

Comprehension and Visualisation of Object-Oriented Code for Inspections

Alastair Dunsmore

EFoCS-33-98

Empirical Foundations of Computer Science (EFoCS)
Department of Computer Science
University of Strathclyde
Livingstone Tower
Glasgow G1 1XH, U.K.

August 1998

E-mail: apd@cs.strath.ac.uk

Abstract

This paper considers the role of comprehension during the preparation and defect detection phases of the software inspection process. Software inspection is generally accepted as a useful technique for finding errors in both documents and code. However, there is no general agreement on how defects are best detected and, in particular, how much understanding of the product is required and how that understanding is best achieved. Some inspection processes provide no guidance. Many advocate fairly informal aids such as checklists. Recently more structured techniques, in the form of scenarios, have been proposed. The need for increased comprehension seems particularly relevant to object-oriented technology as a result of inherent features, which appear to increase inter-component dependencies. This paper reviews the Software Engineering literature investigating the role of comprehension and the related topic of program visualisation during the preparation and defect detection phases of inspection. It considers particular features of object-oriented technology that may require enhanced comprehension during inspection. It draws on similarities between software maintenance and software inspection to suggest that there are potential benefits to be obtained in using comprehension techniques and tools, developed for maintenance, during inspection.

1. Introduction

Software inspection has been around since the early 70's, and has established itself as an effective means of finding defects. It was originally developed by Fagan [14] in 1972. Since Fagan's original work there have been other inspection processes suggested, most being a variation on Fagan's original work. One of the key stages in Fagan's inspection is the preparation stage. This is where an inspector prepares for a group meeting by studying a set of documents. Given the documents to be inspected, according to Fagan [14], each individual inspector was to "understand the design, its intent and logic". To achieve this a certain amount of comprehension is necessary.

There is currently a lack of agreement on the best method or methods to use during the preparation stage of inspection. Several of the processes provide no guidance on how to prepare or detect defects. Checklists have been used as an informal guide for many years (see [23] for example code checklist). More recently more structured aids in the form of scenarios have been proposed [50]. There is at the moment a lack of empirical evidence for the levels of comprehension obtained by using the current methods of defect detection in the preparation stage of inspection.

The process of program comprehension in software maintenance is similar to the preparation stage of code inspections, where inspectors are supposed to gain an understanding of the documents given to them [14]. In some cases as much as 60% [9] of the total time used for software maintenance is spent on program comprehension. This has led to significant research in this area with several cognitive models being proposed which attempt to explain how software maintainers go about understanding code and several tools being created with a visualisation aspect to aid understanding.

In the last two decades there has been a slow but steady increase in the uptake of the object-oriented paradigm in both academia and industry. In 1994, object-oriented languages were the second most used language by program developers in the USA [28]. The object-oriented paradigm has introduced several new ideas including classes, class methods, data encapsulation, etc. These features may have implications for program comprehension, maintenance, testing, and inspection. Jones [28] highlighted several gaps in current knowledge of the object-oriented paradigm, one of which is for guidance on how to inspect object-oriented programs.

This paper looks at the inspection process and suggests that the methods currently used during the preparation and defect detection phases of inspection are not sufficient enough to be able to fully comprehend object-oriented code, and attempts to show how both comprehension and visualisation could help address some of these problems.

The remainder of this paper takes the following form. The next section describes the inspection process and the current levels of comprehension involved in the preparation stage of inspection. Section 3 briefly describes the area of program comprehension. Section 4 provides a brief introduction to the field of visualisation, and how it relates to comprehension. Section 5 discusses some of the problems that the object-oriented paradigm introduces for program comprehension and inspection. Section 6 looks at some tools currently available for visualization and inspection and considers if any of these tools offer any useful features for program comprehension during inspection of code. Section 7 describes possible future tool support. This paper concludes with section 8, summarising this paper and briefly discusses possible areas for future work.

2. Code Inspection

The inspection process was developed by Fagan [14] in 1972, and then updated by Fagan in 1986 [15]. Since the original work of Fagan, there have been several different inspection processes proposed, all a variation on Fagan's original work, attempting inspections using a modified process structure. A list of some of the better known can be found in [35], along with a general description of each. The ideas presented in this paper are based round Fagan's inspection process and are primarily concerned with the inspection of program code documents.

This section gives a brief description of Fagan's original inspection process highlighting the preparation stage and commenting on the current state of defect detection and comprehension in inspection, and finally highlights the amount of comprehension involved in the more popular defect detection methods used in the preparation stage.

2.1 Fagan's Inspection Process

Figure 1 shows a representation of Fagan's original inspection process [14]. The first step involves an Overview, which involves the whole team. This is where the inspection team is given a description of the overall project by the designer, and the appropriate documentation is circulated. In the specific case of program code inspection, code listings and design specification documents are circulated, and the overview description is optional. Also, the inspector's attention is drawn to areas that have been changed or reworked since the design inspections.

In the preparation stage, each member of the group works on their own and attempts to gain an understanding of the documents he/she has been provided with. In Fagan's original description of the preparation stage [14], an additional aid in the form of a checklist is available, which can be used to help increase an inspector's chances of finding problems. In Fagan's inspection, defect detection is a by-product of the understanding process. In other inspection processes, such as the one suggested by Gilb and Graham [18], the focus of the preparation stage is on defect detection. Since Fagan's original description of inspections several other defect detection methods have been suggested, including scenarios [50]. These have been proposed in an attempt to find more errors during the preparation stage, following dissatisfaction with checklists.

The inspection stage is where all members of the team get together and systematically go through the given documentation. If a problem is discovered, it is duly noted, but no attempt is made to fix it. Out of the inspection stage comes a document that contains notes on all the issues raised.

The next stage is rework, where all the errors in the inspection report are investigated by the designer or coder.

The final follow-up stage is used to check that all problems that were raised in the inspection stage have been dealt with. Fagan states that errors that are missed by the follow-up stage can be 10 to 100 times more expensive to solve if discovered further on in the process. It is suggested that if five percent or more of the material has been reworked, then 100 percent re-inspection should be carried out using the same inspection team.

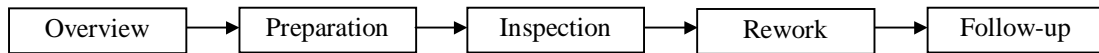


Figure 1: Inspection process created by Fagan [13].

2.2 Comprehension and Defect Detection in Inspection

Fagan describes the preparation process for inspectors as "using the design documentation, literally to do their homework to try to understand the design, its intent and logic" [14, pp. 193]. The objective is not to find errors, but to gain an understanding of the documents provided. The following descriptions illustrate some of the methods currently used to achieve this.

In Fagan's original inspection process [14], the main focus of the preparation phase was to understand the documents presented; defect detection was a by-product of this process. Fagan suggested the use of checklists to help inspectors.

In Gilb and Graham's version of software inspection [18], the emphasis in the individual inspection phase (preparation stage) is towards defect detection. They suggest that an inspector should be free to use any checking method that they feel gives results, but their book also highlights the use of checklists for defect detection.

Strauss and Ebenau [67] believe that during the preparation stage, the inspector should place a higher priority on understanding the work product and that the finding of errors is of secondary concern. Here, as with Gilb and Graham, Strauss and Ebenau suggest that any tools and techniques that can help in defect detection, and are reasonable in their demands on the available preparation time, can be used.

Porter *et al.* [51] suggest in their review of software inspections, that the preparation process focuses on the use of defect detection methods. The defect detection techniques used can vary from intuitive, non-systematic techniques such as ad-hoc or checklist, to what Porter *et al.* describe as a well-defined and highly systematic technique, such as scenarios [50]. Porter states that the most frequently used detection methods in inspection are ad-hoc and checklists. Porter's view of the preparation process is that an inspectors responsibility is to either find as many defects as possible, or focus on a limited set of issues, e.g. appropriate use of hardware interfaces, checking conformance to coding standards, etc.

Marick [39] suggests that the procedure when carrying out an inspection is to have the source code on one side, and a checklist on the other, and then step through the checklist.

From these examples it can be noted that the comprehension and defect detection methods currently used for code inspection vary. In some cases no set method is suggested, leaving the choice up to the individual inspector, e.g. Gilb & Graham and Strauss & Ebenau. In other cases the only methods used in the preparation process are defect detection methods, e.g. checklists, scenarios, which would appear to encourage varying degrees of comprehension.

The next section takes a closer look at the defect detection methods highlighted above, and looks at the levels of comprehension advocated by these methods.

2.3 Ad-hoc Detection, Checklists, and Scenarios

Three defect detection methods currently used during the preparation stage of inspection are ad-hoc, checklist and scenarios. What is not currently known or being investigated is the levels of comprehension afforded by these defect detection methods.

Ad-hoc defect detection is one of the more popular methods [51]. In ad-hoc detection, the inspector uses a non-systematic technique to find defects, e.g. the inspectors are left to their own devices. There is no evidence or reports on the amount of comprehension achieved using ad-hoc methods.

Perhaps the most popular detection method is the checklist [51]. Two of the characteristics of a checklist defined by Gilb and Graham [18] are:

- checklists should be kept updated to reflect experience of frequent defects
- a checklist should concentrate on questions which will turn up major defects

The first point highlights that checklists change over time, to keep up-to-date with the current list of most frequent defects found in previous inspections. The second point emphasises the use of checklists, i.e. delivering a series of questions that have the highest probability of finding major defects. Both of these points highlight the fact that checklists are used to emphasise certain areas of code that have a higher probability of containing errors. This may preclude an inspector gaining a complete understanding of all the code to be inspected. Shown in **Table 1** is an extract from a C++ checklist from Humphrey [23].

Complete	Verify that the code covers all the design.
Strings	Check that all strings are <ul style="list-style-type: none">• identified by pointers and• terminated in NULL
{ } Pairs	Ensure the { } are proper and matched
Calls	Check function call formats: <ul style="list-style-type: none">• Pointers• Parameters• Use of '&'
File Open and Close	Verify that all files are <ul style="list-style-type: none">• Properly declared,• opened and,• closed

Table 1. C++ Checklist

As already highlighted, most of the entries in **Table 1** emphasise certain parts of the code to look at, very few help guide the inspector towards comprehension of the code.

The scenario approach developed by Porter *et al.* [50], was created to address a perceived lack of effectiveness in the use of ad-hoc and checklist defect detection methods. Porter describes a scenario as a "collection of procedures that operationalize strategies for detecting particular classes of defects. Each reviewer executes a single scenario, so multiple reviewers are needed to achieve broad coverage of the

document" [50]. Porters experiments with scenarios have so far been limited to Software Requirement Specifications. **Figure 2** shows one of the scenarios proposed by Porter and Votta [50].

- | |
|--|
| <p>C.2 Incorrect Functionality Scenario</p> <ol style="list-style-type: none">1. For each functional requirement identify all input/output data objects:<ol style="list-style-type: none">(a) Are all values written to each output data object consistent with its intended function?(b) Identify at least one function that uses each output data object.2. For each functional requirement identify all specified system events:<ol style="list-style-type: none">(a) Is the specification of these events consistent with their intended interpretation?3. Develop an invariant for each system mode (i.e. Under what conditions must the system exit or remain in a given mode)?<ol style="list-style-type: none">(a) Can the system's initial conditions fail to satisfy the initial mode's invariant?(b) Identify a sequence of events that allows the system to enter a mode without satisfying the mode's invariant.(c) Identify a sequence of events that allows the system to enter a mode, but never leave (deadlock). |
|--|

Figure 2. Scenario created by Porter and Votta [50]

Scenarios, as with checklists, contain questions, which direct the inspector towards specific code segments. In an attempt to improve on checklists, scenarios also contain questions, which would seem to require deeper understanding. While these might improve an inspector's comprehension level, they are still restrictive to certain areas of functionality. To overcome this, multiple scenarios are used, one for each inspector, in an attempt to gain the widest coverage. However, as with ad-hoc and checklist detection methods, scenarios perhaps do not encourage full program comprehension, only comprehension of specific program parts.

There is currently a small amount of empirical evidence that shows scenarios are an improvement over checklists in terms of defect detection [50], [43] but there is no empirical evidence to compare the amount of comprehension afforded by ad-hoc detection, checklists or scenarios.

More recently, a variation on scenarios has appeared, Perspective-Based Reading [4]. They are similar to scenarios and were also originally designed for Software Requirement Specifications. More recently [31] a version of perspective-based reading techniques for code has been proposed, but no conclusive evidence to their effectiveness for code inspections as compared with other current methods has so far been produced.

2.4 Summary

The inspection process has been around since the early 70's, originally created by Fagan [14], and updated by him and others. Many reports have shown the advantages of using inspection [3], [15], with an increase in the number of defects detected.

In Fagan's inspection process, the preparation stage is used to prepare for the group inspection. Each inspector attempts to fully understand the documents they have been given. Several inspection process guides [18], [67], leave the choice of how to carry out the preparation stage to the inspector, others suggest using checklists and scenarios. The level of comprehension that is offered by both these methods in current inspections is relatively unknown as there is little or no empirical evidence. These methods do however tend to focus an inspectors attention towards certain areas of code, suggesting that full comprehension is not obtained.

3. Program Comprehension

Within the software maintenance process, anywhere up to 60% of the total time is spent on program comprehension [9]. Due to this large amount of effort required there has been a sizeable amount of research carried out in this area [6, 33, 41, 55, 60]. Program comprehension is a technique used by software engineers when they are attempting to understand a program. From this and other research, several cognitive models have been described for program comprehension. Researchers agree however that there is no one cognitive model that explains the behaviour of all programmers during program comprehension, and that in many cases programmers swap between models, depending on the particular problem.

The task of looking at a segment of code and being able to understand it is a common feature to both software maintenance and inspections. This opens up the possibility that some of the research and tools available for software maintenance could also be used in inspections. The remainder of this section describes some of the cognitive strategies discovered, and briefly describes a set of comprehension criteria for program comprehension tools.

3.1 Cognitive Models for Program Comprehension

The following is a summary of some of the better known cognitive models. For more detailed information on the cognitive models refer to [6], [33], [41], [55], [60], [75].

3.1.1 Bottom-up Comprehension

The bottom-up model for program comprehension groups together small chunks of the source code to build up higher levels of abstraction. These abstractions are then grouped together to eventually produce a high level design of the code [60]. This technique is followed if the code being looked at is totally new to the programmer.

3.1.2 Top-down Comprehension

The top-down model for program comprehension is usually used when the code under consideration is familiar [63]. Brooks [6] describes the top-down model as reconstructing knowledge about the application domain and mapping this to the

source code. To achieve this, a global hypothesis is defined, describing the program. The reader then tries to verify this hypothesis. This in turn can cause further hypothesis to be created, building up a hierarchy of hypothesis to be verified. This continues until a level is reached where the hypothesis can be verified or proven to be false.

3.1.3 Systematic and as-needed Comprehension

Littman *et al.* [33] describes two comprehension strategies. Systematic is where a programmer systematically reads through code in detail, looking at both the control-flow and data-flow abstractions, to obtain a thorough understanding of the code. The other strategy is where the programmer only looks at the code related to a particular problem or task. Parts of the code are looked at only when the programmer needs to understand them.

The description of as-needed comprehension could be thought of as describing both the checklist and scenario defect detection methods highlighted in the previous section. Young [75] states that the use of as-needed comprehension can lead to a lack of understanding of the causal relationships and dependencies between program components. This is especially problematic considering the increase in the number of dependencies found in object-oriented code (discussed in Section 5) compared to procedural code.

3.1.4 Integrated Comprehension

Von Mayrhauser and Vans [41] integrated the top-down, bottom-up and knowledge-based strategies. From experiments carried out, they noted that a programmer switches between comprehension models. When the code is familiar, the integrated strategy utilises the top-down model, and when the code is unknown, the bottom-up strategy is used. As an example, when reading a program in the top-down manner, the reader may come across an unknown section of code, at which point the reader swaps to the bottom-up cognitive strategy to determine what this new section does.

3.2 Program Comprehension Tools

With the large program comprehension overhead involved in software maintenance, software engineers are constantly trying to find new ways to reduce the burden of work and reduce the number of mistakes made due to human fallibility. With the power of computers increasing rapidly in the last two decades, it has been possible to create tools to help in the program comprehension process. There are many program comprehension tools currently available [53 presents a bibliography of some program comprehension tools].

Program comprehension tools generally implement a reverse engineering process. Reverse engineering involves analysing a system to identify its components and interrelationships and create representations of the system in another form or at a higher level of abstraction [11]. Many of the recent comprehension tools have attempted to help the software engineer by containing various visualisations.

Tilley *et al.* [69] identified three basic activities involved in the reverse engineering process:

- Data gathering - either through static or dynamic analysis
- Knowledge organisation - creating abstractions for efficient storage and retrieval of the raw data
- Information exploration - through navigation, analysis, and presentation.

Storey *et al.* [66] highlights the important point that it is very unlikely that one program comprehension tool will be able to represent all the activities in the different cognitive models suggested by comprehension literature. A look at some of the tools currently available (see Section 6) would seem to confirm this.

Although there are tools that have been specifically created for the program comprehension process, there are tools that have been created for other purposes, e.g. visualisation that could also be used as comprehension tools. Linos [32] describes a criteria list, in which if one or more points are satisfied, the tool can then be considered a program comprehension tool.

1. Tools that support one or more known mental models for program comprehension (e.g. bottom-up, top-down, etc.)
2. Tools that maintain a repository of architectural or behavioural information about a program
3. Tools that provide a presentation model for visualising information about programs in various ways (i.e. code, graph)
4. Tools that provide transformations for going from one kind of representation to another (i.e. text to graph)
5. Tools that provide partitioning techniques for decomposing large displays into smaller manageable pieces
6. Tools that provide abstraction mechanisms to remove unwanted details from large displays
7. Tools that maintain consistent and up-to-date source-code documentation

3.3 Summary

In the last two decades, much research has gone into discovering how programmers go about the process of understanding code. Several cognitive models have been put forward, but more recent evidence tends to support the theory that programmers do not stick to one comprehension strategy, but move between several, depending on the situation.

The checklist and scenario defect detection methods used in inspection would seem to fit under the heading of as-needed comprehension. This strategy can however lead to a lack of understanding of the dependencies within the code being looked at, especially a problem if the code is object-oriented.

There are many comprehension tools available, many of which have been created to help reverse engineer code in the software maintenance process. Many of the more recent tools have contained some visualisation aspect, but as Storey *et al.* [66] points out, it is unlikely that one tool will be able to represent all the activities in the different cognitive models suggested by comprehension literature. The criteria list

put forward by Linos [32] suggests that many tools that are already available for other areas, including code visualisation, can be used as aids to comprehension.

4. Visualisation

As highlighted in the previous section, many of the current comprehension tools that are appearing have some form of visual element. With the increase in the number of dependencies presented by the object-oriented paradigm (discussed in Section 5), some way will have to be found to help show them. One solution to this problem could be visualisation. The remainder of this section briefly discusses visualisation, current areas of research, and highlights why it is a useful technique for aiding comprehension.

4.1 What is visualisation?

Gershon *et al.* [17] describes visualisation as an interface between two powerful information processing systems, the human mind and the modern computer. Visualisation involves manipulating information, data and knowledge and converting this into a visual representation, which utilises the human visual system. Watson [70] describes visualisation as the process of creating and manipulating a visual image, which allows a mental picture of some situation or phenomenon to be formed.

Within the field of software engineering there are two visualisation areas that are sometimes confused with each other, visual programming and program visualisation. Visual programming relates to any system that allows the user to specify a program in a two (or more) dimensional fashion [56]. With program visualisation, the program is specified as normal, and some form of graphical representation is used to show some aspect of the code or its run-time execution [56]. In this paper we are concerned with program visualisation.

4.2 What is the purpose of program visualisation?

Baecker, one of the founding fathers of visualisation describes program visualisation as "the use of graphics to enhance the art of program presentation and thereby to facilitate the visualisation, understanding, and effective use of computer programs by people" [2]. Roman and Cox [56] describe the purpose of program visualisation as extracting information about some aspect of a program and presenting it in a graphical form. Petre *et al.* [49] states the purpose of program visualisation as trying to find simplicity in a complex artefact (e.g. thousand-line code), to produce a selective representation of a complex abstraction. This is not however as easy as it sounds. There is no easy, straightforward way to generate visualisations for program code, which are abstract visualisations, not physical, unlike visualisations for mathematical functions.

Gershon *et al.* [17] believes that with effective visualisations, it is possible to interact with large amounts of data in a fast and effective manner, and be able to find hidden characteristics, patterns and trends.

Some of the earliest work in visualisation was carried out by Ronald Baecker [1] in 1968. It wasn't until the late 70's, early 80's that advances in technology allowed a gradual increase in the complexity of graphical visualisations. One of the simplest forms of program visualisation that appeared early on was pretty-printing. This aimed to improve the readability of code by improving the code layout, and utilising different fonts, font sizes and shading.

With the rapid development of new technology there has been an explosion in the number of visualisation programs being developed ([24] lists some of the visualisation programs from the 80's and [13] lists some visualisation programs from the 90's). With the continued advances in technology, visualisations have increased in complexity and graphical diversity.

Until 1986, there was no firm taxonomy for program visualisation. Myers [44] introduced the original taxonomy for program visualisation. Myers defined two types of program visualisation, code and data. Each of these types was further subdivided into static and dynamic visualisations. Static visualisations are based on information that can be obtained from the code alone, whereas dynamic visualisations are based on information gathered from an executing program. Myers later added a third type of visualisation to go beside code and data - algorithm [45]. Several other people have attempted to expand the visualisation taxonomy, including Brown [7], Stasko and Patterson [64], and the most recent taxonomy created by Price, Baecker and Small [52].

4.3 Current Visualisation Research

Current visualisation research is very diverse and covers a wide spectrum of topics. The following briefly describes some of the more popular areas.

Program maintenance relies heavily on program comprehension, as many maintainers do not trust or rely on the documentation that is provided, and is rarely complete. Storey *et al.* [65] designed the SHriMP visualisation technique to allow seamless exploration of source code and program structure. The SHriMP technique was designed to allow the use of hybrid program comprehension, since many users of program comprehension techniques rarely stick to the one method. Oman [48] describes several maintenance tools, and highlights that all the tools looked at are code-visualisation tools, hinting that program visualisation could be more beneficial in the maintenance area, than it has been in both the analysis and design areas.

In recent years there has been an increase in the number of parallel and distributed systems. To help with program comprehension and debugging activities, several people have attempted to design visualisation environments. Sharnowski and Cheng [59] have developed the GOLD environment, which is a graphically based debugger for C. It incorporates the top-down debugging methodology, and graphically supports the setting of breakpoints within the code. Hart *et al.* [21] designed a query-based visualisation to help novices understand the workings of a distributed program.

A continuing research problem is the discovery of new visual metaphors for representing information [17] and in the last few years the World Wide Web has emerged as a new area for visualisation research. With the vast amount of information present on the web (millions of pages), and its continued growth, visualisation will have an opportunity to play a more important role in web navigation.

4.4 Why is visualisation being used as a technique to aid program comprehension?

Storey *et al.* [66] suggests that graphical representations are generally accepted as useful comprehension aids. Young [75] believes that software visualisation can aid program comprehension. He comments that software visualisation attempts to aid the process of comprehension by providing visual displays for abstract chunks of textual structures, thus reducing the interpretation load [75]. Oman [48] in his review of maintenance tools, which are angled towards aiding program comprehension, found that all the recent tools were code-visualisation tools.

Young [75] describes the human brain as more suited to being able to process, manipulate and recognise visual images and structures. Roman and Cox [56] put forward a number of reasons for the use of visualisation. These include the important role that imagery plays in human communication in general, the extraordinary high bandwidth of the human vision system, the speed with which humans track and detect visual patterns, and the power of abstraction inherent in pictorial representations. Petre *et al.* [49] suggests several reasons for why graphics are preferred to text. These include more comprehensible, more memorable, more accessible, faster to grasp, and more fun.

Another reason to use visualisation to aid comprehension is as time goes on, programs are becoming larger and more complex. Gershon *et al.* [17] points out that with effective visual interfaces we can interact with large volumes of data in a fast and effective manner, in an attempt to discover hidden characteristics, patterns and trends.

Storey highlights that the majority of the time spent on the maintenance, debugging and code reuse processes is spent on understanding existing programs [66]. If this is the case, then any form of help or extra support for the program comprehension process could be very beneficial.

Petre *et al.* [49] believe that while there are currently many people who still prefer to use paper copies of code when carrying out comprehension and debugging, this will change with the increasing popularity of object-oriented languages which are non-linear, and therefore make the code more difficult to follow.

4.5 Summary

Visualisation tries to link together the image processing systems of the human mind and the modern computer [17]. Program visualisation involves representing the code in a graphical form in an attempt to show some particular aspect. With recent improvements in technology, visualisations have been growing more graphical and complex.

Myers [44] separated visualisations into two areas, code and data. Since we are interested in code inspections, we have looked more closely at code visualisations. These were further sub-divided into static and dynamic visualisations. Current research areas for visualisation include program maintenance and distributed and parallel computing.

With programs growing larger and more complex and with effective visualisations able to interact with large volumes of data [17], more and more tools being developed for comprehension have a series of visualisation aids. Many have discovered that visualisation can help aid the process of program comprehension [75],

[66]. As we will see in the next section, visualisation could be used to help alleviate the dependency problem when attempting to comprehend object-oriented code during inspections.

5. Object-Oriented Code Inspection

The inspection process by Fagan [14], proven to be a cost-effective technique for finding errors [57], [15] was created before the object-oriented paradigm became remotely popular.

In 1994, Jones [28] listed some of the gaps in information about the object-oriented paradigm. One of those gaps was in the area of inspection. Jones noted that "Since formal inspections are the most effective known way of eliminating software defects, software quality assurance personnel are anxiously awaiting some kind of guidance and quantitative data on the use of inspections with object-oriented projects" [28]. A search through the literature has found that since 1994, few papers have been published addressing this concern.

One paper that has mentioned inspecting object-oriented code was by Wilde *et al.* [74]. In this paper, Wilde focused on the job of a maintainer. A maintainer has to be able to understand the program's structure and behaviour. Wilde states that they "decided to focus on the problems of a maintainer trying to understand object-oriented software by reading and statically analyzing it, as in code inspections". Wilde highlighted problems with inheritance, method size, dynamic binding and polymorphism, and class hierarchy, but failed to describe how he went about inspecting their object-oriented code.

As highlighted previously, two techniques currently used in the preparation stage of inspection are checklists and scenarios, which enforce the as-needed comprehension strategy. This strategy can lead to a lack of understanding of some dependencies within a code segment. This, as is shown in the remainder of the section, is a problem which could have serious implications where the inspection of object-oriented code is concerned.

What follows is a look at some of the features of the object-oriented paradigm, including those found problematic by Wilde *et al.* [74], showing ways in which they complicate the comprehension and inspection processes.

5.1 Object-Oriented Problems

The object-oriented paradigm has gained widespread acceptance [10] and has created many benefits for the programmer such as producing better structured and more reliable software for complex systems, greater reusability, more extensibility, and easy maintainability [29]. With these successes, there have also arisen new problems to be tackled.

According to Gamma *et al.* [16], the structure of an object-oriented program at run-time is vastly different to that of its code structure. They claim that the two structures are largely independent. Where the code structure is frozen at compile-time, the run-time structure consists of rapidly changing networks of communicating objects. This makes it very difficult to understand one from the other.

Dependencies exist in all code, but their number are increased in object-oriented languages [10], [73]. A dependency exists where if two entities in a system,

say X and Y are related, then if X is modified, this could have possible repercussions on Y.

Dynamic binding involves not knowing the class of a particular object assigned to a variable, as this is only determined at run time [73], [36], [5], [74]. When a method invocation occurs, only at run time can an object's class be correctly identified. This also increases the complexity of the dependencies in a program.

Polymorphism involves naming consistency [73], [36], [5], [74]. The idea behind polymorphism is that different objects have a method with the same name, and if that method is invoked, the same basic action should occur, no matter the object. If method behaviour is not consistent throughout the program, then comprehension problems may arise.

Booch [5] describes inheritance as "a relationship among classes, wherein one class shares the structure or behaviour defined in one (single inheritance) or more (multiple inheritance) other classes". The complexity of the program comprehension process increases as some classes cannot be fully understood in isolation, and have to be looked at in conjunction with others.

The structural style of object-oriented programs differs from that of conventional programming languages [73]. Method sizes may be very small [74], and this could lead to a wide dispersal of code to perform a given task. This means traversing up and down the object inheritance hierarchy in an attempt to locate where the work is carried out.

A generic class acts as a template and can be instantiated by other classes, objects, etc. [36], [5], mainly acting as a container class. A simplistic example of a generic class is that of an array. An array can be defined to contain elements of any one class. Any subclasses of the defining class can also be placed into the array. This leads to the same problem that occurred with dynamic binding. Only at run time can the contents of a generic class be exactly determined.

The following sections take a closer look at each of the previously mentioned problem areas of the object-oriented paradigm, and also highlight what additional problems they cause for inspection and comprehension.

5.2 Dependencies

Wilde and Huitt [73] describe a dependency in a software system as:

A direct relationship between two entities in the system $X \rightarrow Y$ such that a programmer modifying X must be concerned about possible side effects in Y.

Wilde and Huitt [73] also point out that using polymorphism and hierarchy dramatically increases the kinds of dependencies that need to be considered. Some of the dependencies they highlight include Class-to-Class, Class-to-Methods, Class-to-Message, Class-to-Variable, Method-to-Variable, Method-to-Message, and Method-to-Method. Chen *et al.* [10] highlight three kinds of object-oriented specific dependencies. These are message dependence, class dependence and declaration dependence.

5.2.1 Message Dependence

Method (or function) A is said to be message dependent on method (or function) B if B invokes A [10].

Shown in **Figure 3** is an example of message dependence (Table 1 from [10]). Here we have a template class called `Dlist`. In the `read` method, calls are made to the `read` belonging to the class `Elem` and the `insert` method, which belongs to the class `Dlist`. Both these methods are called by the `Dlist::read` method and are therefore message dependent on the `Dlist::read` method.

```
Template<class Elem>
int Dlist<Elem>::read (FILE *fp)
{
    delall();
    Elem el;
    while ( el.read (fp) > 0) {
        insert(el);
    }
    return count;
}
```

Figure 3. Message dependence in method `Dlist::read`

5.2.2 Class Dependence

Class A is said to be class dependent on class B if one the following properties holds: 1) Class A is a derived class of B, or 2) an instance of class A contains one or more instances of Class B, or 3) an instance of class A has a reference or a pointer to an instance of class B, or 4) class A is a friend¹ class of class B [10].

Shown in **Figure 4** is an example showing class dependence (Figure 1 from [10]). The classes represent a banking application. The classes `Accountdb` and `Transactiondb` inherit from `Recdb`. The class `Accountdb` can have multiple instances of `Transactiondb`. `Accountdb` and `Transactiondb` inherit from `Recdb`, which satisfies property 1. Property 2 is satisfied with the `Accountdb` class containing multiple instances of the `Transactiondb` class.

¹ C++ has an extra keyword `friend`, which allows a class to delegate a list of classes that can access its `private` and `protected` declarations, circumventing the object-oriented ideals of encapsulation, making it harder to inspect classes in isolation.

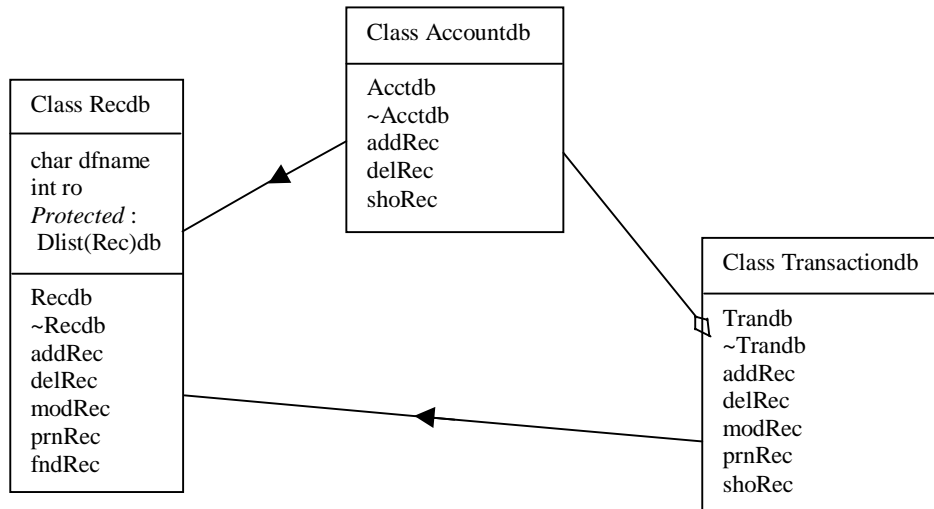


Figure 4. Class dependency

5.2.3 Declaration Dependence

An object (or a variable) *O* is declaration dependent on the class *C* (or type) to which *O* belongs [10].

Figure 5 shows an example of declaration dependence (from Example 3 in [10]). This code shows a declaration dependence between object *a* and class *A*. This is required to be able to identify the method called on execution of the `a.print()` statement.

```

class Base {
    virtual print () = 0;
}
class A : Base {
    void print ();
}
class B : Base {
    void print ();
}

A a;
a.print();
  
```

Figure 5. Declaration dependence example

5.3 Polymorphism and Dynamic Binding

Booch's [5, pp. 513] description of dynamic binding is:

Binding denotes the association of a name (such as a variable declaration) with a class, dynamic binding is a binding in which the name/class association is not made until the object designated by the name is created at execution time.

Another description of dynamic binding states that when a message is sent to a variable holding an object, the actual method implementation that will be called depends on the object's class [73]. Wilde and Huitt [73] believe that this complicates the tracing of dependencies, with different implementations establishing different dependencies. This implies that static analysis of code will not be enough to be able to identify all dependencies. A simple description of polymorphism is that there are a number of different classes of objects that will respond to the same request. A more elaborate description of polymorphism is that of Booch [5, pp.517]:

A concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways.

Figure 6 (based on Figure 5 from [36], [30]) is a small example to demonstrate dynamic binding and polymorphism. In **Figure 6**, we have a class `Object_X` which has one method, `process()`, that displays the message "In class X". There is also `Class_Y`, which inherits from `Object_X`. In this small example, `Object_Y` redefines the method `process()`, and displays the message "In class Y".

The code executes through the main method in the class `Test`. Firstly, two instances are created, one of each class `Object_X` and `Object_Y`. The `process` method for the object assigned to the `local_x` variable is called and this would display "In class X" on the screen. Next, the object in the variable `local_y` is assigned to the `local_x` variable. This is possible because the variable `local_y` is of type `Object_Y`, which is a subclass of `Object_X`. This is polymorphism at work. Finally, the `process` method for the object in the `local_x` variable is called again, but this time because the `local_x` variable actually contains an instance of the `Object_Y` class, it is the `Object_Y` `process` method that is actually called. This is dynamic binding at work.

This example, although simplistic in nature, highlights some of the problems with dynamic binding and polymorphism. The first problem is identifying the class type of the object held by the variable `local_x`. This variable was defined to be of type `Object_X`, meaning that it can hold any instances of `Object_X`, or any of its subclasses, e.g. `Object_Y`. In the **Figure 6** example, it would be possible to statically determine what the class type of `local_x` would be at certain points in the program, but this is due to the simplistic nature of the example. In larger programs, this would be much more difficult, and would have to be determined dynamically at run time. Another problem shown is consistent method behaviour. Both the `Object_X` class and its subclass `Object_Y` have a method with the same name. The names are the same because the `process` method in `Object_Y` carries out the same task as the `process` method in `Object_X`, but carries it out in a specific way for the `Object_Y` class. In a more sophisticated program, the code in these two methods could be more complicated, but they should both carry out a similar process. Wilde points out that if method behaviour is not kept consistent with same name methods, then whoever is reading the code can be led into subtle errors [74].

<pre>class Object_X { public void process() { System.out.println("In class X."); } }</pre>	<pre>class Object_Y extends Object_X { public void process() { System.out.println("In class Y."); } }</pre>
<pre>class Test { public static void main(String[] args) { Object_X local_x = new Object_X(); // local_x refers to an object of class Object_X Object_Y local_y = new Object_Y(); // local_y refers to an object of class Object_Y local_x.process(); // local_x.process() is called local_x=local_y; // local_x now refers to an object of class Object_Y local_x.process(); // Y.process() is called } // End main } // End class Test</pre>	

Figure 6. Example Java code for Polymorphism

5.4 Inheritance

Booch [5, pp. 514] describes inheritance as:

A relationship among classes, wherein one class shares the structure or behaviour defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines an "is-a" hierarchy among classes in which a subclass inherits from one or more generalized superclass; a subclass typically specializes its superclasses by augmenting or redefining existing structure and behaviour.

Inheritance is one of the major features of object-oriented programming. The behaviour of some classes cannot be fully understood by looking at a single class alone, as many classes inherit method and instance definitions from other classes (superclasses). An example of this is shown in **Figure 7**. Here we have `Person` class, our superclass, and `Patient` class, the subclass that inherits methods and definitions from `Person` class.

The `Patient` class is said to extend (inherit) features from the `Person` class. This gives the `Patient` class access to those variables and methods that have been made available to any subclasses. The `Patient` class also contains two constructor methods, one of which requires not only the patient information, i.e. the home address and condition, but also the forename and surname of the patient. `Patient`'s superclass constructor (`Person`), invoked by using the command `super`, requires the forename and surname to be able to create a person.

<pre> class Person { protected String surname; protected String forename; public Person() { } public Person(String s,String f) { } public String person_display() { return surname + " " + forename; } } </pre>	<pre> class Patient extends Person { protected String home_address; protected String condition; public Patient() { super(); } public Patient(String s,String f,String ha,String con) { super(s,f); } public String patient_display() { return home_address + " " + condition; } } </pre>
<pre> // Driver code (not necessarily needed in final document) class Driver { public static void main(String args[] args) { Person person_one = new Person("John","Smith"); Patient person_two = new Patient ("Harry","Bosch","4 Pensilvania Avenue", "Gun-shot wound"); System.out.println(person_one.person_display()); System.out.println(person_two.patient_display()); System.out.println(person_two.person_display()); } } </pre>	

Figure 7. Example Java code for Single Inheritance

The `super` command allows Java to call the constructor method of the superclass (or any other method that has the same name in both a class and its superclass). This requires that to fully understand a method, you have to look at the code of the superclass. It could also be the case that the superclass could use the `super` command and you would have to look up its superclass. This knock on effect means that to be able to understand one class fully, you may have to go and look through several other classes.

In the `Driver` program, `person_two` has been created as a patient. At the end of the driver program, the last two `System.out.println` commands are used to place on the screen the patients information. Calls are made to the methods `patient_display()` and `person_display()` to get the necessary information. When attempting to find the definition of `patient_display`, you would logically begin your search in the `patient` class. In this instance, the `patient_display` method is found in the `patient` class, but if you also looked here for the `person_display` method for the patient you would not find it. In this case you would have to search through the inheritance hierarchy, and look in each superclass until you find it.

The example shown in **Figure 7** uses single inheritance. This feature can be found in all object-oriented languages, i.e. Java, C++, Smalltalk, Python, Eiffel, etc. In some of these object-oriented languages, as well as having single inheritance, they have multiple inheritance. Multiple inheritance is where a subclass inherits features

from two or more classes. This feature can be found in such languages such as C++, Eiffel, and to a small degree in Smalltalk. One example of multiple inheritance is shown in **Figure 8** [36, Figure 3].

The example in **Figure 8** shows two classes `parent1` and `parent2`, and a third class `child` which inherits both from `parent1` and `parent2` (multiple inheritance). The problem with multiple inheritance is locating where a particular method resides. For example, if an instance of the `child` class was created, say `Child_1`, and if we made the call `Child_1.function1()`, you would first look in the `child` class to locate the method, but you would find it wasn't there. Next, you would have to search each of the child's superclasses, `parent1` and `parent2`, to try and find the definition of the method. What if one of the superclasses, e.g. `parent1` also had several superclasses of its own. This can very quickly grow into a large searching problem, and slow down any comprehension process being carried out.

As well as the problems already mentioned, there are also some language specific inheritance problems. In C++ the `friend` keyword gives functions full access to `private` data of a class. The function only has to be declared as a `friend` of a class. The `friend` keyword can hinder inspection by removing much of the logical structure of the system [36]. The Eiffel language contains several complicating features, one of which is repeated inheritance. Repeated inheritance involves inheriting a method once to redefine it, which allows use of the method name, and then a second time to rename it, which allows the use of the original method, but with a different name. Macdonald [36] suggests that when inspecting object-oriented code, which uses repeated inheritance, it becomes very easy to lose track of which bits of code are actually being used.

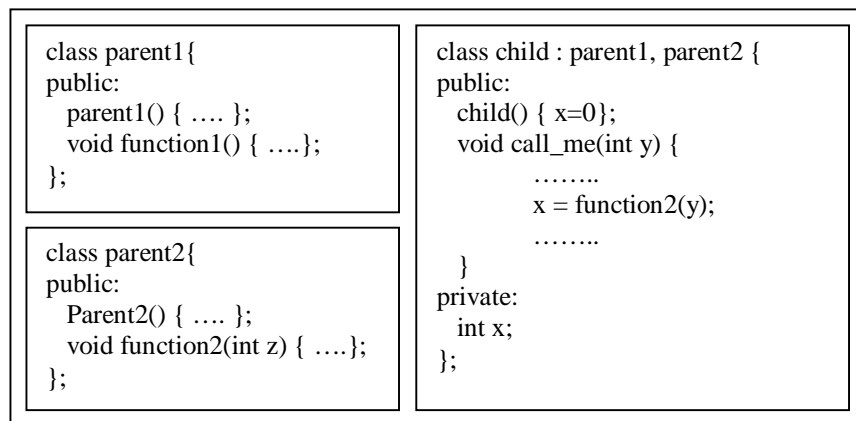


Figure 8. Example C++ code of Multiple Inheritance

Many object-oriented languages also have keywords which define visibility, e.g. `public`, `private`, and `protected`. These limit the access to variables, classes and their methods. **Figure 9** shows an example using visibility keywords (from [36], Figure 4). Here we have two classes, `Class A` which has a `private` integer variable `a`, and `Class B`, which inherits from `Class A`, and has a method called `b`. The `b` method in `Class B` attempts to give the variable `a` a value, but it cannot, because although `Class B` inherits `Class A`, `Class A` has defined the integer variable `a` as being `private`. This means that `a` is available only within `Class A`,

and cannot be accessed by any subclass. Macdonald [36] highlights that with the ability to declare classes as `public`, `protected` or `private`, the visibility rules increase in complexity.

```

class A {
private:
    int a;
};

class B : public A {
public:
    void b() { a = 1 }; // Illegal - B has no access to a
};

```

Figure 9. Visibility restriction in C++ code

Finally, in a study by Daly *et al.* [12] it was found that class structures with high levels of inheritance increase the difficulty of maintenance compared to their corresponding class structures without inheritance.

5.5 Class Structure and Object Hierarchy

The structure style of object-oriented programs is thought to be different from that of traditional languages. Wilde *et al.* [72] found that in object-oriented languages, the size of methods is generally small, but there are a large number of them. The data that he collected is shown in **Table 2** based on results from [72].

System and Domain	Language	Number of Methods	Median Method Size
Bellcore: Interactive Network design aid	C++	1280	1 executable statement
Bellcore: Prototype of noninteractive task planning system	Smalltalk	477	2 noncomment lines
Smalltalk/V: Sample from the environment's class library	Smalltalk	2224	3 noncomment lines

Table 2. Sizes of methods from three object-oriented systems

Wilde *et al.* [74] believe that small methods are not a sign of bad design, but rather a natural consequence of good object-oriented design. Wilde *et al.* [74] also point out that with many small methods, it becomes relatively straight forward to be able to understand each small code segment. This however, does not help the process of being able to understand the overall system behaviour. To be able to fully understand a method, its context of use must be found by following back the chain of calls that reach it, and the effects of the method would have to be followed by looking at all the methods called.


```

template<class Item>
class Queue {
public:

    Queue();
    Queue(const Queue<Item>&);
    virtual ~Queue();

    virtual Queue<Item>& operator=(const Queue<Item>&);
    virtual int operator==(const Queue<Item>&) const;
    int operator!=(const Queue<Item>&) const;

    virtual void clear();
    virtual void append(const Item&);
    virtual void pop();
    virtual void remove(int at);

    virtual int length() const;
    virtual int isEmpty() const;
    virtual const Item& front() const;
    virtual int location(const void*);

protected:
    ...
};

```

Figure 11. Example C++ code for a Generic class

The difference between the two declaration statements is that in the first case, only classes of type `People` could be placed in the queue, whereas in the second declaration, the class `People`, and any of its subclasses could be placed in the queue. The second declaration style introduces the old problem of at compilation time it cannot be exactly stated what the contents of an instantiated generic class might be, i.e. it could be the instantiating class or any of its subclasses.

When inspecting generic code, Macdonald [36] suggests that because of possible inconsistent behaviour of instantiating classes, each one would have to be inspected with respect to the generic class. This problem can increase in size as more instantiating classes are added over time.

5.7 Summary

From experience with inspections over the years, it has been found that there is a limit to the number of lines of code that can be read during the preparation stage, beyond which the number of defects found per thousand lines of code decreases. Fagan [15] suggests that a maximum of two, two-hour inspection sessions are the most that should be carried out each day. He goes on to suggest that good inspections are tied to thoroughness, and that no more than 125 non-commentary source statements per hour are read. Weller [71] confirms this from results taken from over 400 inspections. He found that less than 200 lines of code should be read per hour during the preparation process to get the maximum number of defects per thousand lines of code. It is believed that if the process is rushed, or too much code is looked

at, then the effectiveness of the inspection process is greatly reduced [36], [71], [3]. These factors limit the amount of code that can be looked at during an inspection meeting.

The process of splitting code up for inspection is made more complicated with object-oriented code due to the various dependencies that exist. These include Class-to-Class, Class-to-Methods, Class-to-Message, Class-to-Variable, Method-to-Variable, Method-to-Message, and Method-to-Method [73]. All of these increase the amount of code that has to be looked at to be able to fully understand how a section of code works, and if it is functioning correctly.

Polymorphism and dynamic binding create the problem of not knowing the class of a particular object assigned to a variable until run time. Static analysis cannot provide an adequate list of all the dependencies involved.

Inheritance means that classes cannot be understood in isolation, and that to be able to fully understand them, the class(es) that are inherited would have to be looked at as well.

Most methods are very small in size, many only a line or two, making it difficult to be able to define the behaviour of a program. Because of so many small methods, to be able to understand how one line of code works in some cases, a trace has to be made through the object hierarchy, tracing messages until you reach the method where the work is done.

As with the problem of dynamic binding, the contents of a generic class cannot be exactly known as it is decided at run time. Therefore when inspecting a generic class, each instantiating class and any of its subclasses should also be inspected.

In each of these object-oriented specific features, they depend and refer to other classes. A more organised method of splitting object-oriented code has to be found, as an arbitrary split is not good enough to allow an inspector to be able to completely understand the code they are looking at. Macdonald [36] also highlights a similarity between inspection and testing, "although it is tempting to test a class in isolation, it must actually be tested in context of its parent classes because of the possibility of hidden interactions". Currently, few papers have been published which attempt to resolve this problem.

If, as this author has suggested previously, checklists and scenarios are forms of as-needed comprehension which could lead to a lack of understanding of all the dependencies involved, then perhaps other comprehension strategies could be utilised in place of the current defect detection methods to improve overall comprehension and defect detection, especially with object-oriented code.

6. Tools for Comprehension

This section takes a look at two different types of available tools, inspection and visualisation. The visualisation tools have been created for different object-oriented programming languages. Both the inspection and visualisation tools have not been directly created to support cognitive strategies for program comprehension but they may have features that can help with comprehension. Each tool will be compared to the criteria defined by Linos [32] (see Section 3.2) to see if they offer any comprehension facilities.

6.1 Inspection Tools

There are several tools that have been created for inspections. Macdonald [34] carried out a review of several inspection tools, including ICICLE, Scrutiny, Collaborative Software Inspection, InspeQ, and the Collaborative Software Review System. Since then Macdonald has introduced his own inspection tool ASSIST [38]. Another recent inspection tool WiP [20] has been created to exploit the Internet.

The following sections briefly describe some current inspection tools, and highlight any features they have which could be used to help in the comprehension of code being inspected.

6.1.1 ASSIST

ASSIST [38] (Asynchronous/Synchronous Software Inspection Support Tool) is an inspection tool designed to support any inspection process and allow inspection of any type of document. Facilities within ASSIST include defect finding aids, enhanced document representations, facilities for metric collection and analysis, and the provision of facilities for distributed inspection.

Most inspection tools are based on one specific inspection process method. To remove this obstacle, Macdonald [37] created a generic software inspection template, which would cater for all current inspection processes and be versatile enough to cope with any future processes. This generic template was converted into a process definition language, and was embedded into ASSIST [37].

ASSIST provides an on-line checklist, which can be marked off as each item on the list is covered, as well as a text browser that is used to view the product under inspection. The text browser can be used to highlight areas of the document and add annotations. The annotations describe defects that have been found in the document. A code command help system is available which gives a brief description of the code command selected.

As highlighted in section 3.1.3, checklists are very similar to the as-needed comprehension strategy. Since checklists support this method of comprehension, and ASSIST supports the on-line usage of checklists, ASSIST manages to pass one of the Linos criteria [32].

6.1.2 Scrutiny

Scrutiny [19], [8] is a general inspection support tool which can support distributed inspections. The inspection method used by Scrutiny is based on four phases. The first phase is called **Initiation**, where the inspection team is formed and the Moderator prepares the necessary documentation. Phase two, **Preparation**, involves the inspectors creating their annotations of the presented documents for inspection. Phase three, **Resolution**, is equivalent to Fagan's inspection meeting. The final phase, **Completion**, encompasses both the rework and follow-up stages of Fagan's inspection process.

Scrutiny supports only text documents, but has been designed to be an open tool, in the hope of integrating other tools at a later date. In the product window, which displays the document being inspected, areas of text can be highlighted and an annotation assigned to it. There is no comprehension support available in scrutiny,

and no support for checklists. Scrutiny has no features that fit under any of the comprehension criteria [32].

6.1.3 ICICLE

Intelligent Code Inspection in a C Language Environment (ICICLE) [58] is an intelligent inspection assistant for the inspection of C code. A major difference between this tool and others is that it tries to find common defects itself, in an attempt to help the inspector by removing the more obvious errors in the code. ICICLE achieves this through its own rule-based static debugging tool and the UNIX **lint** tool.

ICICLE supports a two-phase inspection, the individual inspection and the inspection meeting. Group meetings are held in the same room, no distributed facilities are provided. In the individual inspection stage, inspectors can produce comments for each line of code. A referencing system is supplied for variables and functions, allowing quick movement through code, e.g. selecting a variable name would move you to its point of declaration. A hypertext browser is also available, providing domain specific knowledge.

No facilities are provided for any form of comprehension or defect detection. As with the other inspection tools looked at, ICICLE fails the criteria [32] for a comprehension tool.

6.1.4 Collaborative Software Inspection

Collaborative Software Inspection (CSI) [40] is an on-line inspection environment to allow distributed inspections. All material is available on-line, and inspection products are created on-line. CSI supports the Yourdon [76] and Humphrey [22] style of inspection.

CSI was designed to be able to cope with four types of collaborative inspection meeting: 1) same time, same place, 2) same time, different place, 3) different time, same place, 4) different time, different place. CSI supports both synchronous activities, e.g. group meeting, and asynchronous activities, e.g. individual checking. CSI contains a browser that displays the material under inspection, but currently only supports text, and contains hyperlinks from the inspected material to a fault list, note pad, inspection summary, and action list.

CSI contains hyperlinks to allow easy navigation between different areas of the system, but currently contains no help for the preparation stage or for checklists or any other form of defect detection. None of the features in CSI fall into any of the comprehension criteria [32].

6.1.5 WiP

The WiP tool [20] is designed to support distributed inspections, attempting to solve the problem of having a scattered inspection team. WiP utilises the World Wide Web, and is designed to distribute the documents to be inspected, allow annotation of those documents, be able to search related documents, allow selection of a checklist, and gather inspection statistics.

For the preparation stage of inspection, users are given access to source documents and checklists, as well as informal information which could lead to a better understanding of the given documents. The WiP interface also contains hyperlinks to allow easy navigation through the documents.

Annotations that are made during the inspection are not made to the documents directly, to avoid multiplication of data and are instead kept separately and sent back to the main server.

WiP by the authors own admission was designed primarily as an investigation to the possibility of carrying out inspections over the World Wide Web, and not as a complete inspection tool. As with ASSIST, WiP is aimed at enforcing the rigours of the overall inspection process, and as with ASSIST has support for checklists, meaning that it does pass one of the comprehension criteria [32].

6.2 Visualisation Tools

6.2.1 EasyCODE(C++)

EasyCODE [61] is a PC based commercial windows package from Siemens AG Austria, which uses structured programming techniques to visually display programs. It is an improved version of an earlier program called XperCASE, from 1994. The demo version of the program that was looked at was for C/C++, but other versions are available for such languages as Pascal, Basic, Modula2, PL/SQL, and many more. The C/C++ version can also be used on Java code, if one of the set-up files is modified.

The structured diagrams in EasyCODE are created by arranging constructs sequentially or nested, and by filling them with text. Together, both the text and the graphic representation help describe the function of the construct.

EasyCODE can be used to create programs from scratch, or can take existing code and place it into a structured diagram. As well as displaying structured diagrams, the EasyCODE editor can be linked to a compiler, so that a user never needs to move code from one editor to another. The information that EasyCODE uses to construct its diagrams is stored in the source file along with the code.

Figure 12 shows a class method called `check`. In the main EasyCODE window, each rectangle either contains text, or a fuller definition of a function, method or procedure. If you select any of the text area of the screen, you are able to modify or add to the text within that area. At any time you can select from the Insert menu different constructs you can place in your program, e.g. if, for, while, class, etc. If you select the area outside of the method or function, you are taken back to the previous screen.

EasyCODE manages to support three features from the criteria list [32]. Firstly, EasyCODE presents the code in structured diagrams, providing a presentation model for visualisation. Secondly, the structured diagrams are able to present the code in such a fashion as to facilitate the use of some of the comprehension models. The ability to zoom in and out of the methods and functions supports top-down comprehension, where as the structured diagrams themselves support bottom-up comprehension by the layout style used to present the code. Finally, EasyCODE allows decomposition of a class with the zoom in and out technique for methods and functions.

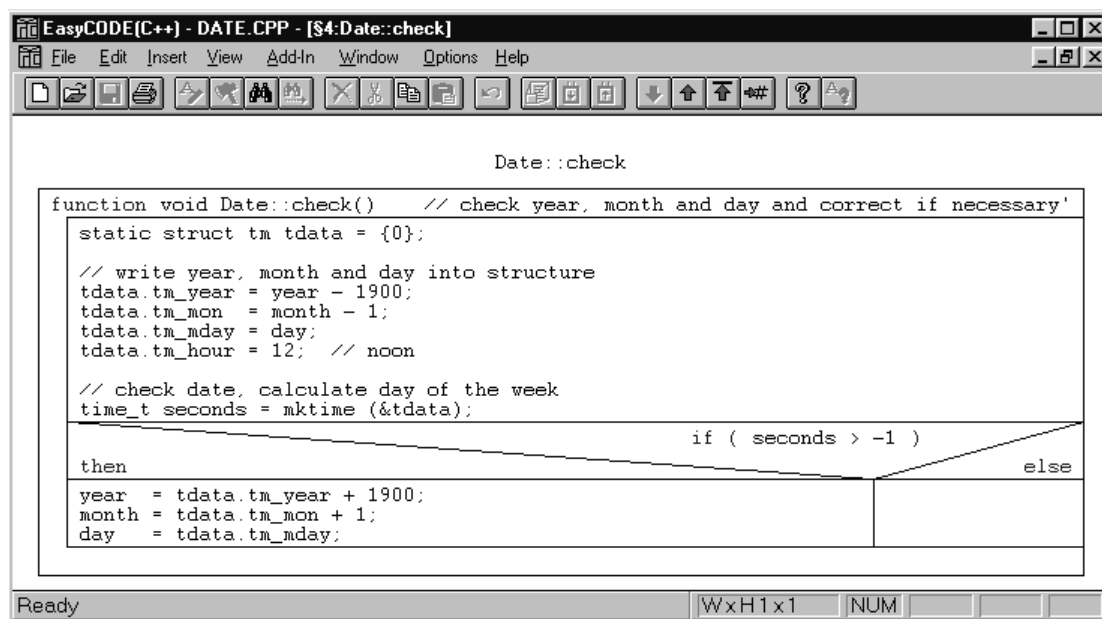


Figure 12. A class method called check() in EasyCODE(C++)

6.2.2 With Class 98

With Class 98 [42] is an object-oriented CASE tool developed by MicroGold Software for windows on the PC. The program allows the construction of graphical models in an object-oriented methodology. With Class allows you to select from several different object-oriented methodologies. These include Unified Method, Rumbaugh, Coad-Yourdon, Booch, Shlaer-Mellor and Martin-Odell. You are not restricted to using one of these for a specific diagram as you can move freely from one type to another, and the current diagram automatically updates itself.

Using With Class, you can build up class diagrams, detailing class attributes and methods. If you select either the class attributes or the class methods, you are presented with a series of forms. These forms allow you to define the specifics. In the case of class attributes you can specify their type, initial values, visibility (e.g. private, protected, etc.) and general comments describing their attributes. The method forms allow you to define method definitions, return types, method code and various other pieces of information.

If you create your diagrams but don't enter any specific code, then you can use With Class 98 to convert your class diagrams into code, leaving only the method specifics to enter. The languages that the diagrams can be converted to consist of C++, Eiffel, Ada, Object Pascal, Smalltalk, Visual Basic, Java, IDL, SQL and others. This is done by way of a script, which can be modified. Other scripts could also be written for other languages if needed.

As well as creating class diagrams, you can also create use case diagrams, state diagrams component diagrams, sequence diagrams, and many others. Several different reports can also be generated, e.g. State Code/Report which contain transition code/reports from a state diagram or Object Code/Report which contain information on object interaction code/reports from an object interaction diagram.

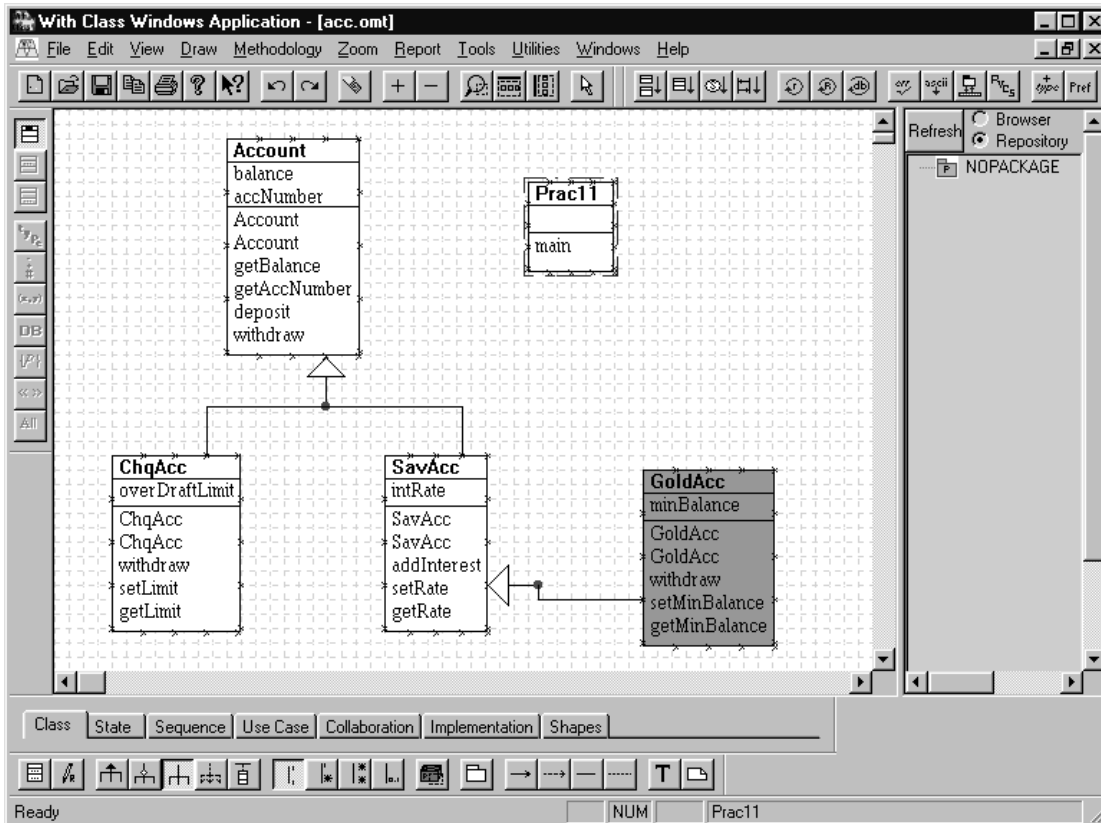


Figure 13. The main window of With Class 98

A useful feature of With Class 98 is its ability to reverse engineer code, e.g. you can direct With Class to a directory that contains code files and it can take the code, and from it create the class diagrams and their relationships to each other. This can be done for C++ header files, Object Pascal, Java, IDL, Visual Basic and ODBC database files. **Figure 13** shows a view of the main With Class 98 window. In the editor is a simple account class diagram and you can see the inheritance hierarchy, the attributes of each class, along with the methods for each class.

With Class 98 fulfils several of the comprehension tool criteria [32]. Firstly, it allows the transformation of one graphical object-oriented methodology to another. Secondly, various windows within the tool decompose the code into individual methods, method declarations, and method bodies. Thirdly, the class hierarchy is presented using a graphical object-oriented methodology, providing information visualisation. Fourthly, top-down comprehension is supported through the use of diagrams representing an overview of the class hierarchy. Individual classes and their methods can then be selected and zoomed in on. Fifthly, With Class 98 provides a repository for storing information on classes and packages. Lastly, because of the partitioning of the code into individual windows, spaces are provided for comments, preconditions, etc. If With Class is used through the entire development of a piece of software then it can be used to keep source-code documentation. All these features help With Class 98 satisfy six out of the seven comprehension criteria.

6.2.3 SNiFF+

SNiFF+ [62] is a software development environment for C, C++, Java, Fortran and IDL. The environment provides many features including version and configuration management, project management, code comprehension and browsing, document building, developing, and debugging. SNiFF+ contains filtering and visualisation techniques that work with large projects (i.e. thousands of files and millions of lines of code).

SNiFF+ requires only the source code, no compilation is required as SNiFF+ has its own parsers. The information created by SNiFF+ parsing the source code is stored in a symbol table, which acts as a common data source for all the tools in the environment.

The SNiFF+ documentation highlights some of its tools that can be used for code comprehension and browsing. The symbol browser can display lists of the methods, classes, macros, interfaces, etc. of the current project. A filter can be used to shorten the lists. The hierarchy browser (**Figure 14**) displays the class hierarchy of the project, a list of all the classes, and the code of the currently selected class. Specific classes can be selected, and their relatives can be shown in a hierarchical diagram, removing any unrelated classes from the diagram, so you can easily see what classes it inherits from and are inherited by. The class browser (**Figure 14**) displays a list of the locally defined and inherited members of a class or structure. A wide range of filtering possibilities based on the inheritance, visibility and type of the members are provided. A small symbol is displayed beside each member, representing visibility and other attributes of the members. They include normal member (yellow: public, blue: protected, dark grey: private), virtual member, member overridden in a subclass, member overrides a member of a base class, member is overridden in a subclass and overrides a member of a base class, and static member. The cross referencer displays a dependency tree showing what refers to or is referred by a selected symbol. The document editor allows the creation and modification of source-code documentation. The retriever (**Figure 14**) allows you to search through the source code for a specific piece of text, and then displays the files and positions of the desired text. The include browser displays a hierarchy of include references between files in the currently selected projects. It is similar in both layout and functionality to the cross referencer tool. The include browser can be used to see which files are included by a particular file and which files include a particular file, as well as checking to see if there are any include files that are not used.

Several features of the SNiFF+ environment make it a comprehension tool [32]. A document editor is provided to allow creation and modification of source-code documentation. Several of the in-built tools provide mechanisms to remove unwanted information. SNiFF+ provides several different windows, some of which have been mentioned previously. These windows decompose the visualisations provided by SNiFF+ into more manageable chunks. The top-down comprehension strategy is supported through the ability to select the names of methods or classes in one window, which then brings up another window displaying the desired code. SNiFF+ provides various presentation models including a class browser and a hierarchy browser. Finally, SNiFF+ provides the ability to move between the different representations offered by the selection of tools.



Figure 14. The Hierarchy Browser, Class Browser and Retriever from SNIFF+

6.2.4 ISVis

ISVis [26], written by Dean F. Jerding, helps visualise interaction patterns in executing programs on the Solaris, SunOS and IRIX platforms. The program was designed to handle large amounts of real information and be able to carry out abstractions, simplifying the data being looked at.

Jerding [27] observed that “program executions are made up of recurring interaction scenarios and that these interaction patterns occur at various levels of abstraction”. This led to his general thesis statement “Visualizing interaction patterns in program executions can facilitate program behavioural understanding

during design recovery, design/implementation validation, and reengineering tasks”. Part of Jerdings work led to the creation of ISVis.

ISVis creates visualisations for C++ code. There are several steps that have to be followed before you can start working with visualisations. Firstly, ISVis needs static information on the code. This can be automatically generated by the CC compiler. Once the static information is obtained, ISVis then uses this to instrument the C++ code. Once the code has been instrumented, the C++ code is recompiled. The instrumentation allows a trace file to be generated, which traces function/method calls that occur during program execution. To obtain the trace, the recompiled code is run. As the code runs, the trace file is generated. Once the trace file is generated, it is read by ISVis, and in conjunction with the static information, is used to generate the visualisations.

The visualisations take the form of interactions of functions, classes, threads, files, or other parts of the system. The interactions can be function calls, function returns, data reference, object instantiation or deletion, or message passing. Looking at these visualisations, it should be possible to see repeated sequences or patterns.

Where ISVis is supposed to help is in the process of abstracting high-level program behaviour from the low-level interaction patterns. Groups of interactions can be selected, highlighted and coloured, then the graphics display can be updated to show how the interactions look with the selected groupings, ideally simplifying the interactions, producing more patterns and greater understanding of the executing program.

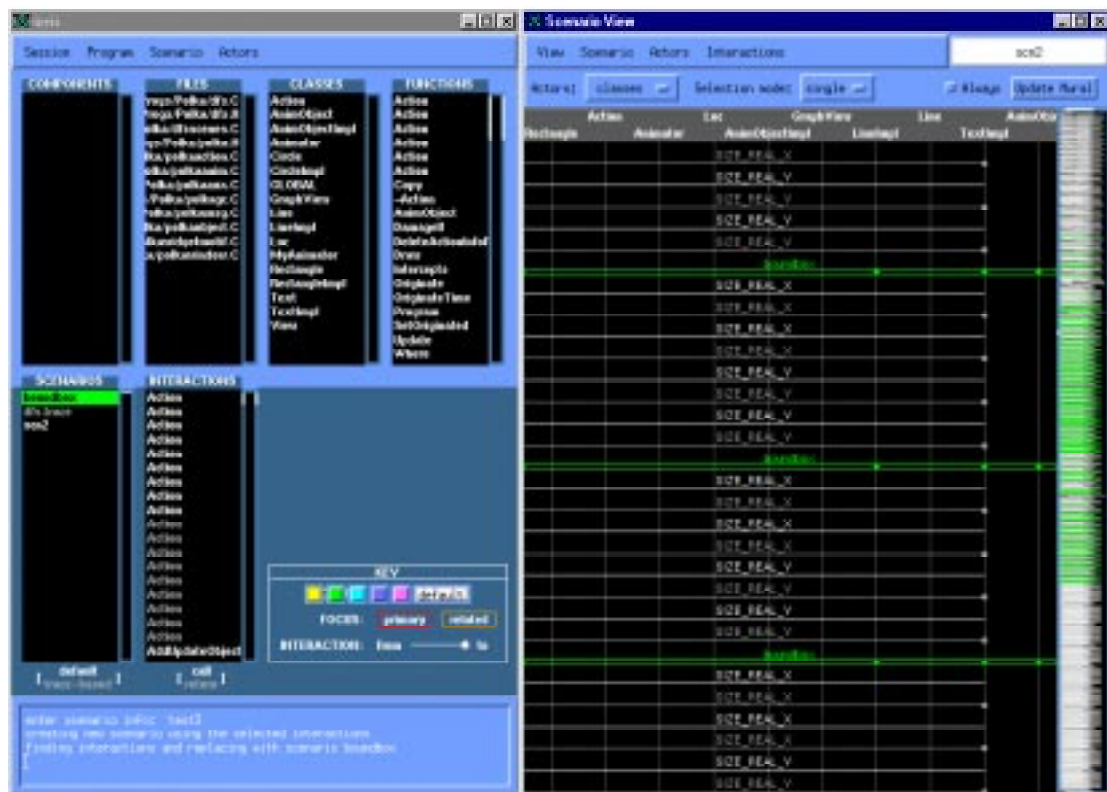


Figure 15. View of the main ISVis window and the Scenario View window

When ISVis displays the program interactions, it also displays an Information Mural (see right side of Scenario View in Figure 15). The information mural is a two dimensional graphical representation used to visualise large amounts of information.

In ISVis, the information mural can be used to navigate through the interactions in time order, or can be used to spot patterns.

Shown in **Figure 15** are two windows from ISVis. The window on the left is the main ISVis window, where you can perform static analysis of code, instrument source code and read in trace information. You are also presented with a summary of what ISVis is looking at, presented in several scrolling boxes. In the window on the right, the Scenario View presents the interactions that occur between classes, functions, files, etc. ISVis relies on being used on a colour display. When building up abstractions, you can colour code them, making them easier to see, and get a feel for any patterns. In the example mural shown in **Figure 15**, several interactions were grouped together under the heading **boundingbox** and this group was given the colour green. A pattern can be seen in the information mural, where some repetitive operation is being carried out in the middle of the execution of the program.

ISVis has a presentation model in its information mural, which represents the interactions of functions, classes, threads, files, etc. These interactions can be grouped together, either to remove unwanted detail or in an attempt to spot interaction patterns. Both of these points conform to the comprehension criteria [32].

6.2.5 Look!

Look! [47] is a C++ debugging and visualisation tool available for Windows, SunOS, Solaris and AIX, and developed by Objective Software Technology Ltd. Look! provides the facility to find out information about an executing object-oriented C++ application.

Various views are available in Look!, and these include object reference structures, object creation relationships, class clusters, object networks, message flow and dynamic class views. To be able to use Look! on an application, you have to compile the code to be visualised and debugged with the compilers debug flag set. Once this is done, the executable can be loaded into Look!. Finally, you can select the views you wish to see, and run the application through Look!.

In the object views, objects are represented by icons which describe the current state and memory allocation of each object. The possible memory allocations available are static, heap, temporary, automatic, aggregate and base. Each of these is represented by its own icon. The background colour of each icon represents the state of the object, e.g. green represents an active object, whereas red represents an orphaned object. For this to happen, at some point an object must have been deleted which was referencing other live objects.

Shown in **Figure 16** is a selection of some of the window views available in Look!. The top left window is the main control panel that allows you to select which of the available views you wish to see, allows the execution of the current program to be started and stopped or moved on a line at a time. A sliding bar is used to control the execution speed of the program.

In the Reference View, a reference relationship diagram is shown. In the example view, three objects can be seen. Each object is an instance of a different bank account; in this case `acc1` is an instance of the cheque account class (`ChqAcc`). The Global object represents the main function. Arrows shown beside Global and `acc1` show that a message is being sent from Global to `acc1`. The particular message is shown in a box, in this case the `setLimit` method of the `ChqAcc` class is being called.

hierarchy and object reference viewer. Look! follows the execution of a program, and all the displays are updated as the program executes. A correlation can be made between what code is being executed, what messages are being passed, and what the states of the objects within the executing program are. These features facilitate easy movement from one representation to another. As with SNIFF+, Look! has many different windows that help decompose the large amounts of information being generated into smaller, easier to follow sections. Many of the features of Look! could be used to support the top-down comprehension process, e.g. view the execution of a program using the Reference Viewer to build up hypotheses of what is occurring during execution, before using other low level viewers like the code or data browsers to confirm the detail. With all of these points, Look! fulfils many of the comprehension criteria [32] to be considered as a comprehension tool.

6.2.6 Jinsight

Jinsight [25] is a Java visualisation tool created by IBM. It shows the execution behaviour of Java programs and is available for Windows 95/NT. Some of the available windows are shown in **Figure 17**.

Jinsight gets its information from traces of a previously executed program. The Java program is compiled as normal, and then a modified Java VM which is supplied with Jinsight is used to run the code. As the program is running the modified Java VM creates trace information and stores it in another file. This file can then be loaded into Jinsight, where different views are available to analyse the trace information.

Jinsight has the ability to show object population, messages, garbage collection, CPU and memory bottlenecks, thread interactions, and deadlocks. Jinsight has several different displays available from which to view the program execution. The Histogram view displays calling and reference relationships among objects. The Execution, Invocation Browser, and Execution Pattern views display sequences of messages among objects as a function of time. There is also a Reference Pattern view, which displays patterns of references among objects. This view can also be used to help find memory leaks. In this view, it can be seen which objects are holding references that are causing problems for the garbage collector.

To help alleviate the problem of information overload with very large traces, Jinsight provides a pattern extraction facility. This allows recurring patterns to be displayed in a single view. This can help remove large amounts of redundant information, simplifying the display of interactions.

Many of Jinsight's views involve sequence displays of messages. A primitive abstraction mechanism is provided to clear up the cluttered and sometimes confusing views, but is not as user friendly as the one provided by LOOK!. No conversions between graphics and code are available, but several presentation modes for visually displaying how the program executed are provided. These features, although not as competent as those found in Look!, are enough for Jinsight to pass the comprehension tool criteria [32].

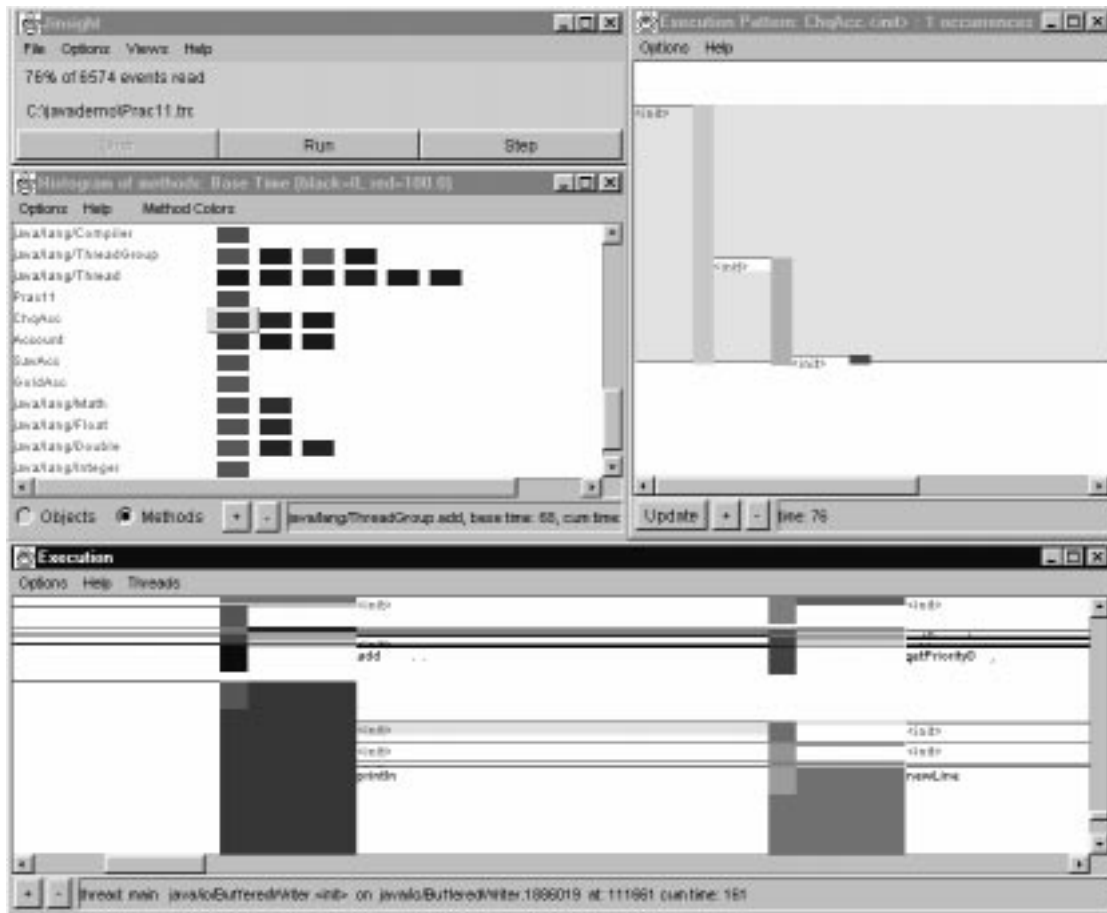


Figure 17. Several windows from the Java Jinsight visualisation tool.

6.3 Summary

Table 3 summarises the list of inspection and visualisation tools looked at, and shows which of the Linos [32] criteria they match. The numbers 1-7 represent each of the criteria in the order they were presented in Section 3.2.

From the inspection tools looked at, only two had any form of comprehension aid, which took the form of an on-line checklist. The checklist can be seen as a form of as-needed comprehension, thereby satisfying the first comprehension tool criteria. The reason there is a lack of useful comprehension aids within inspection tools is because they are usually geared towards easing the burden for the group part of inspection, e.g. summarising defect lists, compiling inspection statistics, allowing for members of the group to be in different places, etc.

Each of the visualisation tools looked at fulfilled one or more of the criteria suggested by Linos for the tool to be accepted as a program comprehension tool. Although all the tools could be used for comprehension, the situations in which each of the tools could be used and their difficulty in use vary.

From the visualisation tools surveyed, the static tools seem to offer the quickest and easiest to learn aids for program comprehension. From the static tools looked at, both With Class 98 and SNIFF+ offer a comprehensive set of features. Look! provided some very useful features and was the best and easiest to use of the

dynamic tools looked at, but it is still debatable whether it could or should be used during inspection due to its dynamic nature.

	1	2	3	4	5	6	7
ASSIST	✓	✗	✗	✗	✗	✗	✗
Scrutiny	✗	✗	✗	✗	✗	✗	✗
ICICLE	✗	✗	✗	✗	✗	✗	✗
Collaborative Software Inspection	✗	✗	✗	✗	✗	✗	✗
WiP	✓	✗	✗	✗	✗	✗	✗
EasyCODE (C++)	✓	✗	✓	✗	✓	✗	✗
WithClass 98	✓	✓	✓	✓	✓	✗	✓
SNiFF+	✓	✗	✓	✓	✓	✓	✓
ISVis	✗	✗	✓	✗	✗	✓	✗
Look!	✓	✗	✓	✓	✓	✓	✗
Jinsight	✗	✗	✓	✗	✗	✓	✗

Table 3. Inspection and Visualisation tools compared to the Linos [32] criteria

The following points highlight some of the useful features found in both the static and dynamic visualisation tools looked at. Further empirical research should be carried out to discover if any of these features would help in detecting defects in inspection.

- Class diagrams showing both class hierarchy and methods
- Ability to zoom in on individual classes and their methods through the use of several windows (as in With Class 98)
- Modify class hierarchy diagrams to show only related classes
- Show all methods and inherited methods for a selected class, detailing their visibility
- Search for all references to a particular class, method or variable and the ability to filter this information
- Show current executing class/line of code
- Check current values of variables within executing program
- Show messages that are passed between objects and the values passed within them

8. Future Tool Support

The previous section looked at several visualisation and inspection tools and compared them with a comprehension tool criteria put forward by Linos [32]. The inspection tools looked at suffered from a lack of comprehension support apart from ASSIST and WiP, which contained support for on-line checklists (a form of as-needed comprehension). This is probably due to the fact that most of the tools were primarily designed to support distributed inspections and enforce the overall inspection process. One common feature between all of the inspection tools was in

the treatment of the code itself, i.e. only utilising the code by itself with no static or dynamic visualisations.

The majority of current inspections rely on paper based material. In part, this can be attributed to the reluctance of some to move to computer aided assistance. One of the suggested benefits of inspections is the inspector learning from the mistakes found in the code being inspected. If the inspector can learn from the mistakes found, then the theory is that the code the inspector writes in the future should have fewer errors. Many feel that introducing computer aids into the inspection process will reduce this effect. Currently there is no empirical evidence to suggest whether this is true or not.

Macdonald [36] believes that if any reasonable inspections on object-oriented code are to be carried out, some form of dynamic inspection may have to be introduced. This author also believes that some form of dynamic inspection may be desirable, but it is not known whether this would be practical or merely distract the inspector. Even if some form of dynamic visualisation is not beneficial, some of the visualisation techniques found within the static tools described in the summary of the previous section could be utilised.

Some of the current inspection tools, including Macdonald's ASSIST have been created with an open architecture to allow for the addition of other modules or editors. In the short term this could allow for the use of some commercially available comprehension or visualisation tools in inspections. In the long term, if research and empirical evidence can show that using comprehension and visualisation can improve inspections of object-oriented code, then a tool that combines all three aspects would be desirable.

7. Conclusions

Current inspections all stem from the original created by Fagan [14] in 1976, and are widely accepted method for finding defects in both code and documents. There are many different variations of the inspection process, but most have a preparation stage before the group inspection, where individual inspectors are required to understand the material and role they have been given. Fagan himself described the process for inspectors as "using the design documentation, literally to do their homework to try to understand the design, its intent and logic".

Comprehension is an integral and important part of the preparation stage of inspection. Rifkin believes strongly about comprehension, stating that "you cannot inspect what you cannot understand" [54]. He also suggests that because of this, new entry criteria for inspection should be defined only allowing code that is comprehensible through to the inspection stage. Many inspection guidelines however leave it up to the individual inspector to decide how to go about comprehending the given documents [18], [67], and to what degree comprehension is required.

Current methods used during the preparation stage; checklists and scenarios are advertised as defect detection aids rather than comprehension aids. In the case of checklists, they generally guide the inspector to look towards certain areas of code that have given the most problems in the past. Scenarios attempt to improve on this by having several people, each with a different perspective from which to look at the software artefact or code, answer several questions they have been given. This method however still tends to focus the attention of the inspector towards certain areas of code. The inspector is not actively encouraged to fully comprehend what

he/she is looking at. There is currently limited research available that can positively describe the impact of ad-hoc, checklist or scenario defect detection methods. The question remains whether increasing the amount of comprehension required (and supported) would enhance their defect detection performance.

Rifkin and Deimel [54] suggest that software engineers have poor strategies when attempting to understand a given document for inspection. If this is the case, it could mean many defects are going unnoticed in inspections. From this Rifkin and Deimel suggest that a better understanding and fuller use of program comprehension techniques could improve an inspectors performance.

Program comprehension is a technique commonly used by software engineers when attempting to understand a section of code. During the software maintenance process, a large percentage of the total time is spent on program comprehension. This can be anywhere up to 60% of the total time [9]. Because of this large amount of effort, research is continuing to determine how a person goes about understanding a program, and what can be done to help them in this process. From some of this research it has emerged that there is no one cognitive model used by programmers to comprehend a program. The more popular models are top-down, bottom-up and integrated, which is a combination of both top-down and bottom-up comprehension. It has emerged that programmers either use one cognitive model throughout, or more frequently swap between models [41].

One of the cognitive models described was that of as-needed comprehension. This strategy involves the software engineer looking only at code related to a particular problem or task. This sounds very much like the description of checklists and scenarios. A problem with as-needed comprehension is that it tends to miss some of the dependencies within code fragments [75].

A search of the current literature has found no guidance on how to inspect object-oriented code, a possibly serious omission, considering the increase in popularity of the paradigm. The object-oriented paradigm introduces many new problems to the inspection process not found in traditional procedural languages, e.g. polymorphism and dynamic binding, inheritance, class structure and object hierarchy, generic classes, and a dramatic increase in the number of dependencies found in program code. A further problem for the inspection process is how to split up object-oriented code for inspection. With average class method sizes being very small, in some cases just a few lines of code, program functionality can be spread right through the program.

All of the previously mentioned object-oriented features increase dramatically the number of dependencies within the code, many more than that of procedural code. This could have a serious impact on the effectiveness of checklists and scenarios, both of which are forms of as-needed comprehension, when used to find defects in an object-oriented inspection. This suggests that other program comprehension strategies should be used when inspecting object-oriented code, which would allow full comprehension of all the dependencies within a code segment.

Program visualisation tools are used to "enhance the art of program presentation and thereby to facilitate the visualisation, understanding, and effective use of computer programs by people" [2]. A visualisation taxonomy by Myers [44] defined two forms of visualisation, code and data. Myers further subdivided code visualisations into static and dynamic visualisations. Static visualisations are based on information that can be obtained purely from the code, whereas dynamic visualisations are based on information gathered from an executing program. In the past, inspections have been restricted to non-execution of code to allow inspection of

code soon after creation and to allow inspectors to learn from mistakes found in inspected code. Static visualisations can fit into this current philosophy of non-execution, as their visualisations are abstracted from the code only. Dynamic visualisations of code go against traditional inspection ideals. Macdonald [36] however highlights that a lack of support for dynamic inspections of object-oriented code could reduce the effectiveness of the defect finding process.

The last decade has seen a dramatic increase in the number of comprehension tools becoming available ([53] presents a bibliography of some comprehension tools). This could be due, in part to the increases in computer technology, but could also be due to a recognised need for additional aids for the program comprehension process. Some of the most recent tools being advocated for use in program comprehension have actually been program visualisation tools [65].

The object-oriented paradigm has also seen an increase in the number of support tools created for it. Many of these have been visualisation tools, e.g. SNiFF+ [68], With Class 98 [42], Jinsight [25], Look! [47], and ISVis [26]. Some of the tools like SNiFF+ and With Class 98 create their visualisations from static source code information, and others such as Look!, Jinsight and ISVis create their visualisations through dynamic, run time execution information. Linos [32] created a comprehension criteria, by which all of these object-oriented tools could be considered as comprehension tools. Both SNiFF+ and With Class 98 static tools offered several useful features to show the relationships within an object hierarchy, and Look! highlights the usefulness and immediacy of dynamic visualisations.

If program comprehension is recognised as being vitally important to the software maintenance process [9], [69], where a complete understanding of the code being looked at is vital, then surely it makes sense to introduce program comprehension into the code inspection process. To further increase the level of comprehension during inspections, especially with object-oriented code, visualisation tools could also be introduced, especially into the preparation stage.

This paper has attempted to investigate the possible usage of program comprehension techniques and visualisation as aids to inspect object-oriented code within the preparation stage of the inspection process. The following are a series of research questions, suggesting possible areas of future research highlighted by ideas from within this paper:

- What are the levels of comprehension obtained by the use of checklists or scenarios during the preparation stage of inspection?
- What are the current industrial practices concerning inspection of object-oriented code? To what degree is comprehension encouraged/required during inspection?
- Can an object-oriented program be meaningfully separated for inspection?
- Can program comprehension techniques and visualisation tools be used to improve the inspectors comprehension of object-oriented code?

References

- [1] R. M. Baecker, Experiments in on-line graphical debugging: The interrogation of complex data structures (summary only), *Proceedings of the First Hawaii International Conference on the Systems Sciences*, pp. 128-129, 1968.
- [2] R. Baecker, Enhancing Program Readability and Comprehensibility with Tools for Program Visualization, in *Proceedings of the 10th International Conference on Software Engineering*, pp. 356-366, April 1988.
- [3] J. Barnard and A. Price, Managing Code Inspection Information, *IEEE Software*, Vol. 11, No. 2, pp. 59-69, 1994.
- [4] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, S. Sørungård, M. V. Zelkowitz, The Empirical Investigation of Perspective-Based Reading, *Empirical Software Engineering, An International Journal*, Volume 1, Number 2, pp 133-164, (Also available as Technical Report ISERN-96-06), Kluwer Academic Publishers, October 1996.
- [5] G. Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings Publishing Company, Inc., 1994.
- [6] R. Brooks, Towards a theory of the comprehension of computer programs, *International Journal of Man-Machine Studies*, 18:543-554, 1983.
- [7] M. H. Brown, *Algorithm Animation*, Ph.D. Dissertation, Brown University, Department of Computer Science, Providence, RI, 1987, published also by MIT Press, 1988.
- [8] Bull HN Information Systems, Inc., U.S. Applied Research Laboratory, *Scrutiny User's Guide*, May 1994.
- [9] G. Canfora, L. Mancini and M. Tortorella, A Workbench for Program Comprehension during Software Maintenance, *4th Workshop on Program Comprehension*, IEEE Computer Society Press, pp. 30-39, 1996.
- [10] X. Chen, W. Tsai, and H. Huang, Omega - an Integrated Environment for C++ Program Maintenance, *International Conference on Software Maintenance*, November 4-8, 1996.
- [11] E. J. Chikofsky and J. H. Cross II, Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, January 1990.
- [12] J. Daly, A. Brooks, J. Miller, M. Roper, and M. Wood, Evaluating inheritance depth on the maintainability of object-oriented software, *Empirical Software Engineering: An International Journal*, 1(2), pp. 109-132, 1996.
- [13] S. Ellershaw and M. Oudshoorn, *Program Visualization - The State of the Art*, Department of Computer Science, University of Adelaide, TR 94-19, 1994.

- [14] M. E. Fagan, Design and Code Inspections to Reduce Errors in Program Development, *IBM Systems Journal*, No. 3, pp. 184-211, 1976.
- [15] M. E. Fagan, Advances in Software Inspections, *IEEE Transactions on Software Engineering*, Vol. 12, No. 7, July 1986.
- [16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns : Elements of reusable object-oriented software*, Addison-Wesley Publishing Company, 1994.
- [17] N. Gershon, S. G. Eick, and S. Card, Information Visualization, *Interactions*, Vol. 2, March + April, 1998.
- [18] T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley, 1993.
- [19] J. W. Gintell, J. Arnold, M. Houde, J. Kruszelnicki, R. McKenney and G. Memmi, Scrutiny: A Collaborative Inspection and Review System, in *Proceedings of the Fourth European Software Engineering Conference*, Garwisch-Partenkirchen, Germany, September 1993.
- [20] L. Harjumaa and I. Tervonen, A WWW-based Tool for Software Inspection, in *31st Hawaii International Conference on Systems Sciences*, Volume III, pp. 379-388, 1998.
- [21] D. Hart, E. Kraemer and G. Roman, Interactive Visual Exploration of Distributed Computations, in *Proceedings of the 11th International Parallel Processing Symposium*, pp. 121-127, April 1997.
- [22] W. S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Mass., 1989.
- [23] W. S. Humphrey, *A Discipline for Software Engineering*, Addison-Wesley Publishing Company, 1995.
- [24] A. Hyrskykari, Development of Program Visualization Systems, *presented at 2nd Czech British Symposium of Visual Aspects of Man-Machine Systems*, March 27 1993, Praha.
- [25] IBM, *Jinsight*, <http://www.alphaworks.ibm.com>
- [26] D. F. Jerding, *ISVis*, <http://www.cc.gatech.edu/morale/tools/isvis/isvis.html>
- [27] D. F. Jerding, *Visualizing Interaction Patterns in Program Execution*, PhD Thesis, Georgia Institute of Technology, November 1997.
- [28] C. Jones, Gaps in the object-oriented paradigm, *IEEE Computer*, Vol. 27, No. 6, June 1994.

- [29] E. H. Khan, M. Al-A'ali, and M. R. Girgis, Object-Oriented Programming for Structured Procedural Programmers, *IEEE Computer*, pp. 48-57, October 1995.
- [30] T. Korson and J.D. McGregor. Understanding Object-Oriented: A Unifying Paradigm", *Communications of the ACM*, Vol. 33, No. 9, September 1990, pp. 40-60.
- [31] O. Laitenberger and J.-M. DeBaud, Perspective-based Reading of Code Documents at Robert Bosch GmbH, *Special Issue on Information and Software Technology*, Nov. 1997.
- [32] P. K. Linos, *A Preliminary Report on Program Comprehension Tools (PCT's)*, Tennessee Technological University, <http://www.csc.tntech.edu/~linos/pcts.html>
- [33] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, Mental models and software maintenance, In *Empirical Studies of Programmers*, pp. 80-98, Ablex Publishing Corporation, 1986.
- [34] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. A Review of Tool Support for Software Inspection, In *Proceedings of the Seventh International Workshop on Computer Aided Software Engineering*, pages 340-349, July 1995.
- [35] F. Macdonald and J. Miller, Modeling Software Inspection Methods for the Application of Tool Support, EFOCS-16-95, December 1995.
- [36] F. Macdonald, J. Miller, A. Brooks, M. Roper and M. Wood, Applying Inspection to Object-Oriented Software, *Software Testing, Verification and Reliability*, Vol. 6, pp. 61-82, 1996.
- [37] F. Macdonald and J. Miller, Automated Generic Support for Software Inspection, *10th International Quality Week*, San Francisco, 27-30 May, 1997.
- [38] F. Macdonald and J. Miller, A Software Inspection Process Definition Language and Prototype Support Tool, *Software Testing, Verification and Reliability*, Vol. 7, No. 2, pp. 99 - 128, June 1997.
- [39] B. Marick, *The craft of software testing : subsystem testing including object-based and object-oriented testing*, Prentice Hall, 1995.
- [40] V. Mashayekhi, J. M. Drake, W. Tsai, and J. Riedl, Distributed, Collaborative Software Inspection, *IEEE Software*, Vol. 10, No. 5, September 1993.
- [41] A. von Mayrhauser and A. Marie Vans, Program Comprehension During Software Maintenance and Evolution, *IEEE Computer*, Vol.28, No. 8, August 1995.
- [42] MicroGold Software Inc., *With Class 98*, <http://www.microgold.com>

- [43] J. Miller, M. Wood and M. Roper, Further Experiences with Scenarios and Checklists, *Empirical Software Engineering*, Vol. 3, pp. 37-64, 1998.
- [44] B. A. Myers, Visual programming, programming by example and program visualization : a taxonomy, *Proceedings of the SIGCHI'86*, pp. 59-66, 1986.
- [45] B. A. Myers, Taxonomies of Visual Programming and Program Visualization, *Journal of Visual Languages and Computing*, 1 (1), pp. 97-123, 1990.
- [46] J. Nielsen and J. T. Richards, The Experience of Learning and Using Smalltalk, *IEEE Software*, May 1989.
- [47] Objective Software Technology Ltd., *Dynamic Visualization of Object Programs written in C++*, <http://www.objectivesoft.com>
- [48] P. Oman, Maintenance Tools, *IEEE Software*, May 1990.
- [49] M. Petre, A. F. Blackwell, and T. R. G. Green, Cognitive Questions in Software Visualisation, To appear in J. Stasko, J. Domingue, B. Price, and M. Brown (Eds.), *Software Visualization: Programming as a Multi-Media Experience*, MIT Press, January 1998.
- [50] A. A. Porter and L. G. Votta, An Experiment To Assess Different Defect Detection Methods For Software Requirements Inspections, *Proceedings of the Sixteenth International Conference on Software Engineering*, pp. 103-112, 1994.
- [51] A. A. Porter, H.P. Siy and L.G. Votta, A Survey of Software Inspection, *Advances in Computers*, Vol. 42, pp. 40-76, November 1996.
- [52] B. A. Price, R. M. Baecker and I. S. Small, A taxonomy of software visualization, *Proceedings of the 25th International Conference on Systems Sciences*, Vol. II, pp. 597-606, 1992.
- [53] *Program Comprehension Tools Bibliography:*
<http://www.csc.tntech.edu/~linos/pctinfo.html>
- [54] S. Rifkin and L. Deimel, Applying Program Comprehension Techniques to Improve Software Inspections, *Presented at the 19th Annual NASA Software Engineering Laboratory Workshop*, Maryland, Nov. 30-Dec. 1, 1994.
- [55] D. J. Robson, K. H. Bennett, B. J. Cornelius, and M. Munro, Approaches to Program Comprehension, *Journal of Systems Software*, Vol. 14, No. 2, pp. 79-84, February 1991.
- [56] G. Roman and K. C. Cox, Program Visualization: The Art of Mapping Programs to Pictures, in *Proceedings of the 14th International Conference on Software Engineering*, May 1992.

- [57] G. W. Russell, Experience with Inspections in Ultralarge-Scale Developments, *IEEE Software*, Vol. 8, No. 1, pp. 25-31, January 1991.
- [58] V. Sembugamoorthy and L. R. Brothers, ICICLE: Intelligent Code Inspection in a C Language Environment. In *Proceedings of the 14th Annual Computer Software and Applications Conference*, pages 146-154, October 1990.
- [59] J. L. Sharnowski and B. H. C. Cheng, A Visualization-based Environment for Top-down Debugging of Parallel Programs, *Proceedings of IEEE International Parallel Processing Symposium*, pp. 640-645, April 1995.
- [60] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, Inc., 1980.
- [61] Siemens AG Austria, *EasyCODE*, <http://www.siemens.at/~easy/easy/en/easycode.htm>
- [62] SNiFF+, Release 2.4, *User's Guide*, TakeFive Software, <http://www.takefive.com>, January 27th, 1998.
- [63] E. Soloway and K. Ehrlich, Empirical studies of programming knowledge, *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, pp. 595-609, September 1984.
- [64] J. T. Stasko and C. Patterson, Understanding and characterising software visualization systems, *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pp. 3-10, 1992.
- [65] M. Storey, K. Wong, F. D. Fracchia, and H. A. Müller, On Integrating Visualization Techniques for Effective Software Exploration, *Proceedings of IEEE Symposium on Information Visualization*, October 20-21, 1997.
- [66] M. Storey, F. D. Fracchia, H. A. Müller, Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization, *Proceedings of the 5th International Workshop on Program Comprehension*, Dearborn, Michigan, U.S.A., pages 17-28, May 28-30, 1997.
- [67] S. H. Strauss and R. G. Ebenau, *Software Inspection Process*, McGraw-Hill, Systems Design & Implementation Series, 1994.
- [68] TakeFive Software, *SNiFF+*, <http://www.takefive.com>
- [69] S. R. Tilley, S. Paul and D. B. Smith, Towards a Framework for Program Comprehension, 4th Workshop on Program Comprehension, *IEEE Computer Society Press*, pp.19-28, 1996.
- [70] A. B. Watson and J. T. Buchanan, *Towards Supporting Software Maintenance with Visualisation Techniques*, Technical Report R5, University of Strathclyde.

- [71] E. F. Weller, Lessons from Three Years of Inspection Data, *IEEE Software*, Vol. 10, No. 5, pp. 38-45, September 1993.
- [72] N. Wilde, A. Chapman, P. Matthews, and R. Huitt, *Describing object oriented software: What maintainers need to know*, SERC-TR-54-F, Software Engineering Research Center, University of Florida, Gainesville, FL, Feb. 1992.
- [73] N. Wilde and R. Huitt, Maintenance Support for Object-Oriented Programs, *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1038-1044, December 1992.
- [74] N. Wilde, P. Matthews, and R. Huitt, Maintaining Object-Oriented Software, *IEEE Software*, Vol. 10, No. 1, January 1993.
- [75] P. Young, *Software Visualisation*, Visualisation Research Group, Centre for Software Maintenance, University of Durham, 1996.
- [76] E. Yourdon, *Structured Walkthroughs*, Prentice Hall, Englewood Cliffs, N.J., 1989.