Alastair Dunsmore
University of Strathclyde

The Role of Comprehension in Object-Oriented Code Inspections

Summary of Work

The following briefly summarises the areas looked at in the past year, and how they have formed the basis for the proposed thesis [2].

Inspection

The software inspection process was originally developed at IBM in 1972 and has established itself as an effective means of finding defects. The inspection process can be described as a series of stages; overview, individual preparation, group inspection, rework and follow-up.

One active area of inspection research is looking at different defect detection methods. During the preparation stage, each inspector works individually and attempts to gain an understanding of the documents they have been given. During this time an inspector can use defect detection methods to help find defects in code documents before the group inspection meeting. Currently the most popular defect detection methods are **ad-hoc**, where inspectors use a non-systematic technique to find defects, and **checklists**, which contain a series of rules to guide the inspector in looking through code. More recently, the **scenario** approach was created to address a perceived lack of effectiveness in the use of ad-hoc and checklist methods. Several articles on inspection have suggested that an inspector should be free to use any checking method during the preparation stage that they feel gives results. This and conflicting comments from other authors highlight the current lack of agreement on the most effective way to detect defects. Both checklists and scenarios were created at a time when the vast majority of inspections were carried out on traditional procedural languages.

One author has suggested that during the preparation stage an inspector should be placing a higher priority on understanding the work product, and that finding errors should be of secondary concern. Another source has suggested that "you cannot inspect what you cannot understand", and that software engineers have poor strategies when attempting to understand a given document for inspection. The suggested solution was that a better understanding and fuller use of program comprehension techniques could improve an inspector's performance.

There have been many tools developed for inspection, specifically designed to support the inspection process. As highlighted in [1], most of these tools do not include much support for the preparation phase of inspection, e.g. there is little in the way of defect detection aids, instead these tools concentrate more on supporting the overall inspection process.

Object-Oriented Paradigm

In 1994, A survey suggested object-oriented programming languages were the second most popular programming language used by development programmers. At that time it was noted in the literature that there was a lack of quantitative data for areas such as inspection, testing and reuse where object-oriented code was concerned.

Many authors have in the last decade highlighted the new features the object-oriented paradigm brings to programming, but at the same time they have also suggested that many of these new features complicate already established software engineering areas. One such area is software maintenance. Dynamic binding has been highlighted as causing problems by increasing the number of dependencies within programs. Tracing these dependencies is vital for effective software maintenance. Another object-oriented feature, polymorphism also increases the number of dependencies within programs.

More recently, John Daly at Strathclyde University carried out several investigations to gauge the opinion of industry towards the object-oriented paradigm. Many developers suggested C++ was a hindrance to the uptake of object technology due to its deficiencies when compared with purer object-oriented languages. The second most suggested reason among developers for understanding problems was inheritance. One reason was if the design used was inappropriate, then inheritance could be even more of a hindrance, whereas a good design could allow inheritance to improve levels of understanding. Along with inheritance, poor design was also highlighted as reasons for difficulties with other object-oriented features such as polymorphism, dynamic binding and software maintenance.

Program Comprehension

Program comprehension is a process used within many areas of software engineering, including software maintenance and software reuse. In software maintenance it is estimated that up to 60% of the total time allotted is spent on program comprehension. Due to this factor, a large amount of research has been carried out, in an attempt to guide and support software engineers in this process.

Several cognitive models have been suggested which attempt to explain how a software engineer goes about the process of understanding code. These include[1] **Bottom-up comprehension**, **Top-Down comprehension**, **Systematic**

[1] A fuller description of the comprehension methods can be found in the thesis proposal [2]

comprehension, **As-needed** and **Integrated comprehension**. Research has suggested that there is no one cognitive model that can explain the behavior of all programmers, and that it is more likely that programmers, depending on the particular problem will swap between models.

The **as-needed** comprehension model involves the software engineer looking only at code related to a particular problem or task. This description of as-needed comprehension is very close to the description of the defect detection methods of checklists and to a lesser extent scenarios. A problem with as-needed comprehension is that it can miss some of the dependencies within code fragments. The effectiveness of as-needed comprehension methods then become questionable when using them for defect detection in object-oriented code, which due to its many features contains a greater number of dependencies than traditional procedural programs.

Visualisation

With the increasing size of programs and the improvements in computing technology, visualisation has become an important support aid for program comprehension. It is believed that effective visualisations can allow interaction with large amounts of data in a fast and effective manner, and help find hidden characteristics, patterns and trends. Several reasons have also been suggested as to why graphics are preferable to text. These include more comprehension, more memorable, more accessible, faster to grasp, and more fun. It has also been suggested that visualisation can help comprehension by providing visual displays for abstract chunks of textual structures. This helps to reduce the interpretation load on the software engineer.

There are currently a large number of visualisation systems all supporting the software comprehension process directly or indirectly (an overview of some of these can be found in [1]). Many of these tools suggest they offer the solution to comprehension problems, but many of the commercial tools contain features that have no empirical backing as to their usefulness.

Thesis Direction and Future Plans

There are several reasons why I have chosen to carry out research into the area of object-oriented code inspections:

- There is currently a lack of formal guidance or empirical research on the inspection of object-oriented code. Most of the current research is based on procedural code. How will inspections cope with the object-oriented paradigm specific features?
- In current (procedural) inspections, to help with defect detection either checklists or scenarios are used, but there is no evidence to show if these techniques would be as effective if used for inspections on object-oriented code.
- It has been suggested by several researchers that comprehension could play an important part during inspection and this could especially help with the inspection of object-oriented code, but currently there is a lack of empirical evidence in this area.

The thesis [2] proposes to investigate the link between comprehension and defect detection in inspection and compare various comprehension, defect detection and visualisation methods and show which of these provides the most effective levels of defect detection for object-oriented code inspection. It is also hoped that as well as investigating static visualisations (visualisations based only on the program text), dynamic visualisations (visualisations based on the execution of code) will also be looked at, which until now have been shied away from by the inspection community.

The vast majority of currently published research in the area of comprehension, defect detection and inspection is based on traditional procedural languages. There is not much in the way of empirical evidence on the effect of the object-oriented paradigm in these areas or the usefulness of comprehension during inspection. It is hoped to provide empirical evidence as to the usefulness of different methods of detecting defects in object-oriented code during inspection, showing the levels of defect detection offered. It is also hoped that it can be shown if an increase in understanding of the program code, through the use of a comprehension method can help increase the number of defects detected during inspection. The empirical evidence is also a first step in confidently building a tool around techniques that have been proven to be effective for particular domains. Currently many commercial tools appear on the market containing features, which are billed as solving comprehension problems, but which appear to have little or no empirical backing.

References

1. A. P. Dunsmore, *Comprehension and Visualisation of Object-Oriented code for Inspections*, Technical Report, EFoCS-33-98, Computer Science Department, University of Strathclyde, 1998.
2. A. P. Dunsmore, *Thesis Proposal*, Empirical Foundations of Computer Science (EFoCS), Department of Computer Science, Strathclyde University, 1998, http://www.cs.strath.ac.uk/~apd/