants), and also the novelty of the techniques. However, even with refinement, inspection is not going to guarantee near 100 percent defect discovery and it is therefore important to recognize that it is only one contribution to the software verification toolset. Code inspection should be used in conjunction with software testing and inspection techniques are vital for defect detection during analysis and design.

Acknowledgments

We are grateful to various anonymous reviewers along the way for their constructive comments on this article.
References


About the Authors

Alastair Dunsmore is a software developer with Formedix (Paisley, UK) which builds clinical trial specification and data-capture systems. His research interests are software inspection and design. He received a PhD in computer science from the University of Strathclyde in Glasgow, UK. Contact him at alastairdunsmore@formedix.com.

Marc Roper is a senior lecturer in the Department of Computer and Information Sciences at the University of Strathclyde, Glasgow, UK. His research interests include developing tools and techniques to support software engineers in analyzing and understanding large software systems. He received an Honors degree in computer science from the University of Reading and a PhD from the Council for National Academic Awards. He is a member of the IEEE Computer Society. Contact him at marc@cis.strath.ac.uk.

Murray Wood is a senior lecturer in the Department of Computer and Information Sciences at the University of Strathclyde, Glasgow, UK. His research interests are software architecture and design, software inspection, and empirical evaluation. He received a PhD in computer science from the University of Strathclyde in Glasgow, UK. He is a member of the IEEE Computer Society. Contact him at murray.wood@cis.strath.ac.uk.

NEW for 2003!
IEEE Security & Privacy
Ensure that your networks operate safely and provide critical services even in the face of attacks. Develop lasting security solutions, with this new peer-reviewed publication. Top security professionals in the field share information you can rely on.

Don’t run the risk! Be secure.
Order your charter subscription today.

http://computer.org/security